Capitulo 5. Implementando Persistencia en Base de Datos

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

5.1. Crear un script de inicio de base de datos

Antes de que podamos comenzar a desarrollar el codigo de persistencia, necesitamos una base de datos. Hemos planeado usar HSQL, la cual es una buena base de datos escrita en Java. Esta base de datos es distribuida con Spring por lo que podemos copiar su archivo jar al directorio lib de la aplicacion web. Copia el archivo `hsqldb.jar` del directorio `"spring-framework-2.5/lib/hsqldb"` al directorio `"springapp/war/WEB-INF/lib"`. Vamos a usar HSQL en modo standalone. Esto significa que tendremos que arrancar la base de datos de forma serparada en lugar de confiar en una base de datos integrada con la propia aplicacion. De esta manera sera mas sencillo ver los cambios hechos en la base de datos cuando ejecutemos la aplicacion.

Necesitamos un script o un archivo de lotes para iniciar la base de datos. Crea un directorio `"db"` dentro del directorio principal `"springapp"`. Este nuevo directorio contendra los archivos de la base de datos. Ahora, añadamos un script de inicio:

Desde Linux/Mac OS X añadimos a:

`"springapp/db/server.sh"`:

```
java -classpath ../war/WEB-INF/lib/hsqldb.jar org.hsqldb.Server -database test
```

No olvides cambiar los permisos de ejecucion con el comando **'chmod +x server.sh'**.

Desde Windows añadimos a:

`"springapp/db/server.bat"`:

```
java -classpath ..\war\WEB-INF\lib\hsqldb.jar org.hsqldb.Server -database test
```

Ahora ya puedes abrir una ventana de comandos, ir al directorio `"springapp/db"` y arrancar la base de datos ejecutando el script de arranque correspondiente a tu sistema operativo.

5.2. Crear una tabla y scripts de prueba de datos

Primero, vamos a ver las sentencias SQL necesarias para crear la tabla. Crea el archivo 'create_products.sql' en el directorio db.

`"springapp/db/create_products.sql" con el siguiente contenido`:

```
CREATE TABLE products (
  id INTEGER NOT NULL PRIMARY KEY,
  description varchar(255),
  price decimal(15,2)
);
CREATE INDEX products_description ON products(description);
```

Ahora necesitamos añadir nuestros datos de prueba. Crea 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);
```

En la seccion siguiente vamos a añadir algunas tareas Ant a su script de construccion de manera que podamos ejecutar los scripts SQL.

5.3. Añadir tareas Ant para ejecutar los scripts SQL y cargar datos de prueba

Vamos a crear tablas y poblarlas con datos de prueba usando el comando incorporado en Ant "sql". Para usarlo necesitamos añadir algunos parametros de conexion a la base de datos en un archivo de propiedades

"springapp/build.properties":

```
# Ant properties for building the springapp

appserver.home=${user.home}/apache-tomcat-6.0.14
# for Tomcat 5 use $appserver.home}/server/lib
# for Tomcat 6 use $appserver.home}/lib
appserver.lib=${appserver.home}/lib

deploy.path=${appserver.home}/webapps

tomcat.manager.url=http://localhost:8080/manager
tomcat.manager.username=tomcat
tomcat.manager.password=s3cret

db.driver=org.hsqldb.jdbcDriver
db.url=jdbc:hsqldb:hsql://localhost
db.user=sa
db.pw=
```

A continuacion añadimos los tareas que necesitamos al archivo de contruccion de Ant. Hay tareas para crear y borrar tablas, y para cargar y borrar datos de prueba.

Añade las siguientes tareas a "springapp/build.xml":

```
    <target name="createTables">
        <echo message="CREATE TABLES USING: ${db.driver} ${db.url}"/>
        <sql driver="${db.driver}"
             url="${db.url}"
             userid="${db.user}"
             password="${db.pw}"
             onerror="continue"
             src="db/create_products.sql">
            <classpath refid="master-classpath"/>
        </sql>
    </target>

