10 Spring-EJB Integration

Although Spring is considered an EJB replacement, sometimes, the choice of “replacement” may not be available and one has to integrate Spring with existing EJB systems. For example, your legacy EJB enterprise application has been working just fine and it would be too costly to re-write it in Spring; however, you might want to re-use some functionality of your existing legacy EJB application without having to re-code it in Spring. In such situations, Spring offers solutions for integrating your Spring-based enterprise application with your existing legacy EJB-based enterprise application.

In this chapter, we demonstrate how Spring-EJB integration works using a concrete, Jboss-based EJB 3 sample. We will create an EJB 3 sample first. This sample will serve as the base for our Spring-EJB integration coverage. It provides an opportunity for us to evidence how those complexities imposed on earlier EJB technologies as discussed in my article “A Retrospective View on EJB 2,” posted on this book’s website, have been taken out from EJB 2 to EJB 3. Then, we demonstrate how you can integrate your Spring 3.x enterprise application with your EJB 3.x legacies. The sample is not only fully verifiable in your environment but also adaptable for your project if you have such needs to meet.

10.1 Creating an EJB 3 Sample

In order to demonstrate Spring support for accessing EJB 3 components, we need to create an EJB 3 sample. In this section, we create such an EJB 3 sample using Eclipse and JPA, which will run on Jboss. To make it as realistic as possible, we create a relevant database on MySQL for this purpose in the next section.

10.1.1 Creating the MySQL Database

The procedure for creating a MySQL database for our EJB 3 sample consists of the following two steps:

1. Creating an ejb3db database with the user credentials of “ejb3admin/ejb3admin” by executing the following command (note that you need to enter root’s password for this step):
Creating the two tables of CUSTOMER and BANKINGTX by executing the following command (note that you need to enter ejb3admin’s password for this step, which is the same as ‘ejb3admin’):

```
cmd>mysql ejb3db -h 127.0.0.1 -u ejb3admin -p < create_ejb3_tables.sql
```

Refer to Listings 10.1 and 10.2 for the contents of the two SQL scripts executed above. Note that we have created two tables: CUSTOMER and BANKINGTX, with the BANKINGTX table containing a foreign key (FK) of customer_id referencing the primary key of id from the CUSTOMER table. This constraint will be reflected in the entities we create as is discussed next.

**Listing 10.1 create_ejb3db.sql**

```sql
create database ejb3db;
grant usage on *.* to ejb3admin@localhost identified by 'ejb3admin';
grant all privileges on ejb3db.* to ejb3admin@localhost;
show databases;
```

**Listing 10.2 create_ejb3tables.sql**

```sql
create table customer (id bigint unsigned not null auto_increment primary key,
                        name varchar(30) not null,
                        password varchar(30) not null,
                        email varchar(30) not null,
                        locked tinyint(1) not null);
create table bankingtx (id bigint unsigned not null auto_increment primary key,
                        description varchar(45) default null,
                        amount decimal(10,2) default null,
                        type varchar(30) not null,
                        customer_id bigint unsigned not null,
                        foreign key (customer_id) references customer (id)
                        on delete cascade on update cascade);
```

10.1.2 Creating Entities with an EJB 3 project

As we know, EJB 3 replaced Entity EJBs with *entities* as specified by the Java Persistence API. In this section, we demonstrate how to create EJB 3 entities using the JPA-enabled Eclipse IDE (Juno version) for our EJB 3 sample, which includes the following two tasks:
Task #1: Creating an EJB 3 sample project

To create our EJB 3 sample, we start with creating a new EJB project by clicking File → New → Other → EJB Project on the Eclipse IDE, as is shown in Figure 10.1.

After clicking Next, you should see a dialog similar to Figure 10.2. Now configure this EJB project as follows:

- **Project name**: Enter a project name like “Customer_EJB3.”
- **Target runtime**: Select a Target Runtime like “Jboss 6.x Runtime” from the drop-down list, since we will run this EJB 3 sample on Jboss, and we need the Jboss JPA runtime to proceed. If this runtime does not show up on your IDE, it means that you need to install it. To install the required Jboss 6.x Runtime for this specific EJB 3 sample, click on the New Runtime... button to bring up the dialog similar to Figure 10.3. If you don’t see the Jboss Community category, click the “Download additional server adapters” link above the “type filter text” box and get it installed. Then you should see those Jboss community server runtimes as shown in Figure 10.3.
- **EJB module version**: Select 3.0.
- **Configuration**: Make sure `<custom>` is selected. Also, click *Modify...* and make sure you have selected all project facets including *EJB Module 3.0, Java 1.6* (note that the default 1.7 is not recommended at this time), *JPA 2.0* and *Jboss 6.x Runtime* as shown in Figure 10.4.

**Note**: You might think why we didn’t create a faceted project in the first place by going through the clicks of File → New → Other → General → Faceted Project, and then add EJB modules and classes to it. This approach turned out to be more problematic than creating the EJB project first and then adding the facets as shown above.

