Acceso a datos

Índice

1 La filosofía del acceso a datos en Spring..... 2
2 Uso de JDBC..... 3
2.1 JDBCTemplate..... 4
2.2 Consultas de selección..... 5
2.3 Consultas de actualización..... 6
3 Uso de JPA..... 7
3.1 Formas de acceder a la "EntityManagerFactory"..... 7
3.2 JpaTemplate..... 9
4 Transaccionalidad declarativa..... 11
4.1 El "Transaction Manager"..... 11
4.2 La anotación @Transactional..... 12
4.3 Transaccionalidad y uso directo de JDBC..... 14

En este tema veremos las facilidades que proporciona Spring para implementar nuestros objetos de la capa de acceso a datos. Veremos que nos permite simplificar el código, reduciendo el código repetitivo, y uniformizar el tratamiento independientemente de la implementación subyacente (JPA, JDBC, ...). Finalmente abordaremos la transaccionalidad, que es un aspecto íntimamente ligado con el acceso a datos, aunque su gestión no suele estar localizada en el código de los DAOs sino en la capa de negocio.

Debido al auge en los últimos años del *big data* en la web, la "punta de lanza" de Spring en la capa de acceso a datos se ha movido más hacia la integración en Spring de bases de datos no relacionales (NoSQL). El proyecto Spring que se ocupa de estos aspectos es Spring Data, que, no obstante, queda fuera del ámbito de estos apuntes.

1. La filosofía del acceso a datos en Spring

Spring proporciona básicamente dos ventajas a la hora de dar soporte a nuestros DAOs:

- Simplifica las operaciones de acceso a datos en APIs tediosos de utilizar como JDBC, proporcionando una capa de abstracción que reduce la necesidad de código repetitivo. Para ello se usan los denominados *templates*, que son clases que implementan este código, permitiendo que nos concentremos en la parte "interesante".
- Define una rica jerarquía de excepciones que modelan todos los problemas que nos podemos encontrar al operar con la base de datos, y que son independientes del API empleado.

Un **template** de acceso a datos en Spring es una clase que encapsula los detalles más tediosos (como por ejemplo la necesidad de abrir y cerrar la conexión con la base de datos en JDBC), permitiendo que nos ocupemos únicamente de la parte de código que hace realmente la tarea (inserción, consulta, ...)

Spring ofrece diversas *templates* entre las que elegir, dependiendo del API de persistencia a emplear. Dada la heterogeneidad de los distintos APIs La implementación del DAO variará según usemos un API u otro , aunque en todos ellos Spring reduce enormemente la cantidad de código que debemos escribir, haciéndolo más mantenible.

Por otro lado, en APIs de acceso a datos como JDBC hay dos problemas básicos con respecto a la **gestión de excepciones**:

- Hay muy pocas excepciones distintas definidas para acceso a datos. Como consecuencia, la más importante, `SQLException`, es una especie de "chica para todo". La misma excepción se usa para propósitos tan distintos como: "no hay conexión con la base de datos", "el SQL de la consulta está mal formado" o "se ha producido una violación de la integridad de los datos". Esto hace que para el desarrollador sea tedioso escribir código que detecte adecuadamente el problema. Herramientas como Hibernate tienen una jerarquía de excepciones mucho más completa, pero son excepciones propias del API, y referenciarlas directamente va a introducir dependencias no deseadas en nuestro código.

- Las excepciones definidas en Java para acceso a datos son *comprobadas*. Esto implica que debemos poner `try/catch` o `throws` para gestionarlas, lo que inevitablemente llena *todos* los métodos de acceso a datos de bloques de gestión de excepciones. Está bien obligar al desarrollador a responsabilizarse de los errores, pero en acceso a datos esta gestión se vuelve repetitiva y propensa a fallos, descuidos o a caer en "malas tentaciones" (¿quién no ha escrito *nunca* un bloque `catch` vacío?). Además, muchos métodos de los DAO generalmente poco pueden hacer para recuperarse de la mayoría de excepciones (por ejemplo, "violación de la integridad"), lo que lleva al desarrollador a poner también `throws` de manera repetitiva y tediosa.

