Spring Data JPA

1. Introduction to Spring Data JPA

What is Spring Data JPA?

Spring Data JPA is a part of the larger Spring Data family that simplifies database access and management using Java Persistence API (JPA). It provides repository support for JPA and helps developers write cleaner, boilerplate-free code for data access layers.

Key Features of Spring Data JPA

1. **Simplified CRUD Operations**: Eliminates the need to write boilerplate code for CRUD operations.
2. **Derived Queries**: Enables query generation from method names.
3. **JPQL and Native Queries**: Allows customization of queries.
4. **Pagination and Sorting**: Provides built-in support for pagination and sorting.
5. **Repository Support**: Offers different types of repositories like `JpaRepository` and `CrudRepository`.

Advantages of Spring Data JPA

1. **Boilerplate-Free**: Reduces repetitive code for database operations.
2. **Integration with Spring Boot**: Seamlessly integrates with Spring Boot, enabling faster development.
3. **Scalability**: Offers support for advanced features like caching, transactions, and lazy loading.
4. **Vendor Independence**: Works with multiple database providers like MySQL, PostgreSQL, H2, and Oracle.

2. Setting Up the Environment

To begin using Spring Data JPA, you need to set up a development environment with the necessary tools and configurations.

Prerequisites

1. **Java Development Kit (JDK)**: Ensure JDK 11 or higher is installed.
2. **Spring Boot**: We'll use Spring Boot to simplify project setup.
3. **Database**: Choose a database like MySQL, PostgreSQL, or H2 (for development/testing).
4. **IDE**: Use an IDE like IntelliJ IDEA, Eclipse, or VS Code.
5. **Build Tool**: Maven or Gradle for dependency management.

Step 1: Create a Spring Boot Project

You can create a Spring Boot project using:

1. **Spring Initializr**: https://start.spring.io
2. **Your IDE**: Most modern IDEs have Spring Boot project generators.

Dependencies to Include:

- Spring Web
- Spring Data JPA
- Database driver (e.g., MySQL Driver)
- Spring Boot DevTools (optional, for live reloading)

Step 2: Add Dependencies in `pom.xml` (Maven)

```
xml
Copy code
<dependencies>
    <!-- Spring Boot Starter for Web -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
```

```xml
    <!-- Spring Boot Starter for Data JPA -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactI
d>
    </dependency>

    <!-- MySQL Driver -->
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <scope>runtime</scope>
    </dependency>

    <!-- Spring Boot DevTools (Optional) -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-devtools</artifactId>
        <scope>runtime</scope>
    </dependency>
</dependencies>
```

Step 3: Configure the Application

Set up the `application.properties` file for database configuration.

Example: MySQL Configuration

```
spring.datasource.url=jdbc:mysql://localhost:3306/ecommerce
spring.datasource.username=root
spring.datasource.password=your_password

spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.diale
```

```
ct.MySQLDialect
```

Key Configurations:

1. `spring.datasource.url`: URL for the database.
2. `spring.datasource.username` and `spring.datasource.password`: Database credentials.
3. `spring.jpa.hibernate.ddl-auto`: Specifies whether Hibernate should manage the schema:
    - `update`: Updates schema automatically (use cautiously in production).
    - `validate`: Validates schema without making changes.
    - `none`: No schema management.

3. Basics of JPA Entities

JPA Entities form the backbone of Spring Data JPA. They represent the tables in your database and are mapped to Java classes. In this section, we'll create and configure entities step by step.

What is a JPA Entity?

A JPA Entity is a lightweight, persistent domain object that:

1. Maps a Java class to a database table.
2. Maps fields in the class to columns in the table.

How to Define a JPA Entity

1. Annotate the class with `@Entity`.
2. Specify the table name using `@Table` (optional; defaults to the class name).
3. Mark the primary key field with `@Id` and specify its generation strategy using `@GeneratedValue`.

Key Annotations for JPA Entities

1. **@Entity**: Declares the class as an entity.

