FRAMEWORK

Spring Data JDBC

Módulo III

harriague+asociados

Emilio Rey | María Agustina Aliaga

INCLUIT
empowering your business

Bases de Datos

Una base de datos es una colección de datos.

El **software** llamado sistema **gestor** de bases de datos o **Database Management System (DBMS)**, permite almacenar y posteriormente acceder a los datos de forma rápida y estructurada.

Un **motor** de base de datos es una **instancia de ese software.**

La **computadora** que corre el motor de la BD es un **servidor** de BD.

JDBC

Java Database Connectivity define un conjunto de APIs standard para acceder a BD relacionales.

Define la forma en que un cliente accede a la Base de Datos.

Su **finalidad** es proveer una API a través de la cual se puedan ejecutar los statements SQL contra una base de datos.

Utiliza **drivers** para conectarse a las diferentes BD.

Para **MySql** vamos a tener que utilizar el driver **J-Connector**.

```
    <!-- MySQL database driver -->
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.9</version>
</dependency>
```

Una vez que se carga el driver, éste se registra como una clase **java.sql.DriverManager, que gestiona una lista de drivers y provee métodos estáticos** para establecer conexiones a la BD.

El **método getConnection()** retorna una interfaz java.sql.Connection.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
```

Arquitectura de JDBC

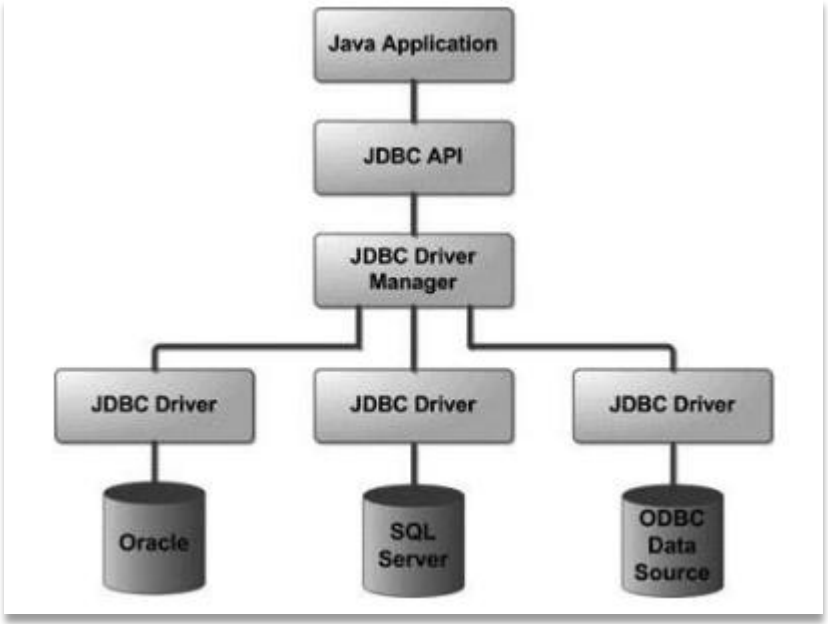
Consiste en **dos capas** principales:

JDBC API: Gestiona la conexión entre la **aplicación** y el **gestor** de BD.

JDBC Driver API: Da soporte a la conexión entre **gestor** y el **driver**.

La API JDBC **utiliza un driver manager y drivers específicos para cada BD**, para proveer una conexión transparente.

Soporta drivers múltiples y concurrentes conectados a múltiples BD diferentes.



Componentes de JDBC

