Spring Data JPA & Database Access

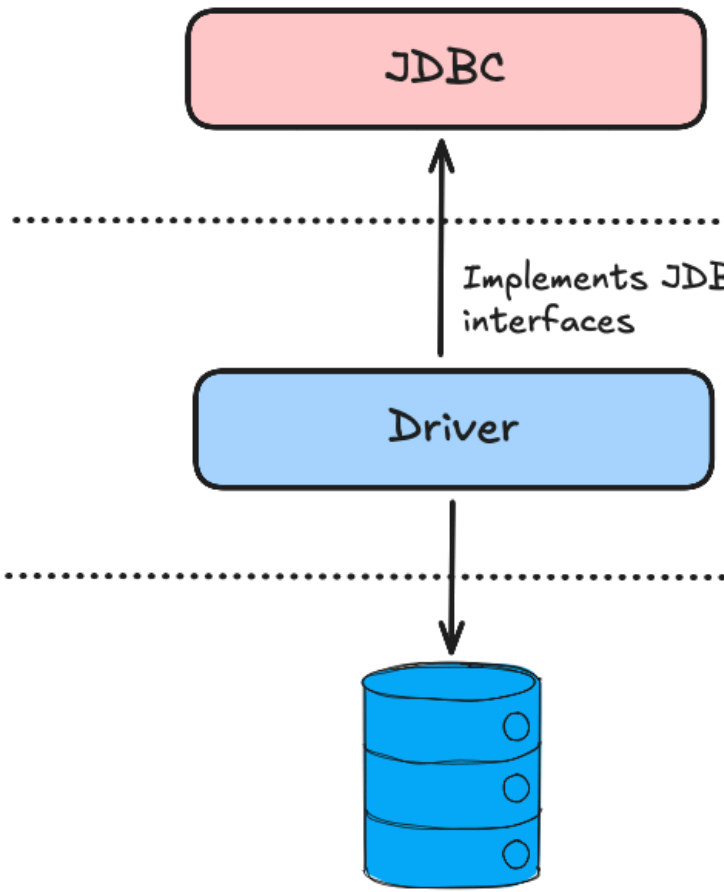
Datamatiker @ EK - 3rd semester

GitHub

Outline

- Recap
- JDBC vs Spring JDBC v.s Spring Data JPA
- JPA Entities
- Repositories
- Relations with JPA (OneToMany, ManyToOne)
- Bidirectional vs Unidirectional Relations
- DTO's with Java Records

Database access with JDBC



Lower level API (interfaces) part of the JDK:

`java.sql`

Database specific driver, in order to talk to a specific Database. Speaks the specific database protocol.
Drivers like MySQL Driver etc.

The actual database, like MySQL, Postgress etc.

Database access with JDBC

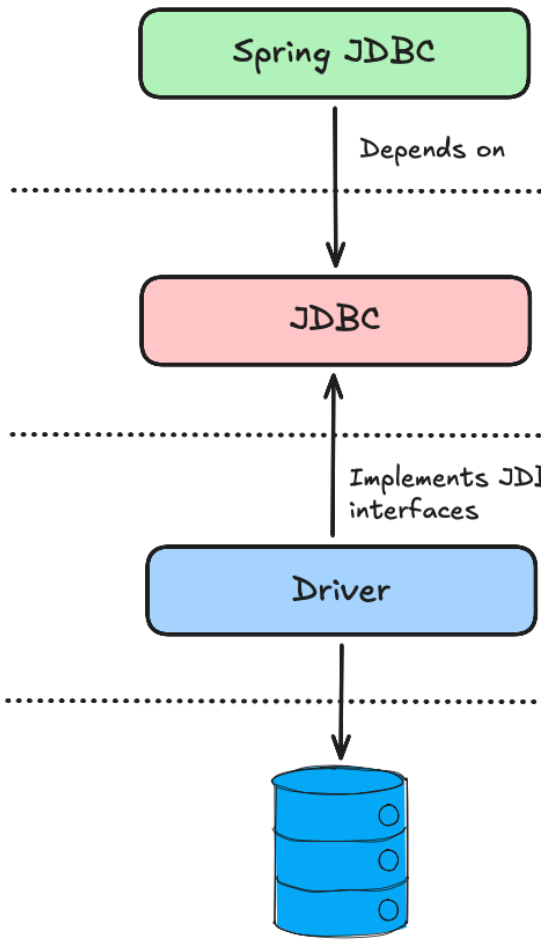
```
public List<Person> findAll() throws SQLException {  1 usage
    String sql = "SELECT * from person";

    try (Connection con = DriverManager.getConnection(url, user, pass);
         PreparedStatement ps = con.prepareStatement(sql);
         ResultSet rs = ps.executeQuery()) {

        List<Person> out = new ArrayList<>();
        while (rs.next()) {
            out.add(PersonMapper.mapRow(rs));
        }
        return out;
    }
}
```

A lot of boilerplate in order to create a simple select statement 🤮
- We need to create different methods manually to get CRUD.
- We need to map the ResultSet to a POJO.
- We need to create tables manually using raw SQL

Database access with Spring JDBC



An abstraction over JDBC, provided through: Spring JDBC
It's part of the Spring framework. Ie.

