CHAPTER 9

Spring ORM Support

In this chapter, you will learn how to integrate *object/relational mapping (ORM)* frameworks into your Spring applications. Spring supports most of the popular ORM frameworks, including Hibernate, JDO, TopLink, iBATIS, and JPA. The focus of this chapter will be on Hibernate and the Java Persistence API (JPA). However, Spring's support for these ORM frameworks is consistent, so you can easily apply the techniques in this chapter to other ORM frameworks as well.

ORM is a modern technology for persisting objects into a relational database. An ORM framework persists your objects according to the mapping metadata you provide, such as the mappings between classes and tables, properties and columns, and so on. It generates SQL statements for object persistence at runtime, so you needn't write database-specific SQL statements unless you want to take advantage of database-specific features or provide optimized SQL statements of your own. As a result, your application will be database independent, and it can be easily migrated to another database in the future. Compared to the direct use of JDBC, an ORM framework can significantly reduce the data access effort of your applications.

Hibernate is a popular open source and high-performance ORM framework in the Java community. Hibernate supports most JDBC-compliant databases and can use specific dialects to access particular databases. Beyond the basic ORM features, Hibernate supports more advanced features like caching, cascading, and lazy loading. It also defines a querying language called *HQL (Hibernate Query Language)* for you to write simple but powerful object queries.

JPA defines a set of standard annotations and APIs for object persistence in both the Java SE and Java EE platforms. JPA is defined as part of the EJB 3.0 specification in JSR-220. JPA is just a set of standard APIs that require a JPA-compliant engine to provide persistence services. You can compare JPA to the JDBC API and a JPA engine to a JDBC driver. Hibernate can be configured as a JPA-compliant engine through an extension module called *Hibernate EntityManager*. This chapter will mainly demonstrate JPA with Hibernate as the underlying engine.

At the time of writing, the latest version of Hibernate is 3.2. Spring 2.0 supports both Hibernate 2.x and 3.x. The support for Hibernate 2.x is provided by the classes and interfaces in the `org.springframework.orm.hibernate` package, while the support for 3.x is in the `org.springframework.orm.hibernate3` package. You must be careful when importing the classes and interfaces to your application. Spring 2.5 supports only Hibernate 3.1 or higher. That means Hibernate 2.1 and Hibernate 3.0 won't be supported any more.

Upon finishing this chapter, you will be able to take advantage of Hibernate and JPA for data access in your Spring applications. You will also have a thorough understanding of Spring's data access module.

9-1. Problems with Using ORM Frameworks Directly

Suppose you are going to develop a course management system for a training center. The first class you create for this system is `Course`. This class is called an *entity class* or a *persistent class*, as it represents a real-world entity and its instances will be persisted to a database. Remember that for each entity class to be persisted by an ORM framework, a default constructor with no argument is required.

```
package com.apress.springrecipes.course;
...
public class Course {

    private Long id;
    private String title;
    private Date beginDate;
    private Date endDate;
    private int fee;

    // Constructors, Getters and Setters
    ...
}
```

For each entity class, you must define an identifier property to uniquely identify an entity. It's a best practice to define an auto-generated identifier, as this has no business meaning and thus won't be changed under any circumstances. Moreover, this identifier will be used by the ORM framework to determine an entity's state. If the identifier value is `null`, this entity will be treated as a new and unsaved entity. When this entity is persisted, an insert SQL statement will be issued; otherwise an update statement will. To allow the identifier to be `null`, you should choose a primitive wrapper type like `java.lang.Integer` and `java.lang.Long` for the identifier.

In your course management system, you need a DAO interface to encapsulate the data access logic. Let's define the following operations in the `CourseDao` interface:

```
package com.apress.springrecipes.course;
...
public interface CourseDao {

    public void store(Course course);
    public void delete(Long courseId);
    public Course findById(Long courseId);
    public List<Course> findAll();
}
```

Usually, when using ORM for persisting objects, the insert and update operations are combined into a single operation (e.g., store). This is to let the ORM framework (not you) decide whether an object should be inserted or updated.

In order for an ORM framework to persist your objects to a database, it must know the mapping metadata for the entity classes. You have to provide mapping metadata to it in its supported format. The native format for Hibernate is XML. However, as each ORM framework may have its own format for defining mapping metadata, JPA defines a set of persistent

annotations for you to define mapping metadata in a standard format that has greater possibilities to be reused in other ORM frameworks.

Hibernate itself also supports the use of JPA annotations to define mapping metadata, so there are essentially three different strategies for mapping and persisting your objects with Hibernate and JPA:

- Using the Hibernate API to persist objects with Hibernate XML mappings
- Using the Hibernate API to persist objects with JPA annotations
- Using JPA to persist objects with JPA annotations

The core programming elements of Hibernate, JPA, and other ORM frameworks resemble those of JDBC. They are summarized in Table 9-1.

Table 9-1. *Core Programming Elements for Different Data Access Strategies*

Concept	JDBC	Hibernate	JPA
Resource	`Connection`	`Session`	`EntityManager`
Resource factory	`DataSource`	`SessionFactory`	`EntityManagerFactory`
Exception	`SQLException`	`HibernateException`	`PersistenceException`

In Hibernate, the core interface for object persistence is `Session`, whose instances can be obtained from a `SessionFactory` instance. In JPA, the corresponding interface is `EntityManager`, whose instances can be obtained from an `EntityManagerFactory` instance. The exceptions thrown by Hibernate are of type `HibernateException`, while those thrown by JPA may be of type `PersistenceException` or other Java SE exceptions like `IllegalArgumentException` and `IllegalStateException`. Note that all these exceptions are subclasses of `RuntimeException`, which you are not forced to catch and handle.

Persisting Objects Using the Hibernate API with Hibernate XML Mappings

To map entity classes with Hibernate XML mappings, you can provide a single mapping file for each class or a large file for several classes. Practically, you should define one for each class by joining the class name with `.hbm.xml` as the file extension for ease of maintenance. The middle extension `hbm` stands for Hibernate metadata.

The mapping file for the `Course` class should be named `Course.hbm.xml` and put in the same package as the entity class.

```
<!DOCTYPE hibernate-mapping
    PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping package="com.apress.springrecipes.course">
    <class name="Course" table="COURSE">
        <id name="id" type="long" column="ID">
            <generator class="identity" />
        </id>
        <property name="title" type="string">
```

```
            <column name="TITLE" length="100" not-null="true" />
        </property>
        <property name="beginDate" type="date" column="BEGIN_DATE" />
        <property name="endDate" type="date" column="END_DATE" />
        <property name="fee" type="int" column="FEE" />
    </class>
</hibernate-mapping>
```

In the mapping file, you can specify a table name for this entity class and a table column for each simple property. You can also specify the column details such as column length, not-null constraints, and unique constraints. In addition, each entity must have an identifier defined, which can be generated automatically or assigned manually. In this example, the identifier will be generated using a table identity column.