2. **@Table**: Specifies the table name and schema.
3. **@Id**: Marks the primary key field.
4. **@GeneratedValue**: Configures the primary key generation strategy.
5. **@Column**: Maps a field to a specific column in the table.
6. **@Transient**: Excludes a field from persistence.
7. **@Embedded** and **@Embeddable**: For composite key or complex value objects.

Code Example: Basic JPA Entity

Entity Class:

```
import jakarta.persistence.*;

@Entity
@Table(name = "products")
public class Product {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(nullable = false, unique = true)
    private String name;

    private double price;

    private int stock;

    // Getters and setters
    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public double getPrice() {
        return price;
    }

    public void setPrice(double price) {
        this.price = price;
    }

    public int getStock() {
        return stock;
    }

    public void setStock(int stock) {
        this.stock = stock;
    }
}
```

Key Points in the Code:

1. `@Entity`: Declares `Product` as a JPA entity.
2. `@Table`: Maps the entity to the `products` table.
3. `@Id` and `@GeneratedValue`: Auto-generate the primary key values.
4. `@Column`: Adds constraints like `nullable=false` and `unique=true`.

Common Primary Key Generation Strategies

- `GenerationType.IDENTITY`: Relies on the database to generate primary keys.

- `GenerationType.SEQUENCE`: Uses a database sequence to generate keys.
- `GenerationType.AUTO`: Chooses the strategy based on the database.

Real-World Example Scenario

E-commerce Application - Product Catalog

In an e-commerce app, each product is represented as an entity. For example:

- The `Product` entity corresponds to the `products` table.
- Attributes like `name`, `price`, and `stock` map to table columns.

Advanced Field Mappings

1. **Date and Time Fields:**

   ```
   @Temporal(TemporalType.DATE)
   private Date createdDate;
   ```

   - `TemporalType.DATE`: Stores only the date.
   - `TemporalType.TIME`: Stores only the time.
   - `TemporalType.TIMESTAMP`: Stores both date and time.

2. **Enum Fields:**

   ```
   @Enumerated(EnumType.STRING)
   private ProductCategory category;
   ```

   - `EnumType.ORDINAL`: Stores the ordinal (numeric) value of the enum.
   - `EnumType.STRING`: Stores the name of the enum.

3. **Large Text Fields:**

   ```
   @Lob
   ```

```
private String description;
```

4. **Transient Fields (Not Persisted):**

```
@Transient
private double discountedPrice;
```

Database Table Correspondence

For the `Product` entity, Spring Data JPA automatically creates a `products` table with the following schema:

```
CREATE TABLE products (
    id BIGINT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(255) NOT NULL UNIQUE,
    price DOUBLE,
    stock INT
);
```

Testing the Entity

Create a repository to test the `Product` entity:

```
import org.springframework.data.jpa.repository.JpaRepository;

public interface ProductRepository extends JpaRepository<Product, Long> {
}
```

Create a simple command-line runner to test CRUD operations:

```
import org.springframework.boot.CommandLineRunner;
import org.springframework.stereotype.Component;

@Component
public class ProductTestRunner implements CommandLineRunner
{

    private final ProductRepository productRepository;

    public ProductTestRunner(ProductRepository productRepos
itory) {
        this.productRepository = productRepository;
    }

    @Override
    public void run(String... args) throws Exception {
        // Save a product
        Product product = new Product();
        product.setName("Laptop");
        product.setPrice(1000.00);
        product.setStock(10);
        productRepository.save(product);

        // Retrieve all products
        System.out.println(productRepository.findAll());
    }
}
```

Output

When the application runs, the following output confirms the product was saved:

```
[Product{id=1, name='Laptop', price=1000.0, stock=10}]
```

```
```

Query Methods in Repositories

1. Derived Query Methods

Derived query methods in Spring Data JPA are methods in repository interfaces that are automatically translated into database queries based on the method name. These methods allow you to write simple and readable query methods without writing raw SQL or JPQL queries.