    <target name="dropTables">
        <echo message="DROP TABLES USING: ${db.driver} ${db.url}"/>
        <sql driver="${db.driver}"
             url="${db.url}"
             userid="${db.user}"
             password="${db.pw}"
             onerror="continue">
            <classpath refid="master-classpath"/>

        DROP TABLE products;

        </sql>
    </target>

    <target name="loadData">
        <echo message="LOAD DATA USING: ${db.driver} ${db.url}"/>
        <sql driver="${db.driver}"
             url="${db.url}"
             userid="${db.user}"
             password="${db.pw}"
             onerror="continue"
             src="db/load_data.sql">
            <classpath refid="master-classpath"/>
        </sql>
    </target>

    <target name="printData">
        <echo message="PRINT DATA USING: ${db.driver} ${db.url}"/>
        <sql driver="${db.driver}"
             url="${db.url}"
             userid="${db.user}"
             password="${db.pw}"
             onerror="continue"
             print="true">
            <classpath refid="master-classpath"/>

        SELECT * FROM products;

        </sql>
    </target>

    <target name="clearData">
        <echo message="CLEAR DATA USING: ${db.driver} ${db.url}"/>
        <sql driver="${db.driver}"
             url="${db.url}"
             userid="${db.user}"
             password="${db.pw}"
             onerror="continue">
            <classpath refid="master-classpath"/>

        DELETE FROM products;

        </sql>
    </target>

    <target name="shutdownDb">
        <echo message="SHUT DOWN DATABASE USING: ${db.driver} ${db.url}"/>
```

```
        <sql driver="${db.driver}"
             url="${db.url}"
             userid="${db.user}"
             password="${db.pw}"
             onerror="continue">
            <classpath refid="master-classpath"/>

        SHUTDOWN;

        </sql>
    </target>
```

Ahora puedes ejecutar **'ant createTables loadData printData'** para preparar los datos de prueba que vamos a usar despues.

5.4. Crear una implementacion para JDBC de un Objeto de Acceso a Datos (DAO)

Comencemos creando un nuevo directorio llamado **"src/springapp/repository"** que contendra cualquier clase que sea usada para el acceso a la base de datos. En este directorio vamos a crear un nuevo interface llamado `ProductDao`. Este sera el interface que definira la funcionalidad de la implementacion DAO que vamos a crear - esto nos permitira elegir en el futuro otra implementacion que se adapte mejor a nuestras necesidades.

"springapp/src/springapp/repository/ProductDao.java":

```
package springapp.repository;

import java.util.List;

import springapp.domain.Product;

public interface ProductDao {

    public List<Product> getProductList();

    public void saveProduct(Product prod);

}
```

A continuacion creamos una clase llamada `JdbcProductDao` que sera la implementacion JDBC de la interface anterior. Spring dispone de un framework de abstraccion JDBC que vamos a usar. La mayor diferencia entre usar JDBC directamente y el framework JDBC de Spring es que no tienes que preocuparte de abrir o cerrar conexiones, o cualquier codigo similar. Todo esto es manejado de manera automatica. Otra ventaja es que no tienes que capturar ninguna excepcion, a menos que quieras. Spring envuelve todas las excepciones de tipo SQLException en un familia de excepciones de tipo unchecked que heredan de DataAccessException. Si lo deseas, puedes capturar esta excepcion, pero puesto que muchas excepciones de base de datos son imposibles de recuperar de ninguna manera, puedes simplemente dejar que esta excepcion se propague hacia un nivel superior. La clase `SimpleJdbcDaoSupport` provee el acceso necesario para obtener un previamente configurado objeto `SimpleJdbcTemplate`, por lo que podemos heredar de esta clase. Todo lo que tenemos que proveer en el contexto de la aplicacion es un `DataSource` convenientemente configurado.

"springapp/src/springapp/repository/JdbcProductDao.java":

