Spring Data JDBC

Spring Data JDBC es una biblioteca poderosa que simplifica las interacciones con la base de datos en una aplicación Spring, proporcionando un enfoque ligero y basado en anotaciones para trabajar con bases de datos relacionales. Es menos opinativa que Spring Data JPA y se centra en un método más simple y explícito de mapear objetos a tablas de la base de datos. A continuación, te guiaré a través de los conceptos básicos de cómo usarlo.

1. Configurar tu Proyecto

Para comenzar, necesitarás incluir Spring Data JDBC en tu proyecto. Si estás usando Maven, agrega la siguiente dependencia a tu `pom.xml`:

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

También necesitarás un controlador JDBC para tu base de datos (por ejemplo, H2, MySQL, PostgreSQL). Por ejemplo, para H2:

```
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
</dependency>
```

Si estás usando Gradle, los equivalentes serían:

```
implementation 'org.springframework.boot:spring-boot-starter-data-jdbc'
runtimeOnly 'com.h2database:h2'
```

2. Configurar tu Base de Datos

En tu `application.properties` o `application.yml`, configura la conexión a la base de datos. Para una base de datos H2 en memoria, podría verse así:

```
spring.datasource.url=jdbc:h2:mem:testdb
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=
spring.h2.console.enabled=true
```

Para una base de datos real como PostgreSQL, ajusta la URL, el nombre de usuario y la contraseña en consecuencia.

3. Definir tu Modelo de Dominio

Crea una clase de entidad simple para representar una tabla en tu base de datos. Spring Data JDBC utiliza convenciones donde el nombre de la clase se mapea al nombre de la tabla (en minúsculas por defecto) y los campos se mapean a columnas.

```
import org.springframework.data.annotation.Id;

public class Person {
    @Id
    private Long id;
    private String firstName;
    private String lastName;

    // Constructor predeterminado (requerido por Spring Data JDBC)
    public Person() {}

    public Person(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    // Getters y setters
    public Long getId() { return id; }
    public void setId(Long id) { this.id = id; }
    public String getFirstName() { return firstName; }
    public void setFirstName(String firstName) { this.firstName = firstName; }
    public String getLastName() { return lastName; }
    public void setLastName(String lastName) { this.lastName = lastName; }
}
```

- `@Id` marca la clave primaria.
- Spring Data JDBC espera un constructor sin argumentos.
- La tabla se llamará `person` a menos que se sobrescriba.

4. Crear un Repositorio

Define una interfaz que extienda `CrudRepository` para manejar operaciones CRUD básicas:

```java
import org.springframework.data.repository.CrudRepository;

public interface PersonRepository extends CrudRepository<Person, Long> {
}
```

¡Eso es todo! No necesitas implementarlo—Spring Data JDBC genera la implementación en tiempo de ejecución.

5. Usar el Repositorio

Inyecta el repositorio en un servicio o controlador y úsalo:

```java
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class PersonService {
    private final PersonRepository repository;

    @Autowired
    public PersonService(PersonRepository repository) {
        this.repository = repository;
    }

    public void savePerson() {
        Person person = new Person("John", "Doe");
        repository.save(person);
        System.out.println("Saved person with ID: " + person.getId());
    }

    public void listPeople() {
        Iterable<Person> people = repository.findAll();
        people.forEach(p -> System.out.println(p.getFirstName() + " " + p.getLastName()));
    }
}
```

6. Ejecutar tu Aplicación

Si estás usando Spring Boot, crea una clase principal con `@SpringBootApplication` y ejecútala:

```java
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

Spring Boot configurará automáticamente el datasource y Spring Data JDBC.

7. Opcional: Consultas Personalizadas

Si necesitas consultas personalizadas, puedes agregar métodos a tu repositorio con `@Query`:

```java
import org.springframework.data.jdbc.repository.query.Query;
import org.springframework.data.repository.CrudRepository;

public interface PersonRepository extends CrudRepository<Person, Long> {
    @Query("SELECT * FROM person WHERE first_name = :name")
    List<Person> findByFirstName(String name);
}
```

Puntos Clave a Recordar

- **Sin Carga Lenta:** A diferencia de JPA, Spring Data JDBC no soporta carga lenta. Las relaciones (por ejemplo, uno a muchos) necesitan un modelado explícito con raíces de agregado.
- **Raíz de Agregado:** Cada entidad es parte de un agregado y solo la raíz se guarda o recupera directamente.
- **Creación de Esquema:** Spring Data JDBC no crea tablas automáticamente. Necesitarás proporcionar el esquema (por ejemplo, a través de `schema.sql` en `src/main/resources`):

```sql
CREATE TABLE person (
    id BIGINT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
    first_name VARCHAR(255),
    last_name VARCHAR(255)
);
```

Flujo de Trabajo de Ejemplo

1. Inicia tu aplicación.
2. Guarda un objeto `Person` usando `repository.save()`.
3. Recupéralo con `repository.findById()` o `findAll()`.

¡Eso es lo básico! Si quieres profundizar en relaciones, mapeos personalizados o algo específico, házmelo saber y adaptaré la explicación. ¿Cómo te va hasta ahora?