Here are a few examples:

- **findByName(String name)**: This method will generate a query like `SELECT e FROM Entity e WHERE e.name = :name`.
- **findAllByName(String name)**: This method will generate a query like `SELECT e FROM Entity e WHERE e.name = :name`.
- **existsByName(String name)**: This method will generate a query like `SELECT CASE WHEN COUNT(e) > 0 THEN true ELSE false END FROM Entity e WHERE e.name = :name`.
- **countByName(String name)**: This method will generate a query like `SELECT COUNT(e) FROM Entity e WHERE e.name = :name`.

These derived methods leverage method naming conventions. The following patterns are automatically recognized by Spring Data JPA:

- `findBy<Property>`: Creates a query to find an entity by a specific property.
- `findAllBy<Property>`: Creates a query to find all entities by a specific property.
- `existsBy<Property>`: Checks if any entity exists with a specific property.
- `countBy<Property>`: Counts the number of entities with a specific property.

Here's how it works:

```
public interface UserRepository extends JpaRepository<User,
Long> {
    List<User> findByName(String name); // Generates a quer
y: SELECT u FROM User u WHERE u.name = :name
}
```

2. Query Creation

Sometimes, derived query methods do not offer the flexibility needed for complex queries. In such cases, you can create your own custom queries using the **@Query** annotation.

Common Methods:

- **findBy**: Used for equality checks.
- **findAllBy**: Retrieves all entities with the specified property.
- **existsBy**: Checks if any entity exists with the specified property.
- **countBy**: Counts the number of entities with the specified property.

Example:

```
public interface UserRepository extends JpaRepository<User,
Long> {
    List<User> findByName(String name);
    List<User> findAllByCity(String city);
    boolean existsByEmail(String email);
    long countByAge(int age);
}
```

Handling Relationships

To handle relationships between entities in query methods, you can use the dot (.) notation. This allows you to filter based on relationships between entities.

Example:

```
public interface UserRepository extends JpaRepository<User,
Long> {
    List<User> findByAddress_City(String city);
}
```

In this example, `findByAddress_City` will generate a query like:

```
sql
Copy code
SELECT u FROM User u WHERE u.address.city = :city
```

Here, `Address` is an embedded object in the `User` entity.

3. Native Queries

Sometimes you may need to write a query in raw SQL to achieve specific functionality that isn't possible with JPQL or derived queries. In such cases, you can use the `@Query` annotation along with native SQL syntax.

```
public interface UserRepository extends JpaRepository<User,
Long> {
    @Query(value = "SELECT * FROM users WHERE email = ?1",
nativeQuery = true)
    User findByEmail(String email);
}
```

In this example:

- The `@Query` annotation is used to specify the native SQL query.
- The `nativeQuery = true` flag indicates that this is a native SQL query.

4. JPQL (Java Persistence Query Language)

JPQL is a powerful language used to write queries in a type-safe manner. It is similar to SQL but operates on entities and their fields, not database tables and columns.

Example of JPQL in Repository:

```
public interface UserRepository extends JpaRepository<User,
Long> {
    @Query("SELECT u FROM User u WHERE u.name = :name")
    List<User> findByNameUsingJPQL(@Param("name") String na
me);
}
```

Explanation:

- `@Query` annotation is used to define JPQL queries.
- `@Param("name")` annotates the method parameter to bind it to the query parameter.
- The query `"SELECT u FROM User u WHERE u.name = :name"` retrieves all users with the specified name.

5. Combining Queries and Custom Logic

You can combine Spring Data JPA queries with custom logic to provide more complex and specific retrieval operations.

```
public interface UserRepository extends JpaRepository<User,
Long> {
    @Query("SELECT u FROM User u WHERE u.name = :name AND
u.age > :age")
    List<User> findByNameAndAgeGreaterThan(@Param("name") S
tring name, @Param("age") int age);
}
```

Key Points to Remember

- **Method naming conventions** make derived query methods powerful and easy to use.
- **JPQL** and **native SQL** are useful for more complex queries or when specific database functions are required.