![Figure 10.2 Project settings for the EJB 3 sample project](image-url)
Figure 10.3 All Jboss Runtimes available as plug-ins on Juno Eclipse IDE
After making sure all project settings have been entered as required as shown in Figure 10.2, click *Next* and then *Next* again. In the new dialog, select “Generic 2.0” for *Platform*, and “Library Provided by Target Runtime” for JPA implementation. We also need to create a MySQL Connection, since we have an EJB 3 database to connect to as described in §10.1.1. You can do this by clicking the “Add connection …” link and referencing Figure 10.5.
Now the dialog should look similar to Figure 10.6. Click Next and then Finish to complete this task. When you are prompted for opening the project in Java EE perspective as shown in Figure 10.7, click Yes. Your newly-created Customer_EJB3 project should look similar to Figure 10.8. The next step is to create JPA entities.

![JPA Facet dialog](image)

*Figure 10.6 JPA Facet dialog*
Task #2: Creating JPA entities

In this step, we create JPA entities directly from the two tables of BANKINGTX and CUSTOMER we created previously. Although you can create those JPA entities manually or copy/paste/modify existing ones (if any) with your project, it’s more convenient to create them using an IDE like Eclipse with such built-in functionality. You may also want to know that behind the scenes a plug-in called Dali takes care of all the details of translating the settings entered on the GUI into actual Java code for the JPA entities created. You can check Eclipse help or online for more information about Dali. You may also want to go through a quick tutorial about JPA if you have never been exposed to it. We’ll provide brief explanations about why we choose certain options over others, but we won’t go too deeply into it, as our main focus here is to create an EJB 3 sample so that we can demonstrate how Spring-EJB integration works.
To create JPA entities for our EJB 3 sample, with your Customer_EJB3 project selected, click File → New → Other → JPA → JPA Entities from Tables, as is shown in Figure 10.9. We select this option out of four possibilities under the JPA category (the other three are JPA Entity, JPA ORM Mapping File, and JPA Project), because we already have a database created a priori to work with. This is the preferred approach that you create your database schema first and then your JPA entities as your domain objects out of your database tables.

Click Next and you should see the Select Tables dialog similar to Figure 10.10. Make sure that you select the proper Connection, Schema, and Tables (in our case, the two tables of BANKINGTX and CUSTOMER we created previously). Also make sure that the “Update class list in persistence.xml file” check box is checked, as we need to have all JPA entity classes included in the persistence.xml file, which is the required configuration file for all JPA entities.

Now click Next and you should see the Table Associations dialog similar to Figure 10.11. Make sure that you click the second association and uncheck the “Generate this association” check box in its editing panel. This is because we don’t have a many-to-many association between customers and banking transactions. What we have here is a one-to-many association between a customer and multiple transactions, as one banking transaction cannot be assigned to more than one customer. If you have a situation that a many-to-many association makes sense, then you need to deal with both.

Now click the first association, and click Next, which should bring up a dialog similar to Figure 10.12. Make sure you have selected the following options:

- **Key Generator**: Select “auto” so that database table keys will be generated automatically.
- **Entity Access**: Select “Field”, which means that the instance variables (fields) of the persistence classes will be mapped to the corresponding table columns in the database. “Property” entity access means that getters of the persistence classes will be used to determine the property names that will be mapped to the table columns in the database. Note that JPA favors “Field” access, while Hibernate favors “Property” access.
- **Associations Fetch**: This is about how persisted entity data should be loaded from the database. Eager fetching (or eager loading) means “loading even if data is not needed right away,” while lazy loading means “do not load until data is needed.” The “Default” is to load property values, including references, eagerly while loading collections lazily. This is a delicate performance issue and should be dealt with carefully according to your application context and requirements.
- **Collection Properties Type**: This is about whether the Set or List type should be used for collections. Using List is just fine for most situations.
- **Source Folder and Package**: Eclipse chooses “<project-name>/ejbModule” for Source Folder by default and we use “com.perfmath.ejb3.jpa” for the package name here.

After clicking Next, a dialog similar to Figure 10.13 shows up, allowing each individual entity to be customized. We don’t have a lot to customize here except that we can optionally change the “t” letter from lower case to upper case in the class name of “Bankingtx” to make it look like “BankingTx.” Click “Finish” to complete the two JPA domain object entity classes for this EJB 3 sample. Listings 10.3 and 10.4 show the JPA classes of Customer.java and BankingTx.java. Some brief discussions are provided following Figure 10.13 and prior to Listing 10.3.
Figure 10.9 Creating JPA entities on Eclipse (Juno)

Figure 10.10 Select Tables step in creating JPA entities
Figure 10.11 Table associations step in creating JPA entities
Figure 10.12 Generate Custom Entities dialog
By examining Listings 10.3 and 10.4, we notice that these entity classes are indeed much simpler than the EJB 2 entity beans such as the Customer entity bean discussed in my article “A Retrospective View on EJB 2.” Secondly, notice how succinct these annotations are – there are only four of them here: @Entity annotation for declaring that this class is a JPA entity class, @Id for declaring that this field is an ID field, @GeneratedValue for specifying how ID values should be generated, and @OneToMany or @ManyToOne for declaring the association type. Of course, our sample here is a very simple one, and you might encounter more complicated situations for your project, in which case, you should consult EJB and JPA dedicated texts.
In addition to the entity classes of Customer.java and BankingTx.java, a JPA specific XML file of persistence.xml has also been created, as shown in Listing 10.5. This is a simple file that specifies the persistence-unit name of “Customer_EJB3” and the entity classes created. This file is generic and we’ll have to overwrite it later when we deploy and test it on Jboss.

