Spring Data JPA

Course notes - Programmering 2 (PROG2)

Version 0.1 - January 2026

by Osman Butt - osnb@ek.dk

Contents

1 Spring Data JPA introduction

Spring Data JPA is an abstraction on top of JPA that replaces most direct JDBC and JdbcTemplate-based data access with repository-based persistence.

If you previously used `JdbcTemplate`, Spring Data JPA shifts your focus:

- from raw SQL to domain models (entities)
- from rows to objects
- from manual mapping to automatic ORM mapping

1.1 ORMs and JPA

Object-Relational Mapping (ORM) is the technique of mapping object-oriented programming concepts to relational database structures. JPA (Jakarta Persistence API) is a specification that defines how to manage relational data in Java applications using ORM.

JPA provides a set of annotations and interfaces to define entity classes, relationships, and queries. By itself, JPA is just a specification and does not provide an implementation. In most Spring Boot applications, JPA is implemented by Hibernate. Hibernate is responsible for:

- generating SQL
- executing JDBC calls
- managing entity state and caching

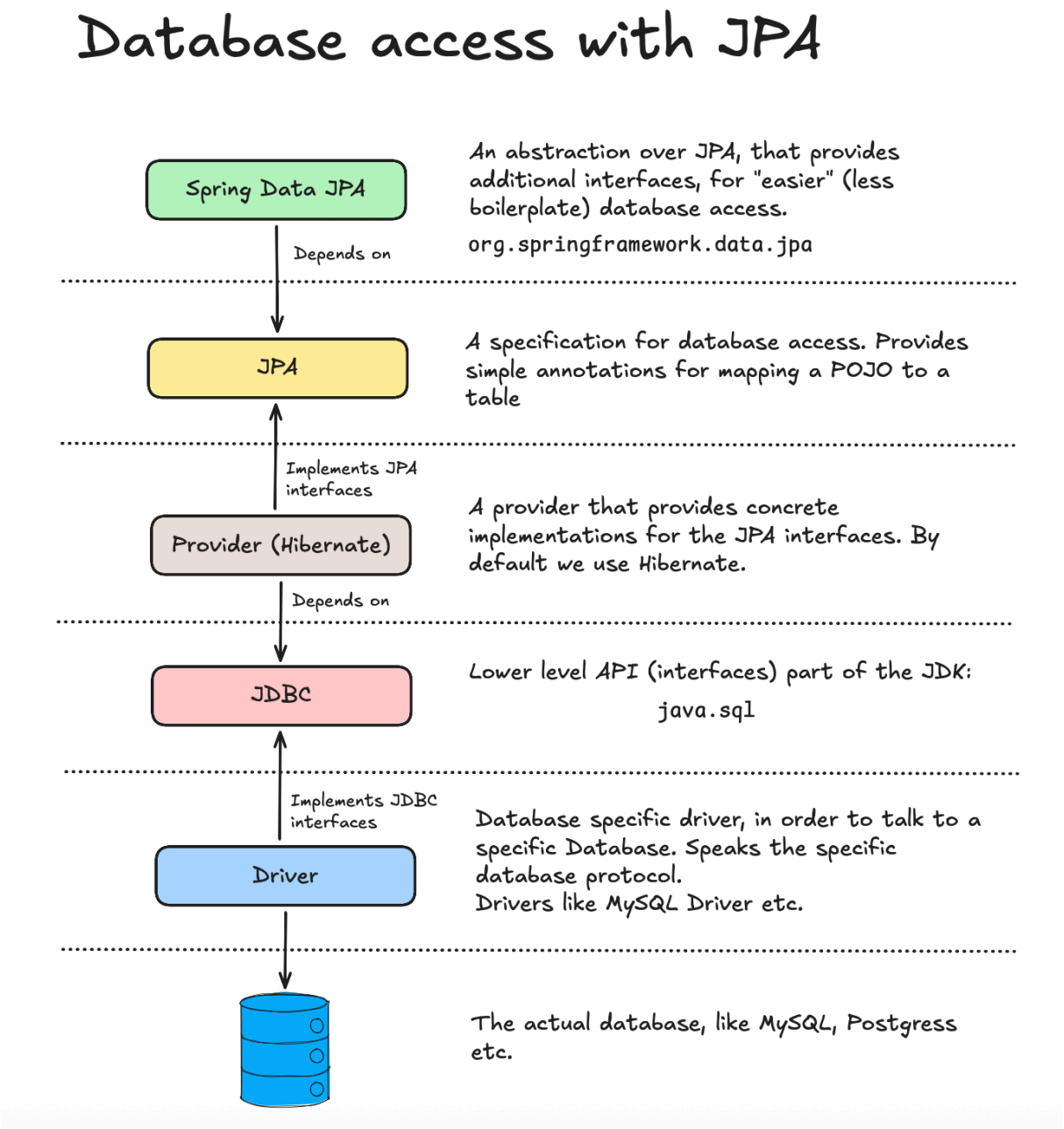
Spring Data JPA builds on top of JPA by providing additional features and abstractions to simplify data access and manipulation. It allows developers to create repositories with minimal boilerplate code, supports query derivation from method names, and provides integration with Spring's transaction management.

1.2 Key Features of Spring Data JPA

- **Repository Abstraction**: Spring Data JPA provides a repository abstraction that allows developers to define repositories as interfaces. The framework automatically generates the implementation at runtime, reducing boilerplate code.
- **Query Derivation**: Spring Data JPA supports query derivation from method names. Developers can define methods in repository interfaces, and Spring Data JPA will generate the corresponding queries based on the method names.
- **Custom Queries**: In addition to query derivation, Spring Data JPA allows developers to define custom queries using JPQL (Java Persistence Query Language) or native SQL.
- **Pagination and Sorting**: Spring Data JPA provides built-in support for pagination and sorting of query results, making it easy to handle large datasets.
- **Auditing**: Spring Data JPA includes auditing features that allow automatic tracking of entity creation and modification timestamps, as well as the user responsible for these actions.
- **Integration with Spring Ecosystem**: Spring Data JPA seamlessly integrates with other Spring projects, such as Spring Boot, Spring Security, and Spring Transaction Management.

1.3 Spring Data JPA Architecture

Spring Data JPA is built on top of the JPA specification and provides a higher-level abstraction for data access. It includes Hibernate as a default JPA provider, but it can also work with other JPA implementations. The overall dependency structure looks like this:



`java.sql` is the standard Java package for JDBC (Java Database Connectivity), which provides a low-level API for interacting with

relational databases. JPA builds on top of JDBC to provide a higher-level abstraction for managing relational data in Java applications.

If you used `JdbcTemplate` before, Hibernate (the JPA provider) now sits between your code and JDBC. You no longer write SQL for most operations - Hibernate generates it for you.

