Capítulo 5. Implementando Persistencia en Base de Datos

Esta es la Parte 5 del tutorial paso a paso sobre sobre cómo desarrollar una aplicación web desde cero usando Spring Framework. En la Parte 1 hemos configurado el entorno y puesto en marcha una aplicación básica. En la Parte 2 hemos mejorado la aplicación que habíamos construido hasta entonces. En la Parte 3 hemos añadido toda la lógica de negocio y los tests unitarios, y en la Parte 4 hemos desarrollado la interface web. Ahora es el momento de introducir persistencia en base de datos. En las partes anteriores hemos visto cómo cargar algunos objetos de negocio definiendo beans en un archivo de configuración. Es obvio que esta solución nunca va a funcionar en el mundo real – cada vez que reiniciemos el servidor obtendremos de nuevo los precios originales. Necesitamos añadir código para persistir esos cambios en una base de datos.

5.1. Creación y rellenado de la base de datos

Antes de que podamos comenzar a desarrollar el código de persistencia, necesitamos una base de datos. En lugar de confiar en una base de datos integrada con la propia aplicación, vamos a utilizar una base de datos separada. Hemos planeado usar MySQL, una buena base de datos de código libre. Sin embargo, los pasos que se muestran a continuación serán similares para otras bases de datos (p. ej. PostgreSQL, HSQL...). Desde el enlace anterior, se puede descargar MySQL para diferentes sistemas operativos. Una vez instalada, cada distribución proporciona sus correspondientes scripts de inicio para arrancar la base de datos.

Primero, creamos el fichero '**springapp.sql**' (en el directorio '**db**') con las sentencias SQL necesarias para la creación de la base de datos springapp. Este fichero creará también la tabla products, que alojará los productos de nuestra aplicación. Además, establecerá los permisos para el usuario springappuser.

'**springapp/db/springapp.sql**':

```
CREATE DATABASE springapp;

GRANT ALL ON springapp.* TO springappuser@'%' IDENTIFIED BY 'pspringappuser';
GRANT ALL ON springapp.* TO springappuser@localhost IDENTIFIED BY 'pspringappuser';

USE springapp;

CREATE TABLE products (
  id INTEGER PRIMARY KEY,
  description varchar(255),
  price decimal(15,2)
);
CREATE INDEX products_description ON products(description);
```

A continuación, necesitamos añadir nuestros datos de prueba. Para ello, creamos el archivo '**load_data.sql**' en el directorio '**db**' con el siguiente contenido:

'**springapp/db/load_data.sql**':

```
INSERT INTO products (id, description, price) values(1, 'Lamp', 5.78);
INSERT INTO products (id, description, price) values(2, 'Table', 75.29);
INSERT INTO products (id, description, price) values(3, 'Chair', 22.81);
```

Por último, ejecutamos las siguientes instrucciones sobre la línea de comandos para crear y rellenar la base de datos:

```
linux:springapp/db# mysql -u root -p
Enter password:
...
mysql> source springapp.sql
mysql> source load_data.sql
```

Para poder acceder desde nuestra aplicación a la base de datos MySQL mediante JPA, debemos incluir las siguientes dependencias en el fichero '**pom.xml**':

Group Id	Artifact Id	Version
mysql	mysql-connector-java	5.1.24
org.hibernate.java-persistence	jpa-api	2.0-cr-1
org.hibernate	hibernate-entitymanager	4.2.0.Final
org.springframework	spring-orm	${org.springframework.version}

5.2. Crear una implementación para JPA de un Objeto de Acceso a Datos (DAO)

Comencemos creando un nuevo paquete llamado '**com.companyname.springapp.repository**' que contendrá cualquier clase que sea usada para el acceso a la base de datos. En este paquete vamos a crear un nuevo interface llamado ProductDao. Éste será el interface que definirá la funcionalidad de la implementación DAO que vamos a crear - esto nos permitirá elegir en el futuro otra implementación que se adapte mejor a nuestras necesidades (p. ej. JDBC, etc.).

'**springapp/src/main/java/com/companyname/springapp/repository/ProductDao.java**':

```
package com.companyname.springapp.repository;

import java.util.List;
```

```
import com.companyname.springapp.domain.Product;

public interface ProductDao {

    public List<Product> getProductList();

    public void saveProduct(Product prod);

}
```

A continuación, creamos una clase llamada `JPAProductDao` que será la implementación JPA de la interface anterior. Spring permite creación automática de beans de acceso a datos mediante la anotación `@Repository`. Asimismo, Spring reconoce las anotaciones del API estándar JPA. Por ejemplo, la anotación `@Persistence` es utilizada en la clase `JPAProductDao` para inyectar automáticamente el `EntityManager`.