Note: The type for the locked field of the Customer entity class was byte when created initially by the JPA utility on the Eclipse IDE. We had to manually change it to boolean to be consistent with the simple type convention observed by Java programmers. This is an example of why IDE-auto-generated artifacts may require applying customizations afterwards. An IDE is convenient, but it cannot replace a developer’s deep understanding of how things work at the source code level that only comes from experience.

Next, we demonstrate how to create an EJB 3 stateless session bean as a session façade for accessing the entities created above.

Listing 10.3 JPA domain object entity class Customer.java

```java
package com.perfmath.ejb3.jpa;

import java.io.Serializable;
import javax.persistence.*
import java.util.List;

/**
 * The persistent class for the customer database table.
 */
@Entity
public class Customer implements Serializable {
    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    private int id;

    private String email;

    private boolean locked;

    private String name;
```
private String password;

//bi-directional many-to-one association to BankingTx
@OneToMany(mappedBy="customer", fetch=FetchType.EAGER)
private List<BankingTx> bankingTxs;

public Customer() {
}

public int getId() {
    return this.id;
}

public void setId(int id) {
    this.id = id;
}

public String getEmail() {
    return this.email;
}

public void setEmail(String email) {
    this.email = email;
}

public boolean getLocked() {
    return this.locked;
}

public void setLocked(boolean locked) {
    this.locked = locked;
}

public String getName() {
    return this.name;
}

public void setName(String name) {
    this.name = name;
}

public String getPassword() {
    return this.password;
}
public void setPassword(String password) {
    this.password = password;
}

public List<BankingTx> getBankingTxs() {
    return this.bankingTxs;
}

public void setBankingTxs(List<BankingTx> bankingTxs) {
    this.bankingTxs = bankingTxs;
}

Listing 10.4 JPA domain object entity class BankingTx.java

package com.perfmath.ejb3.jpa;

import java.io.Serializable;
import javax.persistence.*
import java.math.BigDecimal;

/**
 * The persistent class for the bankingtx database table.
 *
 */
@Entity
public class BankingTx implements Serializable {
    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    private String id;

    private BigDecimal amount;
    private String description;
    private String type;

    //bi-directional many-to-one association to Customer
    @ManyToOne

private Customer customer;

public BankingTx() {
}

public String getId() {
    return this.id;
}

public void setId(String id) {
    this.id = id;
}

public BigDecimal getAmount() {
    return this.amount;
}

public void setAmount(BigDecimal amount) {
    this.amount = amount;
}

public String getDescription() {
    return this.description;
}

public void setDescription(String description) {
    this.description = description;
}

public String getType() {
    return this.type;
}

public void setType(String type) {
    this.type = type;
}

public Customer getCustomer() {
    return this.customer;
}

public void setCustomer(Customer customer) {
    this.customer = customer;
}
Listing 10.5 The persistence.xml file for the EJB 3 sample

```xml
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0" xmlns="http://java.sun.com/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
  <persistence-unit name="Customer_EJB3">
    <class>com.perfmath.ejb3.jpa.BankingTx</class>
    <class>com.perfmath.ejb3.jpa.Customer</class>
  </persistence-unit>
</persistence>
```

10.1.3 Creating an EJB 3 Stateless Session Bean

To create an EJB 3 stateless session bean as a façade for accessing the Customer and BankingTx entities, right-click on the Customer_EJB3 project, and then select File → New → Other → EJB → EJB 3 Session Bean (you can refer back to Figure 10.1 to locate the EJB 3 Session Bean option in the New EJB Project Wizard). After clicking Next, you should see a dialog similar to Figure 10.14 with the following default and manually entered options:

- **Source folder**: This was set to Customer_EJB/ejbModule by default.
- **Session Bean Type**: This was set to Stateless by default, which was also what we needed.
- **Bean Package**: Manually entered “com.perfmath.ejb3.ejb” for this session bean.
- **Bean Name**: Manually entered “CustomerManager” for this session bean.
- **Remote Interface Package and Remote Interface Name**: All populated automatically as shown in Figure 10.14.

In is very interesting to see that this process does not do much. It creates two bare-minimum Java files only: The CustomerManager.java file and CustomerManagerBean.java file as shown in Listings 10.6 and 10.7, respectively (we will have to overwrite these two classes heavily later, but the purpose for showing them here as they are is to give you an opportunity to see what have been created by the IDE in the first place). Note that the former is a remote interface, while the latter is the bean implementation class. The annotations are also minimal: only one for each (@Remote for the remote interface, and @Stateless for the bean implementation class). This is significantly simpler than the EJB 2 session bean created with the EJB 2 sample discussed in my article “A Retrospective View on EJB 2” – All the “octopus” ejbXXX () methods required by EJB 2 are gone now with EJB 3! This is a significant improvement, but EJB 3 components are still more complex than POJOs.
Figure 10.14 Creating a new EJB3 session bean

Listing 10.6 Eclipse-created CustomerManager.java

```java
package com.perfmath.ejb3.ejb;

import javax.ejb.

@Remote
public interface CustomerManager {

}
```

Listing 10.7 Eclipse-created CustomerManagerBean.java

```java
package com.perfmath.ejb3.ejb;

import javax.ejb.
```
import com.perfmath.ejb3.ejb.CustomerManager;

public @Stateless class CustomerManagerBean implements CustomerManager {

}

In order to make it as realistic as possible, we need to make some adjustments here. The first adjustment we would make is to make the CustomerManager interface as shown in Listing 10.6 a more generic business interface by removing the following two statements:

import javax.ejb.
@Remote
We then manually add all business methods as shown in Listing 10.8. We can make a local interface (Listing 10.9 (a)) and a remote interface (Listing 10.9 (b)) by extending the generic interface shown in Listing 10.8 (we can also make a Web Services interface out of it, but it is not needed here). This way, the bean implementation class can implement both the local and remote interfaces as shown is Listing 10.10 so that it can be accessed either locally or remotely.