1.4 Getting Started with Spring Data JPA

1.4.1 Dependencies

To get started with Spring Data JPA, you typically need to include the necessary dependencies in your project, configure the data source, and define your entity classes and repository interfaces.

Unlike JdbcTemplate, you do not configure:

- RowMappers
- SQL statements
- manual transaction boundaries

To use Spring Data JPA in a Spring Boot application, you need to include the following dependency in your `pom.xml`:

```
<!-- Spring Data JPA -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<!-- Test support (includes @DataJpaTest) -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
</dependency>
```

You also need to add a database driver dependency, such as H2, MySQL, etc., depending on your database choice:

MySQL Driver:

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

For H2 Database (in-memory database useful for development and testing):

```
<!-- In Spring Boot 4, h2-console is
     not included by default anymore! -->
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
</dependency>
<!-- h2 console support -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-h2console</artifactId>
</dependency>
```

> Note: In Spring Boot 4, the H2 console is not included by default anymore. You need to add the `spring-boot-h2console` dependency separately to enable it.

1.4.2 Configuring a Data Source

To configure a data source in a Spring Boot application, you can specify the database connection properties in the `application.properties`.

MySQL configuration example:

```
spring.datasource.url=jdbc:mysql://localhost:3306/mydb
spring.datasource.username=ek
spring.datasource.password=ek
```

H2 configuration example:

```
# Note that testdb can be any name you choose
spring.datasource.url=jdbc:h2:mem:testdb
spring.datasource.username=sa
spring.datasource.password=

# Enable H2 console
spring.h2.console.enabled=true

# Set the path for H2 console
spring.h2.console.path=/h2-console
```

1.4.3 Defining Entity Classes

In Spring Data JPA, you define entity classes using JPA annotations. An entity class represents a table in the database. For example:

```
@Entity
public class Person {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private String email;

    public Person() {}

    // Getters and setters
}
```

This `Person` class is annotated with `@Entity`, indicating that it is a JPA entity, i.e. Spring Data JPA will pick it up and map it to a database table. The `@Id` annotation marks the **primary key**, and `@GeneratedValue` specifies the strategy for generating primary key values, in this case using the database's identity column feature (`auto-increment` for MySQL).

1.4.4 Accessing Data with Repositories

To access data in Spring Data JPA, we need to extend one of Spring Data JPA's repository interfaces. The most commonly used one is `JpaRepository`, which provides CRUD operations and pagination support, but there are others like `CrudRepository` and `PagingAndSortingRepository`.

The only thing needed to create a repository is to define an interface that extends `JpaRepository`, specifying the entity type and the type of its primary key. For example:

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

With this interface, Spring Data JPA will automatically provide implementations for common CRUD operations such as saving,

finding, updating, and deleting `Person` entities. The `Long` type parameter indicates that the primary key of the `Person` entity is of type `Long`. So generally speaking, the first parameter is the entity type and the second parameter is the type of the entity's primary key.

Note that no implementation class is needed. Spring Data JPA creates a proxy implementation at runtime.

We don't need to annotate the repository interface with `@Repository`, as Spring Data JPA automatically detects it during component scanning.

1.5 Using Spring Data JPA Repositories

Once you have defined your repository interface, you can use it in your service or controller classes by injecting it using Spring's dependency injection. For example:

```
@Service
public class PersonService {
    private final PersonRepository personRepository;
    public PersonService(PersonRepository personRepository) {
        this.personRepository = personRepository;
    }
    public List<Person> getAllPersons() {
        return personRepository.findAll();
    }
}
```

Many methods are available out-of-the-box, such as `findById`, `save`, `deleteById()`, and `findAll`:

- `List<T> findAll()`: Retrieves all entities of type T, in this case, all `Person` entities.
- `Optional<T> findById(ID id)`: Retrieves an entity by its primary key. Returns an `Optional` that may or may not contain the entity.
- `T save(T entity)`: Saves a given entity. If the entity already exists, it updates it; otherwise, it creates a new one.
- `void deleteById(ID id)`: Deletes the entity with the specified primary key.

1.5.1 `Optional<T>` Explained

Optional is a container object that may or may not contain a non-null value. It is used to avoid null checks and NullPointerExceptions. For example, when you call `findById`, it returns an `Optional<Person>`. You can check if a value is present and retrieve it safely:

```
Optional<Person> personOpt = personRepository.findById(1L);
if (personOpt.isEmpty()) {
    // Handle the case where the person is not found
    // e.g., throw an exception or return a default value
}
```

```
// Safely retrieves the Person object
Person person = personOpt.get();
```

1.6 Schema Generation

With JdbcTemplate, schema management is usually handled manually via SQL scripts. Spring Data JPA can automatically generate database schemas from entity definitions (classes with `@Entity`). You can configure this behavior in the `application.properties` file using the `spring.jpa.hibernate.ddl-auto` property.

```
spring.jpa.hibernate.ddl-auto=update
```

The possible values for `ddl-auto` are:

- `none`: No action will be performed.
- `validate`: Validate the schema, making sure it matches the entities.
- `update`: Update the schema to match the entities (creates missing tables, columns, etc., but does not remove unused ones).
- `create`: Create the schema, dropping existing tables if necessary.
- `create-drop`: Create the schema on startup and drop it on shutdown.

Be cautious when using `create` or `create-drop` in production environments, as they will delete existing data!

1.7 Dummy Data on Startup

You can insert dummy data into your database on application startup by implementing the `CommandLineRunner` interface. The `CommandLineRunner` interface has a single method, `run`, which is executed after the application context is loaded (i.e. after the schema is created/updated).

Here's an example of how to use `CommandLineRunner` to insert dummy data:

```java
@Component
public class InitData implements CommandLineRunner {
    private final PersonRepository personRepository;

    public InitData(PersonRepository personRepository) {
        this.personRepository = personRepository;
    }

    @Override
    public void run(String... args) throws Exception {
        // Insert dummy data
        Person p1 = new Person("John Doe", "jd@ek.dk");
        Person p2 = new Person("Jane Smith", "js@ek.dk");
        personRepository.save(p1);
        personRepository.save(p2);
        // Or save multiple entities at once
        // by passing a list to .saveAll
        // personRepository.saveAll(List.of(p1, p2));
    }
}
```

This `InitData` class is annotated with `@Component`, making it a Spring-managed bean. When the application starts, the `run` method is executed, inserting two `Person` entities into the database using the `PersonRepository`.