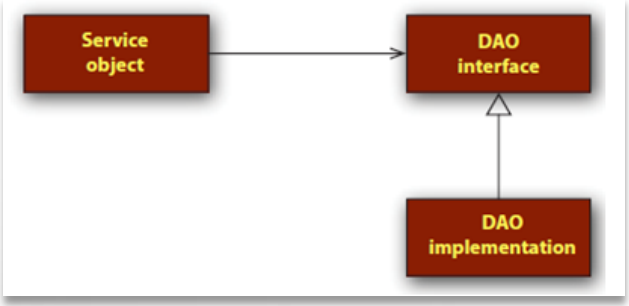
DriverManager	Gestiona una lista de drivers de BD. **Matchea pedidos de conexión de la aplicación java con el driver correspondiente**, el primer driver que reconoce un subprotocolo bajo JDBC será utilizado para establecer una conexión.
Driver	Es una **interfaz que maneja las comunicaciones con el servidor** de BD. El developer no interactúa con drivers sino con DriverManagers.
Connection	Esta interfaz **provee los métodos para conectarse** con la BD.
Statement	Se utilizan **objetos del tipo Statement para enviarlos** a la BD.
ResultSet	Son **objetos que contienen datos traídos de la BD luego de ejecutar una consulta utilizando objetos del tipo Statement**. Actúa como un iterador, permitiendo moverse a través de los datos.
SQLException	**Maneja todos los errores** que ocurren al interactuar con la BD.

Patrón de Diseño DAO

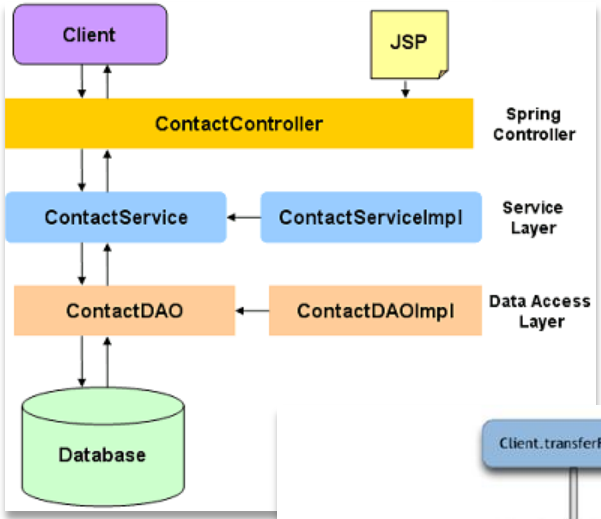
La función del patrón **Data Access Object** es la de **separar** la **lógica de acceso a datos de la lógica de negocio y la de presentación.**

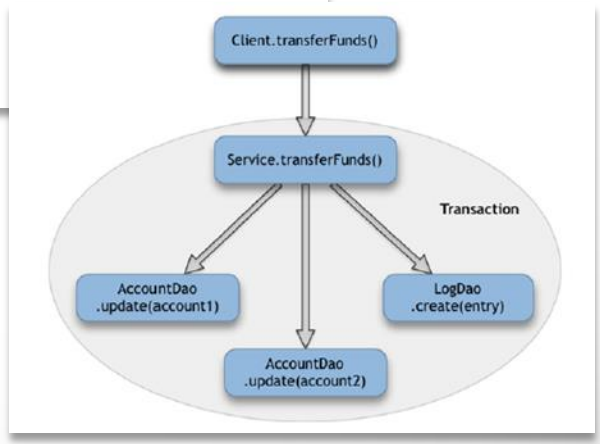
La lógica de acceso a datos es separada en módulos independientes llamados Data Access Objects.

Las **operaciones** de acceso a datos deben ser declaradas en una **interfaz DAO** para permitir diferentes implementaciones dependiendo de las tecnologías a utilizarse.

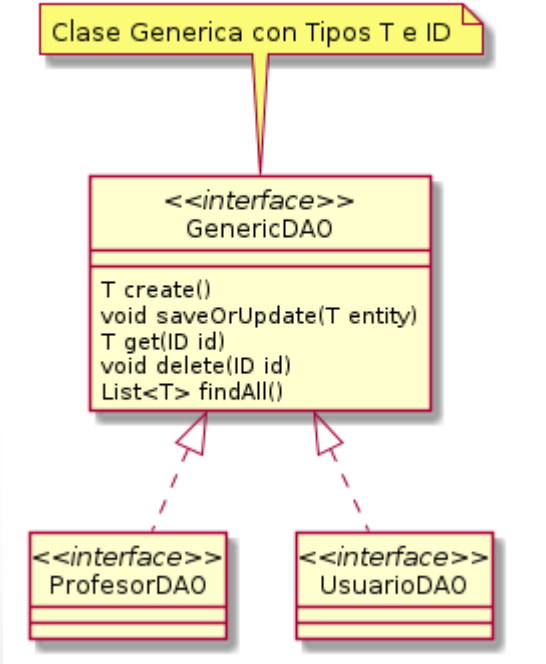


Los objetos de servicio acceden al DAO a través de una interfaz.





Ejemplos de DAO



En el diagrama UML vemos la **interfaz Generic Dao** que es **heredada por dos interfaces, ProfesorDAO y UsuarioDAO**.

Estas son interfaces ya que a su vez **pueden implementarse de diferentes maneras** dependiendo cuál será el origen de los datos.

Entonces tendremos una **ProfesorDAOJdbcImplementation** así como también una **ProfesorDAOHibernateImplementation**, etc.

