SPRING (Acceso a Datos)

VÍCTOR CUSTODIO

Beneficios de Spring en el acceso a datos

Spring proporciona en el acceso a datos las siguientes facilidades:

- Gestión de la transaccionalidad.
- Una jerarquía consistente de excepciones, que oculta los detalles de las excepciones de la tecnología subyacente, JDBC o del framework O/R empleado en su caso.
- Clases abstractas para el soporte de DAOs: Facilitan el uso de DAOs para todas las tecnologías de acceso a datos soportadas, como JDBC plano, Hibernate, JDO y JPA.

Spring y JDBC

Lo que mejor define el valor añadido de Spring en su uso con JDBC, es mostrar los pasos que siempre debemos hacer en esta API:

- 1. Definir parámetros de conexión
- 2. Abrir la conexión
- **3. Especificar la sentencia SQL**
- 4. Preparar y ejecutar dicha sentencia
- 5. Establecer el bucle para recorrer los resultados.
- **6. Hacer el trabajo por cada iteración.**
- 7. Procesar y manejar las excepciones.
- 8. Cerrar la conexión

Aparecen en negrita las que son verdaderamente tarea del desarrollador. Spring nos permite centrarnos en estos 2 pasos

Spring y JDBC

El framework de acceso a datos con JDBC, esta estructurado en 4 paquetes:

- org.springframework.jdbc.**core**: Contiene JdbcTemplate
- org.springframework.jdbc.**datasource**: Contiene una clase de utilidad para el acceso a DataSources, con JNDI y varias implementaciones sencillas de DataSource
- org.springframework.jdbc.**object**: Contiene clases que representan queries, updates y procedimientos almacenados como objetos reutilizables y thread-safe. Esta aproximación se modela por JDO, aunque sin que los objetos esten “conectados” al BBDD.
- org.springframework.jdbc.**support**. Utilidades y Excepciones.

Core.JdbcTemplate

Maneja la creación y liberación de recursos:

- Evita olvidar cerrar la conexión.
- Ejecuta el workflow JDBC, como creación y ejecución de un statement, dejando al código de aplicación únicamente el proporcionar el SQL y la extracción de datos. **Para ello encapsula un DataSource.**
- Ejecuta queries SQL, update Statements o procedimientos almacenados, encapsulando la iteración sobre el resultset y la extracción de los valores de parámetros retornados.
- Captura las excepciones de JDBC, y las transforma en las genéricas y más informativas de Spring.

Core.JdbcTemplate

Creación de un JdbcTemplate

```
JdbcTemplate jdbcTemplate = new JdbcTemplate(miDataSource);
```

Ejecución de sentencias

Los siguientes ejemplos ilustran su uso:

```
int rowCount = this.jdbcTemplate.queryForInt("select count(0) from t_accrual");

int countOfActorsNamedJoe

= this.jdbcTemplate.queryForInt("select count(0) from t_actors where first_name = ?",
new Object[]{"Joe"});

String surname = (String) this.jdbcTemplate

.queryForObject("select surname from t_actor where id = ?", new Object[]{new
Long(1212)}, String.class);
```

Core.JdbcTemplate

Ejecución de sentencias con código de manejo de datos

```
public Usuario findUsuario(Integer id) {
return (Usuario) jdbcTemplate.queryForObject("select * from usuarios where
id="+id, new UsuarioMapper());}
}

public static final class UsuarioMapper implements RowMapper<Usuario>{
public Usuario mapRow(ResultSet rs, int count) throws SQLException {
Usuario u= new Usuario();
u.setId(rs.getString("id"));
u.setNombre(rs.getString("name"));
u.setApellidos(rs.getString("apellidos"));
return u;
}
}
}
```

Core.JdbcTemplate

Si los nombres de las propiedades y las columnas coinciden podemos usar una clase Mapeadora por defecto como es BeanPropertyRowMapper(<T>.class)

```
public List<Usuario> findUsuarios() {
try{
return (List<Usuario>) getJdbcTemplate().query("select * from usuario",
new BeanPropertyRowMapper(Usuario.class));
}catch(EmptyResultDataAccessException e){
return null;
}
}
```

Core.JdbcTemplate

Ejecución de sentencias de modificación

```
this.jdbcTemplate.update("insert into t_actor (first_name, surname)
values (?, ?)", new Object[] {"Leonor", "Watling"});

this.jdbcTemplate.update("update t_actor set weapon = ? where id = ?",
new Object[] {"Banjo", new Long(5276)});

this.jdbcTemplate.update("delete from orders");
```

Otras operaciones (DDL) Se usa el método execute

```
this.jdbcTemplate.execute("create table mytable (id integer, name
varchar(100))");
```

Core.JdbcTemplate

Existen buenas prácticas en el uso de JdbcTemplate:

Debemos tener en cuenta que una vez configurado, la instancia de una clase JdbcTemplate es thread-safe, es decir, aunque internamente maneje un DataSource, el contenedor es capaz de inyectar una referencia compartida en múltiples DAOs, pero sin presentar un estado conversacional.