Note: The above practice of creating a common top non-EJB business interface is called the Business Methods Interface pattern. It provides the developer with the flexibility of swapping EJB out for POJOs when such needs arise later.

If you are not familiar with EJB 3 annotations, note the following annotations used in the bean implementation class:

■ @RemoteBinding: Specifies jndi binding for remote access across JVMs.
■ @LocalBinding: Specifies jndi binding for local access within the same JVM.
■ @PersistenceContext: Specifies the persistence context for the persistence unit of “CustomerEJB3” (note that a persistence unit is a group of entities packaged together in an application module). This makes it possible to inject the EntityManager named em into the CustomerManager bean.

In addition, note the use of the persist and createQuery methods of the entity manager for the various business methods implemented. If you are interested in such details, consult dedicated EJB 3 texts.

Note: Before proceeding further, copy/overwrite the CustomerManager.java, CustomerManagerLocal.java, CustomerManagerRemote.java, and CustomerManager.java files contained in the spring-integration.zip download file into the com.perfmath.ejb3.ejb folder of your project.

Next, we demonstrate how to modify the persistence.xml file to use the Jboss Hibernate JPA implementation in the next section.
Listing 10.8 Generic interface CustomerManager.java

```java
package com.perfmath.ejb3.ejb;

import java.math.BigDecimal;
import java.util.List;

import com.perfmath.ejb3.jpa.Customer;
import com.perfmath.ejb3.jpa.BankingTx;

public interface CustomerManager {
    public void createCustomer(String name, String password, String email, boolean locked);
    public List<Customer> findAllCustomers();
    public Customer findCustomerByName(String name);
    public Customer findCustomerById(int id);
    public void saveTxs(int customerId, BigDecimal amount, String description, String type);
    public List<BankingTx> findAllTxs(int customerId);
}
```

Listing 10.9 (a) Local interface CustomerManagerLocal.java

```java
package com.perfmath.ejb3.ejb;

import javax.ejb.Local;

@Local
public interface CustomerManagerLocal extends CustomerManager {
}
```

Listing 10.9 (b) Remote interface CustomerManagerRemote.java

```java
package com.perfmath.ejb3.ejb;

import javax.ejb.Remote;

@Remote
public interface CustomerManagerRemote extends CustomerManager {
}
```
Listing 10.10 Bean implementation class CustomerManagerBean.java

package com.perfmath.ejb3.ejb;

import java.math.BigDecimal;
import java.util.List;

import javax.ejb.Stateless;
import javax.persistence.*
import org.jboss.ejb3.annotation.LocalBinding;
import org.jboss.ejb3.annotation.RemoteBinding;
import com.perfmath.ejb3.jpa.*;

@Stateless
@RemoteBinding(jndiBinding="CustomerEJB3.0/remote")
@LocalBinding(jndiBinding="CustomerEJB3.0/local")

public class CustomerManagerBean implements CustomerManagerLocal,
        CustomerManagerRemote {
    
    @PersistenceContext(unitName="CustomerEJB3")
    private EntityManager em;

    public void createCustomer(String name, String password, String email, boolean locked) {
        Customer customer = new Customer();
        customer.setName(name);
        customer.setPassword(password);
        customer.setEmail(email);
        customer.setLocked(locked);
        em.persist(customer);
    }

    public void saveTxs(int customerId, BigDecimal amount,
            String description, String type) {
        Customer customer = findCustomerById(customerId);

        BankingTx bankingTx = new BankingTx();
        bankingTx.setCustomer(customer);
        bankingTx.setAmount(amount);
        bankingTx.setDescription(description);
        bankingTx.setType(type);
        em.persist(bankingTx);
    }
}
public List<BankingTx> findAllTxs(int customerId)
{
    Query query = em.createQuery("FROM Customer where id=:id");
    query.setParameter("id", customerId);
    Customer customer = (Customer)query.getSingleResult();

    List<BankingTx>bankingTxs = customer.getBankingTxs();

    return bankingTxs;
}
public Customer findCustomerByName(String customerName)
{
    Query query = em.createQuery("FROM Customer where name=:name");
    query.setParameter("name", customerName);
    Customer customer = (Customer)query.getSingleResult();

    return customer;
}
public Customer findCustomerById(int id)
{
    Query query = em.createQuery("FROM Customer where id=:id");
    query.setParameter("id", id);
    Customer customer = (Customer)query.getSingleResult();

    return customer;
}
public List<Customer> findAllCustomers(){
    Query query = em.createQuery("FROM Customer");
    List<Customer> customerList = query.getResultList();

    return customerList;
}

10.1.4 Modifying persistence.xml to Use the Hibernate JPA Implementation

The persistence.xml file shown in Listing 10.5 is generic and lacks required additional information to be usable for us to deploy and run this EJB 3 sample on a vendor-specific EJB 3 container. We pick Jboss-6.1.0.Final to run this EJB 3 sample as was used for running those EJB 2 examples discussed in my article “A Retrospective View on EJB 2.” Therefore, we need to modify the persistence.xml file shown in Listing 10.5 by replacing it with the persistence.xml file from the spring-intergration.zip file downloadable from this book’s
website. See Listing 10.11 for this modified persistence.xml file. By comparing the two persistence.xml files, notice that we have added the following information:

- **transaction-type**: “JTA” as specified.
- **provider**: “org.hibernate.ejb.HibernatePersistence” as specified.
- **jta-data-source**: “java:/DefaultDS” as specified in the mysql-ds.xml file of the Jboss-6.1.0.Final’s default server configuration as explained in my article “A Retrospective View on EJB 2.”
- **hibernate.dialect**: “org.hibernate.dialect.MySQLDialect” as specified.

