

Desarrollo de Aplicaciones Empresariales con Spring Framework Core 5

ISC. Ivan Venor García Baños



Agenda

1. Presentación
2. Objetivos
3. Contenido
4. Despedida

3. Contenido

- i. Introducción a Spring Framework
- ii. Spring Core
- iii. Spring AOP
- iv. Spring JDBC – Transaction**
- v. Spring ORM – Hibernate 5
- vi. Spring Data JPA
- vii. Fundamentos Spring MVC y Spring REST
- viii. Fundamentos Spring Security
- ix. Seguridad en Servicios REST
- x. Introducción Spring Boot

iv. Spring JDBC - Transaction

iv. Spring JDBC - Transaction (a)

iv.i API JDBC

a. Implementación API JDBC

iv.ii Spring JDBC

a. DAO Support

iv.iii Acceso a Datos con Spring JDBC

a. Templates

b. JdbcDaoSupport

c. Data Sources

d. Soporte para base de datos embebida

Práctica 25 – Parte 1. Configuración Spring JDBC

iv. Spring JDBC - Transaction (b)

iv.iv JdbcTemplate

- a. Select query
- b. Update query (insert, update y delete)
- c. Execute query

Práctica 25 – Parte 2. Implementación IUserDAO

iv.v NamedParameterJdbcTemplate

- a. Select query
- b. Update query (insert, update y delete)
- c. execute query

Práctica 25 – Parte 3. Implementación ICustomerDAO

iv. Spring JDBC - Transaction (c)

iv.vi Simplificando Operaciones JDBC

- a. SimpleJdbcInsert
- b. SimpleJdbcCall
- c. BeanPropertySqlParameterSource
- d. BeanPropertyRowMapper

Práctica 25 – Parte 4. Implementación ICustomer DAO
(mysql)

Práctica 25 – Parte 5. Implementación IAccountDAO

iv. Spring JDBC - Transaction (d)

iv.vii Spring y el Manejo Transaccional

a. ACID

b. Ventajas de Spring Transaction Management

c. PlatformTransactionManagement

iv. Spring JDBC - Transaction (e)

iv.viii Spring Tx

a. Transacciones declarativas configuración por XML

Práctica 26 – Parte 1. Transacciones declarativas
configuración por XML

b. Tipos de Propagación

c. Tipos de Aislamiento

d. Transacciones declarativas configuración por
@Anotaciones

Práctica 26 – Parte 2. Transacciones declarativas
configuración por @Anotaciones

d. Transacciones Programáticas

iv.i API JDBC

Objetivos de la lección

iv.i API JDBC

- Revisar a grandes rasgos el funcionamiento del API JDBC.
- Comprender la complejidad inherente al desarrollo de Data-Access Objects (DAOs) mediante el API JDBC.

iv. Spring JDBC - Transaction - iv.i API JDBC

iv.i API JDBC

a. Implementación API JDBC

iv.i API JDBC (a)

- JDBC (Java Database Connectivity) es el estándar de Java para la comunicación entre programas escritos en Java y un gran número de bases de datos, ya sean comerciales u open source.
- El API JDBC provee un conjunto de Interfaces que estandariza el acceso a bases de datos relacionales.
- Normaliza la mayor parte de las operaciones de acceso a datos, haciéndolas independientes del proveedor de base de datos.
- JDBC no comprueba la sintaxis de sentencias SQL.

iv. Spring JDBC - Transaction - iv.i API JDBC

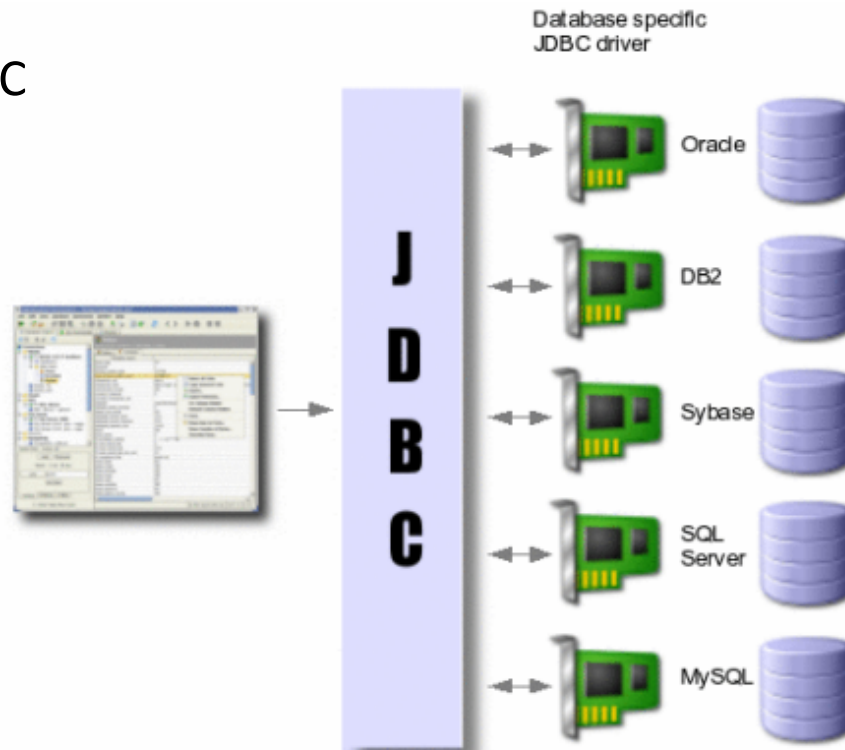
iv.i API JDBC (b)

- JDBC consta de un API genérico, la cual si implementación es distribuida mediante *Drivers*.
- Cada proveedor de base de datos (*vendor*) construye su propia implementación de acceso a datos para su gestor de base de datos.
- Java SE define el API JDBC y cada *vendor* implementa su *Driver* de conexión a base de datos.

iv. Spring JDBC - Transaction - iv.i API JDBC

iv.i API JDBC (c)

- Drivers JDBC



iv. Spring JDBC - Transaction - iv.i API JDBC

iv.i API JDBC (d)

- JDBC define APIs para realizar manipulación de:
 - Creación y liberación de conexiones.
 - Creación de sentencias SQL.
 - Ejecución de sentencias SQL.
 - Obtención de resultados (tuplas) en caso de existir.

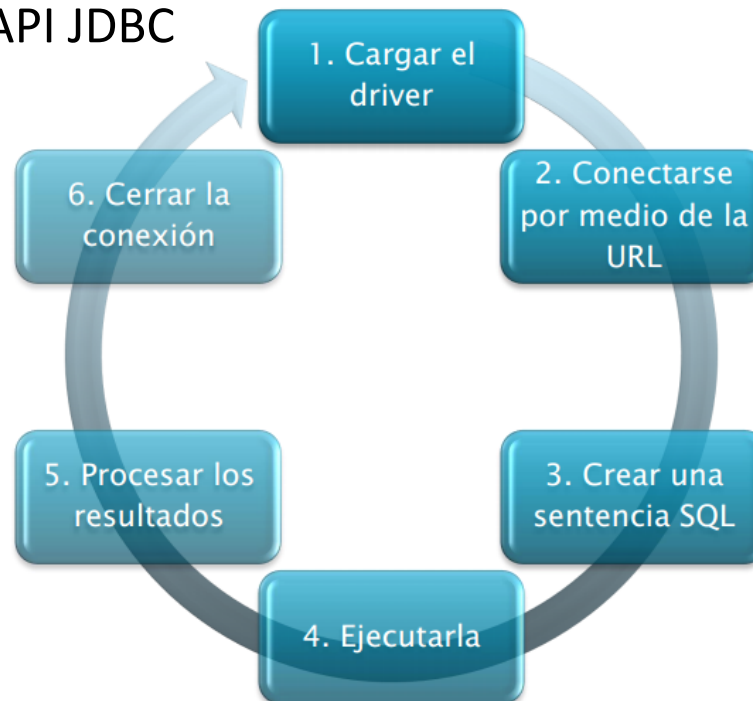
iv. Spring JDBC - Transaction - iv.i API JDBC

iv.i API JDBC

a. Implementación API JDBC

iv.i API JDBC (a)

- Implementación API JDBC



iv. Spring JDBC - Transaction - iv.i API JDBC

iv.i API JDBC (b)

- Implementación API JDBC

```
public static void main(String[] args) {  
    try {  
        Class.forName("com.mysql.jdbc.Driver");  
        String connectionUrl = "jdbc:mysql://localhost/mysql?" +  
                                "user=root&password=123456";  
        Connection con = DriverManager.getConnection(connectionUrl);  
    } catch (SQLException e) {  
        System.out.println("SQL Exception: " + e.toString());  
    } catch (ClassNotFoundException cE) {  
        System.out.println("Class Not Found Exception: " + cE.toString());  
    }  
}
```

iv. Spring JDBC - Transaction - iv.i API JDBC

iv.i API JDBC (c)

```
Connection connection = null;
try {
    Class.forName("com.mysql.jdbc.Driver").newInstance();
    Properties pr = new Properties();
    pr.setProperty("user", "root");
    pr.setProperty("password", "12345");
    connection = DriverManager.getConnection("jdbc:mysql://localhost:3306/mysql", pr);
    System.out.println("JAVA WAY START!");
    System.out.println(connection == null);
} catch (SQLException ex) {
    Logger.getLogger(index.class.getName()).log(Level.SEVERE, null, ex);
} catch (ClassNotFoundException ex) {
    Logger.getLogger(index.class.getName()).log(Level.SEVERE, null, ex);
} catch (InstantiationException ex) {
    Logger.getLogger(index.class.getName()).log(Level.SEVERE, null, ex);
} catch (IllegalAccessException ex) {
    Logger.getLogger(index.class.getName()).log(Level.SEVERE, null, ex);
} finally {
    if (connection != null) {
        try {
            connection.close();
            System.out.println("JAVA WAY STOPED! FINISH HIM!");
            connection = null;
            System.out.println("ULTRAKILL!!!");
        } catch (SQLException ex) {
            Logger.getLogger(index.class.getName()).log(Level.SEVERE, null, ex);
        }
    }
}
```

iv. Spring JDBC - Transaction - iv.i API JDBC

iv.i API JDBC (d)

```
public class PersonaDaoImpl implements PersonaDao {

    public Persona getPersonaById(long id) {
        Connection conn = null;
        PreparedStatement stmt = null;
        ResultSet rs = null;
        try {
            conn = dataSource.getConnection();
            stmt = conn.prepareStatement(
                "select id, nombre, ape_paterno from persona where id=?");
            stmt.setLong(1, id);
            rs = stmt.executeQuery();
            Persona persona = null;
            if (rs.next()) {
                persona = new Persona();
                persona.setId(rs.getLong("id"));
                persona.setNombre(rs.getString("nombre"));
                persona.setApellidoPaterno(rs.getString("ape_paterno"));
            }
            return persona;
        } catch (SQLException e) {
        } finally {
            if (rs != null) {
                try {
                    rs.close();
                } catch (SQLException e) {}
            }
            ...
        }
        return null;
    }
}
```

iv. Spring JDBC - Transaction - iv.i API JDBC

Resumen de la lección

iv.i API JDBC

- Comprendimos la utilidad del API JDBC.
- Verificamos la complejidad de desarrollo que implica desarrollar componentes de acceso a datos mediante el API JDBC.
- Comprobamos que es exhaustiva la cantidad de código de plomería lo que implica desarrollar mediante el API JDBC.
- Comprendimos el ciclo de vida de una conexión a base de datos mediante el API JDBC.

iv. Spring JDBC - Transaction - iv.i API JDBC

Esta página fue intencionalmente dejada en blanco.

iv. Spring JDBC - Transaction - iv.i API JDBC

iv.ii Spring JDBC

Objetivos de la lección

iv.ii Spring JDBC

- Comprender el beneficio de utilizar Spring JDBC.
- Comprender la filosofía de acceso a datos mediante Data-Access Objects (DAOs).
- Conocer el soporte que otorga utilizar Repositorios (DAOs) configurados con Spring Framework.

iv. Spring JDBC - Transaction - iv.ii Spring JDBC

iv.ii Spring JDBC

a. DAO Support

iv.ii Spring JDBC (a)

- Uno de los principales objetivos de Spring es la simplificación de tareas para el desarrollador de aplicaciones Java Empresariales.
- Spring Framework provee un conjunto de APIs para el manejo eficaz y simple de diversas APIs de Java, y JDBC no es la excepción.
- Al trabajar con JDBC plano se hace engorroso escribir código innecesario para manejar excepciones, abrir y cerrar conexiones, así como ejecutar consultas de lectura u escritura.

iv. Spring JDBC - Transaction - iv.ii Spring JDBC

iv.ii Spring JDBC (b)

- Spring JDBC:
 - Abstrae el API JDBC y provee de mecanismos simples para realizar operaciones de acceso a datos, simplificando el uso de JDBC.
 - Spring JDBC implementa detalles de bajo nivel en la abstracción del API JDBC.
- Spring JDBC realiza por defecto tareas tales como, solicitar una conexión al origen de datos (data source), abrir y cerrar conexiones, preparar y ejecutar consultas SQL y, manejar excepciones y transacciones.

iv. Spring JDBC - Transaction - iv.ii Spring JDBC

iv.ii Spring JDBC (c)

- Spring JDBC realiza por defecto tareas tales como:
 - Solicitar una conexión al origen de datos (data source)
 - Abrir la conexión
 - Preparar y ejecutar consultas SQL
 - Manejar excepciones
 - Manejar transacciones, y
 - Cerrar la conexión

iv. Spring JDBC - Transaction - iv.ii Spring JDBC

iv.ii Spring JDBC (d)

- Por tanto, para trabajar con Spring JDBC solamente es necesario:
 - Definir los parámetros de conexión
 - Definir las consultas SQL
 - Desarrollar la lógica necesaria para obtener los resultados de la consulta (ResultSet) mediante un mapeo entre el ResultSet y un POJO. (Para el caso de SELECT).

iv. Spring JDBC - Transaction - iv.ii Spring JDBC

iv.ii Spring JDBC

a. DAO Support

iv.ii Spring JDBC (a)

- ¿Qué es un DAO?
- Un Data-Access Object es un objeto que encapsula y abstrae la lógica requerida para el acceso a datos en un repositorio de datos específico.
- Un DAO puede implementar acceso a datos de origen:
 - Bases de Datos
 - Archivos
 - NoSQL
 - u otros orígenes de datos

iv. Spring JDBC - Transaction - iv.ii Spring JDBC

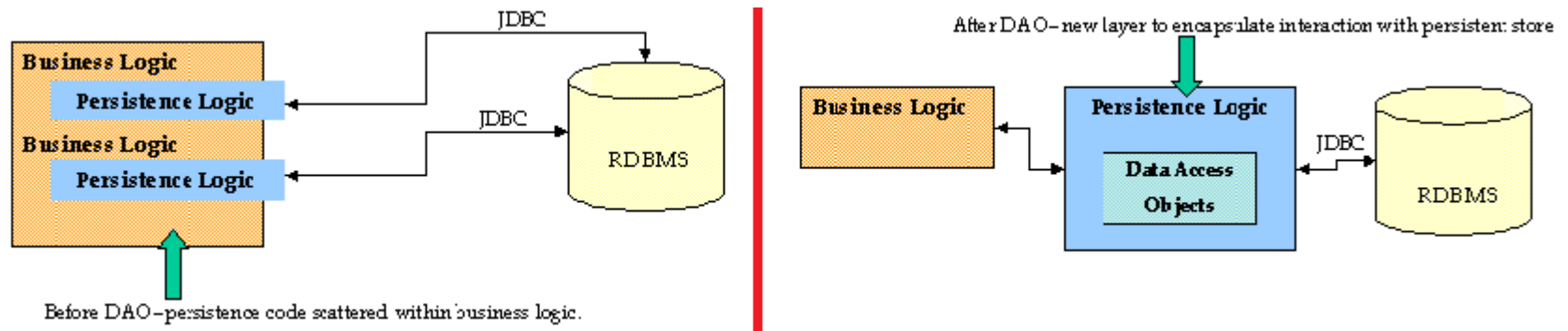
iv.ii Spring JDBC (b)

- ¿Qué es un DAO?
- Un DAO utiliza objetos de negocio o entidades, comúnmente llamados POJOs, los cuales modelan el dominio de la aplicación (objetos de dominio).
- La interfaz de un DAO se encarga de definir los métodos básicos CRUD para crear, recuperar, actualizar y eliminar un objeto de negocio del repositorio que utilice la aplicación.

iv. Spring JDBC - Transaction - iv.ii Spring JDBC

iv.ii Spring JDBC (c)

- La implementación de DAOs es otra forma de separación de responsabilidades.



- Se recomienda implementar un DAO por cada objeto de negocio, así como por cada origen de datos distinto.

iv. Spring JDBC - Transaction - iv.ii Spring JDBC

iv.ii Spring JDBC (d)

- DAO Support
- Spring JDBC ofrece soporte para la implementación de DAOs, independientemente de la tecnología de acceso a datos con la que se quiera trabajar, ya sea JDBC, Hibernate, JPA, JDO, MyBatis, etc.
- El *DAO support* permite cambiar fácilmente de tecnología de persistencia debido a que el código *caller* de la aplicación está acoplado únicamente con la interfaz del DAO y mediante inyección de dependencias se especifica la clase concreta del DAO.

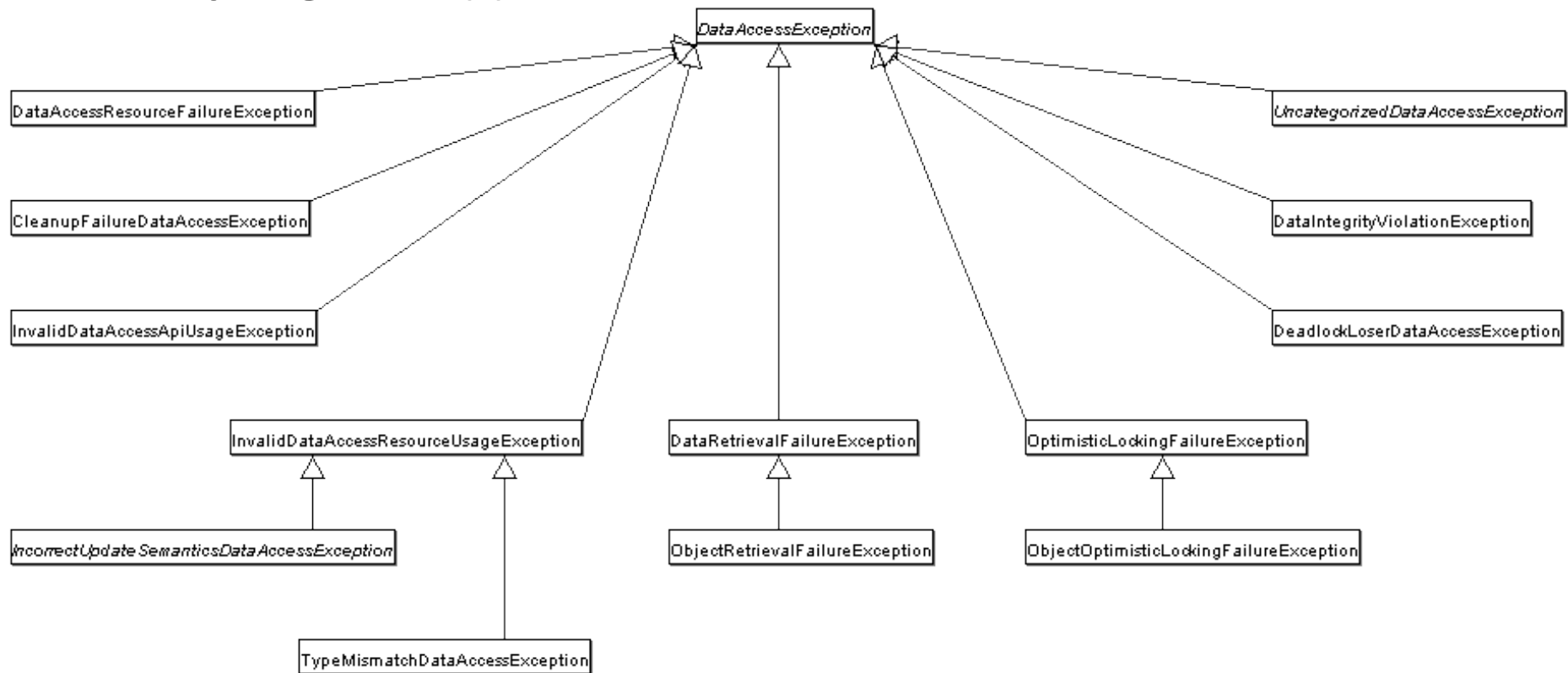
iv. Spring JDBC - Transaction - iv.ii Spring JDBC

iv.ii Spring JDBC (e)

- DAO Support
- El *DAO support* ofrece una jerarquía de excepciones consistente, convierte excepciones específicas de cada tecnología de acceso a datos, por ejemplo del tipo **SQLException**, a una única jerarquía de excepciones **DataAccessException**.
- Por otro lado convierte todas las excepciones *checked* propietarias ya sea del API JDBC, Hibernate, JPA, JDO, etcétera a excepciones *unchecked* del tipo **DataAccessException**.

iv. Spring JDBC - Transaction - iv.ii Spring JDBC

iv.ii Spring JDBC (f)



iv. Spring JDBC - Transaction - iv.ii Spring JDBC

iv.ii Spring JDBC (g)

- DAO Support
- Mediante el uso de la anotación **@Repository** Spring JDBC garantiza que se aplique la conversión de excepciones del tipo **SQLException** o propietarias de alguna tecnología de acceso a datos a excepciones del tipo **DataAccessException**.
- Mediante la anotación **@PersistenceContext** se permite la inyección del objeto **EntityManager** para el caso de implementación de DAOs con JPA.

iv. Spring JDBC - Transaction - iv.ii Spring JDBC

iv.ii Spring JDBC (h)

- DAO Support
- Para el caso de implementación de DAOs con Hibernate, *DAO Support* permite la inyección de **SessionFactory** mediante **@Autowired**.
- A si mismo, para la implementación de DAOs con Spring JDBC, es posible inyectar el objeto **DataSource** configurado en el IoC de Spring mediante **@Autowired**.
- Nota: Spring JDBC requiere un objeto DataSource para abstraer el API JDBC.

iv. Spring JDBC - Transaction - iv.ii Spring JDBC

Resumen de la lección

iv.ii Spring JDBC

- Comprendimos la utilidad de implementar DAOs.
- Comprendimos por qué es importante agilizar el desarrollo de DAOs.
- Analizamos las características del *DAO Support* de Spring JDBC.
- Comprendimos la utilidad de utilizar `@Repository` sobre implementaciones DAO.
- Verificamos como inyectar `EntityManager (JPA)` y `SessionFactory (Hibernate)`.

iv. Spring JDBC - Transaction - iv.ii Spring JDBC

Esta página fue intencionalmente dejada en blanco.

iv. Spring JDBC - Transaction - iv.ii Spring JDBC

iv.iii Acceso a Datos con Spring JDBC

Objetivos de la lección

iv.iii Acceso a Datos con Spring JDBC

- Conocer los beneficios de utilizar Spring JDBC a detalle.
- Conocer qué tareas en la implementación de DAOs se delegan en Spring JDBC y qué otras en el desarrollador.
- Comprender el patrón de diseño template.
- Conocer los templates de Spring JDBC para facilitar el acceso a datos.
- Conocer la clase base JdbcDaoSupport para la implementación de DAOs.
- Conocer las diversas implementaciones de DataSource que implementa Spring JDBC y proveedores terceros.

iv. Spring JDBC - Transaction - iv.iii Acceso a Datos con Spring JDBC

iv.iii Acceso a Datos con Spring JDBC

- a. Templates
- b. JdbcDaoSupport
- c. Data Sources
- d. Soporte para base de datos embebida

Práctica 25 – Parte 1. Configuración Spring JDBC

iv.iii Acceso a Datos con Spring JDBC (a)

- Spring JDBC maneja y facilita muchas de las tareas requeridas para implementar acceso a datos con el API JDBC.

| Acción | Spring | Programador |
|--|--------|-------------|
| Definir parámetros de conexión. | No | Si |
| Abrir conexiones | Si | No |
| Definir sentencias SQL | No | Si |
| Declarar y proveer parámetros para las consultas | No | Si |
| Preparar y ejecutar sentencias SQL | Si | No |

iv. Spring JDBC - Transaction - iv.iii Acceso a Datos con Spring JDBC

iv.iii Acceso a Datos con Spring JDBC (b)