La solución de Spring al primer problema es la definición de una completa jerarquía de excepciones de acceso a datos. Cada problema tiene su excepción correspondiente, por ejemplo `DataAccessResourceFailureException` cuando no podemos conectar con la BD, `DataIntegrityViolationException` cuando se produce una violación de integridad en los datos, y así con otras muchas. Un aspecto fundamental es que estas excepciones *son independientes del API usado para acceder a los datos*, es decir, se generará el mismo `DataIntegrityViolationException` cuando queramos insertar un registro con clave primaria duplicada en JDBC que cuando queramos persistir un objeto con clave duplicada en JPA. La raíz de esta jerarquía de excepciones es `DataAccessException`.

En cuanto a la necesidad de gestionar las excepciones, Spring opta por eliminarla haciendo que todas las excepciones de acceso a datos sean *no comprobadas*. Esto libera al desarrollador de la carga de los `try-catch/throws` repetitivos, aunque evidentemente no lo libera de su responsabilidad, ya que las excepciones tendrán que gestionarse en algún nivel superior.

2. Uso de JDBC

JDBC sigue siendo un API muy usado para el acceso a datos, aunque es tedioso y repetitivo. Vamos a ver cómo soluciona Spring algunos problemas de JDBC, manteniendo las ventajas de poder trabajar "a bajo nivel" si así lo deseamos. Probablemente las ventajas quedarán más claras si primero vemos un ejemplo con JDBC "a secas" y luego vemos el mismo código usando las facilidades que nos da Spring. Por ejemplo, supongamos un método que comprueba que el login y el password de un usuario son correctos, buscándolo en la base de datos con JDBC:

```
private String SQL ="select * from usuarios where login=? and password=?";

public UsuarioTO login(String login, String password) throws DAOException
{
    Connection con=null;
    try {
        con = ds.getConnection();
           PreparedStatement ps = con.prepareStatement(SQL);
        ps.setString(1, login);
        ps.setString(2, password);
```

```
        ResultSet rs = ps.executeQuery();
        if (rs.next()) {
            UsuarioTO uto = new UsuarioTO();
            uto.setLogin(rs.getString("login"));
            uto.setPassword(rs.getString("password"));
            uto.setFechaNac(rs.getDate("fechaNac"));
            return uto;
        }
        else
            return null;
    } catch(SQLException sqle) {
        throw new DAOException(sqle);
    }
    finally {
        if (con!=null) {
            try {
                con.close();
            }
            catch(SQLException sqle2) {
                throw new DAOException(sqle2);
            }
        }
    }
}
```

Se destacan las líneas de código que hacen realmente el trabajo de buscar el registro y devolver la información. El resto es simplemente la infraestructura necesaria para poder hacer el trabajo y gestionar los errores, y que, curiosamente *ocupa más líneas que el código "importante"*. Evidentemente la gestión de errores se habría podido "simplificar" poniendo en la cabecera del método un `throws SQLException`, pero entonces ya estaríamos introduciendo dependencias del API JDBC en la capa de negocio.

Veamos cómo nos puede ayudar Spring a simplificar nuestro código, manteniendo la flexibilidad que nos da SQL. El primer paso será elegir el *template* apropiado.

2.1. JDBCTemplate

Como ya hemos dicho, los *templates* son clases que encapsulan el código de gestión de los detalles "tediosos" del API de acceso a datos. En Spring, para JDBC tenemos varios templates disponibles, según queramos hacer consultas simples, con parámetros con nombre,... Vamos a usar aquí `JdbcTemplate`, que aprovecha las ventajas de Java 5 (autoboxing, genéricos, ...) para simplificar las operaciones. El equivalente si no tenemos Java 5 sería `JdbcTemplate`, que tiene una sintaxis mucho más complicada.