```
1: public interface GenericDAO<T,ID extends Serializable> {
2:     T create() throws BussinessException;
3:     void saveOrUpdate(T entity) throws BussinessException;
4:     T get(ID id) throws BussinessException;
5:     void delete(ID id) throws BussinessException;
6:     List<T> findAll() throws BussinessException;
7: }
```

Ejemplos de interfaz DAO usando generics

```java
/**
 * Base interface for CRUD operations and common queries
 */
public interface IDaoBase<T> {

    public List<T> loadAll();

    public void save(T domain);

    public void update(T domain);

    public void delete(T domain);

    public T get(Serializable id);

    /**
     * Get list by criteria
     * @param detachedCriteria the domain query criteria, include condition and the orders.
     * @return
     *
     */
    public List<T> getListByCriteria(DetachedCriteria detachedCriteria);

    public List<T> getListByCriteria(DetachedCriteria detachedCriteria, int offset, int size);
}
```

```java
public interface ICrudDao<T extends BEntity> {
    public long count();

    public void delete(Long id);

    public List<T> findAll();

    public T findById(Long id);

    public T findByName(String propertyName, Object value);

    public List<T> findDeleted();

    public T save(T t);

    public T update(T t);
}
```

JDBC Clásico vs. Spring JDBC

```
private static final String SQL_INSERT_SPITTER =
    "insert into spitter (username, password, fullname) values (?, ?, ?)";
private DataSource dataSource;
public void addSpitter(Spitter spitter) {
  Connection conn = null;
  PreparedStatement stmt = null;

 try {
   conn = dataSource.getConnection();                    <- Get connection

   stmt = conn.prepareStatement(SQL_INSERT_SPITTER);     <- Create statement

   stmt.setString(1, spitter.getUsername());             <- Bind parameters
   stmt.setString(2, spitter.getPassword());
   stmt.setString(3, spitter.getFullName());

   stmt.execute();                                       <- Execute statement

 } catch (SQLException e) {
   // do something...not sure what, though              <- Handle exceptions (somehow)

 } finally {
   try {
     if (stmt != null) {                                 <- Clean up
       stmt.close();
     }
     if (conn != null) {
       conn.close();
     }
   } catch (SQLException e) {
     // I'm even less sure about what to do here
   }
 }
}
```

Como vemos en este caso, tenemos más de 20 líneas de código para **insertar** un registro en nuestra BD.

Gran parte de lo que escribimos se trata de:

- gestionar la conexión
- los statements
- las excepciones.

Lo mismo sucede con los demás métodos para realizar un simple **CRUD**.

Utilizando un **JDBC Template**, todas las anteriores líneas de código se reducen a lo siguiente:

```
public void create(String name, Integer age) {
   String SQL = "insert into Student (name, age) values (?, ?)";
   jdbcTemplateObject.update( SQL, name, age);
   System.out.println("Created Record Name = " + name + " Age = " + age);
   return;
}
```

Ejemplo JDBC Clásico + Spring

Crear la tabla en la BD y agregar la dependencia de JConnector

```
CREATE TABLE `customer` (
  `CUST_ID` int(10) unsigned NOT NULL AUTO_INCREMENT,
  `NAME` varchar(100) NOT NULL,
  `AGE` int(10) unsigned NOT NULL,
  PRIMARY KEY (`CUST_ID`)
) ENGINE=InnoDB AUTO_INCREMENT=2 DEFAULT CHARSET=utf8;
```

```
<dependencies>

    <!-- Spring framework -->
  <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring</artifactId>
      <version>2.5.6</version>
  </dependency>

    <!-- MySQL database driver -->
  <dependency>
      <groupId>mysql</groupId>
      <artifactId>mysql-connector-java</artifactId>
      <version>5.1.9</version>
  </dependency>

</dependencies>
```

Crear la clase de dominio y una interfaz DAO con el método insertar y buscar por Id.