- Spring JDBC maneja y facilita muchas de las tareas requeridas para implementar acceso a datos con el API JDBC.

| Acción | Spring | Programador |
|--|--------|-------------|
| Iterar sobre cada resultado obtenido por la consulta (ResultSet) | Si | No |
| Realizar la extracción o mapeo de cada resultado obtenido del ResultSet a un objeto de dominio (POJO). | No | Si |
| Procesar cualquier excepción | Si | No |
| Manejar Transacciones | Si | No |
| Cerrar conexiones, sentencias y result sets. | Si | No |

iv. Spring JDBC - Transaction - iv.iii Acceso a Datos con Spring JDBC

iv.iii Acceso a Datos con Spring JDBC

a. **Templates**

b. JdbcDaoSupport

c. Data Sources

d. Soporte para base de datos embebida

Práctica 25 – Parte 1. Configuración Spring JDBC

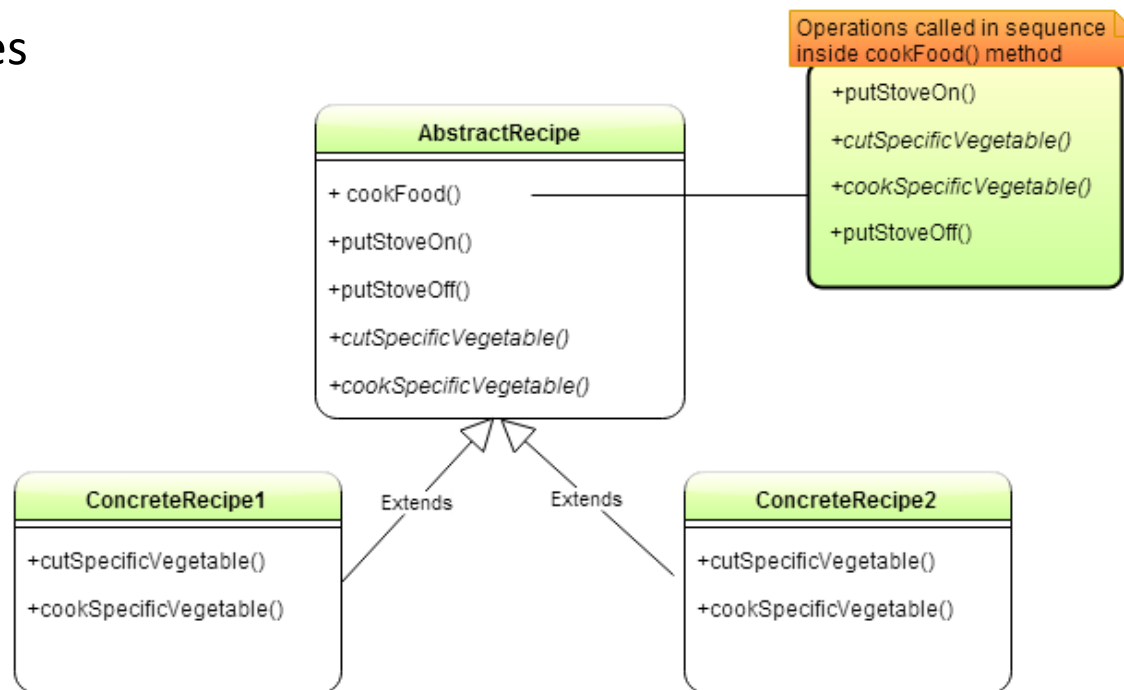
iv.iii Acceso a Datos con Spring JDBC (a)

- Templates
- Spring JDBC provee diferentes mecanismos para acceder a datos por medio de JDBC, todas ellas partiendo de una misma base el objeto JdbcTemplate.
- JdbcTemplate así como todas las clases que provee Spring JDBC implementan el patrón de diseño Template el cual define una plantilla para la ejecución de un algoritmo.

iv. Spring JDBC - Transaction - iv.iii Acceso a Datos con Spring JDBC

iv.iii Acceso a Datos con Spring JDBC (b)

- Templates



iv. Spring JDBC - Transaction - iv.iii Acceso a Datos con Spring JDBC

iv.iii Acceso a Datos con Spring JDBC (c)

- Spring JDBC provee las siguientes clases de acceso a datos.
- **JdbcTemplate**: Es el template clásico de Spring JDBC y es la que trabaja a más bajo nivel y es usada por otras clases tras bambalinas. Utiliza placeholders (“?”) para el reemplazo de parámetros en consultas SQL.
- **NamedParameterJdbcTemplate**: Encapsula o envuelve un objeto JdbcTemplate. Utiliza nombres de parámetros en lugar de placeholders para el reemplazo de parámetros en consultas SQL. Es más fácil de usar que JdbcTemplate para la mayoría de las operaciones básicas CRUD.

iv. Spring JDBC - Transaction - iv.iii Acceso a Datos con Spring JDBC

iv.iii Acceso a Datos con Spring JDBC (d)

- Spring JDBC provee las siguientes clases de acceso a datos.
- **SimpleJdbcInsert** y **SimpleJdbcCall**: Simplifican la codificación de operaciones de inserción y de llamada a procedimientos remotos. Utilizan meta-datos de la base de datos para disminuir la cantidad de configuración que, por lo regular es requerida en JdbcTemplate.
- **MappingSqlQuery**, **SqlUpdate** y **StoredProcedure**: Permiten crear objetos reusables (thread-safe) durante la inicialización de la capa de datos, para realizar consultas, actualizaciones y llamadas a procedimientos remotos similar al modelo de programación de JDO.

iv. Spring JDBC - Transaction - iv.iii Acceso a Datos con Spring JDBC

iv.iii Acceso a Datos con Spring JDBC

a. Templates

b. JdbcDaoSupport

c. Data Sources

d. Soporte para base de datos embebida

Práctica 25 – Parte 1. Configuración Spring JDBC

iv.iii Acceso a Datos con Spring JDBC (a)

- JdbcDaoSupport
- Clase que contiene como atributo una referencia a JdbcTemplate.
- Una clase DAO común puede heredar de ella y por ende tiene acceso a las plantillas. Es usada para no tener que inyectar un JdbcTemplate.
- Requiere inyectar un DataSource.
- No se recomienda pues se prefiere composición en lugar de herencia.

iv. Spring JDBC - Transaction - iv.iii Acceso a Datos con Spring JDBC

iv.iii Acceso a Datos con Spring JDBC (b)

- JdbcDaoSupport

```
import org.springframework.jdbc.core.support.JdbcDaoSupport;
```

```
public class EmployeeDaoImpl extends JdbcDaoSupport implements EmployeeDao{
```

```
    @Override
```

```
    public void insertEmployee(Employee emp) {
```

```
        String query = ...
```

```
        Object[] inputs = ...
```

```
        getJdbcTemplate().update(query, inputs);
```

```
    }
```

```
}
```

iv. Spring JDBC - Transaction - iv.iii Acceso a Datos con Spring JDBC

iv.iii Acceso a Datos con Spring JDBC

- a. Templates
- b. JdbcDaoSupport
- c. Data Sources
- d. Soporte para base de datos embebida

Práctica 25 – Parte 1. Configuración Spring JDBC

iv.iii Acceso a Datos con Spring JDBC (a)

- Data Sources
- Spring obtiene una conexión a base de datos a partir de un objeto DataSource.
- Un DataSource es parte de la especificación del API JDBC y puede relacionarse a una fábrica de conexiones.
- Un DataSource permite ocultar detalles de el manejo de transacciones así como el *pool* de conexiones (en caso de existir) al código de la aplicación.

iv. Spring JDBC - Transaction - iv.iii Acceso a Datos con Spring JDBC

iv.iii Acceso a Datos con Spring JDBC (b)

- Data Sources
- En ambientes productivos difícilmente será responsabilidad del desarrollador configurar los DataSource.
- Como desarrollador únicamente debemos saber que para obtener conexiones a bases de datos lo recomendable, para ambientes productivos, es adquirirlas a través de un DataSource.

iv. Spring JDBC - Transaction - iv.iii Acceso a Datos con Spring JDBC

iv.iii Acceso a Datos con Spring JDBC (c)

- Data Sources
- Cuando implementamos DAOs con Spring JDBC es posible obtener conexiones a partir de:
 - Definición de DataSource por JNDI
 - Definición de DataSource de una implementación de Spring JDBC
 - Definición de DataSource de un proveedor tercero

iv. Spring JDBC - Transaction - iv.iii Acceso a Datos con Spring JDBC

iv.iii Acceso a Datos con Spring JDBC (d)

- Data Sources
- Como implementaciones populares de DataSource por terceros se encuentra Apache Commons DBCP o C3P0.
- Las implementaciones de DataSource de Spring JDBC están orientadas únicamente a pruebas y ambientes single-thread (no provee *pool* de conexiones).

iv. Spring JDBC - Transaction - iv.iii Acceso a Datos con Spring JDBC

iv.iii Acceso a Datos con Spring JDBC (e)

- Interfaces DataSource nativas de Spring JDBC
- **SmartDataSource**: Esta interface hereda de la interface DataSource y define el método **boolean shouldClose(Connection con)**, esta clase es útil para realizar consultas sin tener que cerrar la conexión. Esto es eficiente cuando se trata de compartir y reusar la misma conexión.
- **SingleConnectionDataSource**: Esta clase implementa **SmartDataSource** y envuelve una única instancia **Connection** misma que no es cerrada después de cada uso. No soporta ambientes multi-hilo.

iv. Spring JDBC - Transaction - iv.iii Acceso a Datos con Spring JDBC

iv.iii Acceso a Datos con Spring JDBC (f)

- Interfaces DataSource nativas de Spring JDBC
- **DriverManagerDataSource**: Esta clase implementa el estándar DataSource del API JDBC, se configura a partir de un bean xml (o programáticamente con Java Config) asignando sus propiedades. Esta clase retorna una nueva instancia **Connection** cada que se solicita. No es una implementación de *pool* de conexiones.

iv. Spring JDBC - Transaction - iv.iii Acceso a Datos con Spring JDBC

iv.iii Acceso a Datos con Spring JDBC (g)

- Configuración de DataSource: DriverManagerDataSource.
- Configuración programática con Java Config.

```
DriverManagerDataSource dataSource = new DriverManagerDataSource();  
dataSource.setDriverClassName("org.hsqldb.jdbcDriver");  
dataSource.setUrl("jdbc:hsqldb:hsqldb://localhost:");  
dataSource.setUsername("sa");  
dataSource.setPassword("");
```

iv. Spring JDBC - Transaction - iv.iii Acceso a Datos con Spring JDBC

iv.iii Acceso a Datos con Spring JDBC (h)

- Configuración de DataSource: DriverManagerDataSource.
- Configuración por medio de definición de bean XML.

```
<bean id="dataSource"  
    class="org.springframework.jdbc.datasource.DriverManagerDataSource">  
    <property name="driverClassName" value="${jdbc.driverClassName}"/>  
    <property name="url" value="${jdbc.url}"/>  
    <property name="username" value="${jdbc.username}"/>  
    <property name="password" value="${jdbc.password}"/>  
</bean>  
<context:property-placeholder location="jdbc.properties"/>
```

iv. Spring JDBC - Transaction - iv.iii Acceso a Datos con Spring JDBC

iv.iii Acceso a Datos con Spring JDBC (i)

- Configuración de DataSource: Apache Commons BasicDataSource.
- Configuración por medio de definición de bean XML.

```
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
    destroy-method="close">
    <property name="driverClassName" value="${jdbc.driverClassName}"/>
    <property name="url" value="${jdbc.url}"/>
    <property name="username" value="${jdbc.username}"/>
    <property name="password" value="${jdbc.password}"/>
</bean>
<context:property-placeholder location="jdbc.properties"/>
```

iv. Spring JDBC - Transaction - iv.iii Acceso a Datos con Spring JDBC

iv.iii Acceso a Datos con Spring JDBC (j)

- Configuración de DataSource: MChange C3P0 ComboPooledDataSource
- Configuración por medio de definición de bean XML.

```
<bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource"
    destroy-method="close">
    <property name="driverClass" value="${jdbc.driverClassName}"/>
    <property name="jdbcUrl" value="${jdbc.url}"/>
    <property name="user" value="${jdbc.username}"/>
    <property name="password" value="${jdbc.password}"/>
</bean>
<context:property-placeholder location="jdbc.properties"/>
```

iv. Spring JDBC - Transaction - iv.iii Acceso a Datos con Spring JDBC

iv.iii Acceso a Datos con Spring JDBC

- a. Templates
- b. JdbcDaoSupport
- c. Data Sources

d. Soporte para base de datos embebida

Práctica 25 – Parte 1. Configuración Spring JDBC

iv.iii Acceso a Datos con Spring JDBC (a)

- Soporte para base de datos embebida
- Spring JDBC soporta bases de datos Java embebidas tales como HSQL, H2 y Derby.
- Es posible utilizar el API de Spring JDBC embedded para embeber otros tipos de bases de datos embebidas siempre que implementen `javax.sql.DataSource`.

iv. Spring JDBC - Transaction - iv.iii Acceso a Datos con Spring JDBC

iv.iii Acceso a Datos con Spring JDBC (b)

- ¿Por qué utilizar bases de datos embebidas?
 - Facilidad de definir
 - Muy útil durante el desarrollo, facilita las pruebas
 - Rápido despliegue
- Definición XML.

```
<jdbc:embedded-database id="dataSource" type="H2" generate-name="true">  
  <jdbc:script location="classpath:schema.sql"/>  
  <jdbc:script location="classpath:test-data.sql"/>  
</jdbc:embedded-database>
```

iv. Spring JDBC - Transaction - iv.iii Acceso a Datos con Spring JDBC

iv.iii Acceso a Datos con Spring JDBC (c)

- Definición programática.

```
EmbeddedDatabase db = new EmbeddedDatabaseBuilder() .  
    generateUniqueName(true) .  
    setType(H2) .  
    setScriptEncoding("UTF-8") .  
    ignoreFailedDrops(true) .  
    addScript("schema.sql") .  
    addScripts("user_data.sql", "country_data.sql") .build();
```

...

```
db.shutdown();
```

iv. Spring JDBC - Transaction - iv.iii Acceso a Datos con Spring JDBC

iv.iii Acceso a Datos con Spring JDBC (d)

- Soporte de Bases de Datos Java
 - H2
 - HSQL 1.8.0 +
 - Derby 10.5 +
- Es necesario incluir en el classpath las dependencias correspondientes a cada base de datos Java.

```
<dependency>  
  <groupId>com.h2database</groupId>  
  <artifactId>h2</artifactId>  
  <version>1.4.190</version>  
</dependency>
```

iv. Spring JDBC - Transaction - iv.iii Acceso a Datos con Spring JDBC

iv.iii Acceso a Datos con Spring JDBC

- a. Templates
- b. JdbcDaoSupport
- c. Data Sources
- d. Soporte para base de datos embebida

Práctica 25 – Parte 1. Configuración Spring JDBC

Bonus. Spring Profiles (a)

- Spring permite agrupar beans mediante perfiles (*profiles*).
- Un perfil es un identificador que permite agrupar definiciones de beans para ser registrados en el contenedor de IoC de Spring sólo si dicho perfil se encuentra activo.
- Es posible asociar beans a perfiles mediante configuración de beans por XML, @Anotaciones o vía JavaConfig.
- Es posible utilizar operadores lógicos para generar expresiones a evaluar por los perfiles.

iv. Spring JDBC - Transaction - iv.iii Acceso a Datos con Spring JDBC

Bonus. Spring Profiles (b)

- **@Profile**: La anotación @Profile permite indicar para que perfil corresponde un componente (@Component). El contenedor de IoC sólo levantará los beans registrados en el perfil default y aquellos registrados en los perfiles activos.

@Configuration

@Profile("dev")

```
public class StandaloneDataConfig {  
    @Bean  
    public DataSource dataSource() {  
        return new EmbeddedDatabaseBuilder() .setType(EmbeddedDatabaseType.HSQL).  
            addScript("classpath:com/bank/config/sql/schema.sql").  
            addScript("classpath:com/bank/config/sql/test-data.sql") .build();  
    }  
}
```

iv. Spring JDBC - Transaction - iv.iii Acceso a Datos con Spring JDBC

Bonus. Spring Profiles (c)

@Configuration

@Profile("production")

```
public class JndiDataConfig {  
    @Bean(destroyMethod="close")  
    public DataSource dataSource() throws Exception {  
        Context ctx = new InitialContext();  
        return (DataSource) ctx.lookup("java:comp/env/jdbc/datasource");  
    }  
}
```

iv. Spring JDBC - Transaction - iv.iii Acceso a Datos con Spring JDBC

Bonus. Spring Profiles (d)

- Operadores lógicos aplicados a los perfiles.
 - ! = Operador lógico “not” del perfil.
 - & = Operador lógico “and” del perfil.
 - | = Operador lógico “or” del perfil.
- Es posible agrupar operadores lógicos únicamente utilizando paréntesis.
 - La expresión = “production & us-east | eu-central” no es válida.
 - La expresión = “production & (us-east | eu-central)” es válida.

iv. Spring JDBC - Transaction - iv.iii Acceso a Datos con Spring JDBC

Bonus. Spring Profiles (e)

- Es posible utilizar la anotación `@Profile` como una meta-anotación para crear anotaciones personalizadas y “type-safe”.

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Profile("production")
public @interface Production { }
```

@Production

@Profile("production")

iv. Spring JDBC - Transaction - iv.iii Acceso a Datos con Spring JDBC

Bonus. Spring Profiles (f)

@Configuration

```
public class AppConfig {
```

@Bean

```
@Profile("dev")
```

```
public DataSource dataSourceDev() {
```

```
    return new EmbeddedDatabaseBuilder().setType(EmbeddedDatabaseType.HSQL).
```

```
        addScript("classpath:com/bank/config/sql/schema.sql").
```

```
        addScript("classpath:com/bank/config/sql/test-data.sql").build();
```

```
}
```

```
@Bean(destroyMethod="close")
```

```
@Profile("production")
```

```
public DataSource dataSourceProd() throws Exception {
```

```
    Context ctx = new InitialContext();
```

```
    return (DataSource) ctx.lookup("java:comp/env/jdbc/datasource");
```

```
}
```

```
}
```

iv. Spring JDBC - Transaction - iv.iii Acceso a Datos con Spring JDBC

Bonus. Spring Profiles (g)

```
<beans xmlns="http://www.springframework.org/schema/beans" ...>
```

```
  <beans profile="dev">
```

```
    <jdbc:embedded-database id="dataSource">
```

```
      <jdbc:script location="classpath:com/bank/config/sql/schema.sql"/>
```

```
      <jdbc:script location="classpath:com/bank/config/sql/test-data.sql"/>
```

```
    </jdbc:embedded-database>
```

```
  </beans>
```

```
  <beans profile="production">
```

```
    <jee:jndi-lookup id="dataSource" jndi-name="java:comp/env/jdbc/datasource"/>
```

```
  </beans>
```

```
</beans>
```

iv. Spring JDBC - Transaction - iv.iii Acceso a Datos con Spring JDBC

Bonus. Spring Profiles (h)

- Activación de Perfiles
- AnnotationConfigApplicationContext:

```
AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext();  
ctx.getEnvironment().setActiveProfiles("development");  
ctx.register(Config.class, StandaloneDataConfig.class, JndiDataConfig.class);  
ctx.refresh();
```
- Variable del sistema, propiedades de la JVM, parámetros de ServletContext:

```
spring.profiles.active=development
```

iv. Spring JDBC - Transaction - iv.iii Acceso a Datos con Spring JDBC

Bonus. Spring Profiles (i)

- @ActiveProfiles: Permite habilitar perfiles para pruebas de integración con spring-test y JUnit.

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("/app-config.xml")
@ActiveProfiles("dev")
public class TransferServiceTest {
    @Autowired
    private TransferService transferService;

    @Test
    public void testTransferService() {
        ...
    }
}
```

iv. Spring JDBC - Transaction - iv.iii Acceso a Datos con Spring JDBC

iv.iii Acceso a Datos con Spring JDBC. Práctica 25. (a)

- Práctica 25 – Parte 1. Configuración Spring JDBC.
- Configurar los data sources requeridos para desarrollar DAOs mediante Spring JDBC y profiles.
- Configurar JdbcTemplate y NamedParameterJdbcTemplate

iv. Spring JDBC - Transaction - iv.iii Acceso a Datos con Spring JDBC

Resumen de la lección (a)

iv.iii Acceso a Datos con Spring JDBC

- Comprendimos cual es el propósito de JdbcTemplate y NamedParameterJdbcTemplate.
- Comprendimos el patrón de diseño Template.
- Analizamos las diferencias entre la estrategia de inyección de JdbcTemplate y la herencia de JdbcDaoSupport.
- Conocimos los principales implementaciones data source de spring JDBC y sus finalidades.

iv. Spring JDBC - Transaction - iv.iii Acceso a Datos con Spring JDBC

Resumen de la lección (b)

iv.iii Acceso a Datos con Spring JDBC

- Conocimos las principales implementaciones de data sources de terceros.
- Comprendimos la utilidad de utilizar bases de datos embebidas para desarrollo y pruebas.
- Verificamos como cambiar de implementación de beans mediante Perfiles.

iv. Spring JDBC - Transaction - iv.iii Acceso a Datos con Spring JDBC

Esta página fue intencionalmente dejada en blanco.

iv. Spring JDBC - Transaction - iv.iii Acceso a Datos con Spring JDBC

iv.iv JdbcTemplate

Objetivos de la lección

iv.iv JdbcTemplate

- Conocer la clase core JdbcTemplate a fondo.
- Comprender más en detalle el funcionamiento de JdbcTemplate mediante la llamada a callbacks.
- Conocer la funcionalidad de un RowMapper<T>
- Verificar la correcta configuración de JdbcTemplate en beans DAO.
- Comprender el uso de JdbcTemplate para realizar instrucciones SELECT, INSERT, UPDATE y DELETE.
- Conocer la manera de ejecutar instrucciones DDL.

iv. Spring JDBC - Transaction - iv.iv JdbcTemplate

iv.iv JdbcTemplate

- a. Select query
- b. Update query (insert, update y delete)
- c. Execute query

Práctica 25 – Parte 2. Implementación IUserDAO

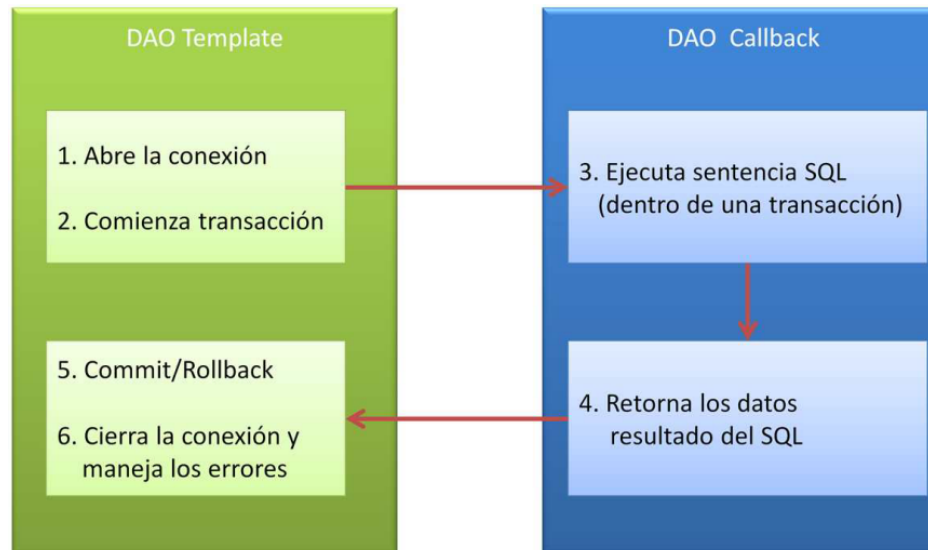
iv.iv JdbcTemplate (a)

- JdbcTemplate es la clase central del core de Spring JDBC.
- Maneja la creación y liberación de recursos (Connection, Statement y ResultSet).
- Ejecuta las operaciones básicas del core de Spring JDBC tal como creación y ejecución de sentencias SQL, implementa la delegación de tareas al desarrollador mediante el patrón de diseño template.
- Ejecuta las consultas e itera sobre los resultados obtenidos (de haberlos) y delega el mapeo de éstos datos, al programador, mediante POJOs.

iv. Spring JDBC - Transaction - iv.iv JdbcTemplate

iv.iv JdbcTemplate (b)