Lo primero que necesitamos es instanciar el *template*. El constructor de `JdbcTemplate` necesita un `DataSource` como parámetro. Como se vio en el tema anterior, los `DataSource` se pueden definir en el fichero XML de los beans, gracias al espacio de nombres `jee`:

```
<jee:jndi-lookup id="ds" jndi-name="jdbc/MiDataSource"
resource-ref="true"/>
```

Con lo que el `DataSource` se convierte en un bean de Spring llamado `ds`. La práctica

habitual es inyectarlo en nuestro DAO y con él inicializar el *template*, que guardaremos en el DAO:

```
import org.springframework.jdbc.core.simple.JbcTemplate;
import org.springframework.stereotype.Repository;
//Resto de imports...
...

@Repository("JDBC")
public class UsuariosDAOJDBC implements IUsuariosDAO {
    private JdbcTemplate jdbcTemplate;

    @Autowired
    public void setDataSource(DataSource ds) {
        this.jdbcTemplate = new JdbcTemplate(ds);
    }

    ...
}
```

Recordemos que la anotación `@Repository` se usa para definir un DAO. Recordemos también que `@Autowired` inyecta la dependencia buscándola por tipo. En este caso no hay ambigüedad, ya que solo hemos definido un `DataSource`.

SimpleJdbcTemplate

Antes de Spring 3, la funcionalidad más "simple" de JDBC en Spring la implementaba `SimpleJdbcTemplate`, mientras que el API de `JdbcTemplate` era más complicado y por tanto solo recomendable para operaciones más complejas que las que vamos a ver aquí. A partir de la versión 3, `SimpleJdbcTemplate` está *deprecated*.

2.2. Consultas de selección

Normalmente en un SELECT se van recorriendo registros y nuestro DAO los va transformando en objetos Java que devolverá a la capa de negocio. En Spring, el trabajo de tomar los datos de un registro y empaquetarlos en un objeto lo hace `RowMapper`. Este es un interface, por lo que nuestro trabajo consistirá en escribir una clase que lo implemente. Realmente el único método estrictamente necesario es `mapRow`, que a partir de un registro debe devolver un objeto. En nuestro caso podría ser algo como:

```
//esto podría también ser private y estar dentro del DAO
//ya que solo lo necesitaremos dentro de él
public class UsuarioTOMapper implements RowMapper<UsuarioTO> {

    public UsuarioTO mapRow(ResultSet rs, int numRow) throws SQLException
{
        UsuarioTO uto = new UsuarioTO();
        uto.setLogin(rs.getString("login"));
        uto.setPassword(rs.getString("password"));
        uto.setFechaNac(rs.getDate("fechaNac"));
        return uto;
    }
}
```

Ahora solo nos queda escribir en el DAO el código que hace el SELECT:

```
private static final String LOGIN_SQL = "select * " +
 "from usuarios where login=? and password=?";

public UsuarioTO login(String login, String password) {
    UsuarioTOMapper miMapper = new UsuarioTOMapper();

    return this.jdbcTemplate.queryForObject(LOGIN_SQL, miMapper,
                                            login, password);
}
```

Como se ve, no hay que gestionar la conexión con la base de datos, preocuparse del `Statement` ni nada parecido. El *template* se ocupa de estos detalles. El método `queryForObject` hace el SELECT y devuelve un `UsuarioTO` ayudado del *mapper* que hemos definido antes. Simplemente hay que pasarle el SQL a ejecutar y los valores de los parámetros.