```
public class Customer
{
    int custId;
    String name;
    int age;
    //getter and setter methods

}
```

```
public interface CustomerDAO
{
    public void insert(Customer customer);
    public Customer findByCustomerId(int custId);
}
```

Crear la implementación de DAO para Jdbc.

```java
public class JdbcCustomerDAO implements CustomerDAO
{
    private DataSource dataSource;

    public void setDataSource(DataSource dataSource) {
        this.dataSource = dataSource;
    }

    public void insert(Customer customer){

        String sql = "INSERT INTO CUSTOMER " +
                "(CUST_ID, NAME, AGE) VALUES (?, ?, ?)";
        Connection conn = null;

        try {
            conn = dataSource.getConnection();
            PreparedStatement ps = conn.prepareStatement(sql);
            ps.setInt(1, customer.getCustId());
            ps.setString(2, customer.getName());
            ps.setInt(3, customer.getAge());
            ps.executeUpdate();
            ps.close();

        } catch (SQLException e) {
            throw new RuntimeException(e);

        } finally {
            if (conn != null) {
                try {
                    conn.close();
                } catch (SQLException e) {}
            }
        }
    }
```

```java
public Customer findByCustomerId(int custId){

    String sql = "SELECT * FROM CUSTOMER WHERE CUST_ID = ?";

    Connection conn = null;

    try {
        conn = dataSource.getConnection();
        PreparedStatement ps = conn.prepareStatement(sql);
        ps.setInt(1, custId);
        Customer customer = null;
        ResultSet rs = ps.executeQuery();
        if (rs.next()) {
            customer = new Customer(
                rs.getInt("CUST_ID"),
                rs.getString("NAME"),
                rs.getInt("Age")
            );
        }
        rs.close();
        ps.close();
        return customer;
    } catch (SQLException e) {
        throw new RuntimeException(e);
    } finally {
        if (conn != null) {
            try {
            conn.close();
            } catch (SQLException e) {}
        }
    }
}
```

Crear un Spring-customer.xml y declarar el bean para la implementación Jdbc de DAO, y en su property dataSource inyectar un bean dataSource.

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

    <bean id="customerDAO" class="com.mkyong.customer.dao.impl.JdbcCustomerDAO">
        <property name="dataSource" ref="dataSource" />
    </bean>

</beans>
```

Crear un Spring-dataSource.xml y declarar el bean dataSource con sus respectivas propiedades.

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

    <bean id="dataSource"
        class="org.springframework.jdbc.datasource.DriverManagerDataSource">

        <property name="driverClassName" value="com.mysql.jdbc.Driver" />
        <property name="url" value="jdbc:mysql://localhost:3306/mkyongjava" />
        <property name="username" value="root" />
        <property name="password" value="password" />
    </bean>

</beans>
```

Spring-module.xml

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

    <import resource="database/Spring-Datasource.xml" />
    <import resource="customer/Spring-Customer.xml" />

</beans>
```

En Main, referenciamos el Spring-module.xml. Realizar un insert y un select e imprimir los datos.

```
public class App
{
    public static void main( String[] args )
    {
        ApplicationContext context =
            new ClassPathXmlApplicationContext("Spring-Module.xml");

        CustomerDAO customerDAO = (CustomerDAO) context.getBean("customerDAO");
        Customer customer = new Customer(1, "mkyong",28);
        customerDAO.insert(customer);

        Customer customer1 = customerDAO.findByCustomerId(1);
        System.out.println(customer1);

    }
}
```

Spring JDBC

Para hacer más sencillo el uso de JDBC Spring provee un **framework que funciona como una interfaz con la BD.**

Los **Templates JDBC** están diseñados para ***proveer plantillas para diferentes operaciones JDBC.***

Los Templates JDBC aún nos permiten **tener bastante control** y escribir las consultas en Lenguaje SQL.