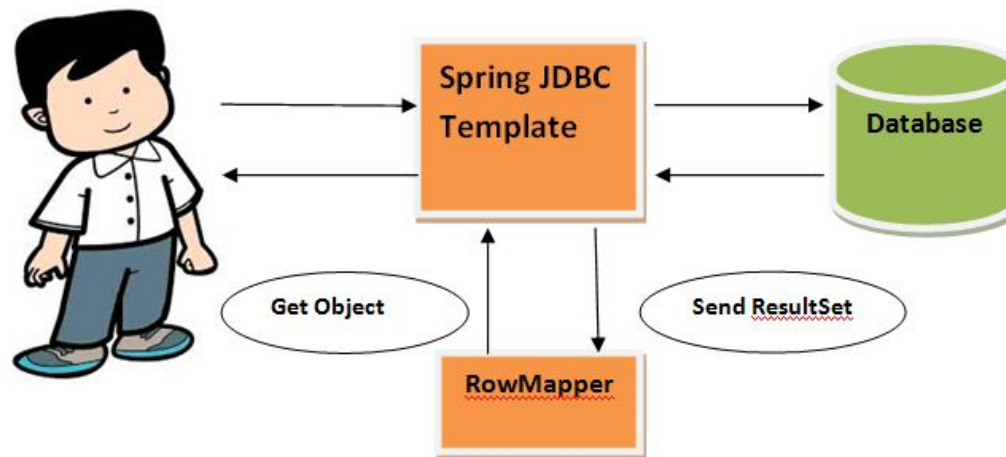
- JdbcTemplate requiere implementar interfaces *callback* para definir el comportamiento requerido por el template.



iv. Spring JDBC - Transaction - iv.iv JdbcTemplate

iv.iv JdbcTemplate (c)

- Ejemplo Callback RowMapper:
- Mapeo de la información obtenida desde un ResultSet a un POJO.



iv. Spring JDBC - Transaction - iv.iv JdbcTemplate

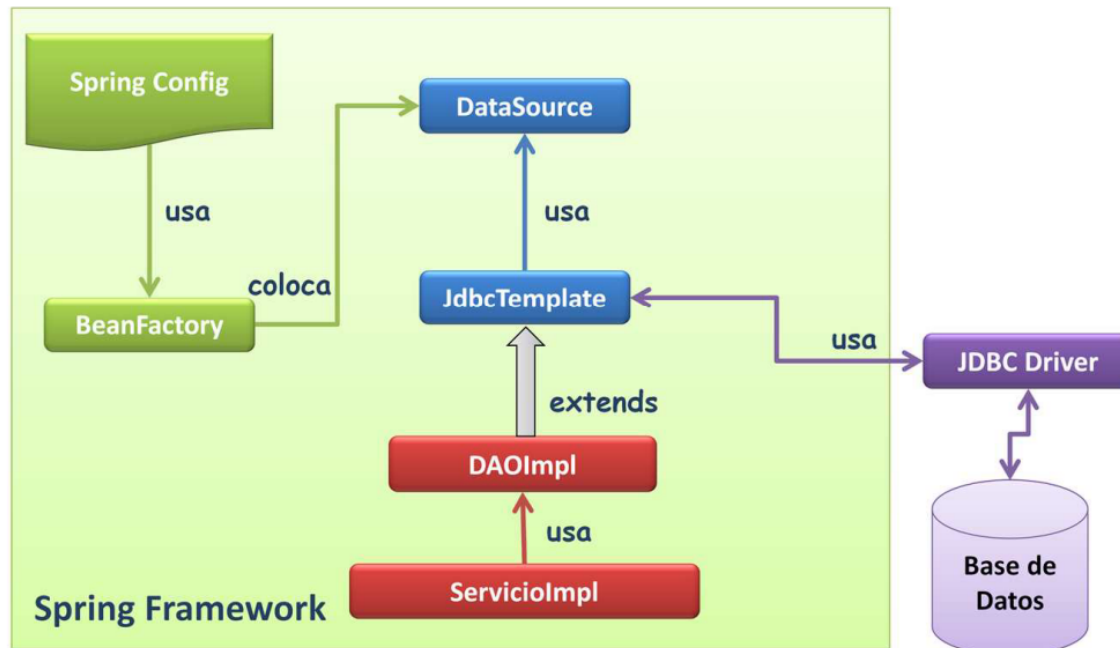
iv.iv JdbcTemplate (d)

- JdbcTemplate debe ser utilizado dentro de la implementación de una interface DAO. Para construir un objeto JdbcTemplate es necesario tener una referencia al DataSource de donde se gestionará la creación y liberación de conexiones, así como el manejo transaccional.
- El DataSource siempre será configurado por Spring IoC.

iv. Spring JDBC - Transaction - iv.iv JdbcTemplate

iv.iv JdbcTemplate (e)

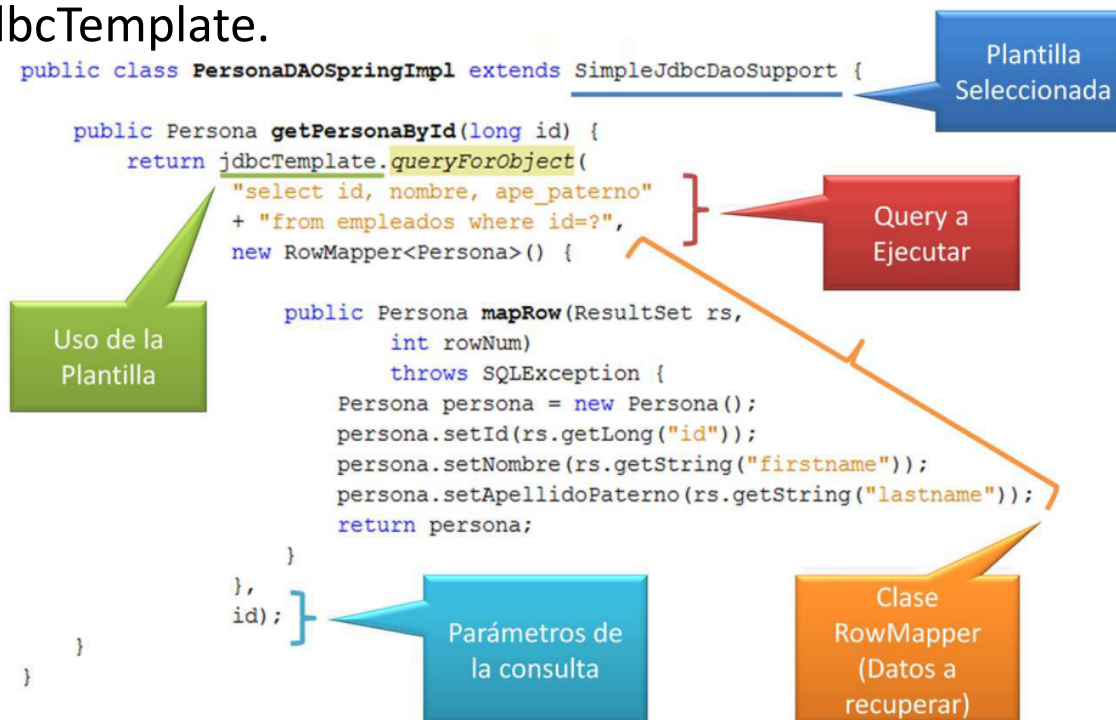
- Configuración JdbcTemplate.



iv. Spring JDBC - Transaction - iv.iv JdbcTemplate

iv.iv JdbcTemplate (f)

- Uso de JdbcTemplate.



iv. Spring JDBC - Transaction - iv.iv JdbcTemplate

iv.iv JdbcTemplate

- a. **Select query**
- b. Update query (insert, update y delete)
- c. Execute query

Práctica 25 – Parte 2. Implementación IUserDAO

iv.iv JdbcTemplate (a)

- Select query: Consultas por una única fila (query for single row).
- **queryForObject**: Retorna una sola columna o un solo objeto, este método espera que exista un único resultado en la consulta. Si se obtienen cero o más de un resultado (fila) lanza excepción **EmptyResultDataAccessException** o **IncorrectResultSizeDataAccessException** respectivamente.

iv. Spring JDBC - Transaction - iv.iv JdbcTemplate

iv.iv JdbcTemplate (b)

- Select query: Consultas por una única fila (query for single row).
- Obteniendo una única columna.

```
String SELECT = "SELECT username FROM USER WHERE id = ?";  
Long id = 1;
```

```
String username = jdbcTemplate.queryForObject(SELECT, String.class, id);
```

```
log.info("username: {}", username);
```

```
username: laura123
```

iv. Spring JDBC - Transaction - iv.iv JdbcTemplate

iv.iv JdbcTemplate (c)

- Select query: Consultas por una única fila (query for single row).
- Obteniendo más de una columna en un objeto POJO

```
User user = jdbcTemplate.queryForObject("SELECT * FROM USER WHERE id = ?", (rs, n) -> {  
    User usr = new User();  
    usr.setId(rs.getLong("USER_ID"));  
    usr.setUsername(rs.getString("USERNAME"));  
    usr.setPassword(rs.getString("PASSWORD"));  
    return usr;  
}, id);  
log.info("user: {}", user);  
user: User(id=2, username=laura123, password=123123)
```

iv. Spring JDBC - Transaction - iv.iv JdbcTemplate

iv.iv JdbcTemplate (d)

- Select query: Consultas por una única fila (query for single row).
- El ejemplo anterior hace uso de un lambda, implementación de `RowMapper<T>`

```
public interface RowMapper<T> {  
    T mapRow(ResultSet rs, int rowNum) throws SQLException;  
}
```

iv. Spring JDBC - Transaction - iv.iv JdbcTemplate

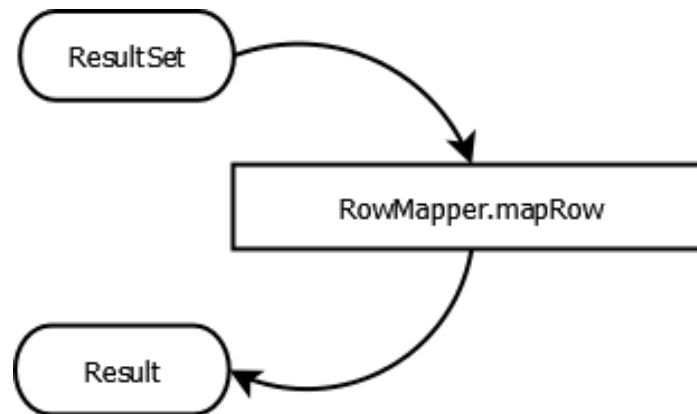
iv.iv JdbcTemplate (e)

```
public class AccountRowMapper implements RowMapper<Account> {  
  
    @Override  
    public Account mapRow(ResultSet rs, int rowNum) throws SQLException {  
        Account account = new Account();  
  
        account.setId(rs.getLong("ACCOUNT_ID"));  
        account.setAccountNumber(rs.getString("ACCOUNT_NUMBER"));  
        account.setCreatedDate(new CustomDate(rs.getDate("CREATED_DATE").getTime()));  
        account.setBalance(rs.getBigDecimal("BALANCE"));  
        return account;  
    }  
}
```

iv. Spring JDBC - Transaction - iv.iv JdbcTemplate

iv.iv JdbcTemplate (f)

- **RowMapper<T>**: Interface genérica que define el método mapRow el cual será indocado desde el template para proveer la estrategia de mapeo de un ResultSet a un POJO.



iv. Spring JDBC - Transaction - iv.iv JdbcTemplate

iv.iv JdbcTemplate (g)

- Select query: Consultas por una única fila (query for single row).
- **queryForMap**: Retorna un solo resultado independientemente si contiene una o más columnas en un objeto Map<String, Object>, este método espera que exista un único resultado en la consulta. Si se obtienen cero o más de un resultado (fila) lanza excepción **EmptyResultDataAccessException** o **IncorrectResultSizeDataAccessException** respectivamente.

iv. Spring JDBC - Transaction - iv.iv JdbcTemplate

iv.iv JdbcTemplate (h)

- Select query: Consultas por una única fila (query for single row).
- Obteniendo una o más columnas en un Map<String, Object>.

```
String SELECT = "SELECT username, password FROM USER WHERE id = ?";  
Long id = 1;
```

```
Map<String, Object> mapResult = jdbcTemplate.queryForMap(SELECT, new Object[] { id });
```

```
log.info("mapResult: {}", mapResult);
```

```
mapResult: {USERNAME=laura123, PASSWORD=123123}
```

iv. Spring JDBC - Transaction - iv.iv JdbcTemplate

iv.iv JdbcTemplate (i)

- Select query: Consultas por cero, una, o más de una fila (query for rows).
- **queryForList**: Retorna una lista del tipo especificado cuando se espera una única columna y se especifica el tipo de la columna, o retorna una lista de Map<String, Object> si el resultado de la consulta devuelve una o más columnas cuando, para el caso de devolver una columna, no se especifica el tipo.

iv. Spring JDBC - Transaction - iv.iv JdbcTemplate

iv.iv JdbcTemplate (j)

- Select query: Consultas por cero, una, o más de una fila (query for rows).
- Obteniendo una lista de una única columna, especificando el tipo.

```
String SELECT = "SELECT username FROM USER WHERE id > ?";
```

```
List<String> usernames = jdbcTemplate.queryForList(SELECT, String.class, new Object[]{0} );
```

```
log.info("usernames: {}", usernames);
```

```
usernames: [xvanhalenx, laura123]
```

iv. Spring JDBC - Transaction - iv.iv JdbcTemplate

iv.iv JdbcTemplate (k)

- Select query: Consultas por cero, una, o más de una fila (query for rows).
- Obteniendo una lista de una única columna, sin especificar el tipo.

```
String SELECT = "SELECT username FROM USER WHERE id > ?";
```

```
List<Map<String, Object>> usernames = jdbcTemplate.queryForList(SELECT, new Object[]{0} );
```

```
log.info("usernames: {}", usernames);
```

```
usernames: [{USERNAME=xvanhalenx}, {USERNAME=laura123}]
```

iv. Spring JDBC - Transaction - iv.iv JdbcTemplate

iv.iv JdbcTemplate (I)

- Select query: Consultas por cero, una, o más de una fila (query for rows).
- Obteniendo una lista de una o más columnas.

String SELECT = "SELECT username, password FROM USER WHERE id > ?";

```
List<Map<String, Object>> usernameAndPassword = jdbcTemplate.queryForList(SELECT,  
                                                                    new Object[]{0} );
```

```
log.info("usernameAndPassword: {}", usernameAndPassword);
```

```
usernameAndPassword: [{USERNAME=xvanhalenx, PASSWORD=123123},  
                      {USERNAME=laura123, PASSWORD=123123}]
```

iv. Spring JDBC - Transaction - iv.iv JdbcTemplate

iv.iv JdbcTemplate (m)

- Select query: Consultas por cero, una, o más de una fila (query for rows) mapeados a un objeto POJO.
- **query**: El método query está excesivamente sobrecargado debido a que internamente JdbcTemplate utiliza la mayoría de los métodos query.

iv. Spring JDBC - Transaction - iv.iv JdbcTemplate

iv.iv JdbcTemplate (n)

- `query(PreparedStatementCreator psc, ResultSetExtractor<T> rse) : T - JdbcTemplate`
- `query(PreparedStatementCreator psc, RowCallbackHandler rch) : void - JdbcTemplate`
- `query(PreparedStatementCreator psc, RowMapper<T> rowMapper) : List<T> - JdbcTemplate`
- `query(String sql, ResultSetExtractor<T> rse) : T - JdbcTemplate`
- `query(String sql, RowCallbackHandler rch) : void - JdbcTemplate`
- `query(String sql, RowMapper<T> rowMapper) : List<T> - JdbcTemplate`
- `query(PreparedStatementCreator psc, PreparedStatementSetter pss, ResultSetExtractor<T> rse) : T - JdbcTemplate`
- `query(String sql, Object[] args, ResultSetExtractor<T> rse) : T - JdbcTemplate`
- `query(String sql, Object[] args, RowCallbackHandler rch) : void - JdbcTemplate`
- `query(String sql, Object[] args, RowMapper<T> rowMapper) : List<T> - JdbcTemplate`
- `query(String sql, PreparedStatementSetter pss, ResultSetExtractor<T> rse) : T - JdbcTemplate`
- `query(String sql, PreparedStatementSetter pss, RowCallbackHandler rch) : void - JdbcTemplate`
- `query(String sql, PreparedStatementSetter pss, RowMapper<T> rowMapper) : List<T> - JdbcTemplate`
- `query(String sql, ResultSetExtractor<T> rse, Object... args) : T - JdbcTemplate`
- `query(String sql, RowCallbackHandler rch, Object... args) : void - JdbcTemplate`
- `query(String sql, RowMapper<T> rowMapper, Object... args) : List<T> - JdbcTemplate`
- `query(String sql, Object[] args, int[] argTypes, ResultSetExtractor<T> rse) : T - JdbcTemplate`
- `query(String sql, Object[] args, int[] argTypes, RowCallbackHandler rch) : void - JdbcTemplate`
- `query(String sql, Object[] args, int[] argTypes, RowMapper<T> rowMapper) : List<T> - JdbcTemplate`

iv. Spring JDBC - Transaction - iv.iv JdbcTemplate

iv.iv JdbcTemplate (o)

| |
|--|
| |
| |
| ● query(String sql, RowMapper<T> rowMapper) : List<T> - JdbcTemplate |
| |
| |
| ● query(String sql, Object[] args, RowMapper<T> rowMapper) : List<T> - JdbcTemplate |
| |
| |
| ● query(String sql, RowMapper<T> rowMapper, Object... args) : List<T> - JdbcTemplate |
| |

iv. Spring JDBC - Transaction - iv.iv JdbcTemplate

iv.iv JdbcTemplate (p)

- Select query: Consultas por cero, una, o más de una fila (query for rows) mapeados a un objeto POJO.
- **query**: Por lo regular se utiliza la implementación que recibe una instancia de **RowMapper<T>** pues es la implementación de más alto nivel y la que menor implementación de código JDBC requiere, por tanto la más fácil de implementar.
- La implementación que utiliza instancias de **ResultSetExtractor<T>** y **RowCallbackHandler** si bien no es común utilizarlas, si son útiles en determinados casos.

iv. Spring JDBC - Transaction - iv.iv JdbcTemplate

iv.iv JdbcTemplate (q)

- **ResultSetExtractor<T>**: Procesa el ResultSet entero, debe iterarse en caso de que la consulta devuelva más de un único resultado mediante la llamada al método *next()*. Se delega total responsabilidad al desarrollador de tratar con el ResultSet entregado.

```
public interface ResultSetExtractor<T> {  
    T extractData(ResultSet rs) throws SQLException, DataAccessException;  
}
```

iv. Spring JDBC - Transaction - iv.iv JdbcTemplate

iv.iv JdbcTemplate (r)

- **RowCallbackHandler:** Procesa el resultado de cada una de las iteraciones de la invocación de *next()* sobre el *ResultSet*. El método *processRow* que define esta interface no devuelve objeto alguno pues esta pensada para *batch processing* así como para el procesamiento de información de la base de datos donde no es necesario obtener hasta el código *caller* los valores obtenidos, facilita el encapsulamiento de algoritmos que simplemente utilizan la información y concluye la ejecución.

```
public interface RowCallbackHandler {  
    void processRow(ResultSet rs) throws SQLException;  
}
```

iv. Spring JDBC - Transaction - iv.iv JdbcTemplate

iv.iv JdbcTemplate (s)

- Select query: Consultas por cero, una, o más de una fila (query for rows) mapeados a un objeto POJO mediante `RowMapper<T>`
- Obteniendo una lista de objetos POJO mediante `RowMapper<T>`

```
String SELECT = "SELECT * FROM ACCOUNT WHERE fk_customer_id = ?";
```

```
List<Account> accounts = jdbcTemplate.query(SELECT, new AccountRowMapper(), 1);
```

```
log.info("accounts: {}", accounts);
```

```
accounts: [Account(id=1, accountNumber=00112233445566, ...),  
           Account(id=2, accountNumber=00112233445577, ...)]
```

iv. Spring JDBC - Transaction - iv.iv JdbcTemplate

iv.iv JdbcTemplate (t)

- Select query: Consultas por cero, una, o más de una fila (query for rows) mapeados a un objeto POJO mediante `ResultSetExtractor<T>`
- Obteniendo una lista de objetos POJO mediante `ResultSetExtractor<T>`

iv. Spring JDBC - Transaction - iv.iv JdbcTemplate

iv.iv JdbcTemplate (u)

String SELECT = "SELECT * FROM ACCOUNT WHERE fk_customer_id = ?";

```
List<Account> accounts = jdbcTemplate.query(SELECT,  
                                           new ResultSetExtractor<List<Account>>() {  
@Override  
public List<Account> extractData(ResultSet rs) throws SQLException, DataAccessException {  
    List<Account> la = new ArrayList<>();  
    while (rs.next()) {  
        Account account = new Account(rs.getLong("ACCOUNT_ID"));  
        ...  
        la.add(account);  
    }  
    return la;  
}, 1);
```

iv. Spring JDBC - Transaction - iv.iv JdbcTemplate

iv.iv JdbcTemplate (v)

- Select query: Consultas por cero, una, o más de una fila (query for rows) sin mapeo mediante RowCallbackHandler
- Obteniendo y manejando resultados mediante RowCallbackHandler

iv. Spring JDBC - Transaction - iv.iv JdbcTemplate

iv.iv JdbcTemplate (w)

```
String SELECT = "SELECT * FROM ACCOUNT WHERE fk_customer_id = ?";
```

```
AccountXmlRowCallbackHandler xmlAccountCallbackHandler = new  
AccountXmlRowCallbackHandler();
```

```
jdbcTemplate.query(SELECT, xmlAccountCallbackHandler, 1);
```

```
xmlConverter.convertFromObjectToXML(xmlAccountCallbackHandler.getAccountList(),  
"account.xml");
```

```
log.info("account.xml: {}", xmlConverter.getXMLAsString("file:account.xml"));
```

iv. Spring JDBC - Transaction - iv.iv JdbcTemplate

iv.iv JdbcTemplate (x)

```
log.info("account.xml: {}", xmlConverter.getXMLAsString("file:account.xml"));
```

```
account.xml: <?xml version="1.0" encoding="UTF-8"?>
<array-list>
  <account xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:type="java:org.certificatic.spring.jdbc.domain.entities.Account">
    <created-date>2019-02-28T00:00:00.000-06:00</created-date>
    <balance>125590.5500</balance>
    <account-number>00112233445566</account-number>
    <customer>
      <id>1</id>
    </customer>
    <id>1</id>
  </account>
</account>
...
</account>
</array-list>
```

iv. Spring JDBC - Transaction - iv.iv JdbcTemplate

iv.iv JdbcTemplate (y)

```
public class AccountXmlRowCallbackHandler implements RowCallbackHandler {  
    private @Getter List<Account> accountList = new ArrayList<>();  
    private AccountRowMapper accountRowMapper = new AccountRowMapper();  
    private int i = 0;  
  
    @Override  
    public void processRow(ResultSet rs) throws SQLException {  
        Account account = accountRowMapper.mapRow(rs, i++);  
        accountList.add(account);  
    }  
  
    public void reset() {  
        this.i = 0;  
        this.accountList = new ArrayList<>();  
    }  
}
```

*xmlConverter.convertFromObjectToXML(
xmlAccountCallbackHandler.getAccountList(),
"account.xml");*

iv. Spring JDBC - Transaction - iv.iv JdbcTemplate

iv.iv JdbcTemplate

- a. Select query
- b. Update query (insert, update y delete)
- c. Execute query

Práctica 25 – Parte 2. Implementación IUserDAO

iv.iv JdbcTemplate (a)

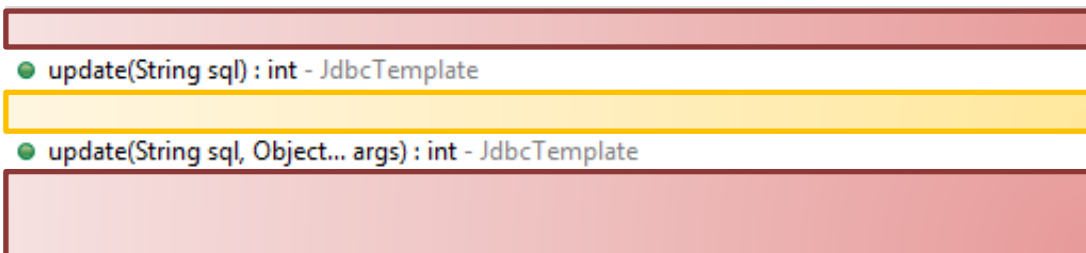
- Update query: Inserción, actualización y eliminación de filas.
- **update**: El método update cuenta con distintos métodos sobrecargados, todos devuelven el número de filas afectadas.

```
● update(PreparedStatementCreator psc) : int - JdbcTemplate  
● update(String sql) : int - JdbcTemplate  
● update(PreparedStatementCreator psc, KeyHolder generatedKeyHolder) : int - JdbcTemplate  
● update(String sql, Object... args) : int - JdbcTemplate  
● update(String sql, PreparedStatementSetter pss) : int - JdbcTemplate  
● update(String sql, Object[] args, int[] argTypes) : int - JdbcTemplate
```

iv. Spring JDBC - Transaction - iv.iv JdbcTemplate

iv.iv JdbcTemplate (b)