Tampoco hay gestión de excepciones, porque Spring captura todas las `SQLException` de JDBC y las transforma en excepciones no comprobadas. Por supuesto, eso no quiere decir que no podamos capturarlas en el DAO si así lo deseamos. De hecho, en el código anterior hemos cometido en realidad un "descuido", ya que podría no haber ningún registro como resultado del SELECT. Para Spring esto es una excepción del tipo `EmptyResultDataAccessException`. Si queremos seguir la misma lógica que en el ejemplo con JDBC, deberíamos devolver `null` en este caso.

```
private static final String LOGIN_SQL = "select * " +
        "from usuarios where login=? and password=?";

public UsuarioTO login(String login, String password) {
   UsuarioTOMapper miMapper = new UsuarioTOMapper();

   try {
       return this.jdbcTemplate.queryForObject(LOGIN_SQL, miMapper,
                                               login, password);
   }
   catch(EmptyResultDataAccessException erdae) {
       return null;
   }
}
```

La amplia variedad de excepciones de acceso a datos convierte a Spring en un *framework* un poco "quisquilloso" en ciertos aspectos. En un `queryForObject` Spring espera obtener *un registro y sólo un registro*, de modo que se lanza una excepción si no hay resultados, como hemos visto, pero también si hay más de uno: `IncorrectResultSizeDataAccessException`. Esto tiene su lógica, ya que `queryForObject` solo se debe usar cuando esperamos como máximo un registro. Si el SELECT pudiera devolver más de un resultado, en lugar de llamar a `queryForObject`, emplearíamos **`query`**, que usa los mismos parámetros, pero devuelve una lista de objetos.

2.3. Consultas de actualización

Las actualizaciones se hacen con el método `update` del *template*. Por ejemplo, aquí tenemos el código que da de alta a un nuevo usuario:

```
private static final String REGISTRAR_SQL =
        "insert into usuarios(login, password, fechaNac) values (?,?,?)";

public void registrar(UsuarioTO uto) {
   this.jdbcTemplate.update(REGISTRAR_SQL, uto.getLogin(),
             uto.getPassword(), uto.getFechaNac());
}
```

3. Uso de JPA

Veamos qué soporte ofrece Spring al otro API de persistencia que hemos visto durante el curso: JPA. El caso de JPA es muy distinto al de JDBC, ya que es de mucho más alto nivel y más conciso que este último. Por eso, aunque Spring implementa un *template* para JPA, también se puede usar directamente el API sin escribir mucho más código. Además otra decisión que debemos tomar es quién gestionará los Entity Managers, si lo haremos "manualmente", al estilo Java SE, o bien a través del servidor de aplicaciones.

3.1. Formas de acceder a la "EntityManagerFactory"

Para poder trabajar con JPA lo primero que necesitamos es una `EntityManagerFactory` a través de la que crear los *Entity Managers*. Spring nos da tres posibilidades para acceder a ella:

- Como se hace en Java SE. Es decir, Entity Managers gestionados por la aplicación. En este caso, la mayor parte de la configuración se hace en el persistence.xml. Esta forma no se recomienda en general, solo para pruebas.
- Usando el soporte del servidor de aplicaciones. Evidentemente esto no funcionará en servidores que no tengan soporte JPA, como Tomcat, pero es la forma recomendada de trabajo en los que sí lo tienen. El acceso a la factoría se hace con JNDI.
- Usando un soporte propio de Spring. Esto funciona incluso en servidores sin soporte "nativo", como Tomcat. La configuración se hace sobre todo en el fichero de configuración de beans de Spring. El problema es que depende del servidor y también de la implementación JPA que estemos usando, pero está explicada con detalle en la documentación de Spring para varios casos posibles (sección 7.8.4.5).