Una buena práctica es tener el dataSource que usemos configurado como bean, y tenerlo como dependencia en las DAOs. Pero en vez de usar internamente el datasource, en el setter crearemos un jdbcTemplate, a partir de él:

```
public void setDataSource(DataSource dataSource) {

this.jdbcTemplate = new JdbcTemplate(dataSource);

}
```

Core.JdbcTemplate

NamedParametersJdbcTemplate.

Clase que amplia JdbcTemplate para ofrecer la posibilidad de crear y ejecutar queries con parámetros nombrados, usando la notacion :nombre en la sql, en vez de ?, como por ejemplo en el lenguaje HQL de Hibernate.

```
private NamedParameterJdbcTemplate namedParameterJdbcTemplate;

public void setDataSource(DataSource dataSource) {
this.namedParameterJdbcTemplate = new
NamedParameterJdbcTemplate(dataSource);
}
public int countOfActorsByFirstName(String firstName) {
String sql = "select count(0) from T_ACTOR where first_name = :first_name";
SqlParameterSource namedParameters = new MapSqlParameterSource("first_name", firstName);
return namedParameterJdbcTemplate.queryForInt(sql,namedParameters);
return u;
}
```

Core.JdbcTemplate

Se observa el uso de SqlParameterSource, para indicar el mapeo de los parámetros por nombre.

También podemos usar **BeanPropertySqlParameterSource**, que toma los campos de una bean por nombre, para hacer el mapeo a los valores de esos campos:

Ej: Dada una clase Actor, con nombre y apellido, podríamos hacer:

```
public int countOfActors(Actor exampleActor) {
String sql = "select count(0) from T_ACTOR where first_name= :nombre and last_name = :apellidos";
SqlParameterSource namedParameters = new BeanPropertySqlParameterSource(exampleActor);
return this.namedParameterJdbcTemplate.queryForInt(sql,namedParameters);
}
```

DataSource

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.

DataSource, con DriverManagerDataSource

```
<bean id="miDS2"
class="org.springframework.jdbc.datasource.DriverManagerDataSource">
<property name="driverClassName“ value="com.mysql.jdbc.Driver”/>
<property name="url" value=“jdbc:mysql:///test/>
<property name="username" value=“root”/>
<property name="password" value=“root”/>
</bean>
```

DataSource, con schema p

```
<!--  Schema p Otra Forma de configurar beans-->
<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" />

<bean id="miDaoJDBCTemplate"
class="accesoADatos.UsuarioDaoJDBCTemplate">
 <property name="dataSource">
<ref local="miDS" />
</property>
```

DataSource

Existen otras implementaciones de DataSource, como:

SingleConnectionDataSource: Encapsula una conexión que no se cierra tras usarla. No apta para entorno multi-thread

TransactionAwareDataSource: Encapsula un datasource añadiéndole capacidades de integrarse con la gestión transaccional de Spring

Core.SQLExceptionTranslator

Esta interfaz proporciona los métodos de conversión de las SQLExceptions, en las excepciones manejo de datos de Spring.

La implementación por defecto es SQLErrorCodesSQLExceptionTranslator. Usa los códigos de error propios de cada fabricante.

Las traducciones se basan en los códigos contenidos en una JavaBean llamada SQLErrorCodes. Esta clase es creada y cargada por SQLErrorCodesFactory, que los crea en función de un fichero de configuración llamado sql-error-codes.xml

Modelado de Operaciones

Spring permite modelar operaciones como objetos Java. Para ello tenemos:

SqlQuery, clase abstracta thread-safe que encapsula una query. Sus subclases deben implementar el método newRowMapper(..), para proporcionar una instancia de RowMapper, que pueda crear una instancia de objeto por fila.

MappingSqlQuery: subclase de SqlQuery. Sus subclases debe implementar mapRow(..), para convertir cada fila del ResultSet en un objeto.

SqlUpdate: Encapsula una modificación SQL. Proporciona métodos update análogos a los métodos execute(..) de los objetos query

SqlFunction: Clase que encapsula una query que sólo devuelve un resultado, normalmente un int, como al ejecutar un select count(*).