We can also specify a `data.sql` file in the `src/main/resources` directory to insert dummy data on startup. Spring Boot will automatically execute this SQL script after the schema is created/updated. For example, you can create a `data.sql` file with the following content. To enable this feature, ensure that the following property is set in `application.properties`:

```
spring.sql.init.mode=deferred
```

Then create a `data.sql` file:

```sql
INSERT INTO person(name, email) VALUES ('John Doe', 'jd@ek.dk');
```

1.8 Query Methods

Spring Data JPA allows you to define query methods in your repository interfaces by simply declaring method signatures. The

framework will automatically generate the corresponding queries based on the method names. For example, you can define a method to find persons by their name:

```
public interface PersonRepository extends JpaRepository<Person,
Long> {
    List<Person> findByName(String name);
}
```

In this example, the `findByName` method will generate a query to find all `Person` entities with the specified name. This is because Spring Data JPA interprets the method name and constructs the appropriate query. The `Name` part of the method name corresponds to the `name` field in the `Person` entity, which is why it works.

You can also define more complex queries using multiple criteria, such as:

```
List<Person> findByNameAndEmail(String name, String email);
List<Person> findByNameOrEmail(String name, String email);
List<Person> findByNameContaining(String substring);
```

These methods will generate queries based on the specified criteria, allowing you to easily retrieve data without writing custom query logic.

For more advanced querying needs, you can use the `@Query` annotation to define custom JPQL or native SQL queries directly in your repository interface. For example:

```
@Query("SELECT p FROM Person p WHERE p.email LIKE %:domain")
List<Person> findByEmailDomain(@Param("domain") String domain);
```

Notice that when using `@Query`, it uses JPQL, which operates on entity objects (the Java `Person` entity) and their fields, not directly on database tables and columns.

This flexibility allows you to leverage the power of JPA while still benefiting from the convenience of Spring Data JPA's repository abstraction.

1.9 Slice Testing JPA Repositories

Spring Data JPA provides support for slice testing of JPA repositories using the `@DataJpaTest` annotation. This annotation configures an in-memory database, scans for `@Entity` classes, and configures Spring Data JPA repositories. It is useful for testing the persistence layer of your application in isolation.

Here's an example of how to use `@DataJpaTest` to test a JPA repository:

```
@DataJpaTest
public class PersonRepositoryTest {
    @Autowired
    private PersonRepository personRepository;

    @BeforeEach
    public void setUp() {
        // Initialize test data
        Person person = new Person("John Doe", "jd@ek.dk");
        personRepository.save(person);
    }

    @Test
    public void testFindByName() {
        List<Person> persons = personRepository.findByName("John
Doe");
        assertThat(persons.size()).isEqualTo(1);
        assertThat(persons.get(0).getEmail()).isEqualTo("jd@ek.dk");
    }
}
```

In this example, the `PersonRepositoryTest` class is annotated with `@DataJpaTest`, which sets up a test context for JPA repositories. The `setUp` method initializes test data before each test, and the `testFindByName` method tests the `findByName` repository method.

This approach allows you to test your JPA repositories in isolation, ensuring that they function correctly without the need for a full application context or external database.

When learning Spring Data JPA, you can use `@DataJpaTest` to quickly try out repository methods and verify their behavior in a controlled environment.

1.10 Dirty checking vs. Explicit save

Dirty checking is a mechanism that automatically detects changes made to managed entities within a transaction and synchronizes those changes with the database when the transaction is committed. This means that you don't need to explicitly call the `save` method to persist changes to an entity; simply modifying the entity's fields is sufficient. To use dirty checking, you need to add the `@Transactional` annotation to the service method that modifies the entity. For example:

```
@Service
public class PersonService {
    private final PersonRepository personRepository;
    public PersonService(PersonRepository personRepository) {
        this.personRepository = personRepository;
    }

    @Transactional
    public void updatePersonEmail(Long id, String newEmail) {
        Optional<Person> personOpt = personRepository.findById(id);
        if (personOpt.isEmpty()) {
            throw new EntityNotFoundException("Person not found with
id: " + id);
        }
        Person person = personOpt.get();
        person.setEmail(newEmail);
        // No need to call:
        // personRepository.save(person);
        // since dirty checking will handle it
    }
}
```

In this example, the `updatePersonEmail` method is annotated with `@Transactional`, which means that any changes made to the `Person` entity within this method will be automatically detected and persisted to the database when the transaction is committed.

However, there are scenarios where you might want to explicitly call the `save` method, such as when you are creating a new entity or when you want to ensure that changes are persisted immediately. In such cases, you can use the `save` method as follows:

```
public Person createPerson(String name, String email) {
    Person person = new Person(name, email);
    return personRepository.save(person); // Explicitly saving the
new entity
}
```

1.10.1 Entity lifecycles

In JPA, entities can be in different states:

- **transient:** not associated with a persistence context, i.e. newly created and not yet saved to the database
- **managed:** associated with a persistence context, i.e. retrieved from the database or saved
- **detached:** no longer associated with a persistence context, i.e. previously managed but now detached
- **removed:** marked for removal from the database, but not yet deleted

Dirty checking only applies to managed entities within a transaction. When an entity is retrieved from the database using a repository method (like `findById`), it becomes a managed entity. Any changes made to this entity will be tracked by the persistence context, and dirty checking will ensure that these changes are persisted when the transaction is committed.

Note that dirty checking only works for managed entities within a transaction. If you retrieve an entity outside of a transaction or if the entity is detached, you will need to explicitly call the `save` method to persist any changes.

In summary, dirty checking is a powerful feature of Spring Data JPA that simplifies the process of persisting changes to entities. By using the `@Transactional` annotation, you can take advantage of dirty

checking to automatically synchronize changes with the database without the need for explicit save calls. However, there are still scenarios where explicit saves are necessary, such as when creating new entities or working with detached entities.

1.11 Logging SQL Statements

It's beneficial to see the SQL statements generated by Spring Data JPA and Hibernate for debugging and performance tuning purposes. You can enable SQL logging by adding the following properties to your `application.properties` file:

```
# Enable SQL logging
spring.jpa.show-sql=true
# Format SQL statements
spring.jpa.properties.hibernate.format_sql=true
```

With these settings, Hibernate will log the SQL statements it generates to the console, making it easier to understand how your JPA queries are translated into SQL.

1.12 Summary

Spring Data JPA is a powerful framework that simplifies data access and manipulation in Java applications using JPA. It provides a repository abstraction, query derivation, custom queries, pagination, sorting, auditing, and seamless integration with the Spring ecosystem. By leveraging Spring Data JPA, it reduces boilerplate code and allows developers to focus on business logic rather than data access details. But this comes at the cost of abstraction, so it's important to understand how it works under the hood to use it effectively.

2 Defining restrictions on entity fields

In addition to defining relationships between entities, it's often necessary to enforce restrictions or constraints on entity fields to ensure data integrity and validity. JPA provides several annotations that allow you to specify these constraints directly on the entity fields. One of the most commonly used annotations is `@Column`, which specifies column properties like name, length, nullability, and uniqueness.