Each application that uses Hibernate requires a global configuration file to configure properties such as the database settings (either JDBC connection properties or a data source's JNDI name), the database dialect, the mapping metadata's locations, and so on. When using XML mapping files to define mapping metadata, you have to specify the locations of the XML files. By default, Hibernate will read the `hibernate.cfg.xml` file from the root of the classpath. The middle extension `cfg` stands for configuration.

```
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
    <session-factory>
        <property name="connection.driver_class">
            org.apache.derby.jdbc.ClientDriver
        </property>
        <property name="connection.url">
            jdbc:derby://localhost:1527/course;create=true
        </property>
        <property name="connection.username">app</property>
        <property name="connection.password">app</property>
        <property name="dialect">org.hibernate.dialect.DerbyDialect</property>
        <property name="show_sql">true</property>
        <property name="hbm2ddl.auto">update</property>

        <mapping resource="com/apress/springrecipes/course/Course.hbm.xml" />
    </session-factory>
</hibernate-configuration>
```

Before you can persist your objects, you have to create tables in a database schema to store the object data. When using an ORM framework like Hibernate, you usually needn't design the tables by yourself. If you set the `hbm2ddl.auto` property to `update`, Hibernate can help you to update the database schema and create the tables when necessary.

Now let's implement the DAO interface in the `hibernate` subpackage using the plain Hibernate API. Before you call the Hibernate API for object persistence, you have to initialize a Hibernate session factory (e.g., in the constructor).

■ **Note** To use Hibernate for object persistence, you have to include `hibernate3.jar` (located in the `lib/hibernate` directory of the Spring installation), `antlr-2.7.6.jar` (located in `lib/antlr`), `asm-2.2.3.jar` (located in `lib/asm`), `cglib-nodep-2.1_3.jar` (located in `lib/cglib`), `dom4j-1.6.1.jar` (located in `lib/dom4j`), `ehcache-1.2.4.jar` (located in `lib/ehcache`), `jta.jar` (located in `lib/j2ee`), `commons-collections.jar` and `commons-logging.jar` (located in `lib/jakarta-commons`), and `log4j-1.2.14.jar` (located in `lib/log4j`) in your classpath. To use Apache Derby as the database engine, you also have to include `derbyclient.jar` (located in the `lib` directory of the Derby installation).

```
package com.apress.springrecipes.course.hibernate;
...
import org.hibernate.Query;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.Configuration;

public class HibernateCourseDao implements CourseDao {

    private SessionFactory sessionFactory;

    public HibernateCourseDao() {
        Configuration configuration = new Configuration().configure();
        sessionFactory = configuration.buildSessionFactory();
    }

    public void store(Course course) {
        Session session = sessionFactory.openSession();
        Transaction tx = session.getTransaction();
        try {
            tx.begin();
            session.saveOrUpdate(course);
            tx.commit();
        } catch (RuntimeException e) {
            tx.rollback();
            throw e;
        } finally {
            session.close();
        }
    }
```

```
    public void delete(Long courseId) {
        Session session = sessionFactory.openSession();
        Transaction tx = session.getTransaction();
        try {
            tx.begin();
            Course course = (Course) session.get(Course.class, courseId);
            session.delete(course);
            tx.commit();
        } catch (RuntimeException e) {
            tx.rollback();
            throw e;
        } finally {
            session.close();
        }
    }

    public Course findById(Long courseId) {
        Session session = sessionFactory.openSession();
        try {
            return (Course) session.get(Course.class, courseId);
        } finally {
            session.close();
        }
    }

    public List<Course> findAll() {
        Session session = sessionFactory.openSession();
        try {
            Query query = session.createQuery("from Course");
            return query.list();
        } finally {
            session.close();
        }
    }
}
```

The first step in using Hibernate is to create a `Configuration` object and ask it to load the Hibernate configuration file. By default, it loads `hibernate.cfg.xml` from the classpath root when you call the `configure()` method. Then you build a Hibernate session factory from this `Configuration` object. The purpose of a session factory is to produce sessions for you to persist your objects.

In the preceding DAO methods, you first open a session from the session factory. For any operation that involves database update, such as `saveOrUpdate()` and `delete()`, you must start a Hibernate transaction on that session. If the operation completes successfully, you commit the transaction. Otherwise, you roll it back if any `RuntimeException` happens. For read-only operations such as `get()` and HQL queries, there's no need to start a transaction. Finally, you must remember to close a session to release the resources held by this session.

You can create the following Main class to test run all the DAO methods. It also demonstrates an entity's typical life cycle.

```
package com.apress.springrecipes.course;
...
public class Main {

    public static void main(String[] args) {
        CourseDao courseDao = new HibernateCourseDao();

        Course course = new Course();
        course.setTitle("Core Spring");
        course.setBeginDate(new GregorianCalendar(2007, 8, 1).getTime());
        course.setEndDate(new GregorianCalendar(2007, 9, 1).getTime());
        course.setFee(1000);
        courseDao.store(course);

        List<Course> courses = courseDao.findAll();
        Long courseId = courses.get(0).getId();

        course = courseDao.findById(courseId);
        System.out.println("Course Title: " + course.getTitle());
        System.out.println("Begin Date: " + course.getBeginDate());
        System.out.println("End Date: " + course.getEndDate());
        System.out.println("Fee: " + course.getFee());

        courseDao.delete(courseId);
    }
}
```

Persisting Objects Using the Hibernate API with JPA Annotations

JPA annotations are standardized in the JSR-220 specification, so they're supported by all JPA-compliant ORM frameworks, including Hibernate. Moreover, the use of annotations will be more convenient for you to edit mapping metadata in the same source file.

The following Course class illustrates the use of JPA annotations to define mapping metadata.

■ **Note** To use JPA annotations in Hibernate, you have to include persistence.jar (located in the lib/j2ee directory of the Spring installation), hibernate-annotations.jar, and hibernate-commons-annotations.jar (located in lib/hibernate) in your classpath.

```
package com.apress.springrecipes.course;
...
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name = "COURSE")
public class Course {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "ID")
    private Long id;

    @Column(name = "TITLE", length = 100, nullable = false)
    private String title;

    @Column(name = "BEGIN_DATE")
    private Date beginDate;

    @Column(name = "END_DATE")
    private Date endDate;

    @Column(name = "FEE")
    private int fee;

    // Constructors, Getters and Setters
    ...
}
```

Each entity class must be annotated with the `@Entity` annotation. You can assign a table name for an entity class in this annotation. For each property, you can specify a column name and column details using the `@Column` annotation. Each entity class must have an identifier defined by the `@Id` annotation. You can choose a strategy for identifier generation using the `@GeneratedValue` annotation. Here, the identifier will be generated by a table identity column.

