El uso de Spring Boot JDBC , sigue siendo una opción bastante habitual con proyectos que tienen consultas SQL complejas o bases de datos legacy en la que preferimos hacer uso de consultas SQL planas y configurar todo un poco a nuestro aire sin abordar temas de frameworks ORM como Hibermate o estandares tipo JPA. Vamos a ver como arrancar esta opción:

```
<dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
    </dependency>

    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-devtools</artifactId>
      <scope>runtime</scope>
      <optional>true</optional>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-test</artifactId>
      <scope>test</scope>
    </dependency>

    <!-- https://mvnrepository.com/artifact/mysql/mysql-connector-java
-->
    <dependency>
      <groupId>mysql</groupId>
```

```
      <artifactId>mysql-connector-java</artifactId>

    </dependency>
```

Una vez tenemos declaradas las dependencias a nivel de Spring Boot el siguiente paso es configurar el dataSource con la cadena de conexión usuario y clave:

```
spring.datasource.url=jdbc:mysql://localhost:8889/rest
spring.datasource.username=root
spring.datasource.password=root
spring.datasource.driver.class=com.mysql.cj.jdbc.Driver
```

Spring Boot JDBC y Templates

Es momento de configurar JDBCTemplates con Spring Boot y generar una implementación básica para una tabla cualquiera en este caso abordaremos el concepto de Persona:

```
package com.arquiitecturajva.jdbc;

public class Persona {
  private String nombre;
  private String apellidos;
  private int edad;

  public String getNombre() {
    return nombre;
  }
  public void setNombre(String nombre) {
    this.nombre = nombre;
  }
  public String getApellidos() {
```

```
    return apellidos;
  }
  public void setApellidos(String apellidos) {
    this.apellidos = apellidos;
  }
  public int getEdad() {
    return edad;
  }
  public void setEdad(int edad) {
    this.edad = edad;
  }
  public Persona(String nombre, String apellidos, int edad) {
    super();
    this.nombre = nombre;
    this.apellidos = apellidos;
    this.edad = edad;
  }
  public Persona(String nombre) {
    this.nombre=nombre;
  }
}
```

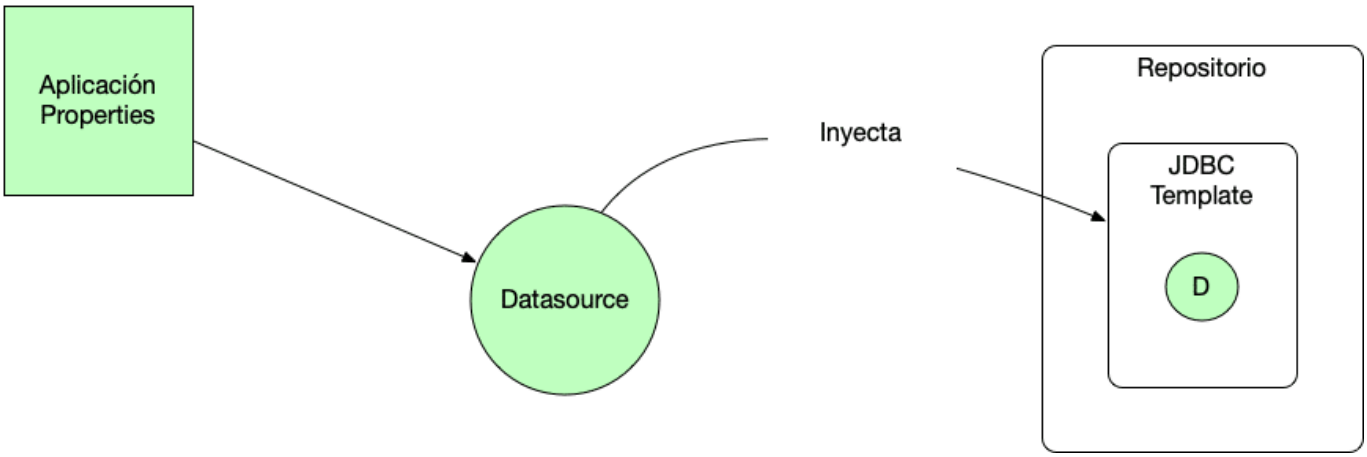
Seguido de la implementación de un repositorio basado en JDBC:

```
package com.arquiitecturajva.jdbc;

import java.util.List;

import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Repository;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
```

```
@Repository
public class PersonaRepository {

  private JdbcTemplate plantilla;

  public PersonaRepository(JdbcTemplate plantilla) {

    this.plantilla = plantilla;
  }
  @Transactional
  public void insertar(Persona persona) {

  plantilla.update("insert into Personas values (?,?,?)",
persona.getNombre(), persona.getApellidos(),
        persona.getEdad());
  }

  public List<Persona> buscarTodos() {

    return plantilla.query("select * from Personas",new
PersonaMapper());
  }
  public Persona buscarUno(String nombre) {

    return plantilla.queryForObject("select * from Personas where
nombre=?",new PersonaMapper(),nombre);
  }
  @Transactional
  public void borrar(Persona persona) {

     plantilla.update("delete from Personas where
```

```
nombre=?",persona.getNombre());
  }
}
```

Esta implementación es extremadamente sencilla y transparente . Quizás para muchas personas demasiado transparente.Hay que recordar que el fichero de application.properties es el encargado de dar de alta un DataSource para nuestra conectividad a la base de datos . Una vez dado de alta este DataSource Spring se encarga de inyectarnos el JDBCTemplate (plantilla) a través de su motor de inyección de dependencias



Es momento de diseñar un par de sencillas pruebas unitarias que se encarguen de confirmar que las consultas funcionan contra la base de datos para ello usaremos unos scripts generales de carga:

```
package com.arquiitecturajva.jdbc;

import static org.junit.jupiter.api.Assertions.assertEquals;

import java.util.List;

import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
```

```
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.jdbc.Sql;

@SpringBootTest
@Sql({ "/schema.sql", "/data.sql" })
class JdbcApplicationTests {

  @Autowired
  PersonaRepository repositorio;
  @Test
  void insertarPersona() {
    Persona persona= new Persona("javier","sanchez",20);
    repositorio.insertar(persona);
    List<Persona> lista=repositorio.buscarTodos();
    assertEquals(2,lista.size());
  }
  @Test
  void borrarPersona() {
    Persona persona= new Persona("pedro");
    repositorio.borrar(persona);
    List<Persona> lista=repositorio.buscarTodos();
    assertEquals(0,lista.size());
  }
  @Test
  void buscarTodos() {
    List<Persona> lista=repositorio.buscarTodos();
    assertEquals(1,lista.size());
  }
  @Test
  void buscarUno() {
    Persona persona=repositorio.buscarUno("pedro");
```

```
    assertEquals("pedro",persona.getNombre());
  }

}
```

Acabamos de configurar Spring Boot JDBC y tenemos todo funcionando.

Otros artículos relacionados

1. Spring Boot Load Data y testing
2. Spring Data Projections y Rendimiento
3. Spring Boot Starter ,un concepto fundamental