```
package springapp.repository;

import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.List;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.jdbc.core.namedparam.MapSqlParameterSource;
import org.springframework.jdbc.core.simple.ParameterizedRowMapper;
import org.springframework.jdbc.core.simple.SimpleJdbcDaoSupport;

import springapp.domain.Product;

public class JdbcProductDao extends SimpleJdbcDaoSupport implements ProductDao {

    /** Logger for this class and subclasses */
    protected final Log logger = LogFactory.getLog(getClass());


    public List<Product> getProductList() {
        logger.info("Getting products!");
        List<Product> products = getSimpleJdbcTemplate().query(
                "select id, description, price from products",
                new ProductMapper());
        return products;
    }

    public void saveProduct(Product prod) {
        logger.info("Saving product: " + prod.getDescription());
        int count = getSimpleJdbcTemplate().update(
            "update products set description = :description, price = :price where id = :id",
            new MapSqlParameterSource().addValue("description", prod.getDescription())
                .addValue("price", prod.getPrice())
```

```
                .addValue("id", prod.getId()));
        logger.info("Rows affected: " + count);
    }

    private static class ProductMapper implements ParameterizedRowMapper<Product> {

        public Product mapRow(ResultSet rs, int rowNum) throws SQLException {
            Product prod = new Product();
            prod.setId(rs.getInt("id"));
            prod.setDescription(rs.getString("description"));
            prod.setPrice(new Double(rs.getDouble("price")));
            return prod;
        }

    }

}
```

Vamos a echarle un vistazo a los dos metodos DAO en esta clase. Puesto que estamos extendiendo `SimpleJdbcSupport` disponemos de un objeto `SimpleJdbcTemplate` preparado y listo para usar. Este objeto es accedido llamando al metodo `getSimpleJdbcTemplate()`.

El primer metodo, `getProductList()` ejecuta una consulta usando `SimpleJdbcTemplate`. Para ello incluimos en el una sentencia SQL y una clase que pueda manejar el mapeo entre el el `ResultSet` y la clase `Product`. En nuestro caso este mapeador es una clase llamada `ProductMapper` que hemos definido como una clase interna del DAO. Por supuesto que esta clase no sera usada fuera del DAO por lo que hacerla interna es una buena solucion.

ProductMapper implementa la interface `ParameterizedRowMapper` que define un unico metodo llamado mapRow , y que por tanto debe ser implementado. Este metodo mapeara los datos de cada fila de la base de datos a una clase que representa la entidad que estas recuperando con tu consulta. Puesto que `RowMapper` es parametizado, el metodo mapRow devuelve el mismo tipo que ha creado.

El segundo metodo, saveProduct, tambien usa `SimplJdbcTemplate`. Esta vez hacemos un `update` pasando la correspondiente sentencia SQL junto con el valor de los parametros mediante un objeto `MapSqlParameterSource`. Usar `MapSqlParameterSource` nos permite usad parametros con nombre en lugar de los caracteres "?" que hubieras necesitado para escribir una sentencia SQL. Los parametros con nombre hacen tu codigo mas explicito y evitan problemas causados por parametros con valores incorrectos (debido a errores de ordenacion, etc). El metodo update devuelve el numero de filas afectadas en la base de datos.

Necesitamos almacenar el valor de la primera clave para cada producto de la clase Product. Esta clave sera usada cuando realicemos cualquier cambio en el objeto y lo volvamos a persistir en la base de datos. Para almacenar esta clave añadimos una variable privada llamada 'id' complementada con sus correspondientes getters y setters.

"springapp/src/springapp/domain/Product.java":

```
package springapp.domain;

import java.io.Serializable;

public class Product implements Serializable {

    private int id;
    private String description;
    private Double price;

    public void setId(int i) {
        id = i;
    }

    public int getId() {
        return 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();
```