Hibernate supports both native XML mapping files and JPA annotations as ways of defining mapping metadata. For JPA annotations, you have to specify the fully qualified names of the entity classes in `hibernate.cfg.xml` for Hibernate to read the annotations.

```
<hibernate-configuration>
    <session-factory>
        ...
        <!-- For Hibernate XML mappings -->
        <!--
```

```
        <mapping resource="com/apress/springrecipes/course/Course.hbm.xml" />
        -->

        <!-- For JPA annotations -->
        <mapping class="com.apress.springrecipes.course.Course" />
    </session-factory>
</hibernate-configuration>
```

In the Hibernate DAO implementation, the `Configuration` class you used is for reading XML mappings. If you use JPA annotations to define mapping metadata for Hibernate, you have to use its subclass, `AnnotationConfiguration`, instead.

```
package com.apress.springrecipes.course.hibernate;
...
import org.hibernate.SessionFactory;
import org.hibernate.cfg.AnnotationConfiguration;

public class HibernateCourseDao implements CourseDao {

    private SessionFactory sessionFactory;

    public HibernateCourseDao() {
        // For Hibernate XML mapping
        // Configuration configuration = new Configuration().configure();

        // For JPA annotation
        Configuration configuration = new AnnotationConfiguration().configure();

        sessionFactory = configuration.buildSessionFactory();
    }
    ...
}
```

Persisting Objects Using JPA with Hibernate As the Engine

In addition to persistent annotations, JPA defines a set of programming interfaces for object persistence. However, JPA is not a persistence implementation itself. You have to pick up a JPA-compliant engine to provide persistence services. Hibernate can be JPA-compliant through the Hibernate EntityManager extension module. With this extension, Hibernate can work as an underlying JPA engine to persist objects.

In a Java EE environment, you can configure the JPA engine in a Java EE container. But in a Java SE application, you have to set up the engine locally. The configuration of JPA is through the central XML file `persistence.xml`, located in the `META-INF` directory of the classpath root. In this file, you can set any vendor-specific properties for the underlying engine configuration.

Now let's create the JPA configuration file `persistence.xml` in the `META-INF` directory of the classpath root. Each JPA configuration file contains one or more `<persistence-unit>` elements. A *persistence unit* defines a set of persistent classes and how they should be persisted. Each

persistence unit requires a name for identification. Here, you assign the name `course` to this persistence unit.

```
<persistence 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_1_0.xsd"
    version="1.0">

    <persistence-unit name="course">
        <properties>
            <property name="hibernate.ejb.cfgfile" value="/hibernate.cfg.xml" />
        </properties>
    </persistence-unit>
</persistence>
```

In this JPA configuration file, you configure Hibernate as your underlying JPA engine by referring to the Hibernate configuration file located in the classpath root. However, as Hibernate EntityManager will automatically detect XML mapping files and JPA annotations as mapping metadata, you have no need to specify them explicitly. Otherwise, you will encounter an `org.hibernate.DuplicateMappingException`.

```
<hibernate-configuration>
    <session-factory>
        ...
        <!-- Don't need to specify mapping files and annotated classes -->
        <!--
        <mapping resource="com/apress/springrecipes/course/Course.hbm.xml" />
        <mapping class="com.apress.springrecipes.course.Course" />
        -->
    </session-factory>
</hibernate-configuration>
```

As an alternative to referring to the Hibernate configuration file, you can also centralize all the Hibernate configurations in `persistence.xml`.

```
<persistence ...>
    <persistence-unit name="course">
        <properties>
            <property name="hibernate.connection.driver_class"
                value="org.apache.derby.jdbc.ClientDriver" />
            <property name="hibernate.connection.url"
                value="jdbc:derby://localhost:1527/course;create=true" />
            <property name="hibernate.connection.username" value="app" />
            <property name="hibernate.connection.password" value="app" />
            <property name="hibernate.dialect"
                value="org.hibernate.dialect.DerbyDialect" />
            <property name="hibernate.show_sql" value="true" />
            <property name="hibernate.hbm2ddl.auto" value="update" />
```

```
        </properties>
    </persistence-unit>
</persistence>'
```

In a Java EE environment, a Java EE container is able to manage the entity manager for you and inject it into your EJB components directly. But when you use JPA outside of a Java EE container (e.g., in a Java SE application), you have to create and maintain the entity manager by yourself.

Now let's implement the `CourseDao` interface in the `jpa` subpackage using JPA in a Java SE application. Before you call JPA for object persistence, you have to initialize an entity manager factory. The purpose of an entity manager factory is to produce entity managers for you to persist your objects.

■ **Note** To use Hibernate as the underlying JPA engine, you have to include `hibernate-entitymanager.jar` and `jboss-archive-browsing.jar` (located in the `lib/hibernate` directory of the Spring installation) in your classpath. As Hibernate EntityManager depends on Javassist (`http://www.jboss.org/javassist/`), you also have to include `javassist.jar` in your classpath. If you have Hibernate installed, you should find it in the `lib` directory of the Hibernate installation. Otherwise, you have to download it from its web site.

```
package com.apress.springrecipes.course.jpa;
...
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.EntityTransaction;
import javax.persistence.Persistence;
import javax.persistence.Query;

public class JpaCourseDao implements CourseDao {

    private EntityManagerFactory entityManagerFactory;

    public JpaCourseDao() {
        entityManagerFactory = Persistence.createEntityManagerFactory("course");
    }

    public void store(Course course) {
        EntityManager manager = entityManagerFactory.createEntityManager();
        EntityTransaction tx = manager.getTransaction();
        try {
            tx.begin();
            manager.merge(course);
            tx.commit();
        } catch (RuntimeException e) {
```

```
            tx.rollback();
            throw e;
        } finally {
            manager.close();
        }
    }

    public void delete(Long courseId) {
        EntityManager manager = entityManagerFactory.createEntityManager();
        EntityTransaction tx = manager.getTransaction();
        try {
            tx.begin();
            Course course = manager.find(Course.class, courseId);
            manager.remove(course);
            tx.commit();
        } catch (RuntimeException e) {
            tx.rollback();
            throw e;
        } finally {
            manager.close();
        }
    }

    public Course findById(Long courseId) {
        EntityManager manager = entityManagerFactory.createEntityManager();
        try {
            return manager.find(Course.class, courseId);
        } finally {
            manager.close();
        }
    }

    public List<Course> findAll() {
        EntityManager manager = entityManagerFactory.createEntityManager();
        try {
            Query query = manager.createQuery(
                "select course from Course course");
            return query.getResultList();
        } finally {
            manager.close();
        }
    }
}
```

The entity manager factory is built by the static method `createEntityManagerFactory()` of the `javax.persistence.Persistence` class. You have to pass in a persistence unit name defined in `persistence.xml` for an entity manager factory.