`org.springframework.jdbc.core.JdbcTemplate`

Lower level API (interfaces) part of the JDK:

`java.sql`

Database specific driver, in order to talk to a specific Database. Speaks the specific database protocol.
Drivers like MySQL Driver etc.

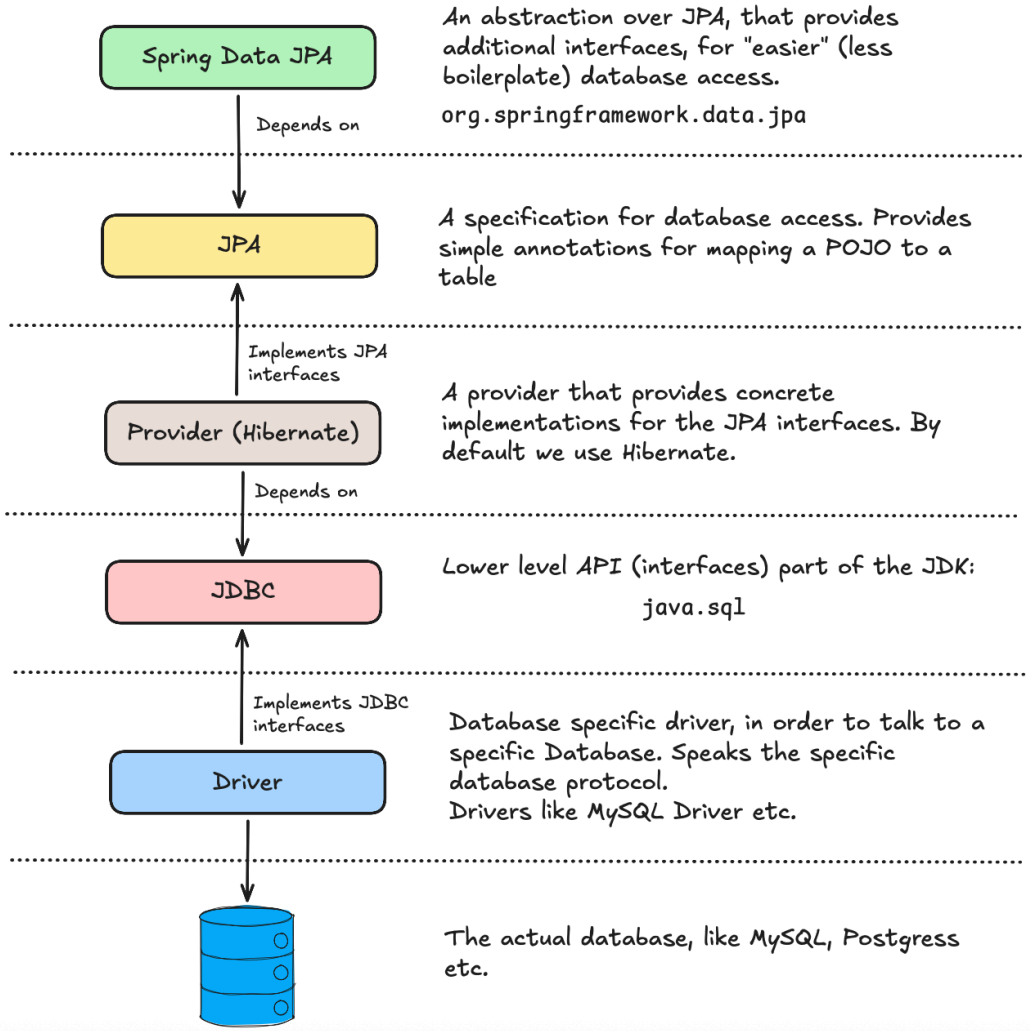
The actual database, like MySQL, Postgress etc.

Database access with Spring JDBC

```
public List<Customer> findAll() {  no usages  new *
    return jdbcTemplate.query(
            sql: "SELECT * FROM person",
            PERSON_ROW_MAPPER
    );
}
```

Less boilerplate in order to create a simple select statement 🙂
- We need to create different methods manually to get CRUD.
- We need to map the ResultSet to a POJO
- We need to create tables manually via raw SQL

Database access with JPA



Database access with JPA

```java
@Entity
class Person {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;  no usages

    public Person() {
    }
}
```

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

No boilerplate 😎
- All CRUD methods are Automatically created for us!
- We only need to tell how to map a Java class to a table!
- Tables can be automatically generated for us. Ie. no raw SQL needed just a little configuration!