- **Dot notation** is used to access properties in relationships between entities.
- **@Query** annotation provides flexibility by allowing custom SQL or JPQL queries.

Pagination and Sorting in Spring Data

In Spring Data, pagination and sorting are simplified through the use of `Pageable` and `Sort` in conjunction with repository interfaces. This approach allows you to easily retrieve paginated and sorted data from your database without writing complex SQL queries.

1. Pagination using Pageable and Page objects

When using repositories in Spring Data, you can leverage the `Pageable` interface to retrieve paginated data. The `Pageable` object is used to encapsulate the information about page size and number, along with sorting criteria.

Steps to use pagination with repositories:

1. **Define your repository interface**: Extend the repository interface provided by Spring Data (`CrudRepository`, `JpaRepository`, etc.). In your interface, you can define methods to fetch paginated results.

Example Repository Interface:

```
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.data.jpa.repository.JpaRepositor
y;
import org.springframework.stereotype.Repository;

@Repository
public interface UserRepository extends JpaRepository<User,
Long> {
    Page<User> findAll(Pageable pageable);
}
```

- `UserRepository`: This interface extends `JpaRepository` and provides basic CRUD operations.
- `findAll(Pageable pageable)`: This method retrieves a paginated list of `User` objects. `Pageable pageable` allows you to specify the page number, page size, and sorting criteria.

1. **Controller layer**: In your controller, you can use the repository method to fetch the paginated data.

Example Controller:

```
import org.springframework.beans.factory.annotation.Autowir
ed;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.PageRequest;
import org.springframework.data.domain.Sort;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestPara
m;
import org.springframework.web.bind.annotation.RestControll
er;

@RestController
public class UserController {

    @Autowired
    private UserRepository userRepository;

    @GetMapping("/users")
    public Page<User> getUsers(
            @RequestParam(value = "page", defaultValue =
"0") int page,
            @RequestParam(value = "size", defaultValue = "1
0") int size) {

        Pageable pageable = PageRequest.of(page, size);
        return userRepository.findAll(pageable);
    }
}
```

2. Sorting using Sort class

For sorting, you can use the `Sort` class along with the `Pageable` object when calling repository methods.

Example with Sorting:

```
import org.springframework.beans.factory.annotation.Autowir
ed;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.PageRequest;
import org.springframework.data.domain.Sort;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestPara
m;
import org.springframework.web.bind.annotation.RestControll
er;

@RestController
public class UserController {

    @Autowired
    private UserRepository userRepository;

    @GetMapping("/users")
    public Page<User> getUsers(
            @RequestParam(value = "page", defaultValue =
"0") int page,
            @RequestParam(value = "size", defaultValue = "1
0") int size,
            @RequestParam(value = "sort", defaultValue = "i
d") String sort) {

        Pageable pageable = PageRequest.of(page, size, Sor
t.by(sort));
```

```
        return userRepository.findAll(pageable);
    }
}
```

- **Sort.by(sort)**: This method of `Sort` creates a `Sort` object based on the field you wish to sort by. The default sort order is ascending. If you want to sort in descending order, you can use `Sort.by(Sort.Direction.DESC, sort)`.

Using `Sort` with Multiple Fields

If you need to sort by multiple fields, you can chain sort criteria:

```
Sort sort = Sort.by("firstName").and(Sort.by("lastName"));
Pageable pageable = PageRequest.of(page, size, sort);
```

Example with multiple sorting fields:

```
@GetMapping("/users")
public Page<User> getUsers(
        @RequestParam(value = "page", defaultValue = "0") i
nt page,
        @RequestParam(value = "size", defaultValue = "10")
int size,
        @RequestParam(value = "sortField", defaultValue =
"id") String sortField,
        @RequestParam(value = "sortOrder", defaultValue =
"ASC") String sortOrder) {

    Sort sort = Sort.by(Sort.Direction.fromString(sortOrde
r), sortField);
    Pageable pageable = PageRequest.of(page, size, sort);
    return userRepository.findAll(pageable);
}
```

Key Points

- **Pageable** is used to control the pagination of data, specifying the page number and size.
- **Page** encapsulates the paginated result, providing access to content, total elements, and total pages.
- **Sort** is used to sort the data by one or more fields, allowing you to specify the sort order (ascending or descending).
- Spring Data makes it easy to implement pagination and sorting by abstracting the complex logic and providing simple method signatures for your repository interfaces.