In the preceding DAO methods, you first create an entity manager from the entity manager factory. For any operation that involves database update, such as `merge()` and `remove()`, you must start a JPA transaction on the entity manager. For read-only operations such as `find()` and JPA queries, there's no need to start a transaction. Finally, you must close an entity manager to release the resources.

You can test this DAO with the similar `Main` class, but this time you instantiate the JPA DAO implementation instead.

```
package com.apress.springrecipes.course;
...
public class Main {

    public static void main(String[] args) {
        CourseDao courseDao = new JpaCourseDao();
        ...
    }
}
```

In the preceding DAO implementations for both Hibernate and JPA, there are only one or two lines that are different for each DAO method. The rest of the lines are boilerplate routine tasks that you have to repeat. Moreover, each ORM framework has its own API for local transaction management.

9-2. Configuring ORM Resource Factories in Spring

Problem

When using an ORM framework on its own, you have to configure its resource factory with its API. For Hibernate and JPA, you have to build a session factory and an entity manager factory from the native Hibernate API and JPA. In this way, you can only manage the session factory and entity manager factory by yourself. Besides, your application cannot utilize the data access facilities provided by Spring.

Solution

Spring provides several factory beans for you to create a Hibernate session factory or a JPA entity manager factory as a singleton bean in the IoC container. These factories can be shared between multiple beans via dependency injection. Moreover, this allows the session factory and the entity manager factory to integrate with other Spring data access facilities, such as data sources and transaction managers.

How It Works

Configuring a Hibernate Session Factory in Spring

First of all, let's modify `HibernateCourseDao` to accept a session factory via dependency injection, instead of creating it directly with the native Hibernate API in the constructor.

```
package com.apress.springrecipes.course.hibernate;
...
import org.hibernate.SessionFactory;

public class HibernateCourseDao implements CourseDao {

    private SessionFactory sessionFactory;

    public void setSessionFactory(SessionFactory sessionFactory) {
        this.sessionFactory = sessionFactory;
    }
    ...
}
```

Now let's see how to declare a session factory that uses XML mapping files in Spring. For this purpose, you have to enable the XML mapping file definition in `hibernate.cfg.xml` again.

```
<hibernate-configuration>
    <session-factory>
        ...
        <!-- For Hibernate XML mappings -->
        <mapping resource="com/apress/springrecipes/course/Course.hbm.xml" />
    </session-factory>
</hibernate-configuration>
```

Then you create a bean configuration file for using Hibernate as the ORM framework (e.g., `beans-hibernate.xml` in the classpath root). You can declare a session factory that uses XML mapping files with the factory bean `LocalSessionFactoryBean`. You can also declare a `HibernateCourseDao` instance under Spring's management.

```
<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-2.5.xsd">

    <bean id="sessionFactory"
        class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
        <property name="configLocation" value="classpath:hibernate.cfg.xml" />
    </bean>

    <bean id="courseDao"
        class="com.apress.springrecipes.course.hibernate.HibernateCourseDao">
        <property name="sessionFactory" ref="sessionFactory" />
    </bean>
</beans>
```

Note that you can specify the `configLocation` property for this factory bean to load the Hibernate configuration file. This property is of type `Resource`, but you can assign a string value to it. The built-in property editor `ResourceEditor` will convert it into a `Resource` object. The preceding factory bean loads the configuration file from the root of the classpath.

Now you can modify the `Main` class to retrieve the `HibernateCourseDao` instance from the Spring IoC container.

```
package com.apress.springrecipes.course;
...
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Main {

    public static void main(String[] args) {
        ApplicationContext context =
            new ClassPathXmlApplicationContext("beans-hibernate.xml");

        CourseDao courseDao = (CourseDao) context.getBean("courseDao");
        ...
    }
}
```

The preceding factory bean creates a session factory by loading the Hibernate configuration file, which includes the database settings (either JDBC connection properties or a data source's JNDI name). Now, suppose you have a data source defined in the Spring IoC container. If you want to use this data source for your session factory, you can inject it into the `dataSource` property of `LocalSessionFactoryBean`. The data source specified in this property will override the database settings in the Hibernate configuration file.

```
<beans ...>
    ...
    <bean id="dataSource"
        class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <property name="driverClassName"
            value="org.apache.derby.jdbc.ClientDriver" />
        <property name="url"
            value="jdbc:derby://localhost:1527/course;create=true" />
        <property name="username" value="app" />
        <property name="password" value="app" />
    </bean>

    <bean id="sessionFactory"
        class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
        <property name="dataSource" ref="dataSource" />
        <property name="configLocation" value="classpath:hibernate.cfg.xml" />
    </bean>
</beans>
```

Or, you may even ignore the Hibernate configuration file by merging all the configurations into `LocalSessionFactoryBean`. For example, you can specify the locations of the XML mapping files in the `mappingResources` property and other Hibernate properties such as the database dialect in the `hibernateProperties` property.

```
<bean id="sessionFactory"
    class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
    <property name="dataSource" ref="dataSource" />
    <property name="mappingResources">
        <list>
            <value>com/apress/springrecipes/course/Course.hbm.xml</value>
        </list>
    </property>
    <property name="hibernateProperties">
        <props>
            <prop key="hibernate.dialect">org.hibernate.dialect.DerbyDialect</prop>
            <prop key="hibernate.show_sql">true</prop>
            <prop key="hibernate.hbm2ddl.auto">update</prop>
        </props>
    </property>
</bean>
```

The `mappingResources` property's type is `String[]`, so you can specify a set of mapping files in the classpath. `LocalSessionFactoryBean` also allows you take advantage of Spring's resource-loading support to load mapping files from various types of locations. You can specify the resource paths of the mapping files in the `mappingLocations` property, whose type is `Resource[]`.

```
<bean id="sessionFactory"
    class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
    ...
    <property name="mappingLocations">
        <list>
            <value>classpath:com/apress/springrecipes/course/Course.hbm.xml</value>
        </list>
    </property>
    ...
</bean>
```

With Spring's resource-loading support, you can also use wildcards in a resource path to match multiple mapping files so that you don't need to configure their locations every time you add a new entity class. Spring's preregistered `ResourceArrayPropertyEditor` will convert this path into a `Resource` array.

```
<bean id="sessionFactory"
    class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
    ...
    <property name="mappingLocations"
        value="classpath:com/apress/springrecipes/course/*.hbm.xml" />
    ...
</bean>
```

If your mapping metadata is provided through JPA annotations, you have to make use of `AnnotationSessionFactoryBean` instead. You have to specify the persistent classes in the `annotatedClasses` property of `AnnotationSessionFactoryBean`.

```
<bean id="sessionFactory" class="org.springframework.orm.hibernate3.➥
    annotation.AnnotationSessionFactoryBean">
    <property name="dataSource" ref="dataSource" />
    <property name="annotatedClasses">
        <list>
            <value>com.apress.springrecipes.course.Course</value>
        </list>
    </property>
    <property name="hibernateProperties">
        <props>
            <prop key="hibernate.dialect">org.hibernate.dialect.DerbyDialect</prop>
            <prop key="hibernate.show_sql">true</prop>
            <prop key="hibernate.hbm2ddl.auto">update</prop>
        </props>
    </property>
</bean>
```

Now you can delete the Hibernate configuration file (i.e., `hibernate.cfg.xml`) because its configurations have been ported to Spring.