```
    }
}
```

Esto completa la implementacion JDBC en nuestra capa de persistencia.

5.5. Implementar tests para la implementacion DAO sobre JDBC

Es el momento de añadir tests a nuestra aplicacion DAO sobre JDBC. Spring dispone de un extenso framework para tests que soporta JUnit 3.8 y 4 asi como TestNG. No podemos cubrir todos ellos en esta guia pero vamos a mostrar una implementacion simple basada en el soporte especifico para JUnit 3.8 de Spring. Necesitamos añadir a nuestro proyecto el archivo jar que contiene el framework de tests de Spring. Copia **spring-test.jar** desde el directorio **"spring-framework-2.5/dist/modules"** hasta el directorio **"springapp/war/WEB-INF/lib"**.

Ahora podemos crear nuestra clase de tests. Extendiendo `AbstractTransactionalDataSourceSpringContextTests` obtenemos un monton de caracteristicas muy utiles de manera totalmente automatica. Conseguimos inyeccion de dependencias desde el contexto de la aplicacion en cualquier metodo setter. Este contexto de aplicacion es cargado por el framework de tests. Todo lo que necesitamos hacer es especificar su nombre en el metodo `getConfigLocations`. Ademas, obtenemos la oportunidad de preparar nuestra base de datos con los datos de prueba apropiados a traves del metodo `onSetUpInTransaction`. Esto es importante, puesto que desconocemos el estado de la base de datos cuando ejecutamos nuestros tests. Puesto que estamos extendiendo un test "Transactional", cualquier cambio que hagamos sera automaticamente cancelado una vez que el test finalice. Los metodos `deleteFromTables` y `executeSqlScript` estan definidos en la superclase, por lo que no tenemos que implementarlos para cada test. Simplemente hay que pasarle los nombres de tabla a vaciar y el nombre del script que contiene los datos de prueba.

"springapp/test/springapp/domain/JdbcProductDaoTests.java":

```
package springapp.repository;

import java.util.List;

public class JdbcProductDaoTests extends AbstractTransactionalDataSourceSpringContextTests {

    private ProductDao productDao;


    public void setProductDao(ProductDao productDao) {
        this.productDao = productDao;
    }

    @Override
    protected String[] getConfigLocations() {
        return new String[] {"classpath:test-context.xml"};
    }

    @Override
    protected void onSetUpInTransaction() throws Exception {
        super.deleteFromTables(new String[] {"products"});
        super.executeSqlScript("file:db/load_data.sql", true);
    }

    public void testGetProductList() {

        List<Product> products = productDao.getProductList();

        assertEquals("wrong number of products?", 3, products.size());

    }

    public void testSaveProduct() {

        List<Product> products = productDao.getProductList();

        for (Product p : products) {
            p.setPrice(200.12);
            productDao.saveProduct(p);
        }

        List<Product> updatedProducts = productDao.getProductList();
        for (Product p : updatedProducts) {
            assertEquals("wrong price of product?", 200.12, p.getPrice());
        }

    }

}
```

Aun no disponemos del archivo que contiene el contexto de la aplicacion, y que es cargado por este test, por lo que vamos a crear este archivo en el directorio **"springapp/test"**:

"springapp/test/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"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

    <!-- the test application context definition for the jdbc based tests -->

    <bean id="productDao" class="springapp.repository.JdbcProductDao">
        <property name="dataSource" ref="dataSource" />
    </bean>

    <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="propertyConfigurer"
          class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
        <property name="locations">
            <list>
                <value>classpath:jdbc.properties</value>
            </list>
        </property>
    </bean>

    <bean id="transactionManager"
          class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <property name="dataSource" ref="dataSource" />
    </bean>

</beans>
```

Hemos definido un `productDao` el cual es la clase que estamos testeando. Ademas hemos definido un `DataSource` con comodines para los valores de configuracion. Sus valores seran ajustados mediante un archivo de propiedades en tiempo de ejecucion. La clase `PropertyPlaceholderConfigurer` que hemos declarado leera este archivo de propiedades y sustituira cada comodin con su valor actual. Esto es conveniente puesto que separa los valores de conexion en su propio archivo, y estos valores a menudo suelen ser cambiados durante el despliegue de la aplicacion. Vamos a poner este nuevo archivo en el directorio **"war/WEB-INF/classes"** por lo que estara disponible cuando ejecutemos la aplicacion ademas de cuando despleguemos la aplicacion web. El contenido de este archivo de propiedades es:

"springapp/war/WEB-INF/classes/jdbc.properties":

```
jdbc.driverClassName=org.hsqldb.jdbcDriver
jdbc.url=jdbc:hsqldb:hsql://localhost
jdbc.username=sa
jdbc.password=
```

Puesto que hemos añadido un archivo de configuracion en el directorio **"test""** y el archivo de propiedades jdbc.properties en el directorio **"WEB-INF/classes"**, vamos a añadir una nueva entrada al classpath para nuestros tests. Esta entrada deberia ir despues de la declaracion de la propiedad `'test.dir'`:

"springapp/build.xml":

```
...
    <property name="test.dir" value="test"/>


    <path id="test-classpath">
        <fileset dir="${web.dir}/WEB-INF/lib">
            <include name="*.jar"/>
        </fileset>
        <pathelement path="${build.dir}"/>
        <pathelement path="${test.dir}"/>
        <pathelement path="${web.dir}/WEB-INF/classes"/>
    </path>

...
```

Ahora disponemos del codigo suficiente para ejecutar nuestros tests y hacerlos pasar pero queremos hacer un cambio adicional al script de Ant. Es una buena practica separar cualquier test de integracion que depende de una base de datos real del resto de los tests. Por ello vamos a añadir una tarea alternativa llamada "dbTests" a nuestro script de Ant, y vamos a excluir los tests sobre la base de datos de la tarea "tests".

"springapp/build.xml":

```
...

    <target name="tests" depends="build, buildtests" description="Run tests">
        <junit printsummary="on"
            fork="false"
            haltonfailure="false"
            failureproperty="tests.failed"
            showoutput="true">
            <classpath refid="test-classpath"/>
            <formatter type="brief" usefile="false"/>

            <batchtest>
                <fileset dir="${build.dir}">
```

```
                    <include name="**/*Tests.*"/>
                    <exclude name="**/Jdbc*Tests.*"/>
                </fileset>
            </batchtest>

        </junit>

        <fail if="tests.failed">
            tests.failed=${tests.failed}
            ***********************************************************
            ***********************************************************
            ****  One or more tests failed!  Check the output ...  ****
            ***********************************************************
            ***********************************************************
        </fail>
    </target>

    <target name="dbTests" depends="build, buildtests,dropTables,createTables,loadData"
            description="Run db tests">
        <junit printsummary="on"
            fork="false"
            haltonfailure="false"
            failureproperty="tests.failed"
            showoutput="true">
            <classpath refid="test-classpath"/>
            <formatter type="brief" usefile="false"/>

            <batchtest>
                <fileset dir="${build.dir}">
                    <include name="**/Jdbc*Tests.*"/>
                </fileset>
            </batchtest>

        </junit>

        <fail if="tests.failed">
            tests.failed=${tests.failed}
            ***********************************************************
            ***********************************************************
            ****  One or more tests failed!  Check the output ...  ****
            ***********************************************************
            ***********************************************************
        </fail>
    </target>

...
```

Hora de ejecutar este test, ejecuta **'ant dbTests'** para ver si el test pasa.

5.6. Resumen

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

- Primero hemos configurado nuestra base de datos y creado los scripts de arranque.
- Hemos creado scripts para crear una tabla en la base de datos y cargar algunos datos de prueba.
- A continuacion hemos añadido algunas tareas a nuestro script de Ant que ejecutaremos cuando necesitemos crear o borrar la tabla, y tambien cuando necesitamos añadir y borrar los datos de prueba.
- Hemos creado una clase DAO que manejara el trabajo de persistencia usando la clase `SimpeJdbcTemplate` de Spring.
- Finalmente hemos creado tests de integracion y sus correspondientes tareas Ant para ejecutarlos.

Debajo puedes ver una captura de pantalla mostrando como debe aparecer tu estructura de directorios despues de seguir todas las instrucciones anteriores.

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