- Update query: Inserción, actualización y eliminación de filas.
- **update**: El método update cuenta con distintos métodos sobrecargados, todos devuelven el número de filas afectadas.



iv. Spring JDBC - Transaction - iv.iv JdbcTemplate

iv.iv JdbcTemplate (c)

- Update query: Inserción, actualización y eliminación de filas.
- Insertando una fila.

```
String INSERT = "INSERT INTO USER(username, password) VALUES (?, ?)";
```

```
int affectedRows = jdbcTemplate.update(INSERT, "user123", "mypass123");
```

iv. Spring JDBC - Transaction - iv.iv JdbcTemplate

iv.iv JdbcTemplate (d)

- Update query: Inserción, actualización y eliminación de filas.
- Actualizando una fila.

```
String UPDATE = "UPDATE USER SET username = ?, password = ? WHERE id = ?";
```

```
int affectedRows = jdbcTemplate.update(UPDATE, new Object[]{"user456", "mypass456", 1});
```

iv. Spring JDBC - Transaction - iv.iv JdbcTemplate

iv.iv JdbcTemplate (e)



- Update query: Inserción, actualización y eliminación de filas.
- Eliminando una fila.

```
String DELETE = "DELETE FROM USER WHERE username = ?";
```

```
int affectedRows = jdbcTemplate.update(DELETE, "user456");
```

iv. Spring JDBC - Transaction - iv.iv JdbcTemplate

iv.iv JdbcTemplate (f)

- En cualquier ORM la forma de comprobar que un registro a sido insertando es verificando que el identificador o llave primaria de la entidad no es nula y mayor que cero.
- ¿Cómo podemos simular este comportamiento en DAOs implementados con Spring JDBC?
 - Insertamos y leemos el último insertado 
¿Cómo sabremos que id fue el último insertado?
 - Insertamos e indicamos que necesitamos el `RETURN_GENERATED_KEYS` 

iv. Spring JDBC - Transaction - iv.iv JdbcTemplate

iv.iv JdbcTemplate (g)

- **PreparedStatementCreator:** Permite personalizar la creación de un PreparedStatement sobre una conexión dada. Esta interface encapsula la construcción de un PreparedStatement para un origen de datos específico, si bien no es común implementarla, es necesaria para poder obtener los RETURN_GENERATED_KEYS.

```
public interface PreparedStatementCreator {  
    PreparedStatement createPreparedStatement(Connection con) throws SQLException;  
}
```

iv. Spring JDBC - Transaction - iv.iv JdbcTemplate

iv.iv JdbcTemplate (h)

- Insertando una fila obteniendo las llaves generadas.

```
final String INSERT = "INSERT INTO USER(username, password) VALUES (?, ?)";
KeyHolder keyHolder = new GeneratedKeyHolder();

int affectedRows = jdbcTemplate.update(INSERT, (Connection conn) -> {
    PreparedStatement ps = conn.prepareStatement(INSERT,
                                                new String[] { "CUSTOMER_ID", "NAME", "LAST_NAME" });
    ps.setString(1, "user123");
    ps.setString(2, "mypass123");
    return ps;
}, keyHolder);

log.info("id: {}", keyHolder.getKey().longValue());
```

iv. Spring JDBC - Transaction - iv.iv JdbcTemplate

iv.iv JdbcTemplate (i)

- Insertando una fila obteniendo las llaves generadas.
- Para obtener las llaves generadas, usando JdbcTemplate, es necesario definir el PreparedStatement facilitando el nombre de las columnas de la tabla.
- **Contras:** Mucho detalle de JDBC a bajo nivel, se pierde la intención del template JdbcTemplate.
- **Solución:** NamedParameterJdbcTemplate

iv. Spring JDBC - Transaction - iv.iv JdbcTemplate

iv.iv JdbcTemplate (j)

- **PreparedStatementSetter:** Permite asignar al PreparedStatement los valores de los parámetros requeridos para ejecutar la consulta.

```
public interface PreparedStatementSetter {  
    void setValues(PreparedStatement ps) throws SQLException;  
}
```

iv. Spring JDBC - Transaction - iv.iv JdbcTemplate

iv.iv JdbcTemplate (h)

- Insertando una fila asignando valores directamente sobre el PreparedStatement usando PreparedStatementSetter.

```
final String INSERT = "INSERT INTO USER(username, password) VALUES (?, ?)";

int affectedRows = jdbcTemplate.update(INSERT, new PreparedStatementSetter() {
    @Override
    public void setValues(PreparedStatement ps) throws SQLException {
        ps.setString(1, "user123");
        ps.setString(2, "mypass123");
    }
});
```

iv. Spring JDBC - Transaction - iv.iv JdbcTemplate

iv.iv JdbcTemplate

- a. Select query
- b. Update query (insert, update y delete)
- c. Execute query

Práctica 25 – Parte 2. Implementación IUserDAO

iv.iv JdbcTemplate (a)

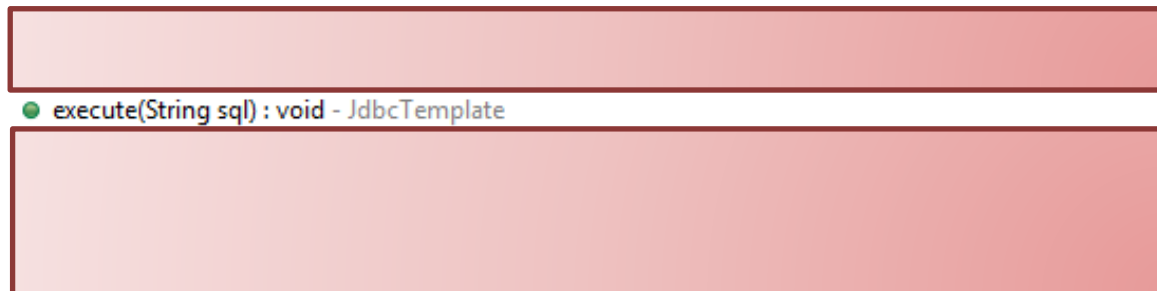
- Execute query: Ejecución de cualquier sentencia SQL.
- **execute**: El método execute cuenta con una exhaustiva definición de métodos sobrecargados, todos devuelven el número de filas afectadas. Por lo regular se utiliza para ejecutar instrucciones DDL.

```
● execute(ConnectionCallback<T> action) : T - JdbcTemplate  
● execute(StatementCallback<T> action) : T - JdbcTemplate  
● execute(String sql) : void - JdbcTemplate  
● execute(CallableStatementCreator csc, CallableStatementCallback<T> action) : T - JdbcTemplate  
● execute(PreparedStatementCreator psc, PreparedStatementCallback<T> action) : T - JdbcTemplate  
● execute(String callString, CallableStatementCallback<T> action) : T - JdbcTemplate  
● execute(String sql, PreparedStatementCallback<T> action) : T - JdbcTemplate
```

iv. Spring JDBC - Transaction - iv.iv JdbcTemplate

iv.iv JdbcTemplate (b)

- Execute query: Ejecución de cualquier sentencia SQL.
- **execute**: El método execute cuenta con una exhaustiva definición de métodos sobrecargados, todos devuelven el número de filas afectadas. Por lo regular se utiliza para ejecutar instrucciones DDL.



iv. Spring JDBC - Transaction - iv.iv JdbcTemplate

iv.iv JdbcTemplate (c)

- Execute query: Ejecución de cualquier sentencia SQL.
- Creando una tabla

```
String CREATE_TABLE = "CREATE TABLE USER ( USER_ID integer identity primary key, "+  
    "USERNAME varchar(100) not null, "+  
    "PASSWORD varchar(100) not null )";
```

```
jdbcTemplate.execute(CREATE_TABLE);
```

iv. Spring JDBC - Transaction - iv.iv JdbcTemplate

iv.iv JdbcTemplate (d)

- Execute query: Ejecución de cualquier sentencia SQL.
- Internamente JdbcTemplate llama al método execute en muchos casos, por ejemplo al invocar al método **queryForObject**.
- **Contras:** No se recomienda el uso del método execute, debido a que su implementación requiere de implementación de interfaces de bajo nivel mismas que implementa JdbcTemplate por nosotros, se pierde la intención del template.
- **Pros:** Permite la ejecución de sentencias DDL.

iv. Spring JDBC - Transaction - iv.iv JdbcTemplate

iv.iv JdbcTemplate (e)

- Diferencias entre execute y update.
- **execute** NO regresa la cantidad de filas afectadas (rows affected)
- **update** SI regresa la cantidad de filas afectadas (rows affected)
- **execute** utiliza Statement para la ejecución de consultas.
- **update** utiliza PreparedStatement para la ejecución de consultas.
- **execute** se recomienda para ejecutar sentencias DDL.
- **update** se recomienda para ejecutar sentencias insert, update y delete.

iv. Spring JDBC - Transaction - iv.iv JdbcTemplate

iv.iv JdbcTemplate

- a. Select query
- b. Update query (insert, update y delete)
- c. Execute query

Práctica 25 – Parte 2. Implementación IUserDAO

iv.iv JdbcTemplate. Práctica 25. (a)

- Práctica 25 – Parte 2. Implementación IUserDAO.
- Implementar la funcionalidad CRUD básica de la interface IUserDAO mediante la aplicación de JdbcTemplate.
- Realizar pruebas funcionales en los 3 diferentes ambientes de base de datos.

iv. Spring JDBC - Transaction - iv.iv JdbcTemplate

Resumen de la lección

iv.iv JdbcTemplate

- Conocimos los principales métodos que provee la clase JdbcTemplate a fondo.
- Analizamos y comprendimos el funcionamiento de JdbcTemplate mediante la llamada a callbacks.
- Verificamos la funcionalidad del callback RowMapper<T>
- Verificamos la correcta configuración de JdbcTemplate en beans DAO.
- Comprendimos el uso de JdbcTemplate para realizar instrucciones SELECT, INSERT, UPDATE y DELETE.
- Analizamos la manera correcta de ejecutar instrucciones DDL.

iv. Spring JDBC - Transaction - iv.iv JdbcTemplate

Esta página fue intencionalmente dejada en blanco.

iv. Spring JDBC - Transaction - iv.iv JdbcTemplate

iv.v NamedParameterJdbcTemplate

Objetivos de la lección

iv.v NamedParameterJdbcTemplate

- Conocer la clase NamedParameterJdbcTemplate a fondo.
- Conocer como enviar parámetros a consultas con parámetros nombrados.
- Conocer la funcionalidad del callback ResultSetExtractor<T> y RowCallbackHandler.
- Comprender el uso de NamedParameterJdbcTemplate para realizar instrucciones SELECT, INSERT, UPDATE y DELETE.

iv. Spring JDBC - Transaction - iv.v NamedParameterJdbcTemplate

iv.v NamedParameterJdbcTemplate

- a. Select query
- b. Update query (insert, update y delete)
- c. Execute query

Práctica 25 – Parte 3. Implementación ICustomerDAO

iv.v NamedParameterJdbcTemplate (a)

- Provee funcionalidad para realizar consultas JDBC mediante el uso de nombres de parámetros, en lugar de placeholders como JdbcTemplate.
- Encapsula un objeto JdbcTemplate y delega mucho del trabajo a él.

iv. Spring JDBC - Transaction - iv.v NamedParameterJdbcTemplate

iv.v NamedParameterJdbcTemplate

- a. **Select query**
- b. Update query (insert, update y delete)
- c. Execute query

Práctica 25 – Parte 3. Implementación ICustomerDAO

iv.v NamedParameterJdbcTemplate (a)

- Select query: Consultas por una única fila (query for single row).
- **queryForObject**: Retorna una sola columna o un solo objeto, este método espera que exista un único resultado en la consulta. Si se obtienen cero o más de un resultado (fila) lanza excepción **EmptyResultDataAccessException** o **IncorrectResultSizeDataAccessException** respectivamente.

iv. Spring JDBC - Transaction - iv.v NamedParameterJdbcTemplate

iv.v NamedParameterJdbcTemplate (b)

- Select query: Consultas por una única fila (query for single row).
- Obteniendo una única columna (a).

```
String SELECT = "SELECT username FROM USER WHERE id = :userId";
```

```
Map<String, Object> map = new HashMap<>();  
map.put("userId", id);
```

```
String username = namedParameterJdbcTemplate.queryForObject(SELECT, map, String.class);
```

iv. Spring JDBC - Transaction - iv.v NamedParameterJdbcTemplate

iv.v NamedParameterJdbcTemplate (c)

- Select query: Consultas por una única fila (query for single row).
- Obteniendo una única columna (b).

```
String SELECT = "SELECT username FROM USER WHERE id = :userId";
```

```
SqlParameterSource paramSource = new MapSqlParameterSource(). addValue("userId", id);
```

```
String username = namedParameterJdbcTemplate.queryForObject(SELECT, paramSource,  
                                                             String.class);
```

iv. Spring JDBC - Transaction - iv.v NamedParameterJdbcTemplate

iv.v NamedParameterJdbcTemplate (d)

- Select query: Consultas por una única fila (query for single row).
- Obteniendo más de una columna en un objeto POJO(a)(RowMapper<T>)

```
SqlParameterSource paramSource = new MapSqlParameterSource(). addValue("userId", id);
```

```
User user = namedParameterJdbcTemplate.queryForObject(  
    "SELECT * FROM USER WHERE id = :userId", paramSource, (rs, n) -> {  
        User usr = new User();  
        usr.setId(rs.getLong("USER_ID"));  
        usr.setUsername(rs.getString("USERNAME"));  
        usr.setPassword(rs.getString("PASSWORD"));  
        return usr;  
    });
```

iv. Spring JDBC - Transaction - iv.v NamedParameterJdbcTemplate

iv.v NamedParameterJdbcTemplate (e)

- Select query: Consultas por una única fila (query for single row).
- Obteniendo más de una columna en un objeto POJO(b)(RowMapper<T>)

```
Map<String, Object> map = new HashMap<>();  
map.put("userId", id);
```

```
User user = namedParameterJdbcTemplate.queryForObject(  
"SELECT * FROM USER WHERE id = :userId", map, new UserRowMapper());
```

iv. Spring JDBC - Transaction - iv.v NamedParameterJdbcTemplate

iv.v NamedParameterJdbcTemplate (f)

```
public class UserRowMapper implements RowMapper<User> {  
  
    @Override  
    public User mapRow(ResultSet rs, int rowNum) throws SQLException {  
        User usr = new User();  
        usr.setId(rs.getLong("USER_ID"));  
        usr.setUsername(rs.getString("USERNAME"));  
        usr.setPassword(rs.getString("PASSWORD"));  
        return usr;  
    }  
}
```

iv. Spring JDBC - Transaction - iv.v NamedParameterJdbcTemplate

iv.v NamedParameterJdbcTemplate (g)

- Select query: Consultas por una única fila (query for single row).
- **queryForMap**: Retorna un solo resultado independientemente si contiene una o más columnas en un objeto Map<String, Object>, este método espera que exista un único resultado en la consulta. Si se obtienen cero o más de un resultado (fila) lanza excepción **EmptyResultDataAccessException** o **IncorrectResultSizeDataAccessException** respectivamente.

iv. Spring JDBC - Transaction - iv.v NamedParameterJdbcTemplate

iv.v NamedParameterJdbcTemplate (h)

- Select query: Consultas por una única fila (query for single row).
- Obteniendo una o más columnas en un Map<String, Object> (a).

String SELECT = "SELECT username, password FROM USER WHERE id = :userId";

```
Map<String, Object> map = new HashMap<>();  
map.put("userId", id);
```

```
Map<String, Object> mapResult = namedParameterJdbcTemplate.queryForMap(SELECT, map);
```

iv. Spring JDBC - Transaction - iv.v NamedParameterJdbcTemplate

iv.v NamedParameterJdbcTemplate (i)

- Select query: Consultas por una única fila (query for single row).
- Obteniendo una o más columnas en un Map<String, Object> (b).

String SELECT = "SELECT username, password FROM USER WHERE id = :userId";

SqlParameterSource paramSource = new MapSqlParameterSource(). addValue("userId", id);

Map<String, Object> mapResult = namedParameterJdbcTemplate.queryForMap(SELECT,
paramSource);

iv. Spring JDBC - Transaction - iv.v NamedParameterJdbcTemplate

iv.v NamedParameterJdbcTemplate (j)

- Select query: Consultas por cero, una, o más de una fila (query for rows).
- **queryForList**: Retorna una lista del tipo especificado cuando se espera una única columna y se especifica el tipo de la columna, o retorna una lista de Map<String, Object> si el resultado de la consulta devuelve una o más columnas cuando, para el caso de devolver una columna, no se especifica el tipo.

iv. Spring JDBC - Transaction - iv.v NamedParameterJdbcTemplate

iv.v NamedParameterJdbcTemplate (k)

- Select query: Consultas por cero, una, o más de una fila (query for rows).
- Obteniendo una lista de una única columna, especificando el tipo (a).

```
String SELECT = "SELECT username FROM USER WHERE id > :userId";
```

```
Map<String, Object> map = new HashMap<>();  
map.put("userId", id);
```

```
List<String> usernames = namedParameterJdbcTemplate.queryForList(SELECT, map,  
                                                                    String.class);
```

iv. Spring JDBC - Transaction - iv.v NamedParameterJdbcTemplate

iv.v NamedParameterJdbcTemplate (I)

- Select query: Consultas por cero, una, o más de una fila (query for rows).
- Obteniendo una lista de una única columna, especificando el tipo (b).

```
String SELECT = "SELECT username FROM USER WHERE id > :userId";
```

```
SqlParameterSource paramSource = new MapSqlParameterSource(). addValue("userId", id);
```

```
List<String> usernames = namedParameterJdbcTemplate.queryForList(SELECT, paramSource,  
                                                                    String.class);
```

iv. Spring JDBC - Transaction - iv.v NamedParameterJdbcTemplate

iv.v NamedParameterJdbcTemplate (m)

- Select query: Consultas por cero, una, o más de una fila (query for rows).
- Obteniendo una lista de una única columna, sin especificar el tipo (a).

```
String SELECT = "SELECT username FROM USER WHERE id > :userId";
```

```
Map<String, Object> map = new HashMap<>();  
map.put("userId", id);
```

```
List<Map<String, Object>> usernames = namedParameterJdbcTemplate.queryForList(SELECT,  
                                                                                   map);
```

```
// [{USERNAME=xvanhalenx}, {USERNAME=laura123}]
```

iv. Spring JDBC - Transaction - iv.v NamedParameterJdbcTemplate

iv.v NamedParameterJdbcTemplate (n)

- Select query: Consultas por cero, una, o más de una fila (query for rows).
- Obteniendo una lista de una única columna, sin especificar el tipo (b).

```
String SELECT = "SELECT username FROM USER WHERE id > :userId";
```

```
SqlParameterSource paramSource = new MapSqlParameterSource(). addValue("userId", id);
```

```
List<Map<String, Object>> usernames = namedParameterJdbcTemplate.queryForList(SELECT,  
                                                                              paramSource);
```

```
// [{USERNAME=xvanhalenx}, {USERNAME=laura123}]
```

iv. Spring JDBC - Transaction - iv.v NamedParameterJdbcTemplate

iv.v NamedParameterJdbcTemplate (o)

- Select query: Consultas por cero, una, o más de una fila (query for rows).
- Obteniendo una lista de una o más columnas (a).

```
String SELECT = "SELECT username, password FROM USER WHERE id > :userId";
```

```
Map<String, Object> map = new HashMap<>();  
map.put("userId", id);
```

```
List<Map<String, Object>> usernameAndPass = namedParameterJdbcTemplate.queryForList(  
    SELECT, map);
```

```
// [{USERNAME=xvanhalenx, PASSWORD=123123},  
    {USERNAME=laura123, PASSWORD=123123}]
```

iv. Spring JDBC - Transaction - iv.v NamedParameterJdbcTemplate

iv.v NamedParameterJdbcTemplate (p)

- Select query: Consultas por cero, una, o más de una fila (query for rows).
- Obteniendo una lista de una o más columnas (b).

```
String SELECT = "SELECT username, password FROM USER WHERE id > :userId";
```

```
SqlParameterSource paramSource = new MapSqlParameterSource(). addValue("userId", id);
```

```
List<Map<String, Object>> usernameAndPass = namedParameterJdbcTemplate.queryForList(  
    SELECT, paramSource);
```

```
// [{USERNAME=xvanhalenx, PASSWORD=123123},  
    {USERNAME=laura123, PASSWORD=123123}]
```

iv. Spring JDBC - Transaction - iv.v NamedParameterJdbcTemplate

iv.v NamedParameterJdbcTemplate (q)

- Select query: Consultas por cero, una, o más de una fila (query for rows).
- **query**: El método query del objeto NamedParameterJdbcTemplate permite recuperar resultados mediante diferentes estrategias tal como ya se han analizado para el objeto JdbcTemplate.

```
● query(String sql, ResultSetExtractor<T> rse) : T - NamedParameterJdbcTemplate  
● query(String sql, RowCallbackHandler rch) : void - NamedParameterJdbcTemplate  
● query(String sql, RowMapper<T> rowMapper) : List<T> - NamedParameterJdbcTemplate  
● query(String sql, Map<String,?> paramMap, ResultSetExtractor<T> rse) : T - NamedParameterJdbcTemplate  
● query(String sql, Map<String,?> paramMap, RowCallbackHandler rch) : void - NamedParameterJdbcTemplate  
● query(String sql, Map<String,?> paramMap, RowMapper<T> rowMapper) : List<T> - NamedParameterJdbcTemplate  
● query(String sql, SqlParameterSource paramSource, ResultSetExtractor<T> rse) : T - NamedParameterJdbcTemplate  
● query(String sql, SqlParameterSource paramSource, RowCallbackHandler rch) : void - NamedParameterJdbcTemplate  
● query(String sql, SqlParameterSource paramSource, RowMapper<T> rowMapper) : List<T> - NamedParameterJdbcTemplate
```

iv. Spring JDBC - Transaction - iv.v NamedParameterJdbcTemplate

iv.v NamedParameterJdbcTemplate (r)

- Select query: Consultas por cero, una, o más de una fila (query for rows).
- **query**: El método query del objeto NamedParameterJdbcTemplate permite recuperar resultados mediante diferentes estrategias tal como ya se han analizado para el objeto JdbcTemplate.