Estas dos últimas posibilidades serían lo que en JPA se conoce como Entity Managers gestionados por el contenedor, con la diferencia de que una de ellas es una implementación nativa y la otra proporcionada por Spring. En cualquier caso, en realidad como desarrolladores todo esto nos da casi igual: elijamos la opción que elijamos Spring va a gestionar la `EntityManagerFactory` por nosotros. Esta se implementa como un bean de Spring, que podemos inyectar en nuestro código usando la anotación estándar de `@PersistenceUnit`. También podemos inyectarlo con `@Autowired` o `@Resource`, como vimos en el tema anterior. Dicho bean debemos definirlo en el fichero XML de

configuración. Aquí tenemos un ejemplo usando JPA gestionado por el contenedor pero en Tomcat (es decir, gestionado en realidad por Spring)

```
<?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"
       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.xsd">

    <context:component-scan base-package="es.ua.jtech"/>

    <jee:jndi-lookup id="miDS" jndi-name="jdbc/MiDataSource"
       resource-ref="true"/>

    <bean id="miEMF" class="org.springframework.orm.jpa.
LocalContainerEntityManagerFactoryBean">
      <property name="dataSource" ref="miDS"/>
      <property name="jpaVendorAdapter">
         <bean class="org.springframework.orm.jpa.vendor.
                        HibernateJpaVendorAdapter">
               <property name="showSql" value="true"/>
               <property name="generateDdl" value="true"/>
               <property name="databasePlatform"
                          value="org.hibernate.dialect.HSQLDialect"/>
        </bean>
      </property>
    </bean>
</beans>
```

Como vemos el bean es una clase propia de Spring, de ahí que no podamos definirlo mediante anotaciones.El identificador del bean (miEMF) es arbitrario e irrelevante para nuestro ejemplo. Especificamos dos propiedades del bean: el datasource que se usará para conectar con la BD (aquí definido con JNDI) y la implementación JPA que estamos usando, en nuestro caso Hibernate. Finalmente, la etiqueta "component-scan" es necesaria para que Spring reconozca y procese las anotaciones `@PersistenceUnit` y `@PersistenceContext`.

Cuidado con la implementación JPA

En nuestro caso la configuración se simplifica porque usamos Hibernate, pero otras implementaciones JPA requieren *load time weaving*, una técnica que manipula el *bytecode* de las clases JPA en tiempo de ejecución y que es necesaria para que funcione la gestión por el contenedor. Si usáramos otra implementación JPA para la que fuera necesario el *load time weaving* habría que configurarlo, configuración que es además dependiente del servidor web en que hagamos el despliegue. Consultad la documentación de Spring para más información.

Con esto ya podemos inyectar el EntityManager o bien la factoría de EntityManagers en nuestro código:

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

```
//faltan los import...
@Repository("JPA")
public class UsuariosDAOJPA implements IUsuariosDAO {

    @PersistenceContext
    EntityManager em;

    public UsuarioTO login(String login, String password) {
        UsuarioTO uto = em.find(UsuarioTO.class, login);
        if (uto!=null && password.equals(uto.getPassword()))
            return uto;
        else
            return null;
    }
}
```

3.2. JpaTemplate

Esta clase facilita el trabajo con JPA, haciéndolo más sencillo. No obstante, al ser JPA un API relativamente conciso, no es de esperar que ahorremos mucho código. De hecho, los propios diseñadores de Spring recomiendan usar directamente el API JPA para aumentar la portabilidad del código. No obstante, vamos a verlo brevemente.

Aquí tenemos el esqueleto de una nueva implementación de `IUsuariosDAO` usando ahora `JpaTemplate` en lugar de `JdbcTemplate`:

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

import es.ua.jtech.spring.dominio.UsuarioTO;
import javax.persistence.EntityManagerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.orm.jpa.JpaTemplate;
import org.springframework.stereotype.Repository;

@Repository("JPATemplate")
public class UsuariosDAOJPATemplate implements IUsuariosDAO {
    private JpaTemplate template;

    @Autowired
    public void setEntityManagerFactory(EntityManagerFactory emf) {
        this.template = new JpaTemplate(emf);
    }

    //Falta la implementación de este método
    public UsuarioTO login(String login, String password) {
        ...
    }