Relationships and Mappings

JPA (Java Persistence API) provides various annotations to define and manage relationships between entities in a database. Let's go through the most common relationships and mappings in JPA along with their explanations, examples, and code snippets.

1. One-to-One: One-to-One Mapping

A `@OneToOne` relationship is used when two entities are related in a one-to-one manner. This means that each instance of the parent entity is related to exactly one instance of the child entity and vice versa.

Example:

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

    private String name;

    @OneToOne
    private Address address; // One-to-one relationship wit
h Address
}
```

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

    private String street;
    private String city;

    @OneToOne(mappedBy = "address")
    private Person person; // Back reference to Person enti
ty
}
```

Explanation:

- `@OneToOne`: This annotation is used on the `address` field in the `Person` class, indicating a one-to-one relationship.
- `mappedBy = "address"`: This specifies that this entity is the inverse side of the relationship, meaning it does not own the relationship. The owning side is defined by the field in `Person`.

2. One-to-Many and Many-to-One: One-to-Many and Many-to-One Relationships

- **One-to-Many**: A parent entity has multiple child entities, but each child is related to only one parent.
- **Many-to-One**: A child entity can be related to multiple parent entities, but a parent entity has only one child.

Example (One-to-Many):

```
@Entity
public class Course {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String title;

    @OneToMany(mappedBy = "course")
    private List<Review> reviews; // One-to-many relationsh
ip with Review
}
```

```
@Entity
public class Review {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String comment;

    @ManyToOne
    private Course course; // Many-to-one relationship with
Course
}
```

Explanation:

- `@OneToMany(mappedBy = "course")`: This annotation on the `reviews` field in the `Course` entity specifies that the `Review` entity's `course` field is the owning side of the relationship.
- `@ManyToOne`: This annotation in the `Review` entity represents a many-to-one relationship with `Course`.

3. Many-to-Many Relationships

A `@ManyToMany` relationship allows an entity to be associated with multiple other entities.

Example:

```
@Entity
public class Student {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;

    @ManyToMany
    @JoinTable(
      name = "student_course",
      joinColumns = @JoinColumn(name = "student_id"),
      inverseJoinColumns = @JoinColumn(name = "course_id"))
    private List<Course> courses; // Many-to-many relations
hip with Course
}
```

```
@Entity
public class Course {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String title;

    @ManyToMany(mappedBy = "courses")
    private List<Student> students; // Many-to-many relatio
nship with Student
}
```

Explanation:

- `@ManyToMany`: This annotation is used to define a many-to-many relationship between `Student` and `Course`.
- `@JoinTable`: This annotation is used to create a join table between `student` and `course` tables to represent the many-to-many relationship.
- `mappedBy = "courses"`: In the `Course` entity, this annotation indicates the inverse side of the relationship.

4. Fetch Types: EAGER vs LAZY Loading

- **EAGER**: By default, JPA fetches the related entity data when fetching the main entity from the database. This can lead to performance issues if the related data is not required immediately.
- **LAZY**: By using `LAZY`, JPA fetches related data only when it is accessed. This can reduce the number of database queries, improving performance.

Example:

```
@Entity
public class Employee {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;

    @OneToOne(fetch = FetchType.LAZY)
    private Department department; // Lazy loading
}
```

Explanation:

- `@OneToOne(fetch = FetchType.LAZY)`: By using `LAZY`, the `department` data will not be fetched until it is specifically accessed in the code.