Si se busca una manera de más alto nivel para acceder a la base de datos, Spring nos da la posibilidad de integrar **frameworks ORM** (Object Relational Mapping) como Hibernate o JPA.

Paquete	Funcionalidad
org.springframework.jdbc.**core**	Posee las clases **JdbcTemplate**, **SimpleJdbcInsert**, **SimpleJdbcCall** and **NamedParameterJdbcTemplate**
org.springframework.jdbc.**datasource**	Clases **utilitarias** para acceder al dataSource.
org.springframework.jdbc.**object**	Acceso a la BD desde un punto de vista orientado a objetos. Permite que las **queries retornen los resultados en forma de objetos del negocio**. **Mapea** los resultados de las queries en columnas y propiedades de los objetos.
org.springframework.jdbc.**support**	Soporta la funcionalidad de **traducción** de las **SQL Exceptions**.

Configurando el Data Source

Para aprovechar al máximo las posibilidades de gestión de acceso a datos que nos provee Spring, es posible **configurar un bean que implemente javax.sql.DataSource.**

La diferencia entre un DataSource y un Connection es que el DataSource provee y gestiona conexiones.

DriverManagerDataSource **(org.springframework.jdbc.datasource)** es la implementación más simple de DataSource.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-3.1.xsd">

    <bean id="dataSource"
        class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <property name="driverClassName">
            <value>${jdbc.driverClassName}</value>
        </property>
        <property name="url">
            <value>${jdbc.url}</value>
        </property>
        <property name="username">
            <value>${jdbc.username}</value>
        </property>
        <property name="password">
            <value>${jdbc.password}</value>
        </property>
    </bean>

    <context:property-placeholder location="jdbc.properties"/>
</beans>
```

```
jdbc.driverClassName=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://localhost:3306/prospring3_ch8
jdbc.username=prospring3
jdbc.password=prospring3
```

También es posible configurar el DataSource en el **código Java** de la siguiente manera:

```
@Configuration
@ComponentScan("org.baeldung.jdbc")
public class SpringJdbcConfig {
    @Bean
    public DataSource mysqlDataSource() {
        DriverManagerDataSource dataSource = new DriverManagerDataSource()
        dataSource.setDriverClassName("com.mysql.jdbc.Driver");
        dataSource.setUrl("jdbc:mysql://localhost:3306/springjdbc");
        dataSource.setUsername("guest_user");
        dataSource.setPassword("guest_password");

        return dataSource;
    }
}
```

Configurando una BD Embebida

Spring provee soporte para BD embebidas como por ejemplo **H2**, Derby, HSQL.

Es recomendable utilizarla cuando queremos **testear** nuestra aplicación.

Estas bases de datos se alojan en la **memoria**, de esta manera es posible **ejecutar nuestros tests** contra una base de datos real.

Esta BD se **crea** y se **destruye** en el **ámbito de ejecución de los tests.**

```
<!-- https://mvnrepository.com/artifact/com.h2database/h2 -->
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <version>1.4.196</version>
    <scope>test</scope>
</dependency>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jdbc="http://www.springframework.org/schema/jdbc"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
        http://www.springframework.org/schema/jdbc
        http://www.springframework.org/schema/jdbc/spring-jdbc-3.1.xsd">

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

</beans>
```

Utilizamos **el tag <jdbc:embedded-database>** para declarar la BD embebida y le asignamos un id.

Instruimos a Spring que **ejecute los scripts** para crear el schema y popular los datos para realizar los tests.

El **primer script** especificado debe ser el archivo que contenga **Data Definition Language (DDL)**.
El **segundo** el que contenga **Data Manipulation Language (DML)**.

Utilizando los Templates JDBC

El framework de Spring JDBC **se ocupa de los detalles de bajo nivel** para permitir que nos concentremos solamente en el código que mueve datos desde y hacia la BD.

La clase JdbcTemplate

JDBC Template es la principal API a través de la que accederemos a funcionalidades como:

- Crear y cerrar **conexiones**
- Ejecutar **statements**
- Iterar sobre los **resultSets** y retornar resultados

.