Let's say that the `email` field in the `Person` entity should be unique and not null. You can enforce these constraints using the `@Column` annotation as follows:

```
@Entity
public class Person {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    @Column(nullable = false, unique = true)
    private String email;
    // Getters and setters
}
```

We can also specify the maximum length of a string field using the `length` attribute of the `@Column` annotation:

```
@Entity
public class Person {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    @Column(nullable = false, unique = true, length = 100)
    private String email;
    // Getters and setters
}
```

This ensures that the `email` field cannot be null, must be unique across all `Person` entities, and has a maximum length of 100

characters. It's also possible to rename the column in the database using the `name` attribute:

```
@Column(name = "email_address", nullable = false, unique = true,
length = 100)
private String email;
```

This will create a column named `email_address` in the database table instead of the default `email`.

It's also possible to rename the table in the database using the `@Table` annotation on the entity class:

```
@Entity
@Table(name = "people")
public class Person {
    // Entity fields and annotations
}
```

This will create a table named `people` in the database instead of the default `person`.

The `@Column` annotation is a powerful tool for enforcing data integrity and validity at the database level. By specifying constraints directly on entity fields, you can ensure that your application maintains consistent and reliable data.

Applying all the annotations from to the `Person` entity, it will generate the following SQL for the `people` table would look like this:

```
CREATE TABLE people (
    id BIGINT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(255),
    email_address VARCHAR(100) NOT NULL UNIQUE
);
```

3 Adding relations to entities

In real-world applications, entities often have relationships with one another. JPA provides several annotations to define these relationships, including `@OneToOne`, `@OneToMany`, `@ManyToOne`, and `@ManyToMany`. Understanding how to model these relationships is crucial for effective data management in Spring Data JPA.

3.1 Types of Relationships

1. **One-to-One (`@OneToOne`)**: A one-to-one relationship exists when one entity is associated with exactly one instance of another entity. For example, a `Person` entity may have a one-to-one relationship with a `Passport` entity, where each person has one passport.
2. **One-to-Many (`@OneToMany`)**: A one-to-many relationship exists when one entity is associated with multiple instances of another entity. For example, a `Customer` entity may have a one-to-many relationship with an `Order` entity, where each customer can have multiple orders.
3. **Many-to-One (`@ManyToOne`)**: A many-to-one relationship is the inverse of a one-to-many relationship. It exists when multiple instances of one entity are associated with a single instance of another entity. For example, multiple `Order` entities may be associated with a single `Customer` entity.
4. **Many-to-Many (`@ManyToMany`)**: A many-to-many relationship exists when multiple instances of one entity are associated with multiple instances of another entity. For example, a `Student` entity may have a many-to-many relationship with a `Course` entity, where each student can enroll in multiple courses, and each course can have multiple students.

3.2 Defining Relationships in JPA

To define relationships in JPA, you use the appropriate annotations on the entity classes. Here are some examples:

3.2.1 One-to-One Relationship

```
@Entity
public class Person {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    // Getters and setters
}
@Entity
public class Passport {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String number;
    @OneToOne
    private Person person;
    // Getters and setters
}
```

The `@OneToOne` annotation tells JPA that each `Person` has one `Passport`. This will create a foreign key in the `Passport` table referencing the `Person` table, and is equivalent to the following in SQL:

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

CREATE TABLE passport (
    id BIGINT PRIMARY KEY AUTO_INCREMENT,
    number VARCHAR(255),
    person_id BIGINT,
    FOREIGN KEY (person_id) REFERENCES person(id)
);
```

This relationship is called a unidirectional (for Java objects, not SQL) one-to-one relationship because only the `Passport` entity has a reference to the `Person` entity. This means in practice that you can navigate from a `Passport` to its associated `Person`, but not the other way around:

```
Passport passport = passportRepository.findById(1L)
            .orElseThrow(() -> new EntityNotFoundException());
Person person = passport.getPerson();
// The following line would not compile, since Person has no
reference to Passport
// Passport passport = person.getPassport();
```

When saving a `Person` and a `Passport`, you need to ensure that the `Person` is saved first, so that it has an ID to reference in the `Passport`:

```
Person person = new Person();
person.setName("John Doe");
personRepository.save(person); // Save Person first to generate ID
Passport passport = new Passport();
passport.setNumber("123456789");
passport.setPerson(person); // Set the reference to Person
passportRepository.save(passport); // Now save Passport
```

If saved in the wrong order, you will encounter a foreign key constraint violation, since the `Passport` would reference a non-existing `Person`. In Java this will throw a transient property value exception.

3.2.1.1 Bidirectional One-to-One Relationship

If you want to navigate both ways, you can make it a bidirectional relationship by adding a reference to `Passport` in the `Person` entity and using the `mappedBy` attribute:

```
@Entity
public class Person {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    @OneToOne(mappedBy = "person")
    private Passport passport;
    // Getters and setters
}
@Entity
public class Passport {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
```

```
    private Long id;
    private String number;
    @OneToOne
    private Person person;
    // Getters and setters
}
```

This in turn allows you to navigate from `Person` to `Passport` as well:

```
Person person = personRepository.findById(1L)
            .orElseThrow(() -> new EntityNotFoundException());
Passport passport = person.getPassport();
```

The important part here is the `mappedBy` attribute, which indicates that the `passport` field in the `Person` entity is mapped by the `person` field in the `Passport` entity. This tells JPA that the `Passport` entity **owns the relationship, and the foreign key will be created in the `Passport` table**.

This bidirectional relationship allows for more flexible navigation between entities, but it also requires careful management of the relationship to avoid issues such as infinite recursion during serialization to JSON. Note that the owning side (the side without `mappedBy`) is the one that controls the relationship and is responsible for updating the foreign key in the database. This means that you still need to set both sides of the relationship when creating new entities:

```
Person person = new Person();
person.setName("John Doe");
Passport passport = new Passport();
passport.setNumber("123456789");
passport.setPerson(person); // Set the reference to Person
person.setPassport(passport); // Set the reference to Passport
personRepository.save(person); // Save Person first to generate ID
passportRepository.save(passport); // Now save Passport
```

3.2.1.2 Serialization Considerations

When using bidirectional relationships, be cautious when serializing entities to JSON (e.g., in REST APIs) to avoid infinite recursion. You can use annotations like `@JsonManagedReference` and

`@JsonBackReference` from the Jackson library to manage serialization.