5. Cascade Types: Persist, Merge, Remove, Detach, etc.

Cascade types define the actions that JPA should take on related entities when an operation is performed on a parent entity. Common cascade types include:

- **PERSIST**: When a parent entity is persisted, its associated child entities are also persisted.
- **MERGE**: When a parent entity is merged, its associated child entities are also merged.
- **REMOVE**: When a parent entity is removed, its associated child entities are also removed.
- **DETACH**: When a parent entity is detached, its associated child entities are also detached.

Example:

```
@Entity
public class Department {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;

    @OneToMany(mappedBy = "department", cascade = CascadeType.ALL)
    private List<Employee> employees; // Cascade all operations to employees
}
```

```
@Entity
public class Employee {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;

    @ManyToOne(cascade = CascadeType.PERSIST)
    private Department department; // Cascade PERSIST opera
tion
}
```

Explanation:

- `@OneToMany(mappedBy = "department", cascade = CascadeType.ALL)`: This annotation on the `employees` field in `Department` indicates that operations (e.g., persist, remove) will be cascaded to the related `Employee` entities.
- `@ManyToOne(cascade = CascadeType.PERSIST)`: In `Employee`, this cascade type allows you to persist an `Employee` and its related `Department` simultaneously.

6. MappedBy: Defining Owning/Non-Owning Sides of Relationships

- **Owning side**: The side of the relationship that holds the mapping (usually with `mappedBy`).
- **Non-owning side**: The side of the relationship that refers to the owning side.

Example:

```

@Entity
public class Teacher {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;
    @OneToMany(mappedBy = "teacher")
    private List<Student> students; // Non-owning side
}
```

```
@Entity
public class Student {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;

    @ManyToOne
    @JoinColumn(name = "teacher_id")
    private Teacher teacher; // Owning side
}
```

Explanation:

- `@OneToMany(mappedBy = "teacher")`: In `Teacher`, this specifies that the `Student` entity is the owning side of the relationship.
- `@ManyToOne`: In `Student`, this specifies that `Teacher` is the owning side of the relationship. The `teacher` field refers back to the `Teacher` entity.

Entity Lifecycle Callbacks

Entity lifecycle callbacks in JPA provide hooks to perform custom actions at different stages of the entity lifecycle, such as before and after persisting, updating, and removing an entity. These annotations allow you to execute logic before or after an entity is persisted, updated, or removed.

1. Lifecycle Callback Annotations

1. **@PrePersist**: This annotation is used to specify a method that will be invoked before an entity is saved to the database.

2. **@PostPersist**: This annotation is used to specify a method that will be invoked after an entity is saved to the database.
3. **@PreUpdate**: This annotation is used to specify a method that will be invoked before an entity is updated in the database.

2. Example Scenarios

Let's go through each annotation with practical examples to understand how they are used and how they can be beneficial.

2.1. @PrePersist

The `@PrePersist` annotation allows you to perform certain actions or validations just before an entity is saved to the database. This can be useful for setting default values or ensuring required fields are populated.

Example Scenario:

Suppose we have an `Order` entity where we want to set the creation timestamp before saving a new order.

```
@Entity
public class Order {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String customerName;

    private BigDecimal amount;

    @Temporal(TemporalType.TIMESTAMP)
    private Date createdDate;

    @PrePersist
    private void setCreationDate() {
        this.createdDate = new Date();
    }

    // Getters and Setters
}
```

Explanation:

- `@PrePersist`: This annotation is used on the `setCreationDate` method.
- `private void setCreationDate()`: This method will be called automatically before the entity is persisted to the database.
- In this method, we set the `createdDate` to the current date and time.

2.2. @PostPersist

The `@PostPersist` annotation allows you to perform actions after an entity has been successfully saved to the database. This is useful for logging purposes or sending notifications.