```
CREATE TABLE Student(
    ID   INT NOT NULL AUTO_INCREMENT,
    NAME VARCHAR(20) NOT NULL,
    AGE  INT NOT NULL,
    PRIMARY KEY (ID)
);
```

Ejemplos de Consultas con JDBC Template:

Insert

```
String insertQuery = "insert into Student (name, age) values (?, ?)";
jdbcTemplateObject.update( insertQuery, name, age);
```

Select

```
String selectQuery = "select * from Student";
List <Student> students = jdbcTemplateObject.query(selectQuery, new StudentMapper());
```

Se utiliza un objeto RowMapper para mapear cada registro a un objeto de la clase Student.

Update

```
String updateQuery = "update Student set age = ? where id = ?";
jdbcTemplateObject.update(updateQuery, age, id);
```

Delete

```
String deleteQuery = "delete from Student where id = ?";
jdbcTemplateObject.update(deleteQuery, id);
```

Consultando y retornando un Objeto

```
String SQL = "select * from Student where id = ?";
Student student = jdbcTemplateObject.queryForObject(
   SQL, new Object[]{10}, new StudentMapper());

public class StudentMapper implements RowMapper<Student> {
   public Student mapRow(ResultSet rs, int rowNum) throws SQLException {
      Student student = new Student();
      student.setID(rs.getInt("id"));
      student.setName(rs.getString("name"));
      student.setAge(rs.getInt("age"));

      return student;
   }
}
```

Ejecutar un Statement SQL o DDL, utilizando el método **execute().**

Crear una tabla:

```
String SQL = "CREATE TABLE Student( " +
   "ID   INT NOT NULL AUTO_INCREMENT, " +
   "NAME VARCHAR(20) NOT NULL, " +
   "AGE  INT NOT NULL, " +
   "PRIMARY KEY (ID));"

jdbcTemplateObject.execute( SQL );
```

Ejemplo: Paso a Paso

1) **Crear una BD** en MySQL llamada TEST.

mysql> CREATE DATABASE TEST;

mysql> USE TEST;

```
CREATE TABLE Student(
   ID   INT NOT NULL AUTO_INCREMENT,
   NAME VARCHAR(20) NOT NULL,
   AGE  INT NOT NULL,
   PRIMARY KEY (ID)
);
```

2) **Configurar el DataSource** en el archivo de **configuración xml** de Spring.

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

3) **Añadir al POM.xml las dependencias:**

mysql-connector para java
org.springframework.jdbc
org.springframework.transaction

```
    <!-- MySQL database driver -->
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.9</version>
</dependency>
```

4) Crear la **interfaz StudentDAO**.java

```
import java.util.List;
import javax.sql.DataSource;

public interface StudentDAO {
     public void setDataSource(DataSource ds);

     public void create(String name, Integer age);

     public Student getStudent(Integer id);

     public List<Student> listStudents();

     public void delete(Integer id);

     public void update(Integer id, Integer age);
}
```

5) Crear la **clase de dominio Student.java**

```
public class Student {
   private Integer age;
   private String name;
   private Integer id;

   public void setAge(Integer age) {
      this.age = age;
   }
   public Integer getAge() {
      return age;
   }
   public void setName(String name) {
      this.name = name;
   }
   public String getName() {
      return name;
   }
   public void setId(Integer id) {
      this.id = id;
   }
   public Integer getId() {
      return id;
   }
}
```

6) Crear la clase **StudentMapper.java, que implementa la interfaz RowMapper.** Row Mapper mapea las filas de un Result Set, a los atributos de un objeto.

```
import java.sql.ResultSet;
import java.sql.SQLException;
import org.springframework.jdbc.core.RowMapper;

public class StudentMapper implements RowMapper<Student> {
   public Student mapRow(ResultSet rs, int rowNum) throws SQLException {
      Student student = new Student();
      student.setId(rs.getInt("id"));
      student.setName(rs.getString("name"));
      student.setAge(rs.getInt("age"));

      return student;
   }
}
```

Mapea los registros de Student a Objetos de la clase Student.

7) Crear la clase **StudentJDBCTemplate.java que implementa los métodos de la interfaz StudentDAO.java**