Next, we create a Jboss specific test client driver for testing this EJB 3 sample in the next section.

**Listing 10.11 Jboss specific persistence.xml file**

```xml
<?xml version="1.0" encoding="UTF-8"?>
persistence
   version="2.0"
   xmlns=http://java.sun.com/xml/ns/persistence
   xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
   xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
   http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
   persistence-unit name="CustomerEJB3"
   transaction-type="JTA">
   provider>org.hibernate.ejb.HibernatePersistence</provider>
   jta-data-source>java:/DefaultDS</jta-data-source>
   <class>com.perfmath.ejb3.jpa.BankingTx</class>
   <class>com.perfmath.ejb3.jpa.Customer</class>
   <properties>
    <property name="hibernate.dialect" value="org.hibernate.dialect.MySQLDialect"/>
   </properties>
   </persistence-unit>
</persistence>
```

10.1.5 Creating a JBoss Specific EJB 3 Test Client

In order to test this EJB 3 sample standalone, we need a test client driver. Listing 10.12 shows such a driver. Since this test client program is similar to the EJB 2 test drivers discussed in my article “A Retrospective View on EJB 2,” we would not explain further about how it works. Instead, in the next section, we explain how to deploy this EJB 3 sample and run it in the same Jboss environment as defined by the same Jboss-6.1.0.Final release we used with the EJB 2 samples discussed in my article “A Retrospective View on EJB 2.”

**Listing 10.12 A Jboss specific EJB 3 test client driver**

```java
package com.perfmath.ejb3.client;

import java.math.BigDecimal;
import java.util.*;
```
import javax.naming. *;
import com.perfmath.ejb3.ejb.*;
import com.perfmath.ejb3.jpa.*;
import com.perfmath.ejb3.util.RandomID;

public class TestCustomerEJB3 {

    public static void main(String[] args) throws Exception {

        InitialContext ctx = null;
        try {
            ctx = getInitialContext();
        } catch (Exception ex) {
            System.out.println("Error in establishing initial context: "+ ex.getMessage());
        }

        CustomerManager customerManager = (CustomerManager)ctx.lookup("CustomerManagerEJB3/remote");

        String customerName = "customer" + (new RandomID()).getId();
        System.out.println("creating " + customerName);
        customerManager.createCustomer(customerName, "password", "customer0@abc.com", false);

        String customerId = customer.getId();
        System.out.println("saving txs for customer with customer ID: "+ customerId);
        customerManager.saveTxs(customerId, new BigDecimal(1000.0*Math.random()), "EJB 3 test", "test");

        System.out.println("find all txs for " + customerName);
        Iterator<BankingTx> iter = txs.iterator();
        while (iter.hasNext()) {
            BankingTx tx = iter.next();
            System.out.println("----------------");
            System.out.println("tx id: "+ tx.getId());
            System.out.println("description: "+ tx.getDescription());
            System.out.println("Type: "+tx.getType());
        }
    }

}
System.out.println("amount: "+tx.getAmount());
}

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);
}

10.1.6 Running the EJB 3 Sample on Jboss

To run this EJB 3 sample on JBoss, perform the following tasks to prepare your environment first:

- Make sure the same Jboss-6.1.0.Final environment you used with the EJB 2 samples as discussed in my article “A Retrospective View on EJB 2” is still available on your machine. If not, install and configure it per instructions given in that article.
- Copy the setJboss.bat and build.xml files from the downloaded Customer_EJB3 project folder mentioned previously into your project. Make sure the JBOSS_HOME and JBOSS_SERVER_CONFIG environment variables are set properly on your system.
- Create a dist folder in your project (if it does not exist) to hold the distribution jar file. Your Customer_EJB3 project should now look similar to Figure 10.15.

(Note: Before proceeding further, make sure you have deleted the EJB 2 sample jar files left in the $JBOSS_INSTALL\server\default\deploy folder if any. In addition, replace “ejb” with “ejb3db” in the <connection-url> element in the mysql-ds.xml file located in the same folder of $JBOSS_INSTALL\server\default\deploy.

