

Class-26: Paging, Sorting & Auditing



by Piali Kanti Samadder

Why use Pagination?

Without Pagination:

- Fetching all records can slow down the application
- High memory usage
- Slow UI response
- Not scalable for large data sets

Goal:

- Fetch data in small manageable chunks
- Improve performance and user experience

What is Pageable?

Pageable represents **pagination instructions**.

It contains:

- `pageNumber` → Which page to fetch (starts at **0**)
- `pageSize` → How many records per page
- `sort` → Sorting rules (optional)

Creating a Pageable Object

Basic Paging

```
Pageable pageable = PageRequest.of(0, 10);
```

- `0` → page number (starts from zero)
- `10` → number of records to return

Paging With Sorting

```
Pageable pageable = PageRequest.of(0, 10, Sort.by("name"));
```

Default sort is **ascending**.

How Spring Automatically Handles Pagination

When Spring sees a `Pageable` parameter in a controller or repository method:