Spring JPA

Victor Custodio

¿Por qué usar Spring para acceso a datos con JDBC?

- **Simplifica el código** de los DAOs en APIs tediosos como JDBC
    - Gracias a los templates
    - Permite usar también directamente el API si lo preferimos
- Ofrece una **rica jerarquía de excepciones** independiente del API de acceso a datos

¿Por qué usar Spring para acceso a datos con JPA?

- **En JPA el beneficio ya no es tan significativo, pero lo hay:**
  - **Gestión de transacciones**
  - **Inyección del EntityManager**

JPA: opciones de configuración

- JPA gestionado por la aplicación

Usado hasta ahora, en persistence.xml, sólo par desarrollo.

- JPA gestionado por el contenedor

No disponible en todos los servidore

  - Directamente, para servidores de aplicaciones, que tendrán soporte "nativo". Se accede a la factoría por JNDI
  - También es posible para servidores web, como Tomcat, el soporte lo proporciona Spring y se realiza en el archivo de configuración de Spring.

Con Spring usaremos ésta

DataSource

- El datasource es una clase Java que encapsula la información necesaria para conectarse a una fuente de datos.
- Al usar la capa de acceso a datos de Spring, podemos configurar el DataSource como una bean más o obtenerlo por JNDI.
- Spring proporciona la implementación DriverManagerDataSource, donde se debe configurar:
  - Driver class name: **driverClassName**
  - URL de conexión: **url**
  - Usuario: **username**
  - Password: **password**

- Existen otras implementaciones de un datasource, con su configuración asociada.

Ejemplo de DataSource

Driver Manager Data Source:

```
<bean id="dataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName" value="com.mysql.jdbc.Driver" />
    <property name="url" value="jdbc:mysql://localhost:3306/test" />
    <property name="username" value="root" />
    <property name="password" value="root" />
</bean>
```

Ejemplo de DataSource

Basic Data Source con esquema p:

```
<bean id="miDS"
class="org.apache.commons.dbcp.BasicDataSource"
p:driverClassName="com.mysql.jdbc.Driver"
p:url="jdbc:mysql:///test"
p:username="root"
p:password="root" />
```

Además, necesitamos un bean que "fabrique" EMs

- Al igual que en el datasource existen varias implementaciones que nos fabrican Ems .

```
<bean id="myEmf"
class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
    <property name="dataSource" ref="dataSource" />
    <property name="packagesToScan" value="modelo.entidades" />
    <property name="jpaVendorAdapter">
      <bean class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter" />
    </property>
    <property name="jpaProperties">
      <props>
        <prop key="hibernate.hbm2ddl.auto">update</prop>
        <prop key="hibernate.dialect">org.hibernate.dialect.MySQL5Dialect</prop>
      </props>
    </property>
  </bean
```

Uso del EM

```
@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;
}
}
```

Transaccionalidad declarativa

- Normalmente se gestiona desde la capa de negocio, aunque está íntimamente ligada al acceso a datos
- Lo primero que necesitamos en Spring es un "Transaction Manager". Hay varias implementaciones, dependiendo del API usado por los DAOs.
- Para JPA:

```
<bean id="transactionManager"
class="org.springframework.orm.jpa.JpaTransactionManager">
  <property name="entityManagerFactory" ref="myEmf" />
</bean>
```

La anotación @Transactional

- Colocada delante de un método, lo hace transaccional. Delante de la clase hace que TODOS los métodos lo sean
- El comportamiento por defecto es **rollback automático** ante excepción no comprobada.
- Para poder usar la anotación es necesario añadir en el archivo xml:
- **<tx:annotation-driven transaction- manager="miTxManager" />**

La anotación @transactional

```
@Service
public class GestorUsuarios {
@Autowired
private UsuariosDAO udao;
@Transactional
public void registrar(UsuarioTO uto) {
udao.registrarEnBD(uto);
udao.registrarEnListasDeCorreo(uto);
}
```

Configurar @Transactional

- Admite una serie de atributos
- **propagation**: en general, todo código que se ejecuta dentro de un ámbito de transacción se ejecutará en esa transacción. Sin embargo, hay varias opciones que especifican el comportamiento si se ejecuta un método de transacción cuando el contexto de la transacción ya existe: por ejemplo, sólo tiene que seguir ejecutándose en la operación existente (el caso más común), o la suspensión de la operación existente y la creación de una nueva transacción.
- **timeout**: ¿cuánto tiempo tiene la transacción para ejecutarse antes de deshacer la transacción subyacente).
- **read-only**: Una transacción de lectura única no puede modificar ningún dato. Las transacciones de sólo lectura pueden ser una optimización de utilidad en algunos casos (como cuando se utiliza Hibernate)..

Librerías necesarias

- Driver de la base de datos.
- Librerías de hibernate-entitymanager.
- Spring-orm
- Spring-context