● `query(String sql, RowMapper<T> rowMapper) : List<T> - NamedParameterJdbcTemplate`

● `query(String sql, Map<String, ?> paramMap, RowMapper<T> rowMapper) : List<T> - NamedParameterJdbcTemplate`

● `query(String sql, SqlParameterSource paramSource, RowMapper<T> rowMapper) : List<T> - NamedParameterJdbcTemplate`

iv. Spring JDBC - Transaction - iv.v NamedParameterJdbcTemplate

iv.v NamedParameterJdbcTemplate (s)

- Select query: Consultas por cero, una, o más de una fila (query for rows) mapeados a un objeto POJO.
- **query**: Por lo regular se utiliza la implementación que recibe una instancia de **RowMapper<T>** pues es la implementación de más alto nivel y la que menor implementación de código JDBC requiere, por tanto la más fácil de implementar.
- La implementación que utiliza instancias de **ResultSetExtractor<T>** y **RowCallbackHandler** si bien no es común utilizarlas, si son útiles en determinados casos.

iv. Spring JDBC - Transaction - iv.v NamedParameterJdbcTemplate

iv.v NamedParameterJdbcTemplate (t)

- **ResultSetExtractor<T>**: Procesa el ResultSet entero, debe iterarse en caso de que la consulta devuelva más de un único resultado mediante la llamada al método *next()*. Se delega total responsabilidad al desarrollador de tratar con el ResultSet entregado.

```
public interface ResultSetExtractor<T> {  
    T extractData(ResultSet rs) throws SQLException, DataAccessException;  
}
```

iv. Spring JDBC - Transaction - iv.v NamedParameterJdbcTemplate

iv.v NamedParameterJdbcTemplate (u)

- **RowCallbackHandler:** Procesa el resultado de cada una de las iteraciones de la invocación de *next()* sobre el ResultSet. El método *processRow* que define esta interface no devuelve objeto alguno pues esta pensada para *batch processing* así como para el procesamiento de información de la base de datos donde no es necesario obtener hasta el código *caller* los valores obtenidos, facilita el encapsulamiento de algoritmos que simplemente utilizan la información y concluye la ejecución.

```
public interface RowCallbackHandler {  
    void processRow(ResultSet rs) throws SQLException;  
}
```

iv. Spring JDBC - Transaction - iv.v NamedParameterJdbcTemplate

iv.v NamedParameterJdbcTemplate (v)

- Select query: Consultas por cero, una, o más de una fila (query for rows) mapeados a un objeto POJO mediante RowMapper<T>
- Obteniendo una lista de objetos POJO mediante RowMapper<T> (a)

```
String SELECT = "SELECT * FROM ACCOUNT WHERE fk_customer_id = :fk_customerId";
```

```
Map<String, Object> map = new HashMap<>();
```

```
map.put("fk_customerId", id);
```

```
List<Account> accounts = namedParameterJdbcTemplate.query(SELECT, map,  
                                                             new AccountRowMapper());
```

```
//Account(id=1, accountNumber=00112233445566, ...),
```

```
Account(id=2, accountNumber=00112233445577, ...)]
```

iv. Spring JDBC - Transaction - iv.v NamedParameterJdbcTemplate

iv.v NamedParameterJdbcTemplate (w)

- Select query: Consultas por cero, una, o más de una fila (query for rows) mapeados a un objeto POJO mediante RowMapper<T>
- Obteniendo una lista de objetos POJO mediante RowMapper<T> (b)

```
String SELECT = "SELECT * FROM ACCOUNT WHERE fk_customer_id = :fk_customerId";
```

```
SqlParameterSource paramSource = new MapSqlParameterSource(). addValue(  
                                                                    "fk_customerId", id);  
List<Account> accounts = namedParameterJdbcTemplate.query(SELECT, paramSource,  
                                                            new AccountRowMapper());  
//Account(id=1, accountNumber=00112233445566, ...),  
Account(id=2, accountNumber=00112233445577, ...)]
```

iv. Spring JDBC - Transaction - iv.v NamedParameterJdbcTemplate

iv.v NamedParameterJdbcTemplate (x)

- Select query: Consultas por cero, una, o más de una fila (query for rows) mapeados a un objeto POJO mediante `ResultSetExtractor<T>`
- Obteniendo una lista de objetos POJO mediante `ResultSetExtractor<T>`
- El envío de parámetros al query es indistintamente mediante `Map<String, Object>` o `SqlParameterSource`.

iv. Spring JDBC - Transaction - iv.v NamedParameterJdbcTemplate

iv.v NamedParameterJdbcTemplate (y)

String SELECT = "SELECT * FROM ACCOUNT WHERE fk_customer_id = :fk_customerId";

```
List<Account> accounts = namedParameterJdbcTemplate.query(SELECT, paramSource
    new ResultSetExtractor<List<Account>>() {
    @Override
    public List<Account> extractData(ResultSet rs) throws SQLException, DataAccessException {
        List<Account> la = new ArrayList<>();
        while (rs.next()) {
            Account account = new Account(rs.getLong("ACCOUNT_ID"));
            ...
            la.add(account);
        }
        return la;
    }
});
```

iv. Spring JDBC - Transaction - iv.v NamedParameterJdbcTemplate

iv.v NamedParameterJdbcTemplate (z)

- Select query: Consultas por cero, una, o más de una fila (query for rows) sin mapeo mediante RowCallbackHandler
- Obteniendo y manejando resultados mediante RowCallbackHandler
- El envío de parámetros al query es indistintamente mediante Map<String, Object> o SqlParameterSource.

iv. Spring JDBC - Transaction - iv.v NamedParameterJdbcTemplate

iv.v NamedParameterJdbcTemplate (a')

```
String SELECT = "SELECT * FROM ACCOUNT WHERE fk_customer_id = :fk_customerId";
```

```
AccountXmlRowCallbackHandler xmlAccountCallbackHandler = new  
AccountXmlRowCallbackHandler();
```

```
namedParameterJdbcTemplate.query(SELECT, map, xmlAccountCallbackHandler);
```

```
xmlConverter.convertFromObjectToXML(xmlAccountCallbackHandler.getAccountList(),  
"account.xml");
```

```
log.info("account.xml: {}", xmlConverter.getXMLAsString("file:account.xml"));
```

iv. Spring JDBC - Transaction - iv.v NamedParameterJdbcTemplate

iv.v NamedParameterJdbcTemplate (b')

```
log.info("account.xml: {}", xmlConverter.getXMLAsString("file:account.xml"));
```

```
account.xml: <?xml version="1.0" encoding="UTF-8"?>
```

```
<array-list>
```

```
  <account xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
    xsi:type="java:org.certificatic.spring.jdbc.domain.entities.Account">
```

```
    <created-date>2019-02-28T00:00:00.000-06:00</created-date>
```

```
    <balance>125590.5500</balance>
```

```
    <account-number>00112233445566</account-number>
```

```
    <customer>
```

```
      <id>1</id>
```

```
    </customer>
```

```
    <id>1</id>
```

```
  </account>
```

```
</account>
```

```
...
```

```
</account>
```

```
</array-list>
```

iv. Spring JDBC - Transaction - iv.v NamedParameterJdbcTemplate

iv.v NamedParameterJdbcTemplate (c')

```
public class AccountXmlRowCallbackHandler implements RowCallbackHandler {
    private @Getter List<Account> accountList = new ArrayList<>();
    private AccountRowMapper accountRowMapper = new AccountRowMapper();
    private int i = 0;

    @Override
    public void processRow(ResultSet rs) throws SQLException {
        Account account = accountRowMapper.mapRow(rs, i++);
        accountList.add(account);
    }

    public void reset() {
        this.i = 0;
        this.accountList = new ArrayList<>();
    }
}
```

*xmlConverter.convertFromObjectToXML(
xmlAccountCallbackHandler.getAccountList(),
"account.xml");*

iv. Spring JDBC - Transaction - iv.v NamedParameterJdbcTemplate

iv.v NamedParameterJdbcTemplate

- a. Select query
- b. Update query (insert, update y delete)**
- c. Execute query

Práctica 25 – Parte 3. Implementación ICustomerDAO

iv.v NamedParameterJdbcTemplate (a)

- Update query: Inserción, actualización y eliminación de filas.
- **update**: El método update cuenta con distintos métodos sobrecargados, todos devuelven el número de filas afectadas.
- El envío de parámetros al query es indistintamente mediante Map<String, Object> o SqlParameterSource.

```
● update(String sql, Map<String,?> paramMap) : int - NamedParameterJdbcTemplate  
● update(String sql, SqlParameterSource paramSource) : int - NamedParameterJdbcTemplate  
● update(String sql, SqlParameterSource paramSource, KeyHolder generatedKeyHolder) : int - NamedParameterJdbcTemplate  
● update(String sql, SqlParameterSource paramSource, KeyHolder generatedKeyHolder, String[] keyColumnNames) : int - NamedParameterJdbcTemplate
```

iv. Spring JDBC - Transaction - iv.v NamedParameterJdbcTemplate

iv.v NamedParameterJdbcTemplate (b)

- Update query: Inserción, actualización y eliminación de filas.
- **update**: El método update cuenta con distintos métodos sobrecargados, todos devuelven el número de filas afectadas.
- El envío de parámetros al query es indistintamente mediante Map<String, Object> o SqlParameterSource.

```
● update(String sql, Map<String,?> paramMap) : int - NamedParameterJdbcTemplate  
● update(String sql, SqlParameterSource paramSource) : int - NamedParameterJdbcTemplate
```

iv. Spring JDBC - Transaction - iv.v NamedParameterJdbcTemplate

iv.v NamedParameterJdbcTemplate (c)

- Update query: Inserción, actualización y eliminación de filas.
- Insertando una fila.

```
String INSERT = "INSERT INTO USER(username, password) VALUES (:username, :password)";
```

```
int affectedRows = namedParameterJdbcTemplate.update(INSERT, map);
```

```
int affectedRows = namedParameterJdbcTemplate.update(INSERT, paramSource);
```

iv. Spring JDBC - Transaction - iv.v NamedParameterJdbcTemplate

iv.v NamedParameterJdbcTemplate (d)

- Update query: Inserción, actualización y eliminación de filas.
- Actualizando una fila.

```
String UPDATE = "UPDATE USER SET username = :username, password = :password  
WHERE id = :id";
```

```
int affectedRows = namedParameterJdbcTemplate.update(UPDATE, map);
```

```
int affectedRows = namedParameterJdbcTemplate.update(UPDATE, paramSource);
```

iv. Spring JDBC - Transaction - iv.v NamedParameterJdbcTemplate

iv.v NamedParameterJdbcTemplate (e)

- Update query: Inserción, actualización y eliminación de filas.
- Eliminando una fila.

```
String DELETE = "DELETE FROM USER WHERE username = :username";
```

```
int affectedRows = namedParameterJdbcTemplate.update(DELETE, map);
```

```
int affectedRows = namedParameterJdbcTemplate.update(DELETE, paramSource);
```

iv. Spring JDBC - Transaction - iv.v NamedParameterJdbcTemplate

iv.v NamedParameterJdbcTemplate (f)

- Insertando una fila obteniendo las llaves generadas.

```
String INSERT = "INSERT INTO USER(username, password) VALUES (:username, :password)";
```

```
KeyHolder keyHolder = new GeneratedKeyHolder();
```

```
int affectedRows = namedParameterJdbcTemplate.update(INSERT, paramSource, keyHolder);
```

```
int affectedRows = namedParameterJdbcTemplate.update(INSERT, paramSource, keyHolder,  
new String[]{ "USER_ID", "ACCOUNT_ID" } );
```

```
log.info("id: {}", keyHolder.getKey().longValue())
```

iv. Spring JDBC - Transaction - iv.v NamedParameterJdbcTemplate

iv.v NamedParameterJdbcTemplate

- a. Select query
- b. Update query (insert, update y delete)
- c. Execute query

Práctica 25 – Parte 3. Implementación ICustomerDAO

iv.v NamedParameterJdbcTemplate (a)

- Execute query: Ejecución de cualquier sentencia SQL.
- **execute**: El método execute cuenta con tres métodos sobre cargados los cuales reciben como argumento una instancia de PreparedStatementCallback<T>. Por lo regular el método execute se utiliza para ejecutar instrucciones DDL, sin embargo NamedParameterJdbcTemplate no implementa una forma simple de ejecutar instrucciones DDL, se recomienda usar JdbcTemplate.

```
● execute(String sql, PreparedStatementCallback<T> action) : T - NamedParameterJdbcTemplate  
● execute(String sql, Map<String,?> paramMap, PreparedStatementCallback<T> action) : T - NamedParameterJdbcTemplate  
● execute(String sql, SqlParameterSource paramSource, PreparedStatementCallback<T> action) : T - NamedParameterJdbcTemplate
```

iv. Spring JDBC - Transaction - iv.v NamedParameterJdbcTemplate

iv.v NamedParameterJdbcTemplate (b)

- Execute query: Ejecución de cualquier sentencia SQL.
- **execute**: El método execute cuenta con tres métodos sobre cargados los cuales reciben como argumento una instancia de PreparedStatementCallback<T>. Por lo regular el método execute se utiliza para ejecutar instrucciones DDL, sin embargo NamedParameterJdbcTemplate no implementa una forma simple de ejecutar instrucciones DDL, se recomienda usar JdbcTemplate.



iv. Spring JDBC - Transaction - iv.v NamedParameterJdbcTemplate

iv.v NamedParameterJdbcTemplate (c)

- **PreparedStatementCallback<T>**: Delega al desarrollador toda la responsabilidad de asignar los parámetros al PreparedStatement y de ejecutar la consulta sobre el PreparedStatement previamente ya creado. Es una interface de muy bajo nivel ya que JdbcTemplate la utiliza sin embargo no se recomienda su uso pues se pierde el sentido de los templates NamedParameterJdbcTemplate y JdbcTemplate. El valor genérico es el valor de retorno de la ejecución.

```
public interface PreparedStatementCallback<T> {  
    T doInPreparedStatement(PreparedStatement ps) throws SQLException,  
                                                                    DataAccessException;  
}
```

iv. Spring JDBC - Transaction - iv.v NamedParameterJdbcTemplate

iv.v NamedParameterJdbcTemplate (d)

```
String INSERT = "INSERT INTO USER(username, password) VALUES (:username, :password)";
```

```
Map<String, Object> map = ...
```

```
Boolean boolean = namedParameterJdbcTemplate.execute(INSERT, map,  
                                                    new PreparedStatementCallback<Boolean>(){  
    @Override  
    public Boolean doInPreparedStatement(PreparedStatement ps) throws SQLException,  
                                         DataAccessException {  
        return ps.execute ();  
    }  
});
```

iv. Spring JDBC - Transaction - iv.v NamedParameterJdbcTemplate

iv.v NamedParameterJdbcTemplate (e)

- Mismo ejemplo implementado con JdbcTemplate

String INSERT = "INSERT INTO USER(username, password) VALUES (?, ?)";

```
Boolean boolean = jdbcTemplate.execute(INSERT,  
                                         new PreparedStatementCallback<Boolean>(){  
@Override  
    public Boolean doInPreparedStatement(PreparedStatement ps) throws SQLException,  
                                         DataAccessException {  
        ps.setString(1, "myUser123");  
        ps.setString(2, "myPass123");  
        return ps.execute();  
    }  
});
```

iv. Spring JDBC - Transaction - iv.v NamedParameterJdbcTemplate

iv.v NamedParameterJdbcTemplate

- a. Select query
- b. Update query (insert, update y delete)
- c. Execute query

Práctica 25 – Parte 3. Implementación ICustomerDAO

iv.v NamedParameterJdbcTemplate. Práctica 25. (a)

- Práctica 25 – Parte 3. Implementación ICustomerDAO.
- Implementar la funcionalidad CRUD básica de la interface ICustomerDAO mediante la aplicación de NamedParameterJdbcTemplate.
- Realizar pruebas funcionales en los 2 de los 3 diferentes ambientes de base de datos, se excluye mysql por el momento.

iv. Spring JDBC - Transaction - iv.v NamedParameterJdbcTemplate

Resumen de la lección

iv.v NamedParameterJdbcTemplate

- Conocimos los principales métodos que provee la clase NamedParameterJdbcTemplate a fondo.
- Verificamos la funcionalidad del callback ResultSetExtractor<T> y RowCallbackHandler.
- Comprendimos el uso de NamedParameterJdbcTemplate para realizar instrucciones SELECT, INSERT, UPDATE y DELETE.
- Comprendimos que para ejecutar instrucciones DDL, se recomienda el uso de JdbcTemplate.

iv. Spring JDBC - Transaction - iv.v NamedParameterJdbcTemplate

Esta página fue intencionalmente dejada en blanco.

iv. Spring JDBC - Transaction - iv.v NamedParameterJdbcTemplate

iv.vi Simplificando Operaciones JDBC

Objetivos de la lección

iv.vi Simplificando Operaciones JDBC

- Conocer otro tipo de templates para la simplificación de ejecución de instrucciones SQL.
- Conocer la interface BeanPropertySqlParameterSource.
- Conocer la interface BeanPropertyRowMapper.

iv. Spring JDBC - Transaction - iv.vi Simplificando Operaciones JDBC

iv.vi Simplificando Operaciones JDBC

- a. SimpleJdbcInsert
- b. SimpleJdbcCall
- c. BeanPropertySqlParameterSource
- d. BeanPropertyRowMapper

Práctica 25 – Parte 4. Implementación ICustomer DAO
(mysql)

Práctica 25 – Parte 5. Implementación IAccountDAO

iv.vi Simplificando Operaciones JDBC (a)

- Spring JDBC provee de otros templates tales como **SimpleJdbcInsert** y **SimpleJdbcCall**, los cuales simplifican y encapsulan la configuración de **JdbcTemplate** mediante la obtención de *metadata* directamente desde el DataSource.

iv. Spring JDBC - Transaction - iv.vi Simplificando Operaciones JDBC

iv.vi Simplificando Operaciones JDBC

- a. **SimpleJdbcInsert**
- b. SimpleJdbcCall
- c. BeanPropertySqlParameterSource
- d. BeanPropertyRowMapper

Práctica 25 – Parte 4. Implementación ICustomer DAO
(mysql)

Práctica 25 – Parte 5. Implementación IAccountDAO

iv.vi Simplificando Operaciones JDBC (a)

- SimpleJdbcInsert
- Provee de una template para ejecutar un insert mediante un PreparedStatement reusable.
- Es thread-safe una una vez instanciado.
- Se recomienda instanciarlo durante la inicialización del DAO correspondiente.

iv. Spring JDBC - Transaction - iv.vi Simplificando Operaciones JDBC

iv.vi Simplificando Operaciones JDBC (b)

- SimpleJdbcInsert
- Configuración.

```
SimpleJdbcInsert insertCustomer = new SimpleJdbcInsert(dataSource).  
withTableName("CUSTOMER");
```

- Uso.

```
Map<String, Object> mapParameters = ...;  
insertCustomer.execute(mapParameters);
```

iv. Spring JDBC - Transaction - iv.vi Simplificando Operaciones JDBC

iv.vi Simplificando Operaciones JDBC (c)

- SimpleJdbcInsert
- Configuración con obtención de GeneratedKey.

```
SimpleJdbcInsert insertCustomer = new SimpleJdbcInsert(dataSource).  
    withTableName("CUSTOMER").usingGeneratedKeyColumns("CUSTOMER_ID");
```

- Uso.

```
KeyHolder keyHolder = insertCustomer.executeAndReturnKeyHolder(new  
    BeanPropertySqlParameterSource(customer) );
```

iv. Spring JDBC - Transaction - iv.vi Simplificando Operaciones JDBC

iv.vi Simplificando Operaciones JDBC (d)

- SimpleJdbcInsert
- A su vez es posible delimitar la cantidad de columnas a insertar mediante el método **usingColumns**.

```
SimpleJdbcInsert insertCustomer = new SimpleJdbcInsert(dataSource).  
    withTableName("CUSTOMER").usingColumns("NAME", "LAST_NAME").  
    usingGeneratedKeyColumns("CUSTOMER_ID");
```

iv. Spring JDBC - Transaction - iv.vi Simplificando Operaciones JDBC

iv.vi Simplificando Operaciones JDBC

- a. SimpleJdbcInsert
- b. SimpleJdbcCall**
- c. BeanPropertySqlParameterSource
- d. BeanPropertyRowMapper

Práctica 25 – Parte 4. Implementación ICustomer DAO
(mysql)

Práctica 25 – Parte 5. Implementación IAccountDAO

iv.vi Simplificando Operaciones JDBC (a)

- SimpleJdbcCall
- Provee de una template para ejecutar stored procedures mediante un objeto reusable.
- Es thread-safe una una vez instanciado.
- Se recomienda instanciarlo durante la inicialización del DAO correspondiente.

iv. Spring JDBC - Transaction - iv.vi Simplificando Operaciones JDBC

iv.vi Simplificando Operaciones JDBC (b)

- SimpleJdbcCall

```
MYSQL
DELIMITER //
CREATE PROCEDURE read_customer_user (
  IN in_customerId INTEGER,
  OUT out_user_id integer,
  OUT out_customer_id integer,
  OUT out_username VARCHAR(100),
  OUT out_password VARCHAR(100),
  OUT out_name VARCHAR(100),
  OUT out_last_name VARCHAR(100))
BEGIN
  SELECT USER_ID, CUSTOMER_ID, USERNAME, PASSWORD, NAME, LAST_NAME
  INTO out_user_id, out_customer_id, out_username, out_password, out_name, out_last_name
  FROM USER_TBL, CUSTOMER_TBL WHERE CUSTOMER_ID = FK_CUSTOMER_ID AND CUSTOMER_ID = in_customerId;
END //
DELIMITER ;
```

iv. Spring JDBC - Transaction - iv.vi Simplificando Operaciones JDBC

iv.vi Simplificando Operaciones JDBC (c)

- SimpleJdbcCall
- Configuración.

```
SimpleJdbcCall readCustomerProcedure = new SimpleJdbcCall(dataSource).  
    withProcedureName("read_customer_user");
```

iv. Spring JDBC - Transaction - iv.vi Simplificando Operaciones JDBC

iv.vi Simplificando Operaciones JDBC (d)

- SimpleJdbcCall
- Uso.

```
SqlParameterSource parameterSource = new MapSqlParameterSource().  
    addValue("in_customerId", id);
```

```
Map<String, Object> out = readCustomerProcedure.execute(parameterSource);
```

```
Integer userId = (Integer) out.get("out_user_id");  
String username = (String) out.get("out_username");  
String name = (String) out.get("out_name");
```

...

iv. Spring JDBC - Transaction - iv.vi Simplificando Operaciones JDBC

iv.vi Simplificando Operaciones JDBC

- a. SimpleJdbcInsert
- b. SimpleJdbcCall
- c. BeanPropertySqlParameterSource
- d. BeanPropertyRowMapper

Práctica 25 – Parte 4. Implementación ICustomer DAO
(mysql)

Práctica 25 – Parte 5. Implementación IAccountDAO

iv.vi Simplificando Operaciones JDBC (a)

- BeanPropertySqlParameterSource
- Se utiliza **Map<String, Object>** o **MapSqlParameterSource** para proveer de parámetros a las consultas utilizando NamedParameterJdbcTemplate.
- Es posible utilizar la clase **BeanPropertySqlParameterSource** ya que ésta implementa **SqlParameterSource**.

iv. Spring JDBC - Transaction - iv.vi Simplificando Operaciones JDBC

iv.vi Simplificando Operaciones JDBC (b)

- BeanPropertySqlParameterSource
- Se utiliza **BeanPropertySqlParameterSource** para pasar parámetros a las consultas directamente desde un POJO, que contenga dichos valores, por ejemplo entidades de dominio.
- Las propiedades del POJO deben llamarse igual que las columnas de la tabla para que **BeanPropertySqlParameterSource** genere un **MapSqlParameterSource** con las llaves que requiere la tabla en la base de datos.

iv. Spring JDBC - Transaction - iv.vi Simplificando Operaciones JDBC

iv.vi Simplificando Operaciones JDBC (c)

- BeanPropertySqlParameterSource
- Si las columnas en la tabla están definidas en forma *snake_case* y las propiedades del POJO en *lowerCamelCase*, igualmente **BeanPropertySqlParameterSource** funciona.
- **BeanPropertySqlParameterSource** sigue la convención Java Beans.

iv. Spring JDBC - Transaction - iv.vi Simplificando Operaciones JDBC

iv.vi Simplificando Operaciones JDBC (d)

- BeanPropertySqlParameterSource

```
SimpleJdbcInsert insertCustomer = new SimpleJdbcInsert(dataSource).  
    withTableName("CUSTOMER").usingGeneratedKeyColumns("CUSTOMER_ID");
```

```
KeyHolder keyHolder = insertCustomer.executeAndReturnKeyHolder(  
    new BeanPropertySqlParameterSource(customer) );
```

```
public class Customer {  
    private Long id;  
    private String name;  
    private String lastName;  
    private User user;  
}
```



```
create table CUSTOMER(  
    CUSTOMER_ID integer identity primary key,  
    NAME varchar(100) not null,  
    LAST_NAME varchar(100) not null  
);
```

iv. Spring JDBC - Transaction - iv.vi Simplificando Operaciones JDBC

iv.vi Simplificando Operaciones JDBC

- a. SimpleJdbcInsert
- b. SimpleJdbcCall
- c. BeanPropertySqlParameterSource
- d. **BeanPropertyRowMapper**

Práctica 25 – Parte 4. Implementación ICustomer DAO
(mysql)

Práctica 25 – Parte 5. Implementación IAccountDAO

iv.vi Simplificando Operaciones JDBC (a)

- BeanPropertyRowMapper
- Similar a BeanPropertySqlParameterSource, la Clase **BeanPropertyRowMapper** ayuda a poder mapear de forma automática un ResultSet a un POJO, por ejemplo entidades de dominio.
- Para realizar el mapeo automático desde un ResultSet a una clase POJO, **BeanPropertyRowMapper** requiere para funcionar que los nombres devueltos por la consulta (SELECT) tengan los mismos nombres que los atributos en la clase POJO.

iv. Spring JDBC - Transaction - iv.vi Simplificando Operaciones JDBC

iv.vi Simplificando Operaciones JDBC (b)

- BeanPropertyRowMapper
- **BeanPropertyRowMapper** sigue la convención Java Beans.
- Existen dos formas de instanciar un objeto **BeanPropertyRowMapper**.