Configuring a JPA Entity Manager Factory in Spring

First of all, let's modify `JpaCourseDao` to accept an entity manager factory via dependency injection, instead of creating it directly in the constructor.

```
package com.apress.springrecipes.course.jpa;
...
import javax.persistence.EntityManagerFactory;

public class JpaCourseDao implements CourseDao {

    private EntityManagerFactory entityManagerFactory;

    public void setEntityManagerFactory(
            EntityManagerFactory entityManagerFactory) {
        this.entityManagerFactory = entityManagerFactory;
    }
    ...
}
```

The JPA specification defines how you should obtain an entity manager factory in Java SE and Java EE environments. In a Java SE environment, an entity manager factory is created manually by calling the `createEntityManagerFactory()` static method of the `Persistence` class.

Now let's create a bean configuration file for using JPA (e.g., `beans-jpa.xml` in the classpath root). Spring provides a factory bean, `LocalEntityManagerFactoryBean`, for you to create an entity manager factory in the IoC container. You must specify the persistence unit name defined in the JPA configuration file. You can also declare a `JpaCourseDao` instance under Spring's management.

```
<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-2.5.xsd">

    <bean id="entityManagerFactory"
        class="org.springframework.orm.jpa.LocalEntityManagerFactoryBean">
        <property name="persistenceUnitName" value="course" />
    </bean>

   <bean id="courseDao"
        class="com.apress.springrecipes.course.jpa.JpaCourseDao">
        <property name="entityManagerFactory" ref="entityManagerFactory" />
    </bean>
</beans>
```

Now you can test this `JpaCourseDao` instance with the `Main` class by retrieving it from the Spring IoC container.

```
package com.apress.springrecipes.course;
...
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Main {

    public static void main(String[] args) {
        ApplicationContext context =
            new ClassPathXmlApplicationContext("beans-jpa.xml");

        CourseDao courseDao = (CourseDao) context.getBean("courseDao");
        ...
    }
}
```

In a Java EE environment, you can look up an entity manager factory from a Java EE container with JNDI. In Spring 2.x, you can perform a JNDI lookup by using the `<jee:jndi-lookup>` element.

```
<jee:jndi-lookup id="entityManagerFactory" jndi-name="jpa/coursePU" />
```

`LocalEntityManagerFactoryBean` creates an entity manager factory by loading the JPA configuration file (i.e., `persistence.xml`). Spring supports a more flexible way to create an entity manager factory by another factory bean, `LocalContainerEntityManagerFactoryBean`. It allows you to override some of the configurations in the JPA configuration file, such as the data source and database dialect. So, you can take advantage of Spring's data access facilities to configure the entity manager factory.

```
<beans ...>
    ...
    <bean id="dataSource"
        class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <property name="driverClassName"
            value="org.apache.derby.jdbc.ClientDriver" />
        <property name="url"
            value="jdbc:derby://localhost:1527/course;create=true" />
        <property name="username" value="app" />
        <property name="password" value="app" />
    </bean>

    <bean id="entityManagerFactory" class="org.springframework.orm.jpa.➥
        LocalContainerEntityManagerFactoryBean">
        <property name="persistenceUnitName" value="course" />
        <property name="dataSource" ref="dataSource" />
        <property name="jpaVendorAdapter">
            <bean class="org.springframework.orm.jpa.vendor.➥
                HibernateJpaVendorAdapter">
                <property name="databasePlatform"
                    value="org.hibernate.dialect.DerbyDialect" />
                <property name="showSql" value="true" />
                <property name="generateDdl" value="true" />
            </bean>
        </property>
    </bean>
</beans>
```

In the preceding bean configurations, you inject a data source into this entity manager factory. It will override the database settings in the JPA configuration file. You can set a JPA vendor adapter to `LocalContainerEntityManagerFactoryBean` to specify JPA engine–specific properties. With Hibernate as the underlying JPA engine, you should choose `HibernateJpaVendorAdapter`. Other properties that are not supported by this adapter can be specified in the `jpaProperties` property.

Now your JPA configuration file (i.e., `persistence.xml`) can be simplified as follows because its configurations have been ported to Spring:

```
<persistence ...>
    <persistence-unit name="course" />
</persistence>
```

9-3. Persisting Objects with Spring's ORM Templates

Problem

When using an ORM framework on its own, you have to repeat certain routine tasks for each DAO operation. For example, in a DAO operation implemented with Hibernate or JPA, you have to open and close a session or an entity manager, and begin, commit, and roll back a transaction with the native API.

Solution

Spring's approach to simplifying an ORM framework's usage is the same as JDBC's—by defining template classes and DAO support classes. Also, Spring defines an abstract layer on top of different transaction management APIs. For different ORM frameworks, you only have to pick up a corresponding transaction manager implementation. Then you can manage transactions for them in a similar way.

In Spring's data access module, the support for different data access strategies is consistent. Table 9-2 compares the support classes for JDBC, Hibernate, and JPA.

Table 9-2. *Spring's Support Classes for Different Data Access Strategies*

Support Class	JDBC	Hibernate	JPA
Template class	`JdbcTemplate`	`HibernateTemplate`	`JpaTemplate`
DAO support class	`JdbcDaoSupport`	`HibernateDaoSupport`	`JpaDaoSupport`
Transaction manager	`DataSourceTransactionManager`	`HibernateTransactionManager`	`JpaTransactionManager`

Spring defines the `HibernateTemplate` and `JpaTemplate` classes to provide template methods for different types of Hibernate and JPA operations to minimize the effort involved in using them. The template methods in `HibernateTemplate` and `JpaTemplate` ensure that Hibernate sessions and JPA entity managers will be opened and closed properly. They will also have native Hibernate and JPA transactions participate in Spring-managed transactions. As a result, you will be able to manage transactions declaratively for your Hibernate and JPA DAOs without any boilerplate transaction code.

