Acceso a datos

Índice

1 La filosofía del acceso a datos en Spring..... 2
2 Uso de JDBC..... 3
2.1 SimpleJDBCTemplate..... 4
2.2 Consultas de selección..... 5
2.3 Consultas de actualización..... 6
3 Uso de JPA..... 7
3.1 Configurar la factoría de Entity Managers..... 7
3.2 JpaTemplate..... 9
3.3 Uso directo del API JPA..... 11
4 Transaccionalidad declarativa..... 12
4.1 El "Transaction Manager"..... 12
4.2 La anotación @Transactional..... 14
4.3 Transaccionalidad y uso directo de JDBC..... 17

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

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);
            }
        }
    }
}
```

En negrita se muestran 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. SimpleJDBCTemplate

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 `SimpleJdbcTemplate`, 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 `SimpleJdbcTemplate` 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.SimpleJdbcTemplate;
import org.springframework.stereotype.Repository;
//Resto de imports...
...

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

    @Autowired
    public void setDataSource(DataSource ds) {
        this.jdbcTemplate = new SimpleJdbcTemplate(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`.

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 `ParameterizedRowMapper` (o `RowMapper` si no estamos usando Java 5). Estos son interfaces, por lo que nuestro trabajo consistirá en escribir una clase que los 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
 protected class UsuarioTOMapper implements
ParameterizedRowMapper<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());
}
```

`SimpleJdbcTemplate` carece de métodos más "avanzados" que impliquen por ejemplo el uso de procedimientos almacenados, de parámetros con nombres, etc. No obstante, "esconde" la referencia a otros templates que tienen todas estas posibilidades, y que pueden obtenerse llamando a los métodos `getJdbcOperations()` y `getNamedParameterJdbcOperations()`. Se recomienda consultar la documentación del API de Spring para más información sobre estas opciones "avanzadas".

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. Vamos a ver las posibilidades que ofrecen ambas opciones.

3.1. Configurar la factoría de Entity Managers

Independientemente de si vamos a usar *templates* o directamente el API JPA, lo primero es configurar la factoría de Entity Managers. Como ya sabemos, dicha factoría la gestiona la propia aplicación en el caso de aplicaciones "de escritorio", y el servidor de aplicaciones en el caso de aplicaciones *enterprise*. Si no tenemos o queremos usar servidor de aplicaciones, podemos usar JPA en un contenedor web como Tomcat gracias a la infraestructura proporcionada por Spring. Aquí tenemos los pasos necesarios para configurar el soporte JPA en versiones de Tomcat superiores a la 5.5.20:

1. Por supuesto, lo primero es asegurarnos de que en el proyecto web hemos incluido las librerías JPA necesarias, por ejemplo la implementación JPA de Hibernate.
2. Supondremos que tenemos solo una unidad de persistencia (en la documentación de Spring se explica cómo hacer la configuración si hay varias). Necesitaremos el `persistence.xml` con la configuración, en nuestro caso:

```
<?xml version="1.0" encoding="UTF-8"?>
<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
http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd">
  <persistence-unit name="TestSpring25"
transaction-type="RESOURCE_LOCAL">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <non-jta-data-source/>
    <class>es.ua.jtech.spring.dominio.UsuarioTO</class>
    <exclude-unlisted-classes>true</exclude-unlisted-classes>
    <properties/>
  </persistence-unit>
</persistence>
```

Como vemos en el listado anterior, usamos Hibernate como implementación JPA y no usamos JTA para la transaccionalidad, ya que Tomcat actualmente no tiene soporte JTA.

3. En JPA las clases están instrumentadas en tiempo de ejecución (lo que en inglés se denomina *class weaving*). Tomcat no ofrece soporte directo para ello, por lo que debemos hacer dos cosas:
4. Cambiar el *classloader* usado por Tomcat. Lo habitual es definirlo en el fichero `context.xml` propio de la aplicación. (o en el `server.xml`, global). Si no tienes muy claro qué son estos ficheros, mejor echarle un vistazo a la documentación de Tomcat.

```
<?xml version="1.0" encoding="UTF-8"?>
<Context path="/test">
    <Loader
loaderClass="org.springframework.instrument.classloading.tomcat.TomcatInstrumentableC
            useSystemClassLoaderAsParent="false"/>
    <!-- aquí vendrían más cosas, por ejemplo podría venir el
DataSource
         que usará JPA -->
    ...
</Context>
```

5. Copiar en el directorio `lib` de Tomcat la librería `spring-tomcat-weaver.jar`, incluida en la distribución de Spring.
6. Y por fin, ya podemos configurar la factoría en el fichero XML de definición de beans. Para ello hay que definir un *bean* de la clase `LocalContainerEntityManagerFactoryBean`:

```
<?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-2.5.xsd
        http://www.springframework.org/schema/jee
http://www.springframework.org/schema/jee/spring-jee-2.5.xsd
        http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-2.5.xsd">
```

```
    <!-- Aquí definimos el DataSource, usando JNDI -->
    <jee:jndi-lookup id="miDataSource" jndi-name="DStestSpring"
resource-ref="true" />

    <!-- Le decimos a Spring que autodetecte el servidor
         e inicialice adecuadamente el "weaving" -->
    <context:load-time-weaver/>

    <!-- ¡Por fin! esta es la dichosa factoría. Necesita una
referencia al Datasource -->
    <bean id="emf"
class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
        <property name="dataSource" ref="miDataSource"/>
    </bean>

 </beans>
```

Los pasos son distintos (y más sencillos) si usamos un servidor de aplicaciones. Consultar la documentación de Spring para la configuración en este caso.