Then, perform the following tasks to run this EJB 3 sample:

- Start up a command prompt and make sure all environment variables, including JAVA_HOME, JBOSS_HOME and JBOSS_SERVER_CONFIG, are set properly on your system (you can use the setJboss.bat script included in the EJB 3 sample project download to set your Jboss environment variables, with my JBOSS_HOME replaced with yours in this script).
- Change to your $JBOSS_INSTALL\bin directory and issue the following command to start up your Jboss server:
  cmd>run.bat –b 0.0.0.0
Open up a command prompt, change to your Customer_EJB3 project directory, execute the "setJboss.bat" script, and issue the following command to deploy this EJB 3 sample into your Jboss environment (of course, you should also have configured ANT per instructions given in my article “A Retrospective View on EJB 2” posted on my website):

```
cmd>ant deploy
```

Open up another command prompt, change to your Customer_EJB3 project directory, execute the "setJboss.bat" script, and issue the following command to run the test client for this EJB 3 sample:

```
cmd>ant run-testCustomerEJB3
```

If everything goes well, you should see an example console output similar to the following:

```
run-testCustomerEJB3:
[java] creating customer501521085
```
This wraps up our endeavor of creating an EJB 3 sample and we move to demonstrating how to integrate Spring with EJB 3 in the next section.

10.2 Accessing EJB 3 Components from Spring

Thanks to the concepts of dependency injection and IoC container, accessing EJB components from Spring has been made really simple. The mechanism is as simple as the following three-step procedure:

1. In your Spring bean as the caller of your EJB component, perform the following three tasks:
   - Create an EJB business interface instance variable in your Spring bean to set up a reference for your Spring bean to use the EJB component you intend to use.
   - Create a corresponding setter method so that the EJB reference can be injected into your Spring bean as if it were a regular dependency.
   - Create a wrapper method that calls the business method(s) of the EJB component.

2. In your Spring bean configuration file, configure the EJB component as a Spring bean using Spring-specific rules. In the same bean configuration file, set the property of the EJB business interface variable as a “ref” property for your consuming Spring bean.

3. In your Spring application, create an ApplicationContext object for your Spring bean and use it by calling its wrapper method(s) with the EJB component working behind the scenes.

In the remainder of this section, we do exactly what we described above to demonstrate how Spring-EJB integration works. The application context might differ from yours, but the procedure should be equally applicable. Let’s start with creating a Spring-integration package in the existing SOBA project next.

10.2.1 Creating a Spring-EJB integration package

For convenience, we use the existing Ant-based SOBA project to demonstrate how Spring-EJB integration works. You can simply follow the steps given below and in the next few sections to complete this exercise.

Assuming that you have a working, Ant-based SOBA project, perform the following tasks:
Add a package of "com.perfmath.spring.soba.integration.ejb" in your existing Ant-based SOBA project.

Copy the jbossall-client.jar file from your $JBOSS_INSTALL/client folder and the CustomerEJB3.0.jar file from your Customer_EJB3 project's dist folder to a folder of your choice. Add these two jar files to the classpath for your Ant-based SOBA project. This step helps make sure that our Spring bean will have access to the EJB business interface to be integrated with and all client utilities needed for performing EJB context lookups.

Note: This EJB 3 sample is Jboss specific and the above jars must be added as described above in order to verify that this sample works in your environment. If you choose a different EJB container vendor, such as GlassFish or WebLogic, you'll have to figure out how to make proper adjustments. Note that although the EJB is a common spec, the performance and scalability that the underlying EJB container exhibits may vary from vendor to vendor, or even from release to release with the same vendor. Before committing to a specific EJB container vendor, do your homework to help make sure that you don’t end up with putting up your enterprise application on a vendor’s EJB container that is so resource demanding that even fastest hardware can hardly meet the moderate performance and scalability requirements of your customers.

In the next section, we create the Spring bean in the integration package as created above.

10.2.2 Creating the Spring Bean that Consumes the EJB Component

We now need to decide how we are going to demonstrate Spring-EJB integration. First, by examining the Customer_EJB3 project we created earlier in this chapter, we know that we have a business interface defined in the CustomerManager.java class as shown in Listing 10.8. This interface defines six business methods, which are implemented by the corresponding bean implementation class named CustomerManagerBean.java as shown in Listing 10.10. Let’s say that we’d like to access some of the methods defined in this interface in a Spring bean. For illustrative purposes, we do not need more than one method; therefore, let’s just pick the createCustomer(…) method. Then, according to the three-step procedure given earlier in this section, we can define a Spring bean interface and its corresponding implementation class as shown in Listings 10.13 (a) and (b). Note the following in the Spring EJB-integration bean implementation class of CustomerServiceImpl.java:

1. The EJB business interface instance variable customerManager
2. The setter method of setCustomerManager(…) for this instance variable
3. The Spring bean wrapper method of createCustomer(…) that calls the createCustomer business method of the CustomerManager stateless session bean

In the next section, we describe how to create the Spring-EJB integration bean configuration file to define the EJB component bean and Spring bean so that both can be loaded up during starting up the Spring application.

Listing 10.13 (a) Spring EJB-integration bean interface CustomerService.java
Listing 10.13 (b) Spring EJB-integration bean implementation class

```
package com.perfmath.spring.soba.integration.ejb;

public interface CustomerService {
    public void createCustomer(String name, String password, String email, boolean locked);
}
```

Listing 10.13 (b) Spring EJB-integration bean implementation class

```
package com.perfmath.spring.soba.integration.ejb;

import com.perfmath.ejb3.ejb.*;

public class CustomerServiceImpl implements CustomerService {
    private CustomerManager customerManager;

    public void setCustomerManager(CustomerManager customerManager) {
        this.customerManager = customerManager;
    }

    public void createCustomer(String name, String password, String email, boolean locked) {
        customerManager.createCustomer(name, password, email, locked);
    }
}
```

10.2.3 Creating Spring-EJB Integration Bean Configuration Files

Next, we need to create a Spring bean configuration file to define a `customerManager` bean out of the EJB component as the dependency of the `customerService` Spring bean, which also has to be defined in the same bean configuration file. In fact, there are three options for defining the EJB business interface bean, resulting in three different bean configuration files as shown in Listings 10.14 (a), (b) and (c), respectively. Each of these options are explained as follows:

- **Spring's JndiObjectFactoryBean**: Spring offers this factory bean to declare a JNDI object reference in its IoC container as shown in Listing 10.14 (a). This approach is simpler than the other two we will introduce next that it does not need to specify the EJB business interface. Note the `jndiEnvironment` property and the `jndiName` property, which are similar to the `getInitialContext()` method and the context lookup statement as we saw in the EJB 3 sample test client shown in Listing 10.12, respectively.