iv. Spring JDBC - Transaction - iv.vi Simplificando Operaciones JDBC

iv.vi Simplificando Operaciones JDBC (c)

- BeanPropertyRowMapper
- Instancia mediante Genéricos

```
new BeanPropertyRowMapper<User>(User.class);
```

- Instancia mediante Factory

```
BeanPropertyRowMapper.newInstance(User.class);
```

iv. Spring JDBC - Transaction - iv.vi Simplificando Operaciones JDBC

iv.vi Simplificando Operaciones JDBC (d)

- BeanPropertyRowMapper

```
String SELECT = "SELECT * FROM CUSTOMER WHERE ID = ?";
```

```
Customer customer = jdbcTemplate.queryForObject(SELECT,  
new BeanPropertyRowMapper<Customer>(Customer.class), id);
```

ó

```
Customer customer = jdbcTemplate.queryForObject(SELECT,  
BeanPropertyRowMapper.newInstance (Customer.class), id);
```

iv. Spring JDBC - Transaction - iv.vi Simplificando Operaciones JDBC

iv.vi Simplificando Operaciones JDBC

- a. SimpleJdbcInsert
- b. SimpleJdbcCall
- c. BeanPropertySqlParameterSource
- d. BeanPropertyRowMapper

Práctica 25 – Parte 4. Implementación ICustomer DAO
(mysql)

Práctica 25 – Parte 5. Implementación IAccountDAO

iv.vi Simplificando Operaciones JDBC. Práctica 25. (a)

- Práctica 25 – Parte 4. Implementación ICustomer DAO (mysql)
- Implementar la funcionalidad CRUD básica de la interface ICustomerDAO mediante la aplicación de las diferentes técnicas aprendidas así como de SimpleJdbcInsert, SimpleJdbcCall y BeanPropertySqlParameterSource.

iv. Spring JDBC - Transaction - iv.vi Simplificando Operaciones JDBC

iv.vi Simplificando Operaciones JDBC

- a. SimpleJdbcInsert
- b. SimpleJdbcCall
- c. BeanPropertySqlParameterSource
- d. BeanPropertyRowMapper

Práctica 25 – Parte 4. Implementación ICustomer DAO
(mysql)

Práctica 25 – Parte 5. Implementación IAccountDAO

iv.vi Simplificando Operaciones JDBC. Práctica 25. (b)

- Práctica 25 – Parte 5. Implementación IAccountDAO
- Implementar la funcionalidad CRUD básica de la interface IAccountDAO mediante la aplicación de las diferentes técnicas aprendidas.

iv. Spring JDBC - Transaction - iv.vi Simplificando Operaciones JDBC

Resumen de la lección

iv.vi Simplificando Operaciones JDBC

- Conocimos otros tipos de templates para la simplificación de instrucciones como Inserts y llamada a stored procedures.
- Verificamos la usabilidad de BeanPropertySqlParameterSource como clase auxiliar para proveer de parámetros a consultas SQL.
- Verificamos la usabilidad de BeanPropertyRowMapper como clase auxiliar para el mapeo automático de resultados desde un ResultSet a un objeto de dominio.

iv. Spring JDBC - Transaction - iv.vi Simplificando Operaciones JDBC

Esta página fue intencionalmente dejada en blanco.

iv. Spring JDBC - Transaction - iv.vi Simplificando Operaciones JDBC

iv.vii Spring y el Manejo Transaccional

Objetivos de la lección

iv.vii Spring y el Manejo Transaccional

- Conocer que es transaccionabilidad así como los diferentes alcances transaccionales para aplicaciones Java EE.
- Verificar las características ACID que toda base de datos debe procurar sobre la ejecución de transacciones.
- Conocer las ventajas del uso de Spring para el manejo transaccional.
- Conocer la interface core PlatformTransactionManager, sus principales implementaciones y funcionalidades.
- Comprender la utilidad del Transaction Manager.

iv. Spring JDBC - Transaction - iv.vii Spring y el Manejo Transaccional

iv.vii Spring y el Manejo Transaccional

- a. ACID
- b. Ventajas de Spring Transaction Management
- c. PlatformTransactionManagement

iv.vii Spring y el Manejo Transaccional (a)

- La aplicación de transaccionabilidad en aplicaciones Java EE es uno de los motivos más fuertes para utilizar Spring Framework pues proporciona una abstracción muy coherente para la gestión y el manejo transaccional.
- Spring Tx, en específico, provee de un modelo consistente a través de múltiples APIs transaccionales como son JTA, JDBC, Hibernate, JPA y JDO.

iv. Spring JDBC - Transaction - iv.vii Spring y el Manejo Transaccional

iv.vii Spring y el Manejo Transaccional (b)

- Spring Tx soporta transacciones declarativas o programáticas.
- Se integra perfectamente a cualquier framework de persistencia.

iv. Spring JDBC - Transaction - iv.vii Spring y el Manejo Transaccional

iv.vii Spring y el Manejo Transaccional (a)

- Existen 2 tipos de transacciones en Java EE, globales y locales, ambas tienen muchas limitaciones.
- Globales: Permite trabajar con múltiples orígenes transaccionales, típicamente bases de datos o colas de mensajes. Es responsabilidad del application server manejar las transacciones mediante JTA, haciéndose difícil de probar.

iv. Spring JDBC - Transaction - iv.vii Spring y el Manejo Transaccional

iv.vii Spring y el Manejo Transaccional (b)

- Globales: Las transacciones globales necesitan ser encontradas como un recurso JNDI, por tanto es necesario implementar JNDI con JTA.
- Locales: Son fáciles de implementar pero no soportan trabajar con múltiples orígenes de datos. Por lo regular las transacciones locales implementadas con JDBC son invasivas.

iv. Spring JDBC - Transaction - iv.vii Spring y el Manejo Transaccional

iv.vii Spring y el Manejo Transaccional

a. ACID

b. Ventajas de Spring Transaction Management

c. PlatformTransactionManagement

iv.vii Spring y el Manejo Transaccional (a)

- ACID
- Toda base de datos debe procurar las 4 características de la transaccionabilidad.
 - Atomicidad (Atomicity)
 - Consistencia (Consistency)
 - Aislamiento (Isolation)
 - Durabilidad (Durability)

iv. Spring JDBC - Transaction - iv.vii Spring y el Manejo Transaccional

iv.vii Spring y el Manejo Transaccional (b)

- Atomicidad (Atomicity)
- Una transacción debe considerarse como una única unidad de trabajo.
- Toda la secuencia de operaciones o instrucciones que realiza la transacción debe ejecutarse satisfactoriamente o insatisfactoriamente, no sólo partes de la transacción.

iv. Spring JDBC - Transaction - iv.vii Spring y el Manejo Transaccional

iv.vii Spring y el Manejo Transaccional (c)

- Consistencia (Consistency)
- Toda base de datos, al ejecutar una transacción, debe asegurar la consistencia de los datos, en este sentido, consistencia, se refiere a la integridad referencial de tablas y registros.
- Llaves primarias únicas, evitar llaves foráneas inexistentes entre otras.

iv. Spring JDBC - Transaction - iv.vii Spring y el Manejo Transaccional

iv.vii Spring y el Manejo Transaccional (d)

- Aislamiento (Isolation)
- Toda base de datos, al ejecutar transacciones concurrentes, debe ser capaz de mantener la ejecución de las instrucciones de cada transacción aisladas entre ellas.
- El aislamiento debe poder evitar problemas como lectura sucia, lectura no repetible y lectura fantasma.

iv. Spring JDBC - Transaction - iv.vii Spring y el Manejo Transaccional

iv.vii Spring y el Manejo Transaccional (e)

- Durabilidad (Durability)
- Una vez que toda transacción a sido completada, la base de datos debe asegurar que la información persistida por la transacción permanezca de forma permanente físicamente y que no pueda ser perdida o eliminada por algún fallo del sistema.

iv. Spring JDBC - Transaction - iv.vii Spring y el Manejo Transaccional

iv.vii Spring y el Manejo Transaccional

a. ACID

b. Ventajas de Spring Transaction Management

c. PlatformTransactionManagement

iv.vii Spring y el Manejo Transaccional (a)

- Ventajas de Spring Transaction Management
- Spring resuelve las desventajas de desarrollar transacciones globales y locales, proveyendo de modelo de programación consistente mismo que puede ser portable a cualquier ambiente de despliegue.
- Spring provee de un API declarativa y programática para el manejo transaccional, por lo regular se utiliza la forma declarativa.

iv. Spring JDBC - Transaction - iv.vii Spring y el Manejo Transaccional

iv.vii Spring y el Manejo Transaccional (b)

- Ventajas de Spring Transaction Management
- Para el manejo transaccional local no es necesario un application server sin embargo, para transacciones globales si, pues Spring Tx abstrae el manejo y configuración de JTA.
- Por lo regular las aplicaciones no requieren un manejo global de transacciones, para ello existen muchas otras estrategias de arquitectura.

iv. Spring JDBC - Transaction - iv.vii Spring y el Manejo Transaccional

iv.vii Spring y el Manejo Transaccional

- a. ACID
- b. Ventajas de Spring Transaction Management
- c. PlatformTransactionManagement

iv.vii Spring y el Manejo Transaccional (a)

- PlatformTransactionManager
- Interface principal que provee del servicio transaccional, es posible crear implementaciones de esta interfaz para realizar transacciones programáticas.
- Las implementaciones de PlatformTransactionManager son beans como cualquier otro bean de la aplicación en el IoC de Spring.

iv. Spring JDBC - Transaction - iv.vii Spring y el Manejo Transaccional

iv.vii Spring y el Manejo Transaccional (b)

- PlatformTransactionManager
- Debido al desacoplamiento de las múltiples APIs transaccionales por cada proveedor, utilizar PlatformTransactionManager facilita el desarrollo y prueba de componentes transaccionales.

```
public interface PlatformTransactionManager {  
    TransactionStatus getTransaction(TransactionDefinition definition) throws  
                                                                    TransactionException;  
    void commit(TransactionStatus status) throws TransactionException;  
    void rollback(TransactionStatus status) throws TransactionException;  
}
```

iv. Spring JDBC - Transaction - iv.vii Spring y el Manejo Transaccional

iv.vii Spring y el Manejo Transaccional (c)

- **TransactionException:** Es una excepción *unchecked*, por tanto no requiere ser manejada, hereda de RuntimeException.
- **TransactionStatus:** Cada llamada al método `getTransaction()` se devuelve un objeto **TransactionStatus** que representa al estado de una transacción donde su configuración depende del objeto **TransactionDefinition**. El objeto `TransactionStatus` devuelto puede ser la representación de una nueva transacción o de una existente.
- **TransactionDefinition:** Es un objeto que configura la transacción (`TransactionStatus`).

iv. Spring JDBC - Transaction - iv.vii Spring y el Manejo Transaccional

iv.vii Spring y el Manejo Transaccional (d)

- TransactionDefinition define:
 - **Aislamiento (isolation)**: Define el grado en que una transacción será aislada del trabajo de otras transacciones.

Por ejemplo: El nivel de aislamiento permitirá o no, a una transacción, ver las escrituras no *commiteadas* de otra transacción.

iv. Spring JDBC - Transaction - iv.vii Spring y el Manejo Transaccional

iv.vii Spring y el Manejo Transaccional (e)

- TransactionDefinition define:
 - **Propagación (propagation)**: Generalmente todo el código ejecutado dentro de una transacción se ejecutará en esa única transacción sin embargo, la ejecución de todo lo relacionado a una única transacción se puede dar mediante la ejecución de distintos **métodos transaccionales** en los cuales es posible especificar el comportamiento que tendrá la transacción cuando un método transaccional sea ejecutado.

Por ejemplo: La ejecución de un método transaccional requiere una nueva transacción o requiere una previamente inicializada.

iv. Spring JDBC - Transaction - iv.vii Spring y el Manejo Transaccional

iv.vii Spring y el Manejo Transaccional (f)

- TransactionDefinition define:
 - **Tiempo de espera (timeout)**: Define el tiempo de espera antes de que la transacción sea *echada para atrás (rollback)*.
 - **Sólo lectura (read-only)**: Permite definir acceso de sólo lectura sobre un método transaccional, esto optimiza la creación de la transacción sin embargo, no es posible realizar operaciones de escritura.

iv. Spring JDBC - Transaction - iv.vii Spring y el Manejo Transaccional

iv.vii Spring y el Manejo Transaccional (g)

- Definición de PlatformTransactionManager
- Las implementaciones de **PlatformTransactionManager** requieren conocer el ambiente de ejecución de las transacciones (JTA, JDBC, Hibernate, etc), independientemente del tipo de configuración a implementar (programática o declarativa).

iv. Spring JDBC - Transaction - iv.vii Spring y el Manejo Transaccional

iv.vii Spring y el Manejo Transaccional (h)

- Definición de PlatformTransactionManager para JDBC

```
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" ... >  
    <property name="driverClassName" value="${jdbc.driverClassName}" />  
    ...  
</bean>
```

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

iv. Spring JDBC - Transaction - iv.vii Spring y el Manejo Transaccional

iv.vii Spring y el Manejo Transaccional (i)

- Definición de PlatformTransactionManager para JTA (requiere application server)

```
<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"
  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">

  <jee:jndi-lookup id="dataSource" jndi-name="jdbc/db"/>
  <bean id="txManager" class="org.springframework.transaction.jta.JtaTransactionManager" />
</beans>
```

iv. Spring JDBC - Transaction - iv.vii Spring y el Manejo Transaccional

iv.vii Spring y el Manejo Transaccional (j)

- Definición de PlatformTransactionManager para JTA (requiere application server)
- Observaciones:
 - JtaTransactionManager NO necesita conocer el origen de datos para implementar transaccionabilidad debido a que utiliza la infraestructura del manejador de transacciones globales que implementa el application server.

iv. Spring JDBC - Transaction - iv.vii Spring y el Manejo Transaccional

iv.vii Spring y el Manejo Transaccional (k)

- Definición de PlatformTransactionManager para JTA (requiere application server)
- Observaciones:
 - Si se utiliza JNDI, en un application server, para obtener una referencia a un DataSource y éste es utilizado en un manejador de transacciones diferente a JTA, el DataSource será NO transaccional para Spring o más bien, para el application server, pues ésta gestiona sus transacciones mediante JTA.

iv. Spring JDBC - Transaction - iv.vii Spring y el Manejo Transaccional

iv.vii Spring y el Manejo Transaccional (I)

- Definición de PlatformTransactionManager para Hibernate

```
<bean id="sessionFactory"
      class="org.springframework.orm.hibernate5.LocalSessionFactoryBean">
  <property name="dataSource" ref="dataSource"/>
  ...
</bean>

<bean id="txManager"
      class="org.springframework.orm.hibernate5.HibernateTransactionManager">
  <property name="sessionFactory" ref="sessionFactory"/>
</bean>
```

iv. Spring JDBC - Transaction - iv.vii Spring y el Manejo Transaccional

iv.vii Spring y el Manejo Transaccional (m)

- Definición de PlatformTransactionManager para Hibernate
- Observaciones:
 - Si se utiliza Hibernate en un application server que utiliza JTA como manejador de transacciones, simplemente se utiliza JtaTransactionManager en lugar de HibernateTransactionManager.

iv. Spring JDBC - Transaction - iv.vii Spring y el Manejo Transaccional

iv.vii Spring y el Manejo Transaccional (n)

- Definición de PlatformTransactionManager
- Observaciones:
 - Para manejar transacciones globales es necesario utilizar un application server y el estándar JTA.
 - Si utiliza JTA como manejador de transacciones necesitará utilizar JtaTransactionManager independientemente de la tecnología de persistencia a utilizada.
 - En cualquier caso el código cliente no necesita saber que TransactionManager maneja las transacciones, es posible cambiar el TransactionManager sin afectar el código cliente.

iv. Spring JDBC - Transaction - iv.vii Spring y el Manejo Transaccional

Resumen de la lección

iv.vii Spring y el Manejo Transaccional

- Conocimos los tipos de transaccionabilidad Global y Local.
- Verificamos el concepto ACID sobre transacciones en bases de datos.
- Comprendimos la utilidad de la interface PlatformTransactionManager.
- Verificamos como configurar las distintas implementaciones de PlatformTransactionManager.

iv. Spring JDBC - Transaction - iv.vii Spring y el Manejo Transaccional

Esta página fue intencionalmente dejada en blanco.

iv. Spring JDBC - Transaction - iv.vii Spring y el Manejo Transaccional

iv.viii Spring Tx

Objetivos de la lección

iv.viii Spring Tx

- Conocer los diferentes tipos de configuración de transacciones declarativas.
- Conocer los tipos de propagación aplicables a Spring Tx.
- Conocer los tipos de aislamiento aplicables a Spring Tx.
- Implementar transaccionabilidad sobre métodos transaccionales.
- Conocer a grandes rasgos la implementación de transacciones programáticas.

iv. Spring JDBC - Transaction - iv.viii Spring Tx

iv.viii Spring Tx

a. Transacciones declarativas configuración por XML

Práctica 26 – Parte 1. Transacciones declarativas
configuración por XML

b. Tipos de Propagación

c. Tipos de Aislamiento

d. Transacciones declarativas configuración por
@Anotaciones

Práctica 26 – Parte 2. Transacciones declarativas
configuración por @Anotaciones

d. Transacciones Programáticas

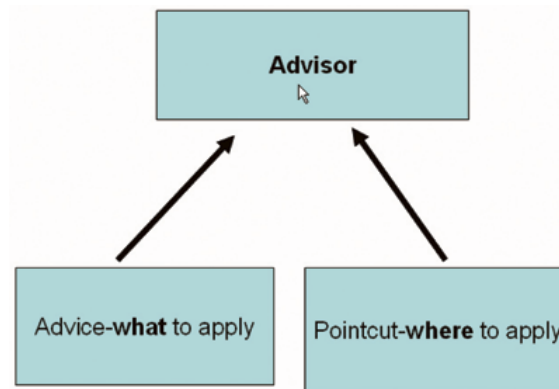
iv.viii Spring Tx (a)

- Transacciones declarativas configuración por XML
- La mayoría de los desarrolladores implementan transaccionabilidad mediante la definición de transacciones declarativas pues es la aproximación no invasiva.
- Es posible implementar transacciones declarativas mediante configuración XML o mediante configuración por @Anotaciones (revisado más adelante).

iv. Spring JDBC - Transaction - iv.viii Spring Tx

iv.viii Spring Tx (b)

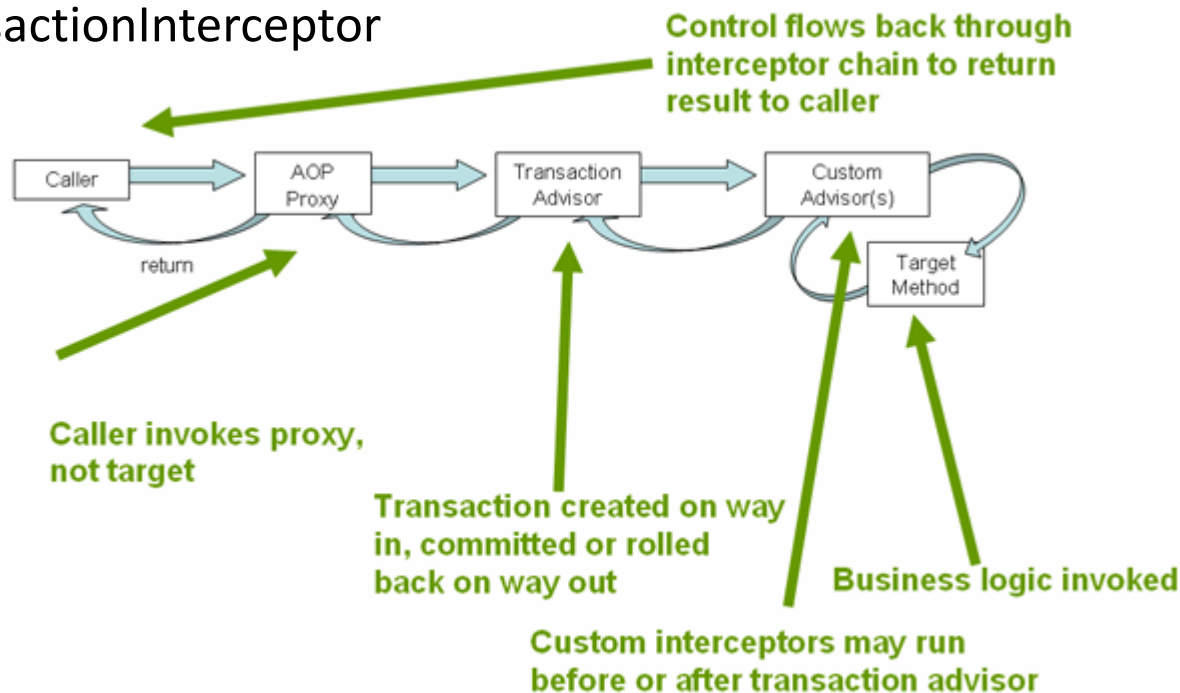
- Transacciones declarativas configuración por XML
- Las transacciones declarativas en Spring se implementan mediante AOP, específicamente mediante un *Advisor*.



iv. Spring JDBC - Transaction - iv.viii Spring Tx

iv.viii Spring Tx (c)

- TransactionInterceptor



iv. Spring JDBC - Transaction - iv.viii Spring Tx

iv.viii Spring Tx (d)

- Configuración de Transacciones declarativas por XML

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:aop="http://www.springframework.org/schema/aop"
  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/tx
  http://www.springframework.org/schema/tx/spring-tx.xsd
  http://www.springframework.org/schema/aop
  http://www.springframework.org/schema/aop/spring-aop.xsd">
  ...
</beans>
```

iv. Spring JDBC - Transaction - iv.viii Spring Tx

iv.viii Spring Tx (e)

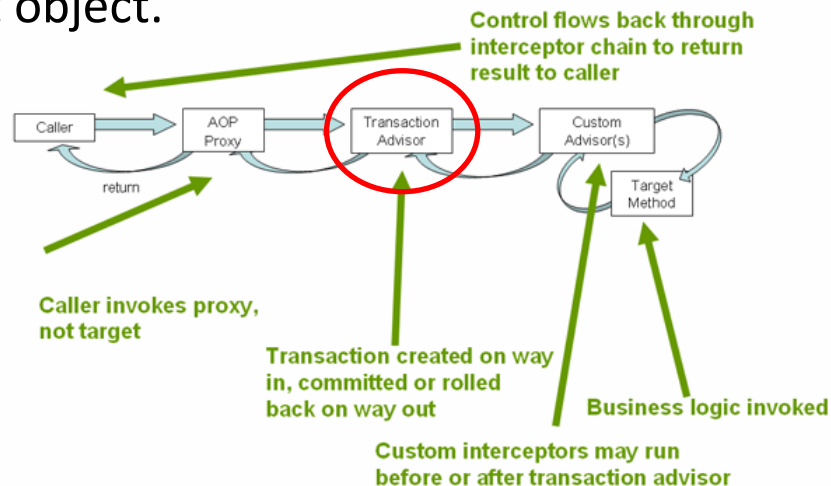
- **<tx:advice />**: Se utiliza para crear un advice manejador de transacciones (*transactional advice*), al mismo tiempo se definen los nombres de los métodos que deseamos crear como transaccionales para que éstos sean aconsejados o interceptados por un aspecto (*advisor*). **<tx:advice>** define las semánticas de los métodos aconsejados (*read-only ó read-write semantics, entre otros*).

```
<tx:advice id="txAdvice" transaction-manager="txManager">  
  <tx:attributes>  
    <tx:method name="get*" read-only="true"/>  
    <tx:method name="*" />  
  </tx:attributes>  
</tx:advice>
```

iv. Spring JDBC - Transaction - iv.viii Spring Tx

iv.viii Spring Tx (f)

- **<tx:method name="get*" />**: Para cada nombre de método se creará un proxy que iniciará la transacción antes de llamar al método del target object y finalizará la transacción (*commit ó rollback*) después de la llamada al target object.



iv. Spring JDBC - Transaction - iv.viii Spring Tx

iv.viii Spring Tx (g)

- El target object se ejecutará en un bloque try catch.
- Si el método aconsejado del target object (*advised object*) finaliza normalmente, mediante AOP, se hace *commit* a la transacción en caso de lanzar una excepción *unchecked*, se hace *rollback*.

iv. Spring JDBC - Transaction - iv.viii Spring Tx

iv.viii Spring Tx (h)

- La configuración del `<tx:advice>` requiere de un aspecto, en este caso un advisor, que defina el pointcut (conjunto de JoinPoints) y el advice (*transactional advice*) estará interceptando el pointcut definido.