How It Works

Using a Hibernate Template and a JPA Template

First, the `HibernateCourseDao` class can be simplified as follows with the help of Spring's `HibernateTemplate`:

```
package com.apress.springrecipes.course.hibernate;
...
import org.springframework.orm.hibernate3.HibernateTemplate;
import org.springframework.transaction.annotation.Transactional;
```

```
public class HibernateCourseDao implements CourseDao {

    private HibernateTemplate hibernateTemplate;

    public void setHibernateTemplate(HibernateTemplate hibernateTemplate) {
        this.hibernateTemplate = hibernateTemplate;
    }

    @Transactional
    public void store(Course course) {
        hibernateTemplate.saveOrUpdate(course);
    }

    @Transactional
    public void delete(Long courseId) {
        Course course = (Course) hibernateTemplate.get(Course.class, courseId);
        hibernateTemplate.delete(course);
    }

    @Transactional(readOnly = true)
    public Course findById(Long courseId) {
        return (Course) hibernateTemplate.get(Course.class, courseId);
    }

    @Transactional(readOnly = true)
    public List<Course> findAll() {
        return hibernateTemplate.find("from Course");
    }
}
```

In this DAO implementation, you declare all the DAO methods to be transactional with the `@Transactional` annotation. Among these methods, `findById()` and `findAll()` are read-only. The template methods in `HibernateTemplate` are responsible for managing the sessions and transactions. If there are multiple Hibernate operations in a transactional DAO method, the template methods will ensure that they will run within the same session and transaction. As a result, you have no need to deal with the Hibernate API for session and transaction management.

The `HibernateTemplate` class is thread-safe, so you can declare a single instance of it in the bean configuration file for Hibernate (i.e., `beans-hibernate.xml`) and inject this instance into all Hibernate DAOs. A `HibernateTemplate` instance requires the `sessionFactory` property to be set. You can inject this property by either setter method or constructor argument.

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
        http://www.springframework.org/schema/tx
```

```
        http://www.springframework.org/schema/tx/spring-tx-2.5.xsd">
    ...
    <tx:annotation-driven />

    <bean id="transactionManager"
        class="org.springframework.orm.hibernate3.HibernateTransactionManager">
        <property name="sessionFactory" ref="sessionFactory" />
    </bean>

    <bean id="hibernateTemplate"
        class="org.springframework.orm.hibernate3.HibernateTemplate">
        <property name="sessionFactory" ref="sessionFactory" />
    </bean>

    <bean name="courseDao"
        class="com.apress.springrecipes.course.hibernate.HibernateCourseDao">
        <property name="hibernateTemplate" ref="hibernateTemplate" />
    </bean>
</beans>
```

To enable declarative transaction management for the methods annotated with `@Transactional`, you have to enable the `<tx:annotation-driven>` element in your bean configuration file. By default, it will look for a transaction manager with the name `transactionManager`, so you have to declare a `HibernateTransactionManager` instance with that name. `HibernateTransactionManager` requires the session factory property to be set. It will manage transactions for sessions opened through this session factory.

Similarly, you can simplify the `JpaCourseDao` class as follows with the help of Spring's `JpaTemplate`. You also declare all the DAO methods to be transactional.

```
package com.apress.springrecipes.course.jpa;
...
import org.springframework.orm.jpa.JpaTemplate;
import org.springframework.transaction.annotation.Transactional;

public class JpaCourseDao implements CourseDao {

    private JpaTemplate jpaTemplate;

    public void setJpaTemplate(JpaTemplate jpaTemplate) {
        this.jpaTemplate = jpaTemplate;
    }

    @Transactional
    public void store(Course course) {
        jpaTemplate.merge(course);
    }
```

```
    @Transactional
    public void delete(Long courseId) {
        Course course = jpaTemplate.find(Course.class, courseId);
        jpaTemplate.remove(course);
    }

    @Transactional(readOnly = true)
    public Course findById(Long courseId) {
        return jpaTemplate.find(Course.class, courseId);
    }

    @Transactional(readOnly = true)
    public List<Course> findAll() {
        return jpaTemplate.find("from Course");
    }
}
```

In the bean configuration file for JPA (i.e., `beans-jpa.xml`), you can declare a `JpaTemplate` instance and inject it into all JPA DAOs. Also, you have to declare a `JpaTransactionManager` instance for managing JPA transactions.

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
        http://www.springframework.org/schema/tx
        http://www.springframework.org/schema/tx/spring-tx-2.5.xsd">
    ...
    <tx:annotation-driven />

    <bean id="transactionManager"
        class="org.springframework.orm.jpa.JpaTransactionManager">
        <property name="entityManagerFactory" ref="entityManagerFactory" />
    </bean>

    <bean id="jpaTemplate"
        class="org.springframework.orm.jpa.JpaTemplate">
        <property name="entityManagerFactory" ref="entityManagerFactory" />
    </bean>

    <bean name="courseDao"
        class="com.apress.springrecipes.course.jpa.JpaCourseDao">
        <property name="jpaTemplate" ref="jpaTemplate" />
    </bean>
</beans>
```

Another advantage of HibernateTemplate and JpaTemplate is that they will translate native Hibernate and JPA exceptions into exceptions in Spring's DataAccessException hierarchy. This allows consistent exception handling for all the data access strategies in Spring. For instance, if a database constraint is violated when persisting an object, Hibernate will throw an org.hibernate.exception.ConstraintViolationException while JPA will throw a javax.persistence.EntityExistsException. These exceptions will be translated by HibernateTemplate and JpaTemplate into DataIntegrityViolationException, which is a subclass of Spring's DataAccessException.

If you want to get access to the underlying Hibernate session or JPA entity manager in HibernateTemplate or JpaTemplate in order to perform native Hibernate or JPA operations, you can implement the HibernateCallback or JpaCallback interface and pass its instance to the execute() method of the template.

```
hibernateTemplate.execute(new HibernateCallback() {
    public Object doInHibernate(Session session) throws HibernateException,
            SQLException {
        ...
    }
};

jpaTemplate.execute(new JpaCallback() {
    public Object doInJpa(EntityManager em) throws PersistenceException {
        ...
    }
};
```

Extending the Hibernate and JPA DAO Support Classes

Your Hibernate DAO can extend HibernateDaoSupport to have the setSessionFactory() and setHibernateTemplate() methods inherited. Then, in your DAO methods, you can simply call the getHibernateTemplate() method to retrieve the template instance.

```
package com.apress.springrecipes.course.hibernate;
...
import org.springframework.orm.hibernate3.support.HibernateDaoSupport;
import org.springframework.transaction.annotation.Transactional;

public class HibernateCourseDao extends HibernateDaoSupport implements
        CourseDao {

    @Transactional
    public void store(Course course) {
        getHibernateTemplate().saveOrUpdate(course);
    }

    @Transactional
    public void delete(Long courseId) {
        Course course = (Course) getHibernateTemplate().get(Course.class,
```

```
                courseId);
        getHibernateTemplate().delete(course);
    }

    @Transactional(readOnly = true)
    public Course findById(Long courseId) {
        return (Course) getHibernateTemplate().get(Course.class, courseId);
    }

    @Transactional(readOnly = true)
    public List<Course> findAll() {
        return getHibernateTemplate().find("from Course");
    }
}
```