- **Spring's SimpleRemoteStatelessSessionProxyFactoryBean**: Spring offers this factory bean to create a local proxy for a remote stateless session bean as shown in Listing 10.14 (b) (the equivalent factory bean for accessing a local stateless session bean would be
LocalStatelessSessionProxyFactoryBean). With this option, it’s necessary to specify a business interface property, which takes the fully qualified EJB 3 interface name for its value. The other parts remain the same as with the previous JndiObjectFactoryBean option.

- **Spring’s `<jee:remote-slsb>` element**: Spring offers this `jee` schema as an alternative to defining EJB business interface bean using either the JndiObjectFactoryBean or the StatelessSessionProxyFactoryBean. As is shown in Listing 10.14 (c), this option also requires defining an EJB business interface property as is the case with the StatelessSessionProxyFactoryBean option.

As is seen, all these three options only differ in *format*. I would recommend using the JndiObjectFactoryBean, since it does not require defining an EJB business interface and it mimics how an EJB component is accessed by a regular EJB client to some extent.

Next, we describe how to create a Spring-EJB integration test driver to test this Spring-EJB integration sample.

**Listing 10.14 (a) Defining an EJB business bean using Spring’s JndiObjectFactoryBean**

(beans-customer_0.xml)

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd">
    <bean id="customerManager" class="org.springframework.jndi.JndiObjectFactoryBean">
        <property name="jndiEnvironment">
            <props>
                <prop key="java.naming.factory.initial">org.jboss.security.jndi.JndiLoginInitialContextFactory</prop>
                <prop key="java.naming.provider.url">localhost:1099</prop>
                <prop key="java.naming.security.principal">admin</prop>
                <prop key="java.naming.security.credentials">admin</prop>
            </props>
        </property>
        <property name="jndiName" value="CustomerManagerEJB3/remote" />
    </bean>
    <bean id="customerService" class="com.perfmath.spring.soba.integration.ejb.CustomerServiceImpl">
        <property name="customerManager" ref="customerManager" />
    </bean>
</beans>
```

**Listing 10.14 (b) Defining an EJB business bean using Spring’s SimpleRemoteStatelessSessionProxyFactoryBean**

(beans-customer_1.xml)
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd">
  <bean id="customerManager"
        class="org.springframework.ejb.access.
              SimpleRemote StatelessSessionProxyFactoryBean">
      <property name="jndiEnvironment">
        <props>
          <prop key="java.naming.factory.initial">org.jboss.security.jndi.
                JndiLoginInitialContextFactory</prop>
          <prop key="java.naming.provider.url">localhost:1099</prop>
          <prop key="java.naming.security.principal">admin</prop>
          <prop key="java.naming.security.credentials">admin</prop>
        </props>
      </property>
      <property name="jndiName" value="CustomerManagerEJB3/remote"/>
      <property name="businessInterface" value="com.perfmath.ejb3.ejb.CustomerManager"/>
    </bean>
  <bean id="customerService"
        class="com.perfmath.spring.soba.integration.ejb.CustomerServiceImpl">
    <property name="customerManager" ref="customerManager"/>
  </bean>
</beans>

Listing 10.14 (c) Defining an EJB business bean using Spring’s <jee:remote-slsb> element
(beans-customer_2.xml)

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:jee="http://www.springframework.org/schema/jee"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd
      http://www.springframework.org/schema/jee
      http://www.springframework.org/schema/jee/spring-jee.xsd">
  <jee:remote-slsb id="customerManager"
                   jndi-names="CustomerManagerEJB3/remote"
                   business-interface="com.perfmath.ejb3.ejb.CustomerManager">
    <jee:environment>
      java.naming.factory.initial=org.jboss.security.jndi.
      JndiLoginInitialContextFactory
    </jee:environment>
  </jee:remote-slsb>
10.2.4 Creating a Spring-EJB Integration Test Driver

In this section, we demonstrate how to create a Spring-EJB integration test driver to test the `CustomerService` bean as shown in Listing 10.13. Such a test driver is shown in Listing 10.15. This program works as follows:

- First, it creates an `ApplicationContext` using one of the bean configuration files as discussed in the previous section (since we are using the `FileSystemXmlApplicationContext` instead of the `ClassPathXmlApplicationContext`, we must place the bean configuration file in the project’s top directory).
- It then uses the usual `context.getBean(…)` method to retrieve the `customerService` bean instance.
- Finally, it calls the `createCustomer(…)` method to create a customer by invoking the same `createCustomer(…)` method of the underlying EJB 3 component.

As is seen, it’s really not complicated to access an EJB component within a Spring bean, thanks to the simplicity of the Spring framework in general. Next, we describe how to run this test driver, given all we have presented so far.

Listing 10.15 A Spring-EJB integration test driver (CustomerServiceMain.java)

