Ejercicios de Acceso a datos en Spring

Índice

Continuaremos en esta sesión con la aplicación de pedidos, simplificando el código del DAO gracias a Spring y añadiendo transaccionalidad declarativa.

1. Uso de JDBC en Spring (1 punto)

Tendrás que añadirle al pom.xml una nueva dependencia con las librerías de Spring para JDBC:

```
<dependency>
   <groupId>org.springframework</groupId>
   <artifactId>spring-jdbc</artifactId>
   <version>3.2.0.RELEASE</version>
</dependency>
```

Vamos a añadirle al DAO un método `List<Pedido> listarPedidos()` que devuelva los pedidos hechos hasta el momento. Usaremos el API de Spring en el código, en concreto `SimpleJdbcTemplate`. Para ello sigue estos pasos:

1. Modifica el interfaz `IPedidosDAO` para incluir la nueva operación.
2. Tienes que definir un nuevo campo en `PedidosDAOJDBC` con el template:

   ```
   private .JdbcTemplate template;
   ```

   Puedes inicializar este campo del mismo modo que en los apuntes, usando el "truco" del método `setDataSource`. Por tanto, debes crear el propio método setDataSource y quitar la anotación @Autowired del campo "ds" y ponerla en el método. Este método debe inicializar la variable ds y el template. Necesitas las dos porque aunque el método que vas a introducir usa el template, el código de insertarPedido usa directamente el "ds".
3. Tendrás que crear la clase Pedido en el paquete `es.ua.jtech.spring`. Será simplemente un javabean con *getters* y *setters*. Ponle los campos correspondientes a las columnas de la BD (id, idProducto, idCliente, unidades).
4. Necesitarás una clase que implemente el interfaz `RowMapper<Pedido>`, que a partir de un registro de la BD devuelva un objeto Pedido. Puede ser la propia `PedidosJDBCDAO` o una clase adicional.
5. Por último tendrás que escribir el código de "listarPedidos". Observa que es mucho más simple de lo que sería si tuvieras que usar JDBC "puro".

Puedes comprobar el funcionamiento con el servlet "ListarPedidos" , disponible en las plantillas de la sesión. Cópialo en el paquete `es.ua.jtech.spring.web` y fíjate que está mapeado con la URL "listarPedidos". El código no es muy "correcto", ya que accede directamente al DAO sin pasar por la capa de negocio, pero así no tendrás que modificar `IPedidosBO` y `PedidosBOSimple`.

2. Transaccionalidad declarativa (1 punto)

Supongamos que cada vez que se hace un pedido se tiene que avisar al proveedor, para que nos sirva los productos. Si el aviso no puede enviarse, el pedido debería anularse y por tanto se debería hacer un *rollback* de las operaciones realizadas en la base de datos.

En las plantillas de la sesión hay un interfaz `IMensajeria` y una clase `MensajeriaDummy`, que lo implementa. `MensajeriaDummy` supuestamente se encarga de enviar el aviso (y decimos "supuestamente" porque en realidad lo único que hace es imprimir un mensaje en la salida estándar). La clase permite simular el mal funcionamiento del aviso a través de una variable static. Sigue estos pasos:

1. Copia el interface `IMensajeria` y la clase `MensajeriaDummy` en el *package* `es.ua.jtech.spring.negocio`
2. Copia el JSP "mensajeria.jsp" en "src/main/webapp". Este JSP te permite ver el "estado del servicio de avisos" y simular el fallo en el aviso poniendo el estado en "off"
3. `PedidosBOSimple` necesitará de un bean de tipo `IMensajeria` para funcionar. Define un campo de ese tipo igual que hiciste con el DAO y anótalo para que Spring gestione el objeto:

```
    @Autowired
    private IMensajeria m;
```