As `HibernateCourseDao` inherits the `setSessionFactory()` and `setHibernateTemplate()` methods, you can inject either of them into your DAO so that you can retrieve the `HibernateTemplate` instance. If you inject a session factory, you will be able to delete the `HibernateTemplate` declaration.

```
<bean name="courseDao"
    class="com.apress.springrecipes.course.hibernate.HibernateCourseDao">
    <property name="sessionFactory" ref="sessionFactory" />
</bean>
```

Similarly, your JPA DAO can extend `JpaDaoSupport` to have `setEntityManagerFactory()` and `setJpaTemplate()` inherited. In your DAO methods, you can simply call the `getJpaTemplate()` method to retrieve the template instance.

```
package com.apress.springrecipes.course.jpa;
...
import org.springframework.orm.jpa.support.JpaDaoSupport;
import org.springframework.transaction.annotation.Transactional;

public class JpaCourseDao extends JpaDaoSupport implements CourseDao {

    @Transactional
    public void store(Course course) {
        getJpaTemplate().merge(course);
    }

    @Transactional
    public void delete(Long courseId) {
        Course course = getJpaTemplate().find(Course.class, courseId);
        getJpaTemplate().remove(course);
    }
```

```
    @Transactional(readOnly = true)
    public Course findById(Long courseId) {
        return getJpaTemplate().find(Course.class, courseId);
    }

    @Transactional(readOnly = true)
    public List<Course> findAll() {
        return getJpaTemplate().find("from Course");
    }
}
```

As `JpaCourseDao` inherits both `setEntityManagerFactory()` and `setJpaTemplate()`, you can inject either of them into your DAO. If you inject an entity manager factory, you will be able to delete the `JpaTemplate` declaration.

```
<bean name="courseDao"
    class="com.apress.springrecipes.course.jpa.JpaCourseDao">
    <property name="entityManagerFactory" ref="entityManagerFactory" />
</bean>
```

9-4. Persisting Objects with Hibernate's Contextual Sessions

Problem

Spring's `HibernateTemplate` can simplify your DAO implementation by managing sessions and transactions for you. However, using `HibernateTemplate` means your DAO has to depend on Spring's API.

Solution

An alternative to Spring's `HibernateTemplate` is to use Hibernate's contextual sessions. In Hibernate 3, a session factory is able to manage contextual sessions for you and allows you to retrieve them by the `getCurrentSession()` method. Within a single transaction, you will get the same session for each `getCurrentSession()` method call. This ensures that there will be only one Hibernate session per transaction, so it works nicely with Spring's transaction management support.

How It Works

To use the contextual session approach, your DAO methods require access to the session factory, which can be injected via a setter method or a constructor argument. Then, in each DAO method, you get the contextual session from the session factory and use it for object persistence.

```
package com.apress.springrecipes.course.hibernate;
...
import org.hibernate.Query;
import org.hibernate.SessionFactory;
import org.springframework.transaction.annotation.Transactional;
```

```
public class HibernateCourseDao implements CourseDao {

    private SessionFactory sessionFactory;

    public void setSessionFactory(SessionFactory sessionFactory) {
        this.sessionFactory = sessionFactory;
    }

    @Transactional
    public void store(Course course) {
        sessionFactory.getCurrentSession().saveOrUpdate(course);
    }

    @Transactional
    public void delete(Long courseId) {
        Course course = (Course) sessionFactory.getCurrentSession().get(
                Course.class, courseId);
        sessionFactory.getCurrentSession().delete(course);
    }

    @Transactional(readOnly = true)
    public Course findById(Long courseId) {
        return (Course) sessionFactory.getCurrentSession().get(
                Course.class, courseId);
    }

    @Transactional(readOnly = true)
    public List<Course> findAll() {
        Query query = sessionFactory.getCurrentSession().createQuery(
                "from Course");
        return query.list();
    }
}
```

Note that all your DAO methods must be made transactional. You can achieve this by annotating each method or the entire class with `@Transactional`. This ensures that the persistence operations within a DAO method will be executed in the same transaction and hence by the same session. Moreover, if a service layer component's method calls multiple DAO methods, and it propagates its own transaction to these methods, then all these DAO methods will run within the same session as well.

In the bean configuration file for Hibernate (i.e., `beans-hibernate.xml`), you have to declare a `HibernateTransactionManager` instance for this application and enable declarative transaction management via `<tx:annotation-driven>`.

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
```

```
        http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
        http://www.springframework.org/schema/tx
        http://www.springframework.org/schema/tx/spring-tx-2.5.xsd">
    ...
    <tx:annotation-driven />

    <bean id="transactionManager"
        class="org.springframework.orm.hibernate3.HibernateTransactionManager">
        <property name="sessionFactory" ref="sessionFactory" />
    </bean>

    <bean name="courseDao"
        class="com.apress.springrecipes.course.hibernate.HibernateCourseDao">
        <property name="sessionFactory" ref="sessionFactory" />
    </bean>
</beans>
```

`HibernateTemplate` will translate the native Hibernate exceptions into exceptions in Spring's `DataAccessException` hierarchy. This allows consistent exception handling for different data access strategies in Spring. However, when calling the native methods on a Hibernate session, the exceptions thrown will be of native type `HibernateException`. If you want the Hibernate exceptions to be translated into Spring's `DataAccessException` for consistent exception handling, you have to apply the `@Repository` annotation to your DAO class that requires exception translation.

```
package com.apress.springrecipes.course.hibernate;
...
import org.springframework.stereotype.Repository;

@Repository
public class HibernateCourseDao implements CourseDao {
    ...
}
```

Then register a `PersistenceExceptionTranslationPostProcessor` instance to translate the native Hibernate exceptions into data access exceptions in Spring's `DataAccessException` hierarchy. This bean post processor will only translate exceptions for beans annotated with `@Repository`.

```
<beans ...>
    ...
    <bean class="org.springframework.dao.annotation.➥
        PersistenceExceptionTranslationPostProcessor" />
</beans>
```

In Spring 2.5, `@Repository` is a stereotype annotation. By annotating this, a component class can be auto-detected through component scanning. You can assign a component name in this annotation and have the session factory auto-wired by the Spring IoC container with `@Autowired`.