```java
package com.perfmath.spring.soba.integration.ejb;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import com.perfmath.spring.soba.util.RandomID;

class CustomerServiceMain {
    public static void main (String[] args) {
        ApplicationContext context = new ClasspathXmlApplicationContext("beans-customer_0.xml");

        CustomerService customerService =
            context.getBean("customerService", CustomerService.class);
```

String customerName = "customer" + (new RandomID(9)).getId();
System.out.println("creating " + customerName);
customerService.createCustomer(customerName, "password", "customer0@abc.com", false);
}

10.2.5 Running the Spring-EJB Test Driver

To run the Spring-EJB integration test driver as shown in Listing 10.15 in the previous section, first, start up your Jboss and make sure that the EJB 3 sample itself works standalone per instructions given in §10.1. Then, simply right-click on the CustomerServiceMain.java class in the Ant-based SOBA project and select Run As → 2 Java Application (if you want, you can test other bean configuration files by substituting the beans-customer_0.xml file in the test driver with beans-customer_1.xml or beans-customer_2.xml). If everything works in your environment, you should see a console output similar to Figure 10.16. You can also check the customer created from your MySQL database.

Figure 10.16 Console output from running the Spring-EJB integration sample

10.3 ACCESSING EJB 2.x COMPONENTS

According to the Spring Reference Documentation, accessing EJB 2 session beans and EJB 3 session beans is largely transparent. This is because the Spring-EJB integration framework handles a home interface if found (EJB 2 style), or perform straight component invocations if no home interface is found (EJB 3 style). This transparency applies no matter how an EJB lookup is performed with one of the three options discussed previously.

At this point, we could have proved the above transparency statement by providing an example using one of the EJB 2 samples (BMP-based or CMP-based) presented in my article “A Retrospective View on EJB 2.” However, most of the readers may not be interested in going further along this line, so we leave it as an exercise for those who may want to pursue it on their own.

10.4 SPRING’S EJB IMPLEMENTATION SUPPORT CLASSES

To be complete, we have to mention that Spring provides convenience classes to help you implement EJB3. Their thinking is that we can put our business logic behind EJBs in POJOs while leave EJBs responsible for transaction demarcation and (optionally) remoting.
To implement a Stateless or Stateful session bean, or a Message Driven bean, we need only derive our implementation class from one of the following Spring classes:

- AbstractStatelessSessionBean for SLSBs
- AbstractStatefulSessionBean for SFSBs
- AbstractMessageDrivenBean for MDBs
- AbstractJmsMessageDrivenBean for JMDBs

However, this looks more like a cross-breeding approach between Spring and EJB, and you might be better off by choosing either Spring or EJB for your new project without getting into that kind of awkward situation of programming in both Spring and EJB. The Spring Reference Documentation explains further how it can be done programmatically, and you should consult it for more information if you indeed have such a need.

10.5 SUMMARY

In this chapter, we focused on demonstrating how Spring-EJB integration works. We first created a MySQL database with two tables of CUSTOMER and BANKINGTX, setting the stage for creating JPA entities for domain objects for our EJB 3 sample. We then demonstrated how to create JPA entities with Eclipse/Dali facilities, based on the two tables described above. This part shows a generally applicable approach to creating JPA entities based on a well-defined database schema.

Following creating the JPA entities, we demonstrated how to create an EJB 3 stateless session bean based on a non-EJB-specific business interface named CustomerManager. We created both a local and a remote EJB 3 interface as well as an EJB 3 stateless session bean implementation class. We went a few extra miles to demonstrate how to configure and run this EJB 3 sample on Jboss to help ensure that the sample actually works.

The second part of this chapter focused on presenting the specific steps for carrying out a Spring-EJB integration project. A three-step procedure was given with a full proof of concept using the EJB 3 sample created in the first part of this chapter. Overall, the Spring-EJB integration framework works by introducing a Spring bean based on the business interface of the underlying EJB 3 component, and then adding it as a property to the Spring integration bean that wraps the methods of the EJB 3 component with its own methods. We introduced the three options that Spring can look up an EJB component, which are:

- Using JndiObjectFactoryBean
- Using [Local|SimpleRemote]StatelessSessionProxyFactoryBean
- Using <jee:remote-slsb> or <jee:local-slsb>

Corresponding to the above three options, three specific bean configuration XML files were provided, showing exactly how each option works. We also created a Spring-EJB integration test driver, which was run on Jboss to demonstrate that the entire Spring-EJB integration suite presented in this chapter works.

This wraps up our coverage on the Spring-EJB integration framework.
10.6 Recommended Reading

If you are interested in learning more about EJB 3, there are many similar texts available. You can also search the Internet for online free tutorials, especially from one of your preferred EJB container vendors.

For more information about Spring-EJB integration, refer to the Spring Reference Documentation 3.2: Chapter 22 Enterprise JavaBeans (EJB) integration.

10.7 Exercises

10.1 What are the differences among those three JPA associations fetch options as shown in Figure 10.12? Use specific examples to explain the pros and cons of each option.

10.2 Prove the Spring-EJB integration transparency statement as discussed in §10.3 by replacing the EJB 3 sample with one of the EJB 2 samples discussed in my article “A Retrospective View on EJB 2” available for downloading from this book’s website.