Mapping Java objects to DB tables

Tells JPA to map to a DB table

```java
@Entity 1 inheritor
public class Person {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(nullable = false) no usages
    private String name;

    @Column(name = "email_address", unique = true) no usages
    private String email;

    public Person() {
    }

    // Getters and Setters
}
```

A default constructor is needed by JPA

```
mysql> describe person;
+---------------+--------------+------+-----+---------+----------------+
| Field         | Type         | Null | Key | Default | Extra          |
+---------------+--------------+------+-----+---------+----------------+
| id            | bigint       | NO   | PRI | NULL    | auto_increment |
| email_address | varchar(255) | YES  | UNI | NULL    |                |
| name          | varchar(255) | NO   |     | NULL    |                |
+---------------+--------------+------+-----+---------+----------------+
3 rows in set (0.01 sec)
```

JPA Entities

- Annotate class with `@Entity`
- Define primary key with `@Id` (and optionally `@GeneratedValue`)
- Map fields to columns (optional: `@Column`)
- Default constructor (no-args) required
- Getters and setters for fields (optional, but recommended)

JPA Repository

- Create an interface that extends `JpaRepository<T, ID>`
  - `T` : Entity type
  - `ID` : Type of the primary key
- Provides CRUD operations out of the box
- Custom query methods can be defined by following naming conventions

```
public interface UserRepository extends JpaRepository<User, Long> {
}
```

Setting up Spring Data JPA

We need to add the following JPA dependencies to our `pom.xml`:

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

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

Setting up a Driver (MySQL)

We also need to add a database driver dependency. For example, for MySQL:

```
<dependency>
    <groupId>com.mysql</groupId>
    <artifactId>mysql-connector-j</artifactId>
    <scope>runtime</scope>
</dependency>
```

Setting up a Driver (H2 Database)

In **Spring Boot 4** we explicitly need to add `spring-boot-h2console` as a dependency, to access the web-based H2 console:

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

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-h2console</artifactId>
</dependency>
```

Configuring database connection

```
# MySQL configuration - REPLACE WITH OWN DB AND USE ENVIRONMENT VARIABLES IN PRODUCTION
spring.datasource.url=jdbc:mysql://localhost:3306/db-name
spring.datasource.username=ek
spring.datasource.password=ek
spring.jpa.hibernate.ddl-auto=update
```

`ddl-auto` values - for schema generation:

- `create` : Creates the database schema on startup, destroying previous data.
- `create-drop` : Creates the database schema on startup and drops it on shutdown.
- `update` : Updates the database schema on startup without destroying.
- `none` : No action will be performed.

Demo

Exercises

Relations with JPA

JPA supports various types of relationships between entities:

- One-to-Many
- Many-to-One
- Many-to-Many
- One-to-One
- Use annotations like `@OneToMany`, `@ManyToOne`, `@ManyToMany`, and `@OneToOne` to define relationships.

> There exists unidirectional and bidirectional relationships - but this is from a Java perspective, not a database perspective.

Many-to-One Relationship

> `@ManyToOne`: Many instances of an entity are associated with one instance of another entity.

```
@Entity
public class Order {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String orderNumber;
    @ManyToOne
    private Customer customer;
    // Getters and setters
}
```

Many-to-One Relationship

This creates the following tables:

```
CREATE TABLE Customer (
    id BIGINT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(255)
);

CREATE TABLE Order (
    id BIGINT PRIMARY KEY AUTO_INCREMENT,
    order_number VARCHAR(255),
    customer_id BIGINT,
    FOREIGN KEY (customer_id) REFERENCES Customer(id)
);
```

Demo

Exercises

One-to-Many Relationship

Use `@OneToMany` on the parent side (the side with the collection) and `@ManyToOne` on the child side.

```
@Entity
public class Customer {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    @OneToMany(mappedBy = "customer")
    private List<Order> orders;
    // Getters and setters
}
```

> ⚠️ `mappedBy` indicates that the `Order` entity owns the foreign key and owns the relationship.

Owning Side vs Inverse Side

In JPA, **owning side** is purely a mapping / SQL responsibility concept:

- The **owning side** is the side that contains the **FK column** (or defines the join table) and therefore is the side whose state JPA uses to generate **UPDATE/INSERT** for that relationship.
- The **inverse side** uses `mappedBy` and is basically: “mirror this relationship; don’t write the FK from here.”

Bidirectional vs Unidirectional Relationships

Unidirectional: Only one entity has a reference to the other.

Bidirectional: Both entities have references to each other.

- Allow navigation (in Java) from both sides.
- More complexity in terms of **managing the relationship** and when **serializing entities to JSON**.

Demo

Exercises

Infinite Recursion with Bidirectional Relationships

When serializing bidirectional relationships to JSON, infinite recursion can occur.

- Use `@JsonManagedReference` and `@JsonBackReference` to manage serialization.

Or even better, use DTO's to avoid exposing entities directly in API responses.

DTOs with Java Records

> DTOs (Data Transfer Objects) are simple objects used to transfer data between layers, often used to avoid exposing entities directly.

Java Records provide a concise way to create **immutable** DTOs.

```
public record UserDTO(Long id, String name, String email) {}
```

Example usage:

```
UserDTO userDTO = new UserDTO(1L, "John Doe", "john@example).com");
Long id = userDTO.id();
String name = userDTO.name();
String email = userDTO.email();
```

Demo

Exercises