'springapp/src/main/java/com/companyname/springapp/repository/JPAProductDao.java':

```
package com.companyname.springapp.repository;

import java.util.List;

import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;

import org.springframework.stereotype.Repository;
import org.springframework.transaction.annotation.Transactional;

import com.companyname.springapp.domain.Product;

@Repository(value = "productDao")
public class JPAProductDao implements ProductDao {

    private EntityManager em = null;

    /*
     * Sets the entity manager.
     */
    @PersistenceContext
    public void setEntityManager(EntityManager em) {
        this.em = em;
    }

    @Transactional(readOnly = true)
    @SuppressWarnings("unchecked")
    public List<Product> getProductList() {
        return em.createQuery("select p from Product p order by p.id").getResultList();
    }

    @Transactional(readOnly = false)
    public void saveProduct(Product prod) {
        em.merge(prod);
    }

}
```

Vamos a echarle un vistazo a los dos métodos DAO en esta clase. El primer método, `getProductList()`, ejecuta una consulta usando el `EntityManager`. Para ello incluimos en él una sentencia SQL que obtiene los objetos persistentes de la clase `Product`. El segundo método, `saveProduct()`, también usa el `EntityManager`. Esta vez hacemos un `merge` para almacenar el producto en la base de datos. Ambos métodos se ejecutan de manera transaccional gracias a la anotación `@Transactional`, con la diferencia de que el método `getProductList()` permite la ejecución de diversas consultas de lectura en paralelo.

Llegados a este punto, debemos modificar la clase `Product` para que se persista correctamente. Para ello modificamos el fichero **'Product.java'** y añadimos las anotaciones de JPA que realizan el mapeo entre los campos del objeto y aquellos de la base de datos. Asimismo, hemos añadido el campo `id` para mapear la clave primaria de la tabla `products`.

'springapp/src/main/java/com/companyname/springapp/domain/Product.java':

```
package com.companyname.springapp.domain;

import java.io.Serializable;

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="products")
public class Product implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id
```

```
    @Column(name = "id")
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Integer id;

    private String description;
    private Double price;

    public Integer getId()
    {
        return id;
    }

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

    public String getDescription() {
        return description;
    }

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

    public Double getPrice() {
        return price;
    }

    public void setPrice(Double price) {
        this.price = price;
    }

    public String toString() {
        StringBuffer buffer = new StringBuffer();
        buffer.append("Description: " + description + ";");
        buffer.append("Price: " + price);
        return buffer.toString();
    }
}
```

Los pasos anteriores completan la implementación JPA en nuestra capa de persistencia.

5.3. Implementar tests para la implementación DAO sobre JPA

Es el momento de añadir tests a nuestra aplicación DAO sobre JPA. Para ello, crearemos la clase '**JPAProductDaoTests**' dentro de paquete '**com.companyname.springapp.repository**' del la carpeta '**src/test/java**'.

'**springapp/src/test/java/com/companyname/springapp/repository/JPAProductDaoTests.java**':

```
package com.companyname.springapp.repository;

import java.util.List;

import static org.junit.Assert.*;

import org.junit.Before;
import org.junit.Test;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.companyname.springapp.domain.Product;


public class JPAProductDaoTests {

    private ApplicationContext context;
    private ProductDao productDao;

    @Before
    public void setUp() throws Exception {
        context = new ClassPathXmlApplicationContext("classpath:test-context.xml");
        productDao = (ProductDao) context.getBean("productDao");
    }

    @Test
    public void testGetProductList() {
        List<Product> products = productDao.getProductList();
        assertEquals(products.size(), 3, 0);
    }

    @Test
    public void testSaveProduct() {
        List<Product> products = productDao.getProductList();
```

```
        Product p = products.get(0);
        Double price = p.getPrice();
        p.setPrice(200.12);
        productDao.saveProduct(p);

        List<Product> updatedProducts = productDao.getProductList();
        Product p2 = updatedProducts.get(0);
        assertEquals(p2.getPrice(), 200.12, 0);

        p2.setPrice(price);
        productDao.saveProduct(p2);
    }
}
```

Aún no disponemos del archivo que contiene el contexto de la aplicación, y que es cargado por este test, por lo que vamos a crear este archivo en una nueva carpeta de recursos de tipo `Source folder` en nuestro proyecto, a la que llamaremos '**src/test/resources**':

'**springapp/src/test/resources/test-context.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:context="http://www.springframework.org/schema/context"
 xmlns:p="http://www.springframework.org/schema/p"
 xmlns:tx="http://www.springframework.org/schema/tx"
 xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd
  http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx.xsd
  http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context.xsd">

    <!-- holding properties for database connectivity /-->
    <context:property-placeholder location="classpath:jdbc.properties"/>

    <!-- enabling annotation driven configuration /-->
    <context:annotation-config/>

    <bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
      <property name="driverClassName" value="${jdbc.driverClassName}"/>
      <property name="url" value="${jdbc.url}"/>
      <property name="username"  value="${jdbc.username}"/>
      <property name="password" value="${jdbc.password}"/>
    </bean>

    <bean id="entityManagerFactory"
 class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean"
 p:dataSource-ref="dataSource"
 p:jpaVendorAdapter-ref="jpaAdapter">
 <property name="loadTimeWeaver">
          <bean class="org.springframework.instrument.classloading.InstrumentationLoadTimeWeaver"/>
        </property>
        <property name="persistenceUnitName" value="springappPU"></property>
    </bean>