```
@Entity
public class Person {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    @OneToOne(mappedBy = "person")
    @JsonManagedReference
    private Passport passport;
    // Getters and setters
}
@Entity
public class Passport {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String number;
    @OneToOne
    @JsonBackReference
    private Person person;
    // Getters and setters
}
```

This makes sure that when serializing a `Person`, the associated `Passport` is included, but when serializing a `Passport`, the associated `Person` is not included, preventing infinite loops.

3.2.1.3 Using `cascade` and `orphanRemoval`

From a design perspective, we may want to enforce that the `Passport` entity cannot exist without a `Person` entity. Furthermore, when we create a new `Person` with a `Passport`, we want to be able to save both entities in one go. This can be achieved using the `cascade` and `orphanRemoval` attributes of the `@OneToOne` annotation.

`cascade` specifies which operations should be cascaded from the parent entity to the associated entity. For example, `CascadeType.PERSIST` will automatically persist the associated entity when the parent entity is persisted. **`orphanRemoval`** indicates

that if the association to the child entity is removed, the child entity should also be deleted from the database.

Here the parent entity is the one with no foreign key (the one with `mappedBy`), i.e. `Person`, and the child entity is the one with the foreign key, i.e. `Passport`.

We can modify the `Person` entity as follows:

```
@Entity
public class Person {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    @OneToOne(mappedBy = "person", cascade = CascadeType.PERSIST,
                orphanRemoval = true)
    @JsonManagedReference
    private Passport passport;
    // Getters and setters
}
```

With this configuration, when we create a new `Person` with a `Passport` and save the `Person`, the `Passport` will be automatically persisted as well. Additionally, if we remove the `Passport` from the `Person`, it will be deleted from the database.

```
Person person = new Person();
person.setName("John Doe");
Passport passport = new Passport();
passport.setNumber("123456789");
passport.setPerson(person); // Set the reference to Person
person.setPassport(passport); // Set the reference to Passport
personRepository.save(person); // Saves both Person and Passport
```

Note that we only need to call `personRepository.save(person)`, since the `cascade` attribute takes care of persisting the `Passport` entity as well.

3.3 Many-to-One and One-to-Many Relationships

A many-to-one relationship exists when multiple instances of one entity are associated with a single instance of another entity. In JPA, you can define a many-to-one relationship using the `@ManyToOne` annotation.

For example, consider a scenario where multiple `Post` entities are associated with a single `Person` entity:

```
@Entity
public class Person {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    // Getters and setters
}
@Entity
public class Post {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String title;
    @ManyToOne
    private Person author;
    // Getters and setters
}
```

In this example, the `Post` entity has a many-to-one relationship with the `Person` entity, indicating that multiple posts can be authored by the same person. This will create a foreign key in the `Post` table referencing the `Person` table.

Just as with one-to-one relationships, this is a unidirectional relationship, since only the `Post` entity has a reference to the `Person` entity. To make it bidirectional, you can add a `@OneToMany` annotation in the `Person` entity:

```
@Entity
public class Person {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
```

```
    private Long id;
    private String name;
    @OneToMany(mappedBy = "author")
    private List<Post> posts = new ArrayList<>();
    // Getters and setters
}
@Entity
public class Post {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String title;
    @ManyToOne
    private Person author;
    // Getters and setters
}
```

This allows you to navigate from a `Person` to their associated `Post` entities as well. Note the use of `mappedBy` in the `@OneToMany` annotation, which indicates that the `posts` field in the `Person` entity is the inverse side of the relationship, i.e. the `Post` entity owns the relationship (the foreign key is in the `Post` table).

When creating new entities with a one-to-many relationship, you need to set both sides of the relationship:

```
Person person = new Person();
person.setName("Jane Doe");
personRepository.save(person); // Save Person first to generate ID

Post post1 = new Post();
post1.setTitle("First Post");
post1.setAuthor(person); // Set the reference to Person
Post post2 = new Post();
post2.setTitle("Second Post");
post2.setAuthor(person); // Set the reference to Person

postRepository.saveAll(List.of(post1, post2)); // Save Posts
```

When saving the `Post` entities, you need to ensure that the `Person` entity is saved first, so that it has an ID to reference in the `Post` entities.

3.4 Many-to-Many Relationships

A many-to-many relationship exists when multiple instances of one entity are associated with multiple instances of another entity. In JPA, you can define a many-to-many relationship using the `@ManyToMany` annotation.

For example, consider a scenario where a `Post` entity can have multiple `Tag` entities, and a `Tag` entity can be associated with multiple `Post` entities:

```
@Entity
public class Post {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String title;
    @ManyToMany
    private List<Tag> tags = new ArrayList<>();
    // Getters and setters
}
@Entity
public class Tag {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;

    // Getters and setters
}
```

In this example, the `Post` entity has a many-to-many relationship with the `Tag` entity. This will create a join table in the database to manage the associations between `Post` and `Tag` entities. Since no `mappedBy` attribute is specified, this is a unidirectional relationship from `Post` to `Tag`. When saving a `Post` with associated `Tag` entities, you need to ensure that the `Tag` entities are saved first, since the `Post` entity is the owning side of the relationship (from a JPA perspective):

```
Tag tag1 = new Tag();
tag1.setName("Java");
```

```
Tag tag2 = new Tag();
tag2.setName("Spring");
tagRepository.saveAll(List.of(tag1, tag2)); // Save Tags first to
generate IDs

Post post = new Post();
post.setTitle("Introduction to Spring Data JPA");
post.getTags().add(tag1); // Associate Tag with Post
post.getTags().add(tag2); // Associate Tag with Post
postRepository.save(post); // Save Post
```

In this example, we first create and save the `Tag` entities to ensure they have IDs. Then, we create a `Post` entity and associate the `Tag` entities with it before saving the `Post` entity.

The relationship can also be made bidirectional by adding a `@ManyToMany` annotation in the `Tag` entity with the `mappedBy` attribute:

```
@Entity
public class Post {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String title;
    @ManyToMany
    private List<Tag> tags = new ArrayList<>();
    // Getters and setters
}
@Entity
public class Tag {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    @ManyToMany(mappedBy = "tags")
    private List<Post> posts = new ArrayList<>();
    // Getters and setters
}
```

This allows you to navigate from a `Tag` to its associated `Post` entities as well. Note the use of `mappedBy` in the `@ManyToMany` annotation in

the `Tag` entity, which indicates that the `Post` entity owns the relationship (the join table is managed by the `Post` entity).

3.5 Fetch Types

By default, Spring Data JPA uses different fetch strategies for different types of relationships. This is because some relationships may involve loading large amounts of data, which can impact performance. So to optimize performance, Spring Data JPA has two different ways of fetching related entities: eager and lazy loading.

- **Eager Loading**: In eager loading, related entities are loaded immediately along with the parent entity. This means that when you retrieve a parent entity, all its related entities are also fetched from the database in the same query or through additional queries. Eager loading is typically used for `@OneToOne` and `@ManyToOne` relationships by default.
- **Lazy Loading**: In lazy loading, related entities are not loaded until they are explicitly accessed. This means that when you retrieve a parent entity, its related entities are not fetched from the database until you try to access them. Lazy loading is typically used for `@OneToMany` and `@ManyToMany` relationships by default.

You can specify the fetch type for a relationship using the `fetch` attribute of the relationship annotations. For example, to specify lazy loading for a `@ManyToOne` relationship, you can do the following:

```
@ManyToOne(fetch = FetchType.LAZY)
private Person author;
```

Conversely, to specify eager loading for a `@OneToMany` relationship, you can do the following:

```
@OneToMany(mappedBy = "author", fetch = FetchType.EAGER)
private List<Post> posts = new ArrayList<>();
```

When using lazy loading, be cautious about accessing lazily loaded associations outside of a transactional context, as this can lead to `LazyInitializationException`. To avoid this, ensure that you

access lazy-loaded associations within a transaction or consider using eager loading for critical associations.

3.6 DTOs and Java Records

3.6.1 Java Records

Java Records, provide a concise syntax for declaring classes that are primarily used to store data. They are immutable by default and automatically generate boilerplate code such as constructors, getters, `equals()`, `hashCode()`, and `toString()` methods. To define a record, you use the `record` keyword followed by the record name and its components.

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

To create an instance of a record, you simply call its constructor with the required values:

```
PersonDTO personDTO = new PersonDTO(1L, "John Doe", "jd@ek.dk");
```

To access the fields of a record, you use the automatically generated accessor methods:

```
Long id = personDTO.id();
String name = personDTO.name();
String email = personDTO.email();
```

Note that there are no setter methods by default, as records are immutable. If you need to create a modified version of a record, you can create a new instance with the desired changes.

```
PersonDTO updatedPersonDTO = new PersonDTO(personDTO.id(), "Jane
Doe", personDTO.email());
```

3.6.2 DTOs (Data Transfer Objects)

When dealing with entity relationships, especially in bidirectional associations, it's common to encounter issues such as infinite recursion during serialization (e.g., when returning entities in a REST API). To mitigate this, you can use Data Transfer Objects (DTOs), which are simple objects that carry data between processes.

Java Records are perfect to use here, since they by default provide a concise syntax for immutable data carriers. By using DTOs, you can control the data that is exposed to clients and avoid serialization issues. For example, instead of returning the `Person` entity directly in your API response, you can create a `PersonDTO` record that only includes the necessary fields, like the one shown above.

You can then map your entities to DTOs before returning them in your API responses, ensuring that you avoid serialization issues and control the data exposed to clients.

To map entities to DTOs, you can map them by using a custom method, or doing it manually:

```
public List<PersonDTO> findAll() {
    List<Person> persons = personRepository.findAll();
    List<PersonDTO> dtoList = new ArrayList<>();
    for (Person person : persons) {
        dtoList.add(toDTO(person));
    }
    return dtoList;
}

public PersonDTO toDTO(Person person) {
    return new PersonDTO(person.getId(), person.getName(),
person.getEmail());
}
```

3.7 Summary

Modeling relationships between entities is a fundamental aspect of using JPA effectively. By understanding and utilizing the various relationship annotations provided by JPA, you can create complex data models that accurately represent the relationships in your application domain. Whether it's one-to-one, one-to-many, many-to-one, or many-to-many relationships, JPA provides the tools necessary to manage these associations seamlessly within your Spring Data JPA applications.

4 Transactions in Spring Data JPA

Transactions are a fundamental concept in database management that ensure data integrity and consistency. A transaction is a sequence of operations performed as a single logical unit of work. This means that either all operations within the transaction are completed successfully, or none of them are applied, maintaining the integrity of the database. To give an example, consider a banking application where you need to transfer money from one account to another. This operation involves two steps: debiting the amount from the sender's account and crediting it to the receiver's account. If either of these steps fails, the entire transaction should be rolled back to prevent inconsistencies in the account balances. In the following, we will use the banking example to illustrate how transactions work in Spring Data JPA.

In Spring Data JPA, transactions are managed using the `@Transactional` annotation, which allows you to define the scope of a transaction at the method or class level. Within transactional methods, all database operations are executed within a single transaction context. If any operation fails, the entire transaction is rolled back, ensuring that the database remains in a consistent state. As mentioned earlier, dirty checking is a key feature of JPA that works in conjunction with transactions. When an entity is modified within a transactional context, JPA automatically detects the changes and synchronizes them with the database when the transaction is committed. This means that you don't need to explicitly call the `save` method to persist changes to an entity; simply modifying the entity's fields is sufficient. To use dirty checking, you need to add the `@Transactional` annotation to the service method that modifies the entity.

Here's an example of how to use transactions in a banking application:

```
@Service
public class BankingService {
    private final AccountRepository accountRepository;

    public BankingService(AccountRepository accountRepository) {
        this.accountRepository = accountRepository;
    }

    @Transactional
    public void transferMoney(
        Long fromAccountId,
        Long toAccountId,
        BigDecimal amount
        ) {
        Account fromAccount = accountRepository
            .findById(fromAccountId)
            .orElseThrow(
                () -> new EntityNotFoundException(
                    "Account not found: " + fromAccountId
                    ));
        Account toAccount = accountRepository
            .findById(toAccountId)
            .orElseThrow(
                () -> new EntityNotFoundException(
                    "Account not found: " + toAccountId
                    ));
        fromAccount.debit(amount);
        toAccount.credit(amount);
        // No need to call save, dirty checking will handle it
    }
}
```

In this example, the `transferMoney` method is annotated with `@Transactional`, which means that all operations within this method are executed within a single transaction. If any operation fails (e.g., if an account is not found), the entire transaction is rolled back, and no changes are applied to the database. The debit and credit operations on the `Account` entities are automatically detected by JPA's dirty checking mechanism, and the changes are persisted when the transaction is committed.

We can simulate a failure by throwing an exception during the transfer process:

```
@Transactional
public void transferMoney(
    Long fromAccountId,
    Long toAccountId,
    BigDecimal amount
    ) {
    Account fromAccount = accountRepository
        .findById(fromAccountId)
        .orElseThrow(
                () -> new EntityNotFoundException(
                    "Account not found: " + fromAccountId
                    ));
    Account toAccount = accountRepository
        .findById(toAccountId)
        .orElseThrow(
                () -> new EntityNotFoundException(
                    "Account not found: " + toAccountId
                    ));
    fromAccount.debit(amount);
    // Call save explicitly to illustrate the point
    accountRepository.save(fromAccount);
    // Simulate a failure
    if (true) {
        throw new RuntimeException("Simulated failure during
transfer");
    }
    toAccount.credit(amount);
    accountRepository.save(toAccount);
}
```

In this modified example, we simulate a failure by throwing a `RuntimeException` after debiting the `fromAccount`. When this exception is thrown, the transaction is rolled back, and neither the debit nor the credit operation is applied to the database. This ensures that the account balances remain consistent, and no partial updates occur.

4.1 Read-Only Transactions

In addition to standard transactions, Spring Data JPA also supports read-only transactions. Read-only transactions are used for operations that only involve reading data from the database without making any modifications. By marking a transaction as read-only, you can optimize performance and reduce the overhead associated with write operations. To define a read-only transaction, you can use the `@Transactional` annotation with the `readOnly` attribute set to `true`:

```
@Transactional(readOnly = true)
public List<Account> getAllAccounts() {
    return accountRepository.findAll();
}
```

In this example, the `getAllAccounts` method is marked as a read-only transaction. This indicates to the underlying database that no modifications will be made during the transaction, allowing for potential optimizations.

4.2 Summary

Transactions are a critical aspect of database management that ensure data integrity and consistency. Spring Data JPA provides robust support for transactions through the `@Transactional` annotation, allowing you to define the scope of transactions at the method or class level. By leveraging transactions, you can ensure that multiple database operations are executed as a single logical unit of work, with automatic rollback in case of failures. Additionally, JPA's dirty checking mechanism simplifies the process of persisting changes to entities within transactional contexts. Whether you're performing complex operations like money transfers or simply reading data, understanding and utilizing transactions in Spring Data JPA is essential for building reliable and consistent applications.

5 Query methods in Spring Data JPA

5.1 Query Methods Overview

Spring Data JPA provides a powerful feature called query methods, which allows you to define custom queries directly in your repository interfaces using method names. This feature enables you to perform complex database operations without writing explicit SQL or JPQL queries. Instead, Spring Data JPA derives the queries based on the method names you define in your repository interfaces.

For example, consider a `Person` entity with fields `name` and `email`. You can create a repository interface for the `Person` entity and define query methods as follows:

```
public interface PersonRepository extends JpaRepository<Person,
Long> {
    List<Person> findByName(String name);
    List<Person> findByEmailContaining(String emailFragment);
    List<Person> findByNameAndEmail(String name, String email);
}
```

In this example, the `PersonRepository` interface extends `JpaRepository`, which provides basic CRUD operations. The custom query methods `findByName`, `findByEmailContaining`, and `findByNameAndEmail` are defined based on the naming conventions provided by Spring Data JPA. When you call these methods, Spring Data JPA automatically generates the corresponding SQL queries to retrieve the desired data from the database.

The naming conventions for query methods are based on the properties of the entity and the desired query operation. Some common keywords used in method names include:

- `findBy`: Retrieves entities based on the specified criteria. Example: `findByName` retrieves all persons with the given name.
- `And`: Combines multiple criteria using a logical AND. Example: `findByNameAndEmail` retrieves persons with the specified name and email.

- `Or`: Combines multiple criteria using a logical OR. Example: `findByNameOrEmail` retrieves persons with either the specified name or email.
- `Containing`: Performs a LIKE operation to find entities containing a specific substring. Example: `findByEmailContaining` retrieves persons whose email contains the specified fragment.
- `GreaterThan`, `LessThan`: Compares values using greater than or less than operators. Example: `findByAgeGreaterThan` retrieves persons older than the specified age.
- `Between`: Finds entities with values within a specified range. Example: `findByAgeBetween` retrieves persons whose age is within the specified range.
- `OrderBy`: Specifies the sorting order of the results. Example: `findByNameOrderByEmailAsc` retrieves persons with the specified name, sorted by email in ascending order.

These keywords can be combined to create complex queries based on your application's requirements. Spring Data JPA supports a wide range of keywords and operators, allowing you to build sophisticated queries using intuitive method names. All implementations of these methods are provided automatically at runtime by Spring Data JPA, so you don't need to write any implementation code.

5.2 Custom Queries with `@Query` Annotation

It is also possible to define your own custom queries using the `@Query` annotation. This allows you to write JPQL or native SQL queries directly in your repository methods. For example:

```
public interface PersonRepository extends JpaRepository<Person,
Long> {
    @Query("SELECT p FROM Person p WHERE p.name = :name")
    List<Person> findPersonsByName(@Param("name") String name);

    @Query(value = "SELECT * FROM person WHERE email LIKE
%:emailFragment%", nativeQuery = true)
    List<Person> findPersonsByEmailFragment(@Param("emailFragment")
```

```
String emailFragment);
}
```

In this example, the `findPersonsByName` method uses a JPQL query to retrieve persons by name, while the `findPersonsByEmailFragment` method uses a native SQL query to retrieve persons whose email contains a specific fragment.

A JPQL query is similar to SQL, but it operates on entity objects and their properties rather than directly on database tables and columns. This allows you to write queries that are database-agnostic and work across different database systems. JPQL supports various features such as joins, aggregations, and subqueries, making it a powerful tool for querying data in Spring Data JPA.

A join in JPQL allows you to retrieve related entities based on their relationships. For example, consider a scenario where you have a `Person` entity and a `Post` entity, with a one-to-many relationship between them (one person can have many posts). You can use a join in JPQL to retrieve all posts authored by a specific person:

```
public interface PostRepository extends JpaRepository<Post, Long> {
    @Query("SELECT p FROM Post p JOIN FETCH p.author a WHERE a.name
= :authorName")
    List<Post> findPostsByAuthorName(@Param("authorName") String
authorName);
}
```

The `JOIN FETCH` clause is used to fetch the associated `author` entity along with the `Post` entities, allowing you to access the author's details without triggering additional queries.

5.3 Summary

Query methods in Spring Data JPA provide a convenient way to define custom queries directly in repository interfaces using method names. This feature allows you to perform complex database operations without writing explicit SQL or JPQL queries, simplifying data access and manipulation in your applications. Additionally, the

`@Query` annotation enables you to define custom JPQL or native SQL queries for more advanced querying needs. By leveraging query methods and custom queries, you can build powerful and flexible data access layers in your Spring Data JPA applications.

6 Projections in Spring Data JPA

Projections in Spring Data JPA allow you to retrieve a subset of an entity's attributes rather than the entire entity. This is particularly useful when you want to optimize performance by fetching only the necessary data from the database, reducing the amount of data transferred over the network and minimizing memory usage. Projections can be defined using interfaces or classes, and Spring Data JPA automatically maps the query results to the projection type.

It is especially useful when dealing with large entities or when you only need a few fields for a specific operation, such as displaying a list of items in a user interface, or when wanting to avoid lazy loading of related entities.

For example, consider a `Person` entity with fields `id`, `name`, `email`, and `age`. You can define a projection interface to retrieve only the `name` and `email` fields as follows:

```
public interface PersonNameEmailProjection {
    String getName();
    String getEmail();
}

public interface PersonRepository extends JpaRepository<Person,
Long> {
    List<PersonNameEmailProjection> findByAgeGreaterThan(int age);
}
```

In this example, the `PersonNameEmailProjection` interface defines the fields to be retrieved. The `findByAgeGreaterThan` method in the `PersonRepository` returns a list of `PersonNameEmailProjection` instances, containing only the `name` and `email` fields of persons older than the specified age.

6.1 Summary

Projections in Spring Data JPA provide a powerful way to optimize data retrieval by allowing you to fetch only the necessary attributes

of an entity. By defining projection interfaces or classes, you can reduce the amount of data transferred from the database, improve performance, and minimize memory usage. Projections are particularly useful when dealing with large entities or when you only need a subset of fields for specific operations. By leveraging projections, you can build efficient and responsive applications using Spring Data JPA.

7 Entity mappings beyond relationships

There exists many advanced mapping features in JPA beyond basic entity relationships. Here are some common advanced mapping techniques and best practices:

7.1 Embeddables (`@Embeddable` / `@Embedded`)

Use embeddables for value objects that belong to an entity and have no independent identity (e.g., Address).

```
@Embeddable
public class Address {
    private String street;
    private String city;
}

@Entity
public class Person {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    @Embedded
    private Address address;
}
```

Benefits: keeps the domain model expressive and avoids additional tables.

7.2 Composite keys (`@EmbeddedId` / `@IdClass`)

When a natural key spans multiple columns use `@EmbeddedId` (preferred) or `@IdClass`.

```
@Embeddable
public class OrderKey implements Serializable {
    private Long customerId;
    private Long orderNumber;
}

@Entity
public class Order {
    @EmbeddedId private OrderKey id;
    private BigDecimal total;
}
```

Guideline: avoid composite keys unless they model real domain uniqueness; prefer surrogate `@Id` where practical.

7.3 Enum mapping

When dealing with enums in entities, use the `@Enumerated` annotation to specify how the enum should be stored in the database. There are two strategies: `ORDINAL` and `STRING`. The `ORDINAL` strategy stores the enum's ordinal value (its position in the enum declaration), while the `STRING` strategy stores the enum's name. We can add it to our `Post` entity as follows:

```
@Entity
public class Post {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String title;
    @Enumerated(EnumType.STRING)
    private PostStatus status;
    // Getters and setters
}

public enum PostStatus {
    DRAFT,
    PUBLISHED,
    ARCHIVED
}
```

In this example, the `Post` entity has a `status` field of type `PostStatus`, which is an enum. The `@Enumerated(EnumType.STRING)` annotation specifies that the enum should be stored as a string in the database.

7.4 Mapped Superclasses (`@MappedSuperclass`)

Use `@MappedSuperclass` for base classes that provide common mappings to multiple entities without being entities themselves. A common use case is to define audit fields like `createdAt` and `updatedAt`.

```
@MappedSuperclass
public abstract class Auditable {
    private LocalDateTime createdAt;
    private LocalDateTime updatedAt;
    // Constructors, getters and setters
}

@Entity
public class Person extends Auditable {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
}
```

Another example is to create a base entity class that includes common fields such as `id`. This base class can then be extended by other entity classes to inherit these common fields and their mappings. We use an `Abstract` class to make sure that the base class itself is not treated as an entity.

```
@MappedSuperclass
public abstract class BaseEntity {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    // constructors, getters and setters
}

@Entity
public class Person extends BaseEntity {
    private String name;
    private String email;
    // constructors, getters and setters
}

@Entity
public class Post extends BaseEntity {
    private String title;
    private String content;
    // constructors, getters and setters
}
```

8 Pagination and Sorting with Spring Data JPA

When dealing with large datasets, it is often necessary to retrieve data in smaller, more manageable chunks rather than loading the entire dataset into memory at once. This is where pagination and sorting come into play. Spring Data JPA provides built-in support for pagination and sorting through the `Pageable` and `Sort` interfaces.

The `JpaRepository` interface extends `ListPagingAndSortingRepository`, which in turn extends `PagingAndSortingRepository` and provides methods for pagination and sorting. You can use these methods to retrieve paginated and sorted data from your repositories.

8.1 Pagination

Pagination allows you to retrieve a specific subset of data from a larger dataset. In Spring Data JPA, you can achieve pagination by using the `Pageable` interface in your repository methods. The `Pageable` interface allows you to specify the page number, page size, and sorting criteria for the data retrieval.

To use pagination, you can create a `Pageable` object using the `PageRequest` class and pass it to the repository method:

```
Pageable pageable = PageRequest.of(0, 10); // First page, 10 items
per page
Page<Person> personPage = personRepository.findAll(pageable);
List<Person> persons = personPage.getContent(); // Get the list of
persons
int totalPages = personPage.getTotalPages(); // Get total number of
pages
long totalElements = personPage.getTotalElements(); // Get total
number of elements
```

In this example, we create a `Pageable` object that specifies the first page (page index 0) with a page size of 10 items. We then call the `findAll` method of the `PersonRepository`, which returns a `Page<Person>` object containing the paginated data. We can retrieve

the list of persons, total number of pages, and total number of elements from the `Page` object.

8.2 Sorting

Sorting allows you to order the retrieved data based on specific criteria. In Spring Data JPA, you can achieve sorting by using the `Sort` interface in your repository methods. The `Sort` interface allows you to specify the sorting direction (ascending or descending) and the properties to sort by. For example, consider the same `Person` entity and repository interface:

```
Sort sort = Sort.by(Sort.Direction.ASC, "name"); // Sort by name in
ascending order
List<Person> sortedPersons = personRepository.findAll(sort);
```

In this example, we create a `Sort` object that specifies sorting by the `name` property (must match the entity's field name) in ascending order. We then call the `findAll` method of the `PersonRepository`, which returns a list of persons sorted by name.

It's also possible to combine pagination and sorting in a single query by using the `Pageable` interface, which includes sorting capabilities:

```
Pageable pageable = PageRequest.of(0, 10,
Sort.by(Sort.Direction.DESC, "email")); // First page, 10 items per
page, sorted by email in descending order
Page<Person> personPage = personRepository.findAll(pageable);
List<Person> persons = personPage.getContent(); // Get the list of
persons
```

In this example, we create a `Pageable` object that specifies the first page with a page size of 10 items, sorted by the `email` property in descending order. We then call the `findAll` method of the `PersonRepository`, which returns a `Page<Person>` object containing the paginated and sorted data.