Aviso:

Hay ciertas cuestiones dependientes de la implementación JPA. Por ejemplo, en caso de usar Hibernate necesitaremos copiar al `lib` de Tomcat la librería `jboss-archive-browsing.jar`, incluida en la carpeta `hibernate` de la distribución de Spring. En caso de usar Toplink, habrá que hacer lo propio con la librería `toplink-essentials.jar`.

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. Vamos a verlo siguiendo con el ejemplo anterior. Aquí tenemos el esqueleto de una nueva implementación de `IUsuariosDAO` usando ahora `JpaTemplate` en lugar de `SimpleJdbcTemplate`:

```
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). Como siempre, usamos la anotación `@Autowired` para que Spring resuelva automáticamente la dependencia y la inyecte en este caso a través del *setter*. 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);
}
```

Aviso:

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*.

3.3. Uso directo del API JPA

Usando directamente el API JPA tenemos la ventaja de que nuestro código no dependerá en absoluto de los APIs de Spring, haciéndolo mucho más portable. De hecho, los propios desarrolladores de Spring valoran más esta portabilidad que la capa de abstracción que ofrece `JpaTemplate` y recomiendan usar directamente el API JPA.

De cualquier modo, las llamadas al API JPA siguen necesitando de un Entity Manager. ¿Cómo lo obtenemos en Spring?. Afortunadamente Spring soporta las anotaciones estándar `@PersistenceUnit` y `@PersistenceContext`, que como hemos visto en el módulo de EJB inyectan en nuestro código un `EntityManagerFactory` y un `EntityManager`, respectivamente. La elección de uno u otro dependerá de si queremos gestionar manualmente los Entity Manager o vamos a usar la transaccionalidad declarativa de Spring y por tanto los puede gestionar este de manera automática. Como por el momento no hemos tratado la transaccionalidad declarativa, veamos cómo inyectaríamos la factoría de Entity Managers y gestionaríamos estos manualmente:

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

import es.ua.jtech.spring.dominio.UsuarioTO;
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.PersistenceUnit;
import javax.persistence.Query;
import org.springframework.stereotype.Repository;
import java.util.List;

@Repository("JPA")
public class UsuariosDAOJPA  implements IUsuariosDAO  {
    @PersistenceUnit
    private EntityManagerFactory factoria;


    public UsuarioTO login(String login, String password) {
        EntityManager em = null;
        try {
            em = factoria.createEntityManager();
            Query q = em.createQuery("SELECT u FROM UsuarioTO u
```

```
WHERE" +
                                        "u.login=:login AND
u.password=:password");
            q.setParameter("login", login);
            q.setParameter("password", password);
            return (UsuarioTO) q.getSingleResult();
        }
        finally {
            if (em!=null)
                em.close();
        }
    }


    public void registrar(UsuarioTO uto) {
        EntityManager em=null;

        try {
            em = factoria.createEntityManager();
            em.getTransaction().begin();
            em.persist(uto);
            em.getTransaction().commit();
        }
        finally {
            if (em!=null)
                em.close();
        }
    }

}
```

Hay que tener en cuenta que al no usar *templates* perdemos la jerarquía de excepciones de Spring, y 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.annotation.PersistenceExceptionTranslationPostProcesso
```

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. Continuaremos aquí con la última versión del ejemplo e intentaremos usar JPA dentro de Tomcat. La configuración del XML con los beans sería algo como:

```
<?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-2.5.xsd
        http://www.springframework.org/schema/jee
http://www.springframework.org/schema/jee/spring-jee-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">


    <jee:jndi-lookup id="miDataSource" jndi-name="DStestSpring"
resource-ref="true" />

    <context:load-time-weaver/>

    <bean id="emf"
class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
        <property name="dataSource" ref="miDataSource"/>
    </bean>

    <!-- Elegimos el tipo apropiado de "Transaction Manager" (en
nuestro caso, JPA) -->
    <bean id="miTxManager"
class="org.springframework.orm.jpa.JpaTransactionManager">
        <property name="entityManagerFactory" ref="emf"/>
    </bean>

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

Donde como siempre, en negrita se destaca lo nuevo introducido en el fichero. En Spring, el que gestiona las transacciones es el "Transaction Manager", del que hay varias implementaciones. Nosotros hemos escogido la JPA. En el caso particular de JPA el Transaction Manager debe recibir una referencia al Entity Manager Factory. Con la última línea de XML decimos que vamos a usar anotaciones para definir transaccionalidad en lugar de XML.

En el caso de estar usando JDBC, necesitaríamos una implementación distinta de

Transaction Manager que no referencia un Entity Manager Factory sino 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-2.5.xsd
        http://www.springframework.org/schema/jee
http://www.springframework.org/schema/jee/spring-jee-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">


    <!-- no necesitamos weaving ni Entity Manager Factory
           para JDBC, pero sí DataSource -->
    <jee:jndi-lookup id="miDataSource" jndi-name="DStestSpring"
resource-ref="true" />

    <!-- Elegimos el tipo apropiado de "Transaction Manager" (ahora
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 lo debemos dar de alta en una tabla aparte 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.

Aviso:

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 tiene el mismo significado que en EJBs.
- 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`.