    //Falta la implementación de este método
    public void registrar(UsuarioTO uto) {
        ...
    }
```

Como se ve, para instanciar un `JpaTemplate` necesitamos un `EntityManagerFactory` (de ahí el trabajo que nos habíamos tomado en la sección anterior). En este caso usamos la anotación `@Autowired` para que Spring resuelva automáticamente la dependencia y la

inyecte a través del *setter*. También podríamos haber usado `@PersistenceUnit`, que al ser estándar hace nuestro código más portable. Una vez creado el Template, se puede reutilizar en todos los métodos ya que es *thread-safe*, al igual que en JDBC.

Ahora veamos cómo implementar los métodos del DAO. En lugar de ver sistemáticamente el API de `JpaTemplate`, nos limitaremos a mostrar un par de ejemplos. Primero, una consulta de selección:

```
private static String LOGIN_JPAQL = "SELECT u FROM UsuarioTO u
                                     WHERE u.login=?1 AND u.password=?2";
public UsuarioTO login(String login, String password) {
   List<UsuarioTO> lista;
   lista = this.template.find(LOGIN_JPAQL, login, password);
   return (UsuarioTO) lista.get(0);
}
```

Obsérvese que en el ejemplo anterior **no es necesario instanciar ni cerrar ningún Entity Manager**, ya que la gestión la lleva a cabo el *template*. En cuanto al API de acceso a datos, como se ve, el método `find` nos devuelve una lista de resultados y nos permite pasar parámetros a JPAQL gracias a los *varargs* de Java 5. En el ejemplo anterior hemos "forzado un poco" el API de `JpaTemplate` ya que `find` devuelve siempre una lista y en nuestro caso está claro que no va a haber más de un objeto como resultado. Se ha hecho así para mantener el paralelismo con el ejemplo JDBC, aunque aquí quizá lo más natural sería buscar por clave primaria. Tampoco se han tratado adecuadamente los errores, como que no haya ningún resultado en la lista.

Otros métodos de `JpaTemplate` nos permiten trabajar con parámetros con nombre y con *named queries*.

Las actualizaciones de datos no ofrecen gran ventaja con respecto al API directo de JPA, salvo la gestión automática del Entity Manager, por ejemplo:

```
public void registrar(UsuarioTO uto) {
   this.template.persist(uto);
}
```

¡Cuidado con la transaccionalidad!

Spring gestiona automáticamente los Entity Manager en función de la transaccionalidad de los métodos. Si no declaramos ningún tipo de transaccionalidad, como en el ejemplo anterior, podemos encontrarnos con que al hacer un `persist` no se hace el `commit` y el cambio no tiene efecto. En el último apartado del tema veremos cómo especificar la transaccionalidad, pero repetimos que es importante darse cuenta de que *si no especificamos transaccionalidad, la gestión automática de los entity manager no funcionará adecuadamente*.

Una ventaja del uso de *templates* es la jerarquía de excepciones de Spring. En caso de usar directamente el API JPA nos llegarán las propias de la implementación JPA que estemos usando. Si queremos que Spring capture dichas excepciones y las transforme en excepciones de Spring, debemos definir un bean de la clase

`PersistenceExceptionTranslationPostProcessor` en el XML de definición de beans:

```
<bean class="org.springframework.dao. <!--Esto debería estar en 1 sola
línea-->
annotation.PersistenceExceptionTranslationPostProcessor"/>
```

Como se ve, lo más complicado de la definición anterior ¡es el nombre de la clase!

4. Transaccionalidad declarativa

Abordamos aquí la transaccionalidad porque es un aspecto íntimamente ligado al acceso a datos, aunque se suele gestionar desde la capa de negocio en lugar de directamente en los DAOs. Vamos a ver qué facilidades nos da Spring para controlar la transaccionalidad de forma declarativa.

4.1. El "Transaction Manager"

Lo primero es escoger la estrategia de gestión de transacciones: si estamos en un servidor de aplicaciones podemos usar JTA, pero si no, tendremos que recurrir a la implementación nativa del API de acceso a datos que estemos usando. Spring implementa clases propias para trabajar con JTA o bien con transacciones JDBC o JPA. Estas clases gestionan las transacciones y se denominan "Transaction Managers". Necesitaremos definir uno de ellos en el fichero XML de definición de beans.

Por ejemplo, en el caso de estar usando JDBC sin soporte JTA, como en Tomcat, necesitaríamos una implementación de Transaction Manager que referencia un DataSource.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:jee="http://www.springframework.org/schema/jee"
       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.xsd
        http://www.springframework.org/schema/jee
        http://www.springframework.org/schema/jee/spring-jee.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd
        http://www.springframework.org/schema/tx
        http://www.springframework.org/schema/tx/spring-tx.xsd">


    <!-- el DataSource que usará el Transaction Manager -->
    <jee:jndi-lookup id="miDataSource" jndi-name="DStestSpring"
resource-ref="true" />

    <!-- Elegimos el tipo apropiado de "Transaction Manager" (JDBC) -->
    <bean id="miTxManager"
```

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

    <!-- Decimos que para este Transaction Manager vamos a usar
anotaciones -->
    <tx:annotation-driven transaction-manager="miTxManager"/>
 </beans>
```

4.2. La anotación @Transactional

Para entender mejor el uso de esta anotación, vamos a plantear un ejemplo. Supongamos que al registrar un nuevo usuario, además de introducir sus datos en la tabla de usuarios, también lo debemos dar de alta en otra tabla para enviarle publicidad, pero si no es posible este alta de publicidad por lo que sea (dirección de email no válida, digamos) hay que anular toda la operación. Supongamos que al detectar el error nuestro DAO lanzaría un `DataAccessException` que, recordemos, es la raíz de la jerarquía de excepciones de acceso a datos en Spring.

Colocaremos `@Transactional` delante de los métodos que queramos hacer transaccionales. Si la colocamos delante de una clase, estamos diciendo que **todos** sus métodos deben ser transaccionales. En nuestro caso:

```
//Faltan los imports, etc.
...
@Service
public class GestorUsuarios {
  @Autowired
  @Qualifier("JPA")
  private IUsuariosDAO udao;

  public void setUdao(IUsuariosDAO udao) {
      this.udao = udao;
  }

  @Transactional
  public void registrar(UsuarioTO uto) {
      udao.registrar(uto);
      udao.altaPublicidad(uto);
  }
}
```

El comportamiento por defecto de `@Transactional` es **realizar un *rollback* si se ha lanzado alguna excepción no comprobada**. Recordemos que, precisamente, `DataAccessException` era de ese tipo. Por tanto, se hará automáticamente un *rollback* en caso de error.

¡Cuidado con los métodos no públicos!

Todos los métodos que deseamos hacer transaccionales deben ser públicos, no es posible usar `@Transactional` en métodos `protected` o `private`. La razón es que cuando hacemos un método transaccional y lo llamamos desde cualquier otra clase quien está recibiendo la llamada en realidad es el gestor de transacciones. El gestor comienza y acaba las transacciones y "entre medias" llama a nuestro método de acceso a datos, pero eso no lo podrá hacer si este no es

`public`. Por la misma razón, la anotación no funcionará si el método transaccional es llamado desde la misma clase que lo define, aunque esto último se puede solucionar haciendo la configuración adecuada.

`@Transactional` tiene varias implicaciones por defecto (aunque son totalmente configurables, como ahora veremos):

- La propagación de la transacción es REQUIRED. Esto significa que se requiere una transacción abierta para que el método se ejecute en ella. Si no hubiera ninguna, Spring automáticamente la crearía. Esto funciona igual que los EJBs del JavaEE estándar.

Transaccionalidad declarativa en JavaEE

En el bloque de Aplicaciones Enterprise se verá con mucho más detalle todo el tema de la transaccionalidad declarativa, y los diferentes tipos de gestión de la transacción típicos en JavaEE. Por el momento nos limitaremos a casos relativamente simples.

- Cualquier excepción no comprobada dispara el *rollback* y cualquiera comprobada, no.
- La transacción es de lectura/escritura (en algunos APIs de acceso a datos, por ejemplo, Hibernate, las transacciones de "solo lectura" son mucho más eficientes).
- El *timeout* para efectuar la operación antes de que se haga *rollback* es el que tenga por defecto el API usado (no todos soportan *timeout*).

Todo este comportamiento se puede configurar a través de los atributos de la anotación, como se muestra en la siguiente tabla:

Propiedad	Tipo	Significado
propagation	enum: Propagation	nivel de propagación (opcional)
isolation	enum: Isolation	nivel de aislamiento (opcional)
readOnly	boolean	solo de lectura vs. de lectura/escritura
timeOut	int (segundos)	
rollbackFor	array de objetos Throwable	clases de excepción que deben causar rollback
rollbackForClassName	array con nombres de objetos Throwable	nombres de clases de excepción que deben causar rollback
noRollbackFor	array de objetos Throwable	clases de excepción que no deben causar rollback
noRollbackForClassName	array con nombres de objetos Throwable	nombres de clases de excepción que no deben causar rollback

Por ejemplo supongamos que al producirse un error en la llamada a `altaPublicidad()` lo que se genera es una excepción propia de tipo `AltaPublicidadException`, que es comprobada pero queremos que cause un *rollback*:

```
@Transactional(rollbackFor=AltaPublicidadException.class)
public void registrar(UsuarioTO uto) {
      udao.registrar(uto);
      udao.altaPublicidad(uto);
}
```

Finalmente, destacar que podemos poner transaccionalidad "global" a una clase y en cada uno de los métodos especificar atributos distintos:

```
//Faltan los imports, etc.
...
@Service
//Vale, el REQUIRED no haría falta ponerlo porque es la opción
//por defecto, pero esto es solo un humilde ejemplo!
@Transactional(propagation=Propagation.REQUIRED)
public class GestorUsuarios {
  @Autowired
  @Qualifier("JPA")
  private IUsuariosDAO udao;

  public void setUdao(IUsuariosDAO udao) {
      this.udao = udao;
  }

  @Transactional(readOnly=true)
  public UsuarioTO login(String login, String password) {
      return udao.login(login, password);
  }

  @Transactional(rollbackFor=AltaPublicidadException.class)
  public void registrar(UsuarioTO uto) {
      udao.registrar(uto);
      udao.altaPublicidad(uto);
  }

  public void eliminarUsuario(UsuarioTO uto) {
       udao.eliminar(uto);
  }
}
```

Nótese que `eliminarUsuario` es también transaccional, heredando las propiedades transaccionales de la clase.

4.3. Transaccionalidad y uso directo de JDBC

Si usamos el API JDBC directamente, sin ninguna de las facilidades que nos da Spring, se nos va a plantear un problema con la transaccionalidad declarativa: si cada método del DAO abre y cierra una conexión con la BD (lo más habitual), va a ser imposible hacer un rollback de las operaciones que hagan distintos métodos, ya que la conexión ya se habrá cerrado. Para evitar este problema, Spring nos proporciona la clase `DataSourceUtils`,

que nos permite "liberar" la conexión desde nuestro punto de vista, pero mantenerla abierta automáticamente gracias a Spring, hasta que se cierre la transacción y no sea necesaria más. Su uso es muy sencillo. Cada vez que queremos obtener una conexión hacemos:

```
//El DataSource se habría resuelto por inyección de dependencias
@Autowired
private DataSource ds;

...
Connection con = DataSourceUtils.getConnection(ds);
```

Y cada vez que queremos liberarla:

```
DataSourceUtils.releaseConnection(con, ds);
```

Nótese que al liberar la conexión no se puede generar una SQLException, al contrario de lo que pasa cuando cerramos con el `close` de JDBC, lo que al menos nos ahorra un `catch`.