    <bean id="jpaAdapter"
 class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter"
 p:database="${jpa.database}"
 p:showSql="${jpa.showSql}"/>

    <bean id="transactionManager" class="org.springframework.orm.jpa.JpaTransactionManager"
 p:entityManagerFactory-ref="entityManagerFactory"/>

    <tx:annotation-driven transaction-manager="transactionManager"/>

    <!-- Scans the classpath of this application for @Components to deploy as beans -->
    <context:component-scan base-package="com.companyname.springapp.repository" />
    <context:component-scan base-package="com.companyname.springapp.service" />

</beans>
```

Hemos definido un `productDao` el cual es la clase que estamos testeando. Además hemos definido un `DataSource` con comodines para los valores de configuración. Sus valores serán tomados de un archivo de propiedades en tiempo de ejecución. El bean `property-placeholder` que hemos declarado leerá este archivo de propiedades y sustituirá cada comodín con su valor actual. Esto es conveniente puesto que separa los valores de conexión en su propio archivo, y estos valores a menudo suelen ser cambiados durante el despliegue de la aplicación. Vamos a poner este nuevo archivo tanto en el directorio '**src/test/resources**' como en el directorio '**webapp/WEB-INF/classes**' por lo que estará disponible cuando ejecutemos los tests y cuando despleguemos la aplicación web. El contenido de este archivo de propiedades es:

'**springapp/src/test/resources/jdbc.properties**' y '**springapp/src/main/webapp/WEB-INF/classes/jdbc.properties**':

```
jdbc.driverClassName=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://localhost:3306/springapp
jdbc.username=springappuser
jdbc.password=pspringappuser
hibernate.dialect=org.hibernate.dialect.MySQLDialect

jpa.database = MYSQL
```

```
hibernate.generate_statistics = true
hibernate.show_sql = true
jpa.showSql = true
jpa.generateDdl = true
```

Por otro lado, la configuración del bean `entityManager` requerirá la creación de un fichero donde se defina la unidad de persistencia. Por defecto, este fichero se deberá llamar '**`persistence.xml`**' y deberá crearse bajo el directorio '**`META-INF`**' dentro del directorio de recursos de la aplicación '**`src/main/resources`**'

'**`springapp/src/main/resources/META-INF/persistence.xml`**':

```
<?xml version="1.0" encoding="UTF-8"?>
<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="springappPU" transaction-type="RESOURCE_LOCAL">
 </persistence-unit>
</persistence>
```

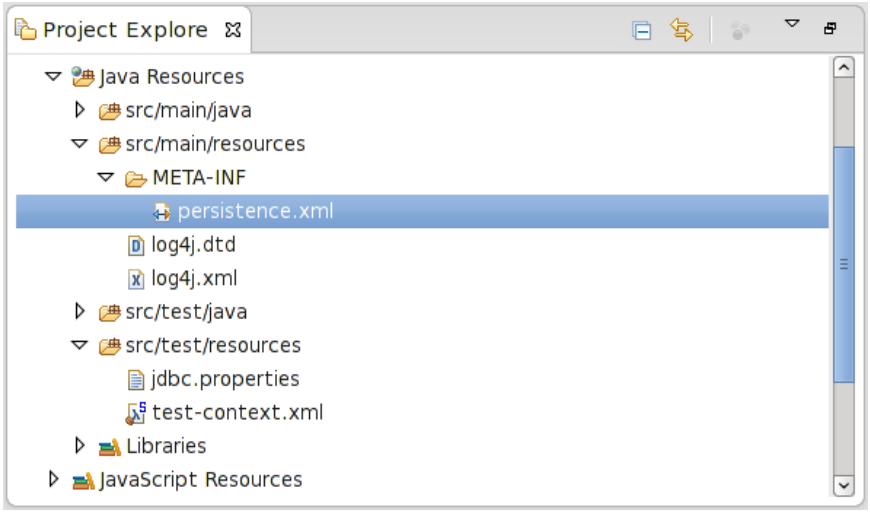
Ahora disponemos del codigo suficiente para ejecutar los tests de '**`JPAProductDaoTests`**' y hacerlos pasar.

5.4. Resumen

Ya hemos completado la capa de persistencia y en la próxima parte vamos a integrarla con nuestra aplicación web. Pero primero, resumamos rápidamente todo lo que hemos hecho en esta parte.

- Primero hemos configurado nuestra base de datos y ejecutado las sentencias SQL para crear una tabla en la base de datos y cargar algunos datos de prueba.
- Hemos creado una clase DAO que manejará el trabajo de persistencia mediante JPA usando la clase `Product`.
- Finalmente hemos creado tests de integración para comprobar su funcionamiento.

A continuación puedes ver una captura de pantalla que muestra el aspecto que debería tener la estructura de directorios del proyecto después de seguir todas las instrucciones anteriores.



La estructura de directorios del proyecto al final de la parte 5