Example Scenario:

Suppose we have an `Order` entity and we want to send an email notification after an order is persisted.

```
@Entity
public class Order {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String customerName;

    private BigDecimal amount;

    @PostPersist
    private void sendOrderConfirmation() {
        // Logic to send email notification
        System.out.println("Sending order confirmation emai
l to " + this.customerName);
    }

    // Getters and Setters
}
```

Explanation:

- `@PostPersist`: This annotation is used on the `sendOrderConfirmation` method.
- `private void sendOrderConfirmation()`: This method will be automatically invoked after the entity has been saved to the database.
- In this method, you can add logic to send an email or perform other post-persistence actions.

2.3. @PreUpdate

The `@PreUpdate` annotation allows you to perform actions before an entity is updated in the database. This can be used for validation or logging purposes.

Example Scenario:

Suppose we have an `Order` entity and we want to validate that the `amount` does not decrease when an order is updated.

```
@Entity
public class Order {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String customerName;

    private BigDecimal amount;

    private BigDecimal previousAmount; // To store the prev
ious value of amount

    @PreUpdate
    private void validateAmountChange() {
        if (this.amount.compareTo(this.previousAmount) < 0)
{
            throw new IllegalArgumentException("Order amoun
t cannot be decreased");
        }
    }

    // Getters and Setters
}
```

Explanation:

- `@PreUpdate`: This annotation is used on the `validateAmountChange` method.
- `private void validateAmountChange()`: This method will be automatically invoked before an entity is updated in the database.
- In this method, we compare the current `amount` with `previousAmount`. If the current `amount` is less than the previous one, an exception is thrown, preventing a decrease./

JPA Mappings, Paging & Sorting, Auditing, and Soft Deletes

1. JPA Mappings

What is JPA Mapping?

JPA (Java Persistence API) mappings define how Java objects (entities) are mapped to relational database tables. These mappings allow the ORM (Object-Relational Mapping) framework to handle database operations seamlessly.

Key Annotations in JPA Mappings

1. **@Entity**
    - Marks a class as a JPA entity.
    - Example:

    ```
    @Entity
    public class Product {
        @Id
    ```

```
    @GeneratedValue(strategy = GenerationType.IDENTIT
Y)
    private Long id;

    private String name;
}
```

2. **@Table**
   - Specifies the table name for the entity.
   - Example:

   ```
   @Entity
   @Table(name = "products")
   public class Product {
       // Fields
   }
   ```

3. **@Id** and **@GeneratedValue**
   - Define the primary key and its generation strategy.
   - Strategies include AUTO, IDENTITY, SEQUENCE, and TABLE.

4. **@Column**
   - Maps a field to a specific column.
   - Example:

   ```
   @Column(name = "product_name", nullable = false)
   private String name;
   ```

5. **Relationships in JPA**
   - **@OneToOne**: Maps a one-to-one relationship.
   - **@OneToMany**: Maps a one-to-many relationship.
   - **@ManyToOne**: Maps a many-to-one relationship.
   - **@ManyToMany**: Maps a many-to-many relationship.
   - Example:

```
@Entity
public class Order {
    @OneToMany(mappedBy = "order")
    private List<OrderItem> items;
}
```

6. **@Embedded** and **@Embeddable**

   - Used for embedding value types in entities.
   - Example:

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

@Entity
public class Customer {
    @Embedded
    private Address address;
}
```

Best Practices

- Use meaningful names for entities and fields.
- Always define `@Id` and generation strategies.
- Avoid lazy-loading pitfalls by understanding fetch types (`LAZY` vs. `EAGER`).

---

2. JPA Paging and Sorting

What is Paging and Sorting?

Paging and Sorting in JPA allow retrieving large datasets efficiently by dividing them into smaller chunks and ordering the results.

Key Concepts

1. **Paging**: Dividing the data into pages with a specific number of records.
2. **Sorting**: Ordering the data based on one or more fields.

Using Spring Data JPA for Paging and Sorting

Repository Example

```
@Repository
public interface ProductRepository extends JpaRepository<Pr
oduct, Long> {
    Page<Product> findAll(Pageable pageable);
    Page<Product> findByNameContaining(String name, Pageabl
e pageable);
}
```

Service Example