```
import java.util.List;
import javax.sql.DataSource;
import org.springframework.jdbc.core.JdbcTemplate;

public class StudentJDBCTemplate implements StudentDAO {
   private DataSource dataSource;
   private JdbcTemplate jdbcTemplateObject;

   public void setDataSource(DataSource dataSource) {
      this.dataSource = dataSource;
      this.jdbcTemplateObject = new JdbcTemplate(dataSource);
   }
   public void create(String name, Integer age) {
      String SQL = "insert into Student (name, age) values (?, ?)";
      jdbcTemplateObject.update( SQL, name, age);
      System.out.println("Created Record Name = " + name + " Age = " + age);
      return;
   }
   public Student getStudent(Integer id) {
      String SQL = "select * from Student where id = ?";
      Student student = jdbcTemplateObject.queryForObject(SQL,
         new Object[]{id}, new StudentMapper());

      return student;
   }
   public List<Student> listStudents() {
      String SQL = "select * from Student";
      List <Student> students = jdbcTemplateObject.query(SQL, new StudentMapper());
      return students;
   }
   public void delete(Integer id) {
      String SQL = "delete from Student where id = ?";
      jdbcTemplateObject.update(SQL, id);
      System.out.println("Deleted Record with ID = " + id );
      return;
   }
   public void update(Integer id, Integer age){
      String SQL = "update Student set age = ? where id = ?";
      jdbcTemplateObject.update(SQL, age, id);
      System.out.println("Updated Record with ID = " + id );
      return;
   }
```

8) Crear la clase **Main** de la aplicación donde creamos un objeto del tipo StudentJDBCTemplate, insertamos datos, los mostramos, y los modificamos.

```
import java.util.List;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import com.tutorialspoint.StudentJDBCTemplate;

public class MainApp {
   public static void main(String[] args) {
      ApplicationContext context = new ClassPathXmlApplicationContext("Beans.xml");

      StudentJDBCTemplate studentJDBCTemplate =
         (StudentJDBCTemplate)context.getBean("studentJDBCTemplate");

      System.out.println("------Records Creation--------" );
      studentJDBCTemplate.create("Zara", 11);
      studentJDBCTemplate.create("Nuha", 2);
      studentJDBCTemplate.create("Ayan", 15);

      System.out.println("------Listing Multiple Records--------" );
      List<Student> students = studentJDBCTemplate.listStudents();

      for (Student record : students) {
         System.out.print("ID : " + record.getId() );
         System.out.print(", Name : " + record.getName() );
         System.out.println(", Age : " + record.getAge());
      }

      System.out.println("----Updating Record with ID = 2 -----" );
      studentJDBCTemplate.update(2, 20);

      System.out.println("----Listing Record with ID = 2 -----" );
      Student student = studentJDBCTemplate.getStudent(2);
      System.out.print("ID : " + student.getId() );
      System.out.print(", Name : " + student.getName() );
      System.out.println(", Age : " + student.getAge());
   }
}
```

9) En el archivo de **configuración Beans.xml** agregamos el bean de StudentJDBCTemplate.java que va a hacer referencia al bean dataSource.

```
<?xml version = "1.0" encoding = "UTF-8"?>
<beans xmlns = "http://www.springframework.org/schema/beans"
   xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation = "http://www.springframework.org/schema/beans
   http://www.springframework.org/schema/beans/spring-beans-3.0.xsd ">

   <!-- Initialization for data source -->
   <bean id="dataSource"
      class = "org.springframework.jdbc.datasource.DriverManagerDataSource">
      <property name = "driverClassName" value = "com.mysql.jdbc.Driver"/>
      <property name = "url" value = "jdbc:mysql://localhost:3306/TEST"/>
      <property name = "username" value = "root"/>
      <property name = "password" value = "password"/>
   </bean>

   <!-- Definition for studentJDBCTemplate bean -->
   <bean id = "studentJDBCTemplate"
      class = "com.tutorialspoint.StudentJDBCTemplate">
      <property name = "dataSource" ref = "dataSource" />
   </bean>

</beans>
```

A modo exploratorio, pueden investigar acerca de:

- NamedParameterJDBCTemplate
- SimpleJDBC (SimpleJdbcInsert, SimpleJdbcCall)