```
<aop:config>
  <aop:pointcut id="dataAccessExecution"
    expression="execution(* x.y.service.dao..*.*(..))"/>
  <aop:advisor advice-ref="txAdvice" pointcut-ref="dataAccessExecution"/>
</aop:config>

<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" ... />

<bean id="txManager" p:dataSource-ref="dataSource"
  class="org.springframework.jdbc.datasource.DataSourceTransactionManager" />
```

iv. Spring JDBC - Transaction - iv.viii Spring Tx

iv.viii Spring Tx (i)

```
<tx:advice id="txAdvice" transaction-manager="txManager">
  <tx:attributes>
    <tx:method name="get*" read-only="true"/>
    <tx:method name="*" />
  </tx:attributes>
</tx:advice>

<aop:config>
  <aop:pointcut id="dataAccessExecution" expression="execution(* x.y.service.dao..*.*(..))"/>
  <aop:advisor advice-ref="txAdvice" pointcut-ref="dataAccessExecution"/>
</aop:config>

<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" ... />
<bean id="txManager" p:dataSource-ref="dataSource"
      class="org.springframework.jdbc.datasource.DataSourceTransactionManager" />
```

iv. Spring JDBC - Transaction - iv.viii Spring Tx

iv.viii Spring Tx (j)

- Definición de Rollback Rules
- Por defecto Spring Tx ejecuta rollback sobre excepciones del tipo **RuntimeException** (*unchecked*) no manejadas es decir, todo **método transaccional** que lance excepciones del tipo **RuntimeException** será echado para atrás (*rollback*).
- Excepciones de tipo *checked* no serán echadas para atrás por default.
- Es posible configurar que excepciones ejecutarán *rollback* sobre métodos transaccionales incluyendo del tipo *checked*.

iv. Spring JDBC - Transaction - iv.viii Spring Tx

iv.viii Spring Tx (k)

- Definición de Rollback Rules: Rollback for.
- Define para que tipos de excepciones se permite hacer *rollback*.

```
<tx:advice id="txAdvice" transaction-manager="txManager">  
  <tx:attributes>  
    <tx:method name="get*" read-only="true" rollback-for="CustomApplicationException"/>  
    <tx:method name="*" />  
  </tx:attributes>  
</tx:advice>
```

iv. Spring JDBC - Transaction - iv.viii Spring Tx

iv.viii Spring Tx (I)

- Definición de Rollback Rules: No Rollback for.
- Define para que tipos de excepciones NO se permite hacer *rollback*.

```
<tx:advice id="txAdvice" transaction-manager="txManager">  
  <tx:attributes>  
    <tx:method name="get*" no-rollback-for="SincronizationApplicationException"/>  
    <tx:method name="*" />  
  </tx:attributes>  
</tx:advice>
```

iv. Spring JDBC - Transaction - iv.viii Spring Tx

iv.viii Spring Tx (m)

- Múltiple definición de *transactional advices*
- Es posible definir múltiples *transactional advices* con diferentes reglas de rollback así como diferentes niveles de propagación o aislamiento.
- Únicamente es necesario definir múltiples **<tx:advice>** con diferente *id* y definir diferentes *pointcuts* con diferente *id* y definir los advisors necesarios que referencien al *pointcut* indicado con el **<tx:advice>** indicado.

iv. Spring JDBC - Transaction - iv.viii Spring Tx

iv.viii Spring Tx (n)

- Múltiple definición de *transactional advices*

```
<aop:config>
  <aop:pointcut id="defaultServiceOperation" expression="execution(* service.*Service.*(..))"/>
  <aop:pointcut id="noTxServiceOperation" expression="execution(* service.ddl.DefaultDdlManager.*(..))"/>
  <aop:advisor pointcut-ref="defaultServiceOperation" advice-ref="defaultTxAdvice"/>
  <aop:advisor pointcut-ref="noTxServiceOperation" advice-ref="noTxAdvice"/>
</aop:config>

<tx:advice id="defaultTxAdvice">
  <tx:attributes>
    <tx:method name="get*" read-only="true"/>
    <tx:method name="*" />
  </tx:attributes>
</tx:advice>

<tx:advice id="noTxAdvice">
  <tx:attributes>
    <tx:method name="*" propagation="NEVER"/>
  </tx:attributes>
</tx:advice>
```

iv. Spring JDBC - Transaction - iv.viii Spring Tx

iv.viii Spring Tx

a. Transacciones declarativas configuración por XML

Práctica 26 – Parte 1. Transacciones declarativas
configuración por XML

b. Tipos de Propagación

c. Tipos de Aislamiento

d. Transacciones declarativas configuración por
@Anotaciones

Práctica 26 – Parte 2. Transacciones declarativas
configuración por @Anotaciones

d. Transacciones Programáticas

iv.viii Spring Tx. Práctica 26. (a)

- Práctica 26 – Parte 1. Transacciones declarativas configuración por XML
- Implementar transaccionabilidad declarativa básica con configuración por XML sobre métodos transaccionales logrando ejecutar commit y rollback de forma automática mediante DataSourceTransactionManager.

iv. Spring JDBC - Transaction - iv.viii Spring Tx

iv.viii Spring Tx

a. Transacciones declarativas configuración por XML

Práctica 26 – Parte 1. Transacciones declarativas
configuración por XML

b. Tipos de Propagación

c. Tipos de Aislamiento

d. Transacciones declarativas configuración por
@Anotaciones

Práctica 26 – Parte 2. Transacciones declarativas
configuración por @Anotaciones

d. Transacciones Programáticas

iv.viii Spring Tx (a)

- Tipos de Propagación.
- El tipo de propagación de una transacción define el comportamiento de la transacción cuando ésta interactúa con otras transacciones, o mejor dicho con otros métodos transaccionales.
- Mediante la definición del tipo de propagación es posible definir si para cada invocación a un método transaccional se utilice la misma transacción o, por ejemplo, que cada método transaccional inicie su propia transacción.

iv. Spring JDBC - Transaction - iv.viii Spring Tx

iv.viii Spring Tx (b)

- Tipos de Propagación.
- Existen 7 tipos de propagación.
 - REQUIRED (default)
 - SUPPORTS
 - MANDATORY
 - REQUIRES_NEW
 - NOT_SUPPORTED
 - NEVER
 - NESTED

iv. Spring JDBC - Transaction - iv.viii Spring Tx

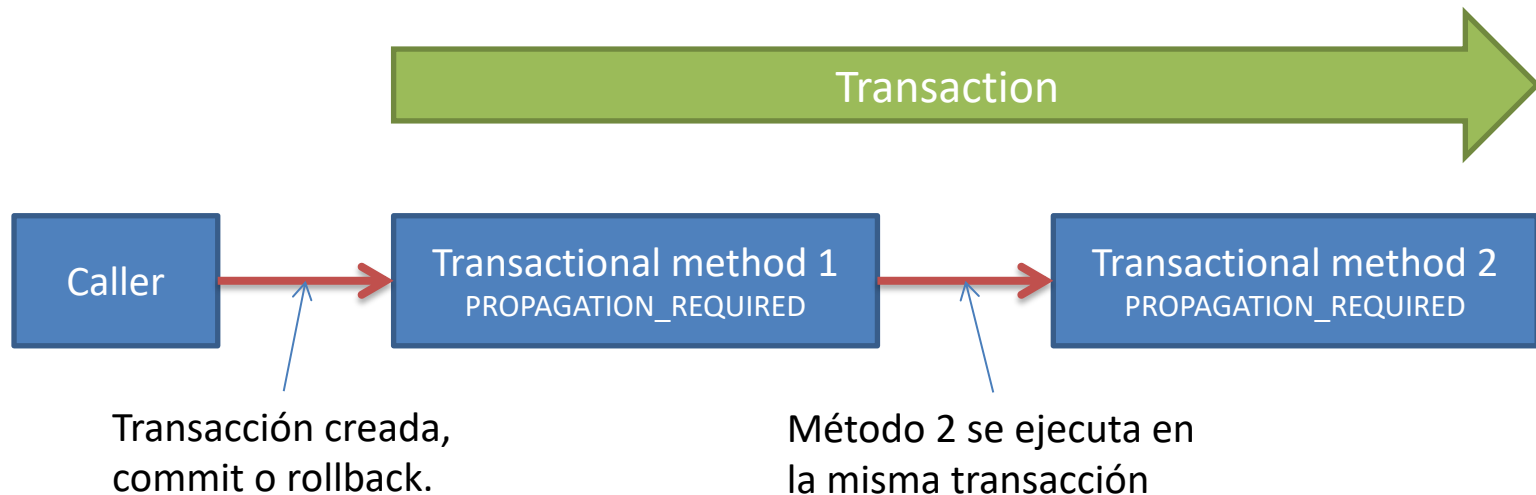
iv.viii Spring Tx (c)

- Tipos de Propagación: REQUIRED (default)
- El tipo **PROPAGATION_REQUIRED** indica que el método transaccional requiere ejecutarse dentro de una transacción.
- Si antes de la llamada al método transaccional ya existe una transacción, la utiliza, si no creará una transacción nueva.

iv. Spring JDBC - Transaction - iv.viii Spring Tx

iv.viii Spring Tx (d)

- Tipos de Propagación: REQUIRED (default)



iv. Spring JDBC - Transaction - iv.viii Spring Tx

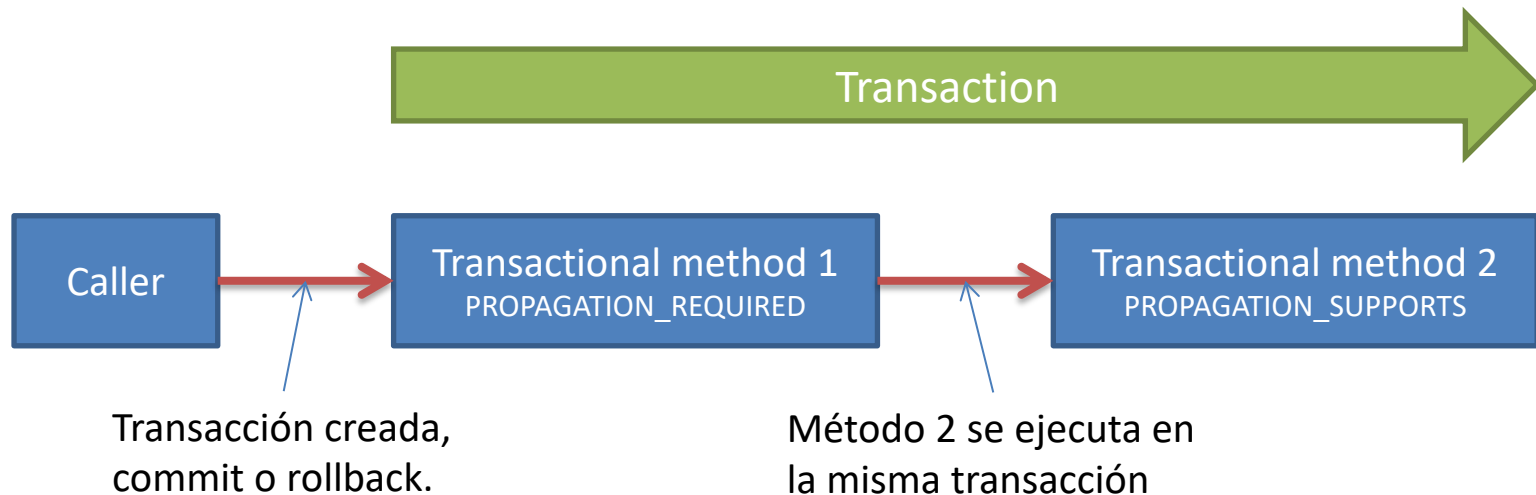
iv.viii Spring Tx (e)

- Tipos de Propagación: SUPPORTS
- El tipo **PROPAGATION_SUPPORTS** indica que el método transaccional no requiere forzosamente una transacción sin embargo, la soporta. En otras palabras un método transaccional con PROPAGATION_SUPPORTS puede ejecutarse dentro de una transacción si es que existe alguna previamente creada.

iv. Spring JDBC - Transaction - iv.viii Spring Tx

iv.viii Spring Tx (f)

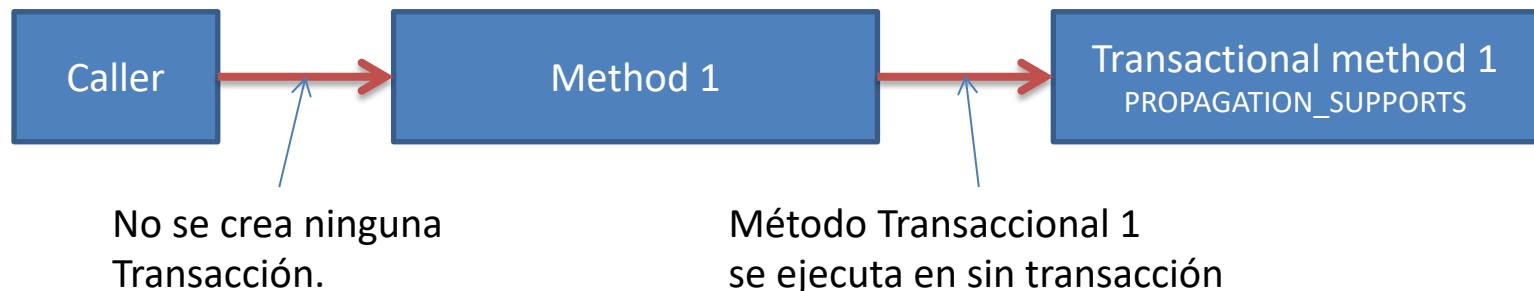
- Tipos de Propagación: SUPPORTS (a)



iv. Spring JDBC - Transaction - iv.viii Spring Tx

iv.viii Spring Tx (g)

- Tipos de Propagación: SUPPORTS (b)



iv. Spring JDBC - Transaction - iv.viii Spring Tx

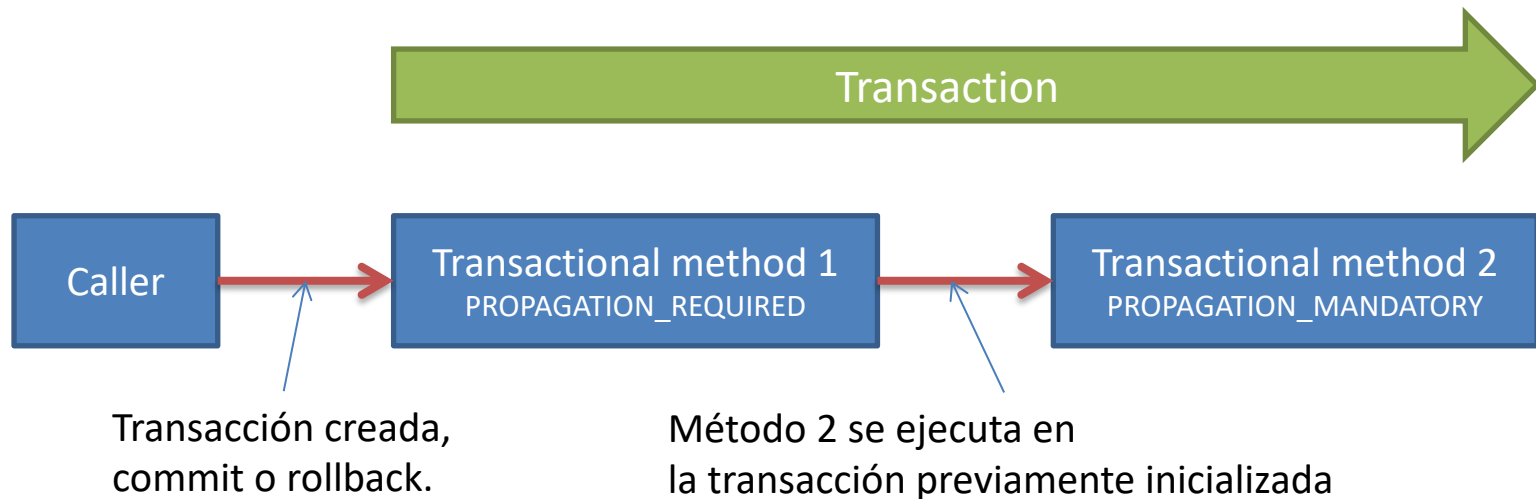
iv.viii Spring Tx (h)

- Tipos de Propagación: MANDATORY
- El tipo **PROPAGATION_MANDATORY** indica que el método transaccional requiere forzosamente ejecutarse en una transacción previamente inicializada, de lo contrario lanzará una excepción.

iv. Spring JDBC - Transaction - iv.viii Spring Tx

iv.viii Spring Tx (i)

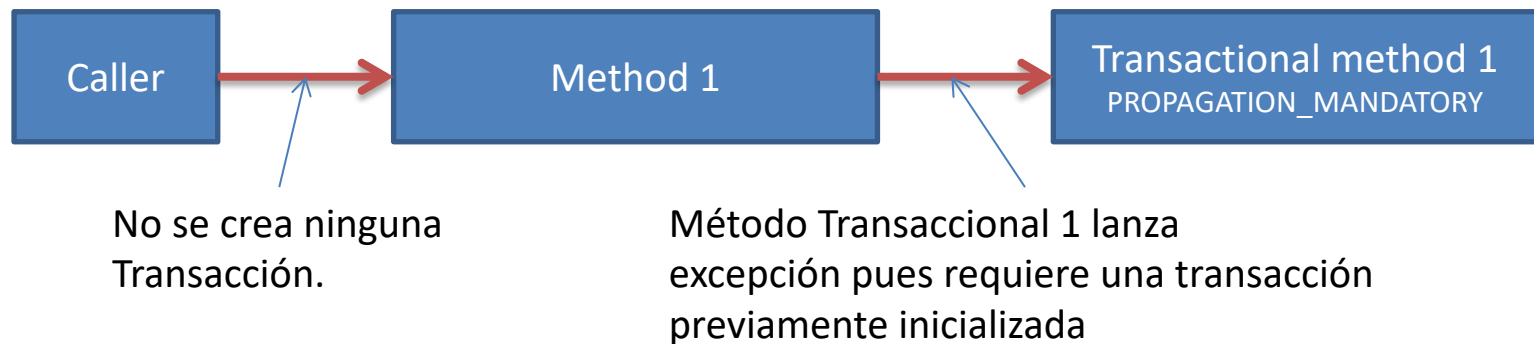
- Tipos de Propagación: MANDATORY (a)



iv. Spring JDBC - Transaction - iv.viii Spring Tx

iv.viii Spring Tx (j)

- Tipos de Propagación: MANDATORY (b)



iv. Spring JDBC - Transaction - iv.viii Spring Tx

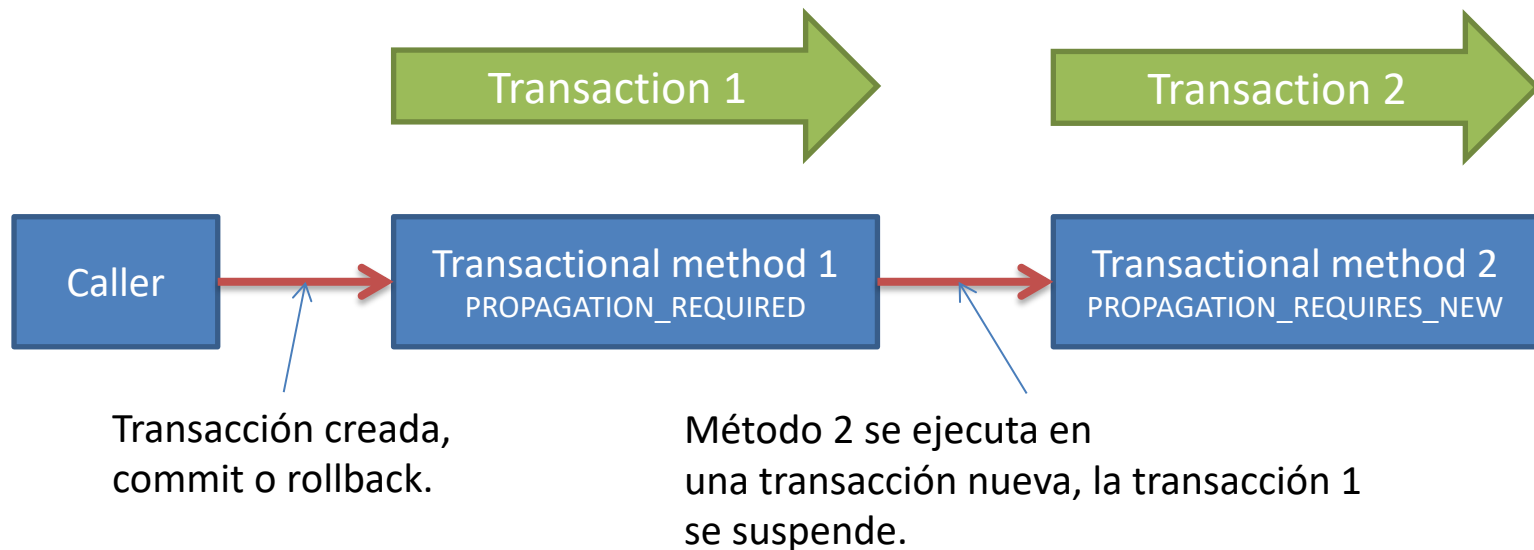
iv.viii Spring Tx (k)

- Tipos de Propagación: `REQUIRES_NEW`
- El tipo **`PROPAGATION_REQUIRES_NEW`** indica que el método transaccional requiere forzosamente ejecutarse en una transacción nueva.
- Si no existe una transacción previamente inicializada, creará una nueva transacción.
- Si existe una transacción previamente inicializada, la suspende y creará una nueva transacción.

iv. Spring JDBC - Transaction - iv.viii Spring Tx

iv.viii Spring Tx (I)

- Tipos de Propagación: `REQUIRES_NEW`



iv. Spring JDBC - Transaction - iv.viii Spring Tx

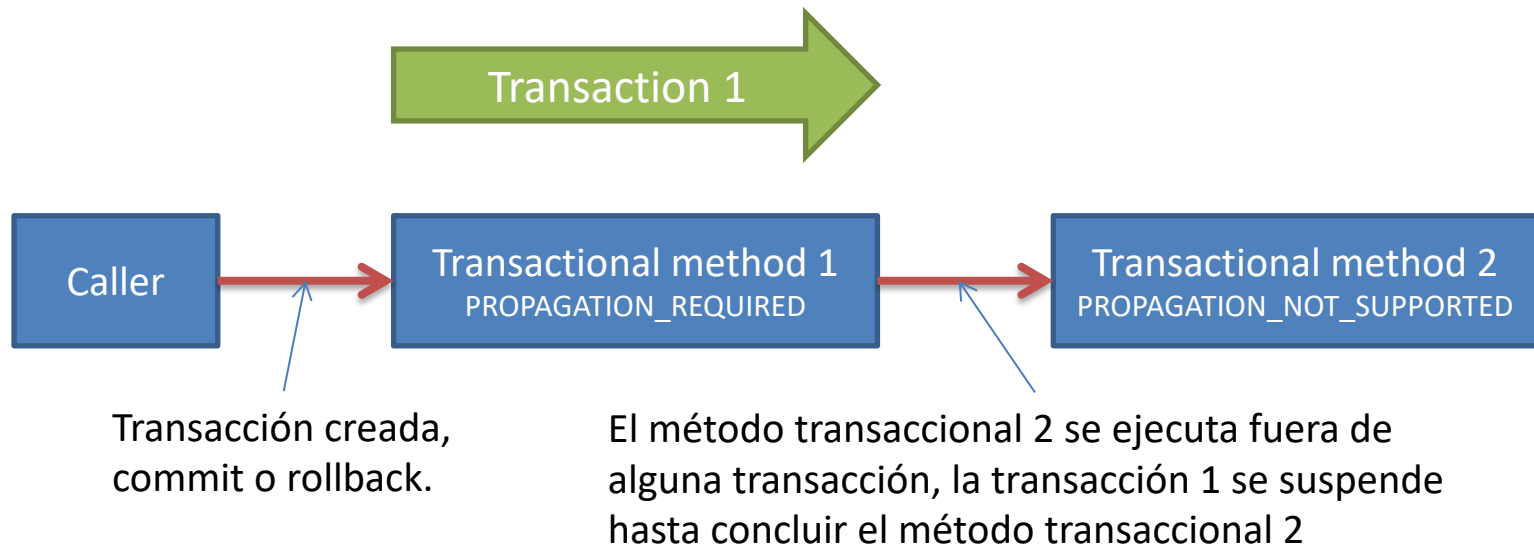
iv.viii Spring Tx (m)

- Tipos de Propagación: NOT_SUPPORTED
- El tipo **PROPAGATION_NOT_SUPPORTED** indica que el método transaccional no requiere ejecutarse en una transacción y que no la soporta.
- Si existe una transacción previamente inicializada, la suspenderá hasta que finalice la ejecución del método transaccional con propagación NOT_SUPPORTED.
- Si no existe una transacción previamente inicializada, no crea ninguna transacción.
- Este tipo de propagación sólo está disponible para JtaTransactionManager.

iv. Spring JDBC - Transaction - iv.viii Spring Tx

iv.viii Spring Tx (n)

- Tipos de Propagación: NOT_SUPPORTED



iv. Spring JDBC - Transaction - iv.viii Spring Tx

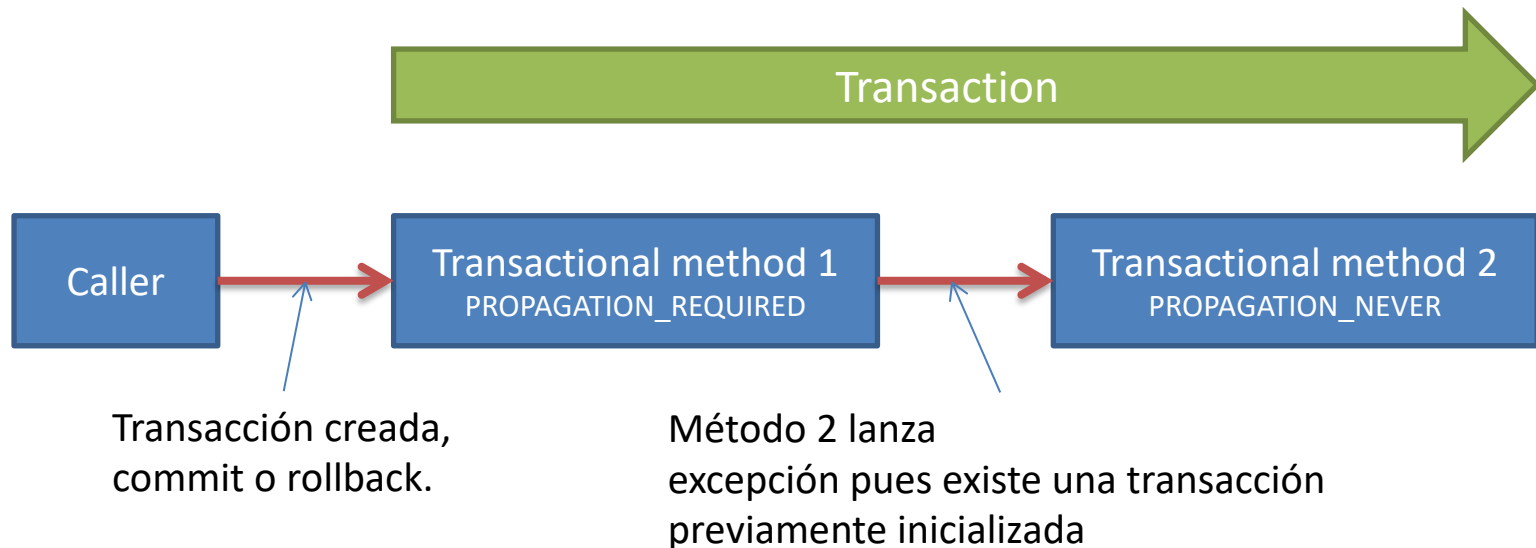
iv.viii Spring Tx (o)

- Tipos de Propagación: NEVER
- El tipo **PROPAGATION_NEVER** indica que el método transaccional no requiere ejecutarse en una transacción y que no la permite.
- Si existe una transacción previamente inicializada, lanzará una excepción.
- Si no existe una transacción previamente inicializada, no crea ninguna transacción.

iv. Spring JDBC - Transaction - iv.viii Spring Tx

iv.viii Spring Tx (p)

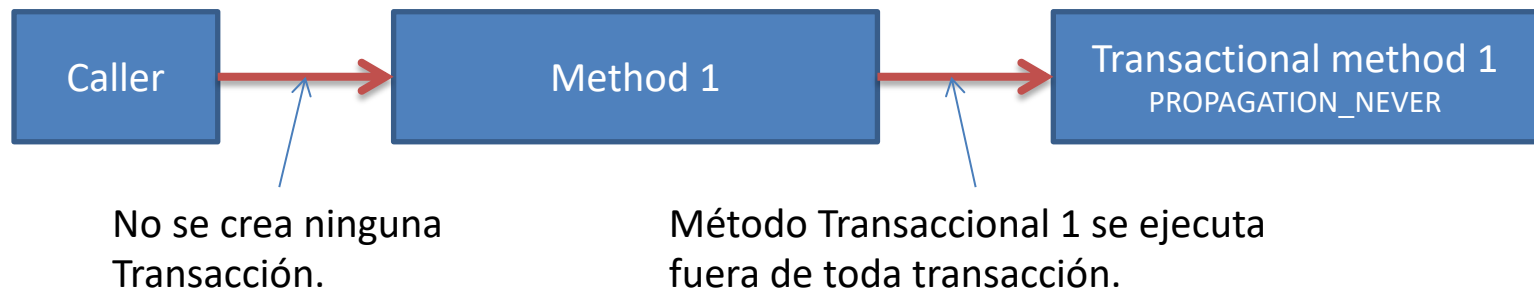
- Tipos de Propagación: NEVER (a)



iv. Spring JDBC - Transaction - iv.viii Spring Tx

iv.viii Spring Tx (q)

- Tipos de Propagación: NEVER (b)



iv. Spring JDBC - Transaction - iv.viii Spring Tx

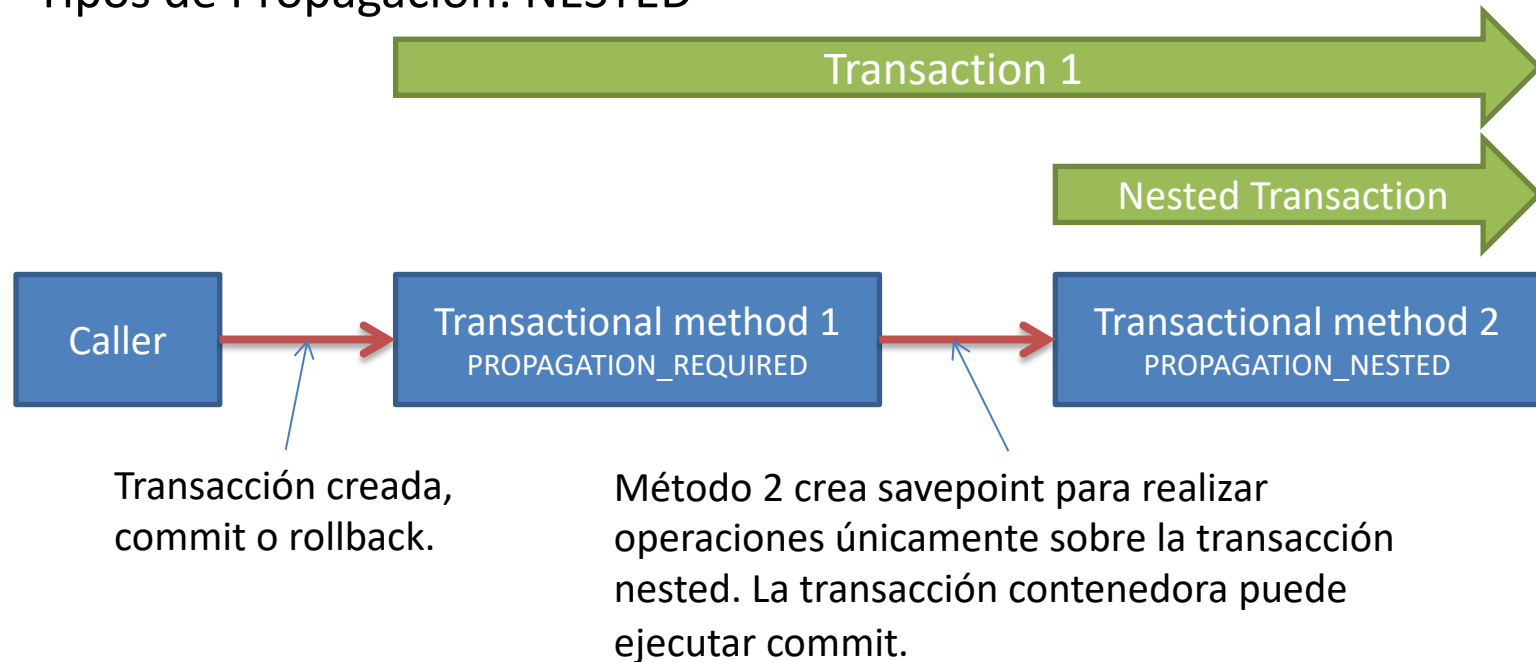
iv.viii Spring Tx (r)

- Tipos de Propagación: NESTED
- El tipo **PROPAGATION_NESTED** indica que el método transaccional requiere ejecutarse en una transacción anidada.
- Si existe una transacción previamente inicializada, creará una transacción anidada. La transacción anidada es independiente de la transacción que la contiene, es posible hacer *rollback* sobre la transacción anidada y *commit* sobre la transacción contenedora o viceversa.
- Si no existe una transacción previamente inicializada, creara una nueva transacción.
- Propagación sólo disponible para DataSourceTransactionManager.

iv. Spring JDBC - Transaction - iv.viii Spring Tx

iv.viii Spring Tx (s)

- Tipos de Propagación: NESTED



iv. Spring JDBC - Transaction - iv.viii Spring Tx

iv.viii Spring Tx

a. Transacciones declarativas configuración por XML

Práctica 26 – Parte 1. Transacciones declarativas
configuración por XML

b. Tipos de Propagación

c. Tipos de Aislamiento

d. Transacciones declarativas configuración por
@Anotaciones

Práctica 26 – Parte 2. Transacciones declarativas
configuración por @Anotaciones

d. Transacciones Programáticas

iv.viii Spring Tx (a)

- Tipos de Aislamiento.
- El tipo de aislamiento de una transacción define el nivel de bloqueo que una transacción ejecutará sobre cierto conjunto de datos, para evitar que dos o más transacciones manipulen los mismos datos de forma concurrente.
- El aislamiento previene mantener información corrupta en la base de datos.

iv. Spring JDBC - Transaction - iv.viii Spring Tx

iv.viii Spring Tx (b)

- Tipos de Aislamiento
- Existen diferentes tipos de problemas al trabajar con transacciones concurrentes.
 - Lectura sucia
 - Lectura no repetible
 - Lectura fantasma

iv. Spring JDBC - Transaction - iv.viii Spring Tx

iv.viii Spring Tx (c)

- Tipos de Aislamiento
- Existen 5 tipos de aislamiento.
 - DEFAULT
 - READ_UNCOMMITTED
 - READ_COMMITTED
 - REPEATABLE_READ
 - SERIALIZABLE

iv. Spring JDBC - Transaction - iv.viii Spring Tx

iv.viii Spring Tx (d)

- Tipos de Aislamiento
- **DAFAULT**: El tipo de **ISOLATION_DEFAULT** utiliza la estrategia de aislamiento por defecto del manejador de base de datos.
- **READ_UNCOMMITTED**: El tipo **ISOLATION_READ_UNCOMMITTED** permite obtener escrituras no *commiteadas* por otra transacción, es posible que ocurra lectura sucia, lectura no repetible y lectura fantasma.

iv. Spring JDBC - Transaction - iv.viii Spring Tx

iv.viii Spring Tx (e)

- Tipos de Aislamiento
- **READ_COMMITTED**: El tipo **ISOLATION_READ_COMMITTED** permite únicamente obtener escrituras *commiteadas* por otras transacciones, es posible que ocurra lectura no repetible y lectura fantasma.
- Este es el nivel de aislamiento por defecto en la mayoría de los manejadores de bases de datos.
- Se previene la lectura sucia.

iv. Spring JDBC - Transaction - iv.viii Spring Tx

iv.viii Spring Tx (f)

- Tipos de Aislamiento
- **REPEATABLE_READ**: El tipo **ISOLATION_REPEATABLE_READ** permite a una transacción poder siempre acceder a la misma información durante su ejecución, es posible que ocurra lectura fantasma.
- Se previene lectura sucia y lectura no repetible.

iv. Spring JDBC - Transaction - iv.viii Spring Tx

iv.viii Spring Tx (g)

- Tipos de Aislamiento
- **SERIALIZABLE**: El tipo **ISOLATION_SERIALIZABLE** previene lectura sucia, lectura no repetible y lectura fantasma.
- Indica el mayor nivel de aislamiento, debido a que bloquea rangos de filas en tablas para que se prevengan la lectura sucia, lectura no repetible y lectura fantasma.
- Ocasiona problemas de performance debido a que una transacción con aislamiento serializable bloquea a otras transacciones y éstas no podrán ejecutarse hasta que la transacción serializable desbloquee el rango de filas en las tablas que utiliza.

iv. Spring JDBC - Transaction - iv.viii Spring Tx

iv.viii Spring Tx

a. Transacciones declarativas configuración por XML

Práctica 26 – Parte 1. Transacciones declarativas
configuración por XML

b. Tipos de Propagación

c. Tipos de Aislamiento

d. Transacciones declarativas configuración por
@Anotaciones

Práctica 26 – Parte 2. Transacciones declarativas
configuración por @Anotaciones

d. Transacciones Programáticas

iv.viii Spring Tx (a)

- Transacciones declarativas configuración por @Anotaciones
- Es posible configurar transaccionabilidad directamente sobre clases o métodos transaccionales mediante la anotación **@Transactional**.
- Habilitar configuración de transacciones por @Anotaciones:

```
<tx:annotation-driven transaction-manager="txManager"/>
```

ó

```
@EnableTransactionManagement
```

iv. Spring JDBC - Transaction - iv.viii Spring Tx

iv.viii Spring Tx (b)

- Configuración por default de @Transactional
- Por default @Transactional (así como <tx:advice>) tiene la siguiente configuración:
 - propagation: PROPAGATION_REQUIRED
 - isolation: ISOLATION_DEFAULT
 - Transacción de lectura y escritura (readonly = false)
 - timeout: -1. Valor por defecto del manejador de base de datos, o ninguno si el manejador no lo soporta.
 - Cualquier excepción RuntimeException (*unchecked*) ejecuta *rollback*.

iv. Spring JDBC - Transaction - iv.viii Spring Tx

iv.viii Spring Tx (c)

- Definición de @Transactional

@Component

@Transactional(readOnly = true)

```
public class TransactionalService implements ITransactionalService {
```

```
    public BusinessObject getBusinessObject(Long id) { ... }
```

```
    @Transactional(readOnly = false, propagation = Propagation.REQUIRES_NEW)
```

```
    public void insertBusinessObject(BusinessObject businessObject) {  
        throw new UnsupportedOperationException();  
    }
```

```
    protected void updateBusinessObject(BusinessObject businessObject) { ... }
```

```
    ...
```

```
}
```

iv. Spring JDBC - Transaction - iv.viii Spring Tx

iv.viii Spring Tx (d)

- Múltiple definición de Transaction Managers con @Transactional
- Solo JTA permite transacciones globales para crear transaccionabilidad sobre múltiples orígenes de datos.
- No es común que una aplicación necesite más de un Transaction Manager (sólo aquellas con transacciones locales con múltiples orígenes de datos).
- Es posible configurar múltiples Transaction Managers, y obtener transaccionabilidad en diferentes orígenes de datos sin embargo, las transacciones de ambos Transaction Managers son independientes.

iv. Spring JDBC - Transaction - iv.viii Spring Tx

iv.viii Spring Tx (e)

- Múltiple definición de Transaction Managers con @Transactional (a)

```
public class TransactionalService {  
  
    @Transactional(value="order")  
    public void setSomething(String name) { ... }  
  
    @Transactional(transactionManager="account")  
    public void doSomething() { ... }  
}  
  
<tx:annotation-driven/>  
  
<bean id="txManager1"  
      class="..DataSourceTransactionManager">  
    ...  
    <qualifier value="order"/>  
</bean>  
  
<bean id="txManager2"  
      class="..DataSourceTransactionManager">  
    ...  
    <qualifier value="account"/>  
</bean>
```

iv. Spring JDBC - Transaction - iv.viii Spring Tx

iv.viii Spring Tx (f)

- Múltiple definición de Transaction Managers con @Transactional (b)

```
public class TransactionalService {  
  
    @Transactional  
    public void setSomething(String name) { ... }  
  
    @Transactional("account")  
    public void doSomething() { ... }  
}
```

```
<tx:annotation-driven  
    transaction-manager="txManager1"/>  
  
<bean id="txManager1"  
    class="..DataSourceTransactionManager">  
    ...  
</bean>  
  
<bean id="txManager2"  
    class="..DataSourceTransactionManager">  
    ...  
    <qualifier value="account"/>  
</bean>
```

iv. Spring JDBC - Transaction - iv.viii Spring Tx

iv.viii Spring Tx (g)

- Combinación de Transaccionabilidad y Aspectos
- La transaccionabilidad se implementa mediante aspectos.
- Es posible implementar aspectos sobre métodos transaccionales.
- Únicamente es requerido ordenar la cadena de ejecución de los Advices.

iv. Spring JDBC - Transaction - iv.viii Spring Tx

iv.viii Spring Tx (h)

- Combinación de Transaccionabilidad y Aspectos
- Configurar ordenamiento de *Transactional Advice*.

`<tx:annotation-driven order="100"/>`

ó

`@EnableTransactionManagement (order="100")`

iv. Spring JDBC - Transaction - iv.viii Spring Tx

iv.viii Spring Tx (i)

- Combinación de Transaccionabilidad y Aspectos
- Configurar ordenamiento de *AOP Advices* (a)

```
@Aspect
@Component("profilingAspect")
public class ProfilingAspect implements Ordered {

    private int order = 101;
    //getter y setter
}
```

iv. Spring JDBC - Transaction - iv.viii Spring Tx

iv.viii Spring Tx (j)

- Combinación de Transaccionabilidad y Aspectos
- Configurar ordenamiento de *AOP Advices* (b)

```
@Aspect  
@Component("profilingAspect")  
@Order(101)  
public class ProfilingAspect {  
  
    ...  
}
```

iv. Spring JDBC - Transaction - iv.viii Spring Tx

iv.viii Spring Tx

a. Transacciones declarativas configuración por XML

Práctica 26 – Parte 1. Transacciones declarativas
configuración por XML

b. Tipos de Propagación

c. Tipos de Aislamiento

d. Transacciones declarativas configuración por
@Anotaciones

Práctica 26 – Parte 2. Transacciones declarativas
configuración por @Anotaciones

d. Transacciones Programáticas

iv.viii Spring Tx. Práctica 26. (a)

- Práctica 26 – Parte 2. Transacciones declarativas configuración por @Anotaciones
- Implementar transaccionabilidad declarativa básica con configuración por @Anotaciones sobre métodos transaccionales logrando ejecutar commit y rollback de forma automática mediante DataSourceTransactionManager.
- Implementar Aspectos mediante transacciones declarativas.

iv. Spring JDBC - Transaction - iv.viii Spring Tx

iv.viii Spring Tx

a. Transacciones declarativas configuración por XML

Práctica 26 – Parte 1. Transacciones declarativas
configuración por XML

b. Tipos de Propagación

c. Tipos de Aislamiento

d. Transacciones declarativas configuración por
@Anotaciones

Práctica 26 – Parte 2. Transacciones declarativas
configuración por @Anotaciones

d. Transacciones Programáticas

iv.viii Spring Tx (a)

- Transacciones programáticas
- Permiten una amplia flexibilidad para implementar transaccionabilidad debido a la invocación explícita de *commit* y *rollback* sin necesidad de tener que lanzar excepciones.
- Se vuelve difícil de mantener.

iv. Spring JDBC - Transaction - iv.viii Spring Tx

iv.viii Spring Tx (b)

- Transacciones programáticas
- Existen 2 formas de implementar transacciones programáticas:
 - Mediante TransactionTemplate.
 - Utilizando el PlatformTransactionManager directamente.
- No se recomienda la implementación de transacciones programáticas.
- En caso de optar por transacciones programáticas se recomienda utilizar **TransactionTemplate**.

iv. Spring JDBC - Transaction - iv.viii Spring Tx

iv.viii Spring Tx (c)

- Utilizando TransactionTemplate
- Similar a JdbcTemplate, Spring implementa el patrón de diseño template para facilitar la implementación de transaccionabilidad al desarrollador.
- El uso de **TransactionTemplate** mantiene un alto acoplamiento entre el código cliente y la infraestructura de Spring.
- Implementa interfaces *callback* para definir el cuerpo de la transacción permitiendo devolver o no algún valor por la transacción.

iv. Spring JDBC - Transaction - iv.viii Spring Tx

iv.viii Spring Tx (d)

- TransactionTemplate, callback TransactionCallback<T>.

```
TransactionTemplate transactionTemplate = new TransactionTemplate(transactionManager);  
transactionTemplate.setPropagationBehavior(TransactionDefinition.PROPGATION_REQUIRED);  
transactionTemplate.setIsolationLevel(TransactionDefinition.ISOLATION_SERIALIZABLE);  
transactionTemplate.setTimeout(30);
```

```
BusinessObject bo = transactionTemplate.execute(new TransactionCallback<BusinessObject>() {  
    public BusinessObject doInTransaction(TransactionStatus status) {  
        try {  
            ...  
            return new BusinessObject();  
        } catch (RuntimeException ex) {  
            status.setRollbackOnly(); throw ex; //puede omitirse el throw  
        }  
    }  
});
```

iv. Spring JDBC - Transaction - iv.viii Spring Tx

iv.viii Spring Tx (e)

- TransactionTemplate, callback TransactionCallbackWithoutResult.

```
TransactionTemplate transactionTemplate = new TransactionTemplate(transactionManager);  
transactionTemplate.setPropagationBehavior(TransactionDefinition.PROPGATION_REQUIRED);  
transactionTemplate.setIsolationLevel(TransactionDefinition.ISOLATION_SERIALIZABLE);  
transactionTemplate.setTimeout(30);
```

```
transactionTemplate.execute(new TransactionCallbackWithoutResult() {  
    public void doInTransactionWithoutResult(TransactionStatus status) {  
        try {  
            ...  
        } catch (RuntimeException ex) {  
            status.setRollbackOnly(); throw ex; //puede omitirse el throw  
        }  
    }  
});
```

Para mayor referencia consultar el paquete test:
org.certificatic.spring.tx.test.programatictx

iv. Spring JDBC - Transaction - iv.viii Spring Tx

iv.viii Spring Tx (f)

- Utilizando PlatformTransactionManager
- Es posible utilizar directamente el API PlatformTransactionManager para manejar transacciones.
- Únicamente es necesario una referencia al bean *transactionManager*.
- La definición de **TransactionDefinition** y la invocación explícita a los métodos *commit* y *rollback* del objeto **TransactionStatus** deben implementarse programáticamente.

iv. Spring JDBC - Transaction - iv.viii Spring Tx

iv.viii Spring Tx (g)

- Utilizando PlatformTransactionManager

```
DefaultTransactionDefinition transactionDefinition = new DefaultTransactionDefinition();  
transactionDefinition.setName("transactionName");  
transactionDefinition.setIsolationLevel(TransactionDefinition.ISOLATION_SERIALIZABLE);  
transactionDefinition.setPropagationBehavior(TransactionDefinition.PROPGATION_REQUIRED);
```

```
TransactionStatus status = transactionManager.getTransaction(transactionDefinition);  
try {  
    ...  
    transactionManager.commit(status);  
} catch (RuntimeException ex) {  
    transactionManager.rollback(status);  
}
```

Para mayor referencia consultar el paquete test:
org.certificatic.spring.tx.test.programatictx

iv. Spring JDBC - Transaction - iv.viii Spring Tx

Resumen de la lección

iv.viii Spring Tx

- Conocimos las formas de configurar transaccionabilidad mediante configuración por XML y anotaciones en Spring Tx.
- Comprendimos los diferentes tipos de propagación.
- Comprendimos los diferentes tipos de aislamiento.
- Analizamos la forma de implementar transaccionabilidad programática mediante TransactionTemplate y directamente haciendo uso del API PlatformTransactionManager.

iv. Spring JDBC - Transaction - iv.viii Spring Tx

Esta página fue intencionalmente dejada en blanco.

iv. Spring JDBC - Transaction - iv.viii Spring Tx