```
package com.apress.springrecipes.course.hibernate;
...
import org.hibernate.SessionFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Repository;

@Repository("courseDao")
public class HibernateCourseDao implements CourseDao {

    private SessionFactory sessionFactory;

    @Autowired
    public void setSessionFactory(SessionFactory sessionFactory) {
        this.sessionFactory = sessionFactory;
    }
    ...
}
```

Then, you can simply enable the `<context:component-scan>` element and delete the original `HibernateCourseDao` bean declaration.

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-2.5.xsd
        http://www.springframework.org/schema/tx
        http://www.springframework.org/schema/tx/spring-tx-2.5.xsd">

    <context:component-scan
        base-package="com.apress.springrecipes.course.hibernate" />
    ...
</beans>
```

9-5. Persisting Objects with JPA's Context Injection

Problem

In a Java EE environment, a Java EE container is able to manage entity managers for you and inject them into your EJB components directly. An EJB component can simply perform persistence operations on an injected entity manager without caring much about the entity manager creation and transaction management.

Similarly, Spring provides `JpaTemplate` to simplify your DAO implementation by managing entity managers and transactions for you. However, using Spring's `JpaTemplate` means your DAO is dependent on Spring's API.

Solution

An alternative to Spring's `JpaTemplate` is to use JPA's context injection. Originally, the `@PersistenceContext` annotation is used for entity manager injection in EJB components. Spring can also interpret this annotation by means of a bean post processor. It will inject an entity manager into a property with this annotation. Spring ensures that all your persistence operations within a single transaction will be handled by the same entity manager.

How It Works

To use the context injection approach, you can declare an entity manager field in your DAO and annotate it with the `@PersistenceContext` annotation. Spring will inject an entity manager into this field for you to persist your objects.

```
package com.apress.springrecipes.course.jpa;
...
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.persistence.Query;

import org.springframework.transaction.annotation.Transactional;

public class JpaCourseDao implements CourseDao {

    @PersistenceContext
    private EntityManager entityManager;

    @Transactional
    public void store(Course course) {
        entityManager.merge(course);
    }

    @Transactional
    public void delete(Long courseId) {
        Course course = entityManager.find(Course.class, courseId);
        entityManager.remove(course);
    }

    @Transactional(readOnly = true)
    public Course findById(Long courseId) {
        return entityManager.find(Course.class, courseId);
    }
```

```
    @Transactional(readOnly = true)
    public List<Course> findAll() {
        Query query = entityManager.createQuery("from Course");
        return query.getResultList();
    }
}
```

You can annotate each DAO method or the entire DAO class with `@Transactional` to make all of these methods transactional. This ensures that the persistence operations within a DAO method will by executed in the same transaction and hence by the same entity manager.

In the bean configuration file for JPA (i.e., `beans-jpa.xml`), you have to declare a `JpaTransactionManager` instance and enable declarative transaction management via `<tx:annotation-driven>`. You have to register a `PersistenceAnnotationBeanPostProcessor` instance to inject entity managers into properties annotated with `@PersistenceContext`.

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
        http://www.springframework.org/schema/tx
        http://www.springframework.org/schema/tx/spring-tx-2.5.xsd">
    ...
    <tx:annotation-driven />

    <bean id="transactionManager"
        class="org.springframework.orm.jpa.JpaTransactionManager">
        <property name="entityManagerFactory" ref="entityManagerFactory" />
    </bean>

    <bean name="courseDao"
        class="com.apress.springrecipes.course.jpa.JpaCourseDao" />

    <bean class="org.springframework.orm.jpa.support.➥
        PersistenceAnnotationBeanPostProcessor" />
</beans>
```

In Spring 2.5, a `PersistenceAnnotationBeanPostProcessor` instance will be registered automatically once you enable the `<context:annotation-config>` element. So, you can delete its explicit bean declaration.

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-2.5.xsd
```

```
        http://www.springframework.org/schema/tx
        http://www.springframework.org/schema/tx/spring-tx-2.5.xsd">

    <context:annotation-config />
    ...
</beans>
```

This bean post processor can also inject the entity manager factory into a property with the `@PersistenceUnit` annotation. This allows you to create entity managers and manage transactions by yourself. It's no different from injecting the entity manager factory via a setter method.

```
package com.apress.springrecipes.course.jpa;
...
import javax.persistence.EntityManagerFactory;
import javax.persistence.PersistenceUnit;

public class JpaCourseDao implements CourseDao {

    @PersistenceUnit
    private EntityManagerFactory entityManagerFactory;
    ...
}
```

`JpaTemplate` will translate the native JPA exceptions into exceptions in Spring's `DataAccessException` hierarchy. However, when calling native methods on a JPA entity manager, the exceptions thrown will be of native type `PersistenceException`, or other Java SE exceptions like `IllegalArgumentException` and `IllegalStateException`. If you want JPA exceptions to be translated into Spring's `DataAccessException`, you have to apply the `@Repository` annotation to your DAO class.

```
package com.apress.springrecipes.course.jpa;
...
import org.springframework.stereotype.Repository;

@Repository("courseDao")
public class JpaCourseDao implements CourseDao {
    ...
}
```

Then register a `PersistenceExceptionTranslationPostProcessor` instance to translate the native JPA exceptions into exceptions in Spring's `DataAccessException` hierarchy. You can also enable `<context:component-scan>` and delete the original `JpaCourseDao` bean declaration, as `@Repository` is a stereotype annotation in Spring 2.5.

```
<beans ...>
    ...
    <context:component-scan
        base-package="com.apress.springrecipes.course.jpa" />
```

```
    <bean class="org.springframework.dao.annotation.➥
        PersistenceExceptionTranslationPostProcessor" />
</beans>
```

9-6. Summary

In this chapter, you have learned how to integrate two popular ORM frameworks—Hibernate and JPA—into your Spring applications. Spring's support for different ORM frameworks is consistent, so you can easily apply these techniques to other ORM frameworks as well.

When using an ORM framework, you have to configure its resource factory (e.g., a session factory in Hibernate and an entity manager factory in JPA) with its API. Spring provides several factory beans for you to create an ORM resource factory as a singleton bean in the IoC container, so it can be shared between multiple beans via dependency injection. This resource factory can also take advantage of Spring's data access facilities, such as data sources and transaction managers.

When using an ORM framework on its own, you have to repeat certain routine tasks for each DAO operation. Spring simplifies an ORM framework's usage by providing template classes, DAO support classes, and transaction manager implementations. You can use them in a similar way for different ORM frameworks, as well as for JDBC.

For Hibernate and JPA, you can also use their plain APIs to implement DAOs that can be integrated with Spring's ORM support. In this way, your DAOs don't need to depend on Spring's API, and they can achieve the same benefits as using template classes.

In the next chapter, you will learn how to build web-based applications with the Spring MVC framework.