Spring Data JDBC

Spring Data JDBC is a powerful library that simplifies database interactions in a Spring application by providing a lightweight, annotation-driven approach to working with relational databases. It's less opinionated than Spring Data JPA and focuses on a simpler, more explicit way of mapping objects to database tables. Below, I'll walk you through the basics of how to use it.

1. Set Up Your Project

To get started, you'll need to include Spring Data JDBC in your project. If you're using Maven, add the following dependency to your `pom.xml`:

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

You'll also need a JDBC driver for your database (e.g., H2, MySQL, PostgreSQL). For example, for H2:

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

If you're using Gradle, the equivalents would be:

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

2. Configure Your Database

In your `application.properties` or `application.yml`, configure the database connection. For an H2 in-memory database, it might look like this:

```
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
```

For a real database like PostgreSQL, adjust the URL, username, and password accordingly.

3. Define Your Domain Model

Create a simple entity class to represent a table in your database. Spring Data JDBC uses conventions where the class name maps to the table name (lowercase by default), and fields map to columns.

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

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

    // Default constructor (required by Spring Data JDBC)
    public Person() {}

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

    // Getters and 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` marks the primary key.
- Spring Data JDBC expects a no-args constructor.
- The table will be named `person` unless overridden.

4. Create a Repository

Define an interface that extends `CrudRepository` to handle basic CRUD operations:

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

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

That’s it! You don’t need to implement it—Spring Data JDBC generates the implementation at runtime.

5. Use the Repository

Inject the repository into a service or controller and use it:

```
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. Run Your Application

If you’re using Spring Boot, create a main class with `@SpringBootApplication` and run it:

```
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 will auto-configure the datasource and Spring Data JDBC.

7. Optional: Custom Queries

If you need custom queries, you can add methods to your repository with `@Query`:

```
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);
}
```

Key Points to Remember

- **No Lazy Loading:** Unlike JPA, Spring Data JDBC doesn't support lazy loading. Relationships (e.g., one-to-many) need explicit modeling with aggregate roots.
- **Aggregate Root:** Each entity is part of an aggregate, and only the root is saved or retrieved directly.
- **Schema Creation:** Spring Data JDBC doesn't create tables automatically. You'll need to provide the schema (e.g., via `schema.sql` in `src/main/resources`):

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

Example Workflow

1. Start your app.
2. Save a `Person` object using `repository.save()`.
3. Retrieve it with `repository.findById()` or `findAll()`.

That's the basics! If you want to dive deeper into relationships, custom mappings, or something specific, let me know what you're aiming for, and I'll tailor the explanation. How's this working for you so far?