```
@Service
public class ProductService {
    @Autowired
    private ProductRepository productRepository;

    public Page<Product> getPagedProducts(int page, int siz
e, String sortBy) {
        Pageable pageable = PageRequest.of(page, size, Sor
t.by(sortBy));
        return productRepository.findAll(pageable);
    }
}
```

Controller Example

```
@RestController
@RequestMapping("/api/products")
public class ProductController {

    @Autowired
    private ProductService productService;

    @GetMapping("/paged")
    public ResponseEntity<Page<Product>> getPagedProducts(
            @RequestParam(defaultValue = "0") int page,
            @RequestParam(defaultValue = "10") int size,
            @RequestParam(defaultValue = "id") String sortB
y) {
        return ResponseEntity.ok(productService.getPagedPro
ducts(page, size, sortBy));
    }
}
```

Best Practices

- Validate input parameters (`page` , `size` , `sortBy`).
- Use `Pageable` and `Sort` to handle complex pagination and sorting scenarios.
- Always paginate large datasets to avoid performance issues.

---

3. Auditing

What is Auditing?

Auditing tracks metadata such as **creation** and **modification timestamps** and the user who performed the changes.

Enabling Auditing in Spring Data JPA

Configuration

```
@Configuration
@EnableJpaAuditing
public class JpaConfig {
    // Enable JPA Auditing
}
```

Base Class

```
@MappedSuperclass
@EntityListeners(AuditingEntityListener.class)
public abstract class Auditable {
    @CreatedDate
    @Column(updatable = false)
    private LocalDateTime createdDate;

    @LastModifiedDate
    private LocalDateTime lastModifiedDate;

    // Getters and Setters
}
```

Entity Example

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

    private String name;
}
```

Best Practices

- Use `@MappedSuperclass` for reusability.
- Avoid manual timestamp handling in services.
- Track created/modified by user for better auditing.

---

4. Soft Deletes

What is Soft Delete?

Soft deletes mark a record as deleted without physically removing it from the database. This is typically done using a `deleted` flag.

Implementing Soft Deletes

Base Class for Soft Deletes

```
@MappedSuperclass
public abstract class SoftDeletable {
    @Column(name = "deleted", nullable = false)
    private boolean deleted = false;

    public boolean isDeleted() {
        return deleted;
    }

    public void setDeleted(boolean deleted) {
        this.deleted = deleted;
    }
}
```

Entity Example

```
@Entity
public class Product extends SoftDeletable {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;
}
```

Repository Example

```
@Repository
public interface ProductRepository extends JpaRepository<Pr
oduct, Long> {
    @Query("SELECT p FROM Product p WHERE p.deleted = fals
e")
```

```
    List<Product> findAllNotDeleted();
}
```

Service Example

```
@Service
public class ProductService {
    @Autowired
    private ProductRepository productRepository;

    public void deleteProduct(Long id) {
        Product product = productRepository.findById(id)
                .orElseThrow(() -> new ResourceNotFoundExce
ption("Product not found"));
        product.setDeleted(true);
        productRepository.save(product);
    }
}
```

Combining Soft Deletes with Auditing

Modified Base Class

```
@MappedSuperclass
@EntityListeners(AuditingEntityListener.class)
public abstract class AuditableSoftDeletable {
    @CreatedDate
    @Column(updatable = false)
    private LocalDateTime createdDate;

    @LastModifiedDate
    private LocalDateTime lastModifiedDate;

    @Column(nullable = false)
    private boolean deleted = false;

    private LocalDateTime deletedDate;
    public void setDeleted(boolean deleted) {
        this.deleted = deleted;
        this.deletedDate = deleted ? LocalDateTime.now() :
null;
    }
}
```

Entity Example

```
@Entity
public class Product extends AuditableSoftDeletable {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;
}
```

Best Practices

- Always query non-deleted records explicitly using a custom query or filter.
- Log delete operations for audit and debugging.
- Use `@MappedSuperclass` to combine auditing and soft delete logic.