4. Modifica el código de "insertarPedido" de `GestorPedidosSimple` para que
   - Después de llamar al DAO llame al método "enviarAvisoPedido" del objeto de tipo `IMensajeria`
   - Sea transaccional, de modo que si se produce un `PedidosException` se haga un *rollback*.
   - Para que todo funcione, tendrás además que configurarlo en el fichero `src/main/webapp/WEB-INF/beans.xml`. Consulta apuntes y/o transparencias para ver un ejemplo:
     1. Definir un "transaction manager" para JDBC. Cuidado, su propiedad "dataSource" debe referenciar ("ref") al DataSource que definiste en la sesión anterior.
     2. Usar la etiqueta `tx:annotation-driven` y referenciar el "transaction-manager" que acabas de definir. Para esta etiqueta necesitas el espacio de nombres "tx" en el XML, así que puedes cambiar la cabecera del beans.xml por esta, que ya lo tiene incluido:

```
<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:jee="http://www.springframework.org/schema/jee"
  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/context
    http://www.springframework.org/schema/context/spring-context-3.2.xsd
```

```
    http://www.springframework.org/schema/jee
     http://www.springframework.org/schema/jee/spring-jee-3.2.xsd
    http://www.springframework.org/schema/tx
    http://www.springframework.org/schema/tx/spring-tx-3.2.xsd">
```

5. Cambia el código de "insertarPedido" en el DAO para que la conexión con la BD (variable "con") se obtenga y se libere a través de la clase de Spring `DataSourceUtils`, ya que si Spring no controla la conexión no será posible hacer el *rollback*.
6. Comprueba que todo funciona, accediendo a "mensajeria.jsp" para poner el estado del "servidor de avisos" a OFF e insertando un pedido como hacías hasta ahora. Si todo es correcto, se generará un PedidosException y el pedido no debería aparecer en la BD al no hacerse un "commit" de la operación.

3. Uso de JPA en Spring (1 punto)

Vamos a crear una clase de negocio que se ocupe de los productos usando el API JPA. Para simplificar, nos permitirá únicamente mostrar productos sabiendo su código. Sigue estos pasos:

1. Introduce en la sección de dependencias del pom.xml las dependencias de Spring ORM, Hibernate JPA y slf4j:

```
<dependency>
   <groupId>org.springframework</groupId>
   <artifactId>spring-orm</artifactId>
   <version>3.2.0.RELEASE</version>
</dependency>
<dependency>
   <groupId>org.hibernate</groupId>
   <artifactId>hibernate-entitymanager</artifactId>
   <version>3.5.6-Final</version>
</dependency>
<dependency>
   <groupId>org.slf4j</groupId>
   <artifactId>slf4j-log4j12</artifactId>
   <version>1.5.8</version>
   <scope>runtime</scope>
</dependency>
```

2. Crea la clase `ProductoEntity` en el package `es.ua.jtech.spring` con el siguiente código (cuidado, faltan getters y setters)

```
package es.ua.jtech.spring;

import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name="productos")
public class ProductoEntity {
```

```
    @Id
    private int id;
    private String nombre;

    //¡¡Faltan getters y setters, genéralos!!
}
```

3. Como fichero `persistence.xml` puedes usar el siguiente. Créalo simplemente como un XML, dentro de "src/main/resources". Crea allí una carpeta "META-INF" y pon el "persistence.xml" ahí dentro. ¡No lo confundas con el "META-INF" de "webapp"!.

```
<persistence version="1.0" xmlns="http://java.sun.com/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd">

    <persistence-unit name="pedidosSpringPU"
transaction-type="RESOURCE_LOCAL">
        <class>es.ua.jtech.spring.ProductoEntity</class>
    </persistence-unit>

</persistence>
```

4. Crea el interface `IProductosDAO` en el package `es.ua.jtech.spring.datos`

```
package es.ua.jtech.spring.datos;

import es.ua.jtech.spring.ProductoEntity;

public interface IProductosDAO {
    public ProductoEntity buscarProducto(int id);
}
```

5. Crea una clase `ProductosDAOJPA` que implemente el interfaz anterior, en el mismo package. Inyecta en ella el EntityManager con la anotación @PersistenceContext, e implementa el método "buscarProducto" usando el API JPA. Finalmente, anota la clase con `@Repository`, para que sea un bean de Spring.
6. Introduce la configuración necesaria en el fichero "beans.xml" como se muestra en los apuntes. Básicamente tienes que definir un bean de la clase `org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean`. Consulta apuntes o transparencias para ver las propiedades que debe tener. Te valdrán las que se muestran allí, salvo por la propiedad "dataSource", cuyo "ref" debe referenciar el identificador del DataSource que definiste en la sesión 1 del módulo.
7. Para probar la implementación usa el servlet `VerProducto`, disponible en las plantillas de la sesión. Tendrás que llamarlo pasándole el `id` del producto que quieres ver como parámetro. Por ejemplo, verProducto?id=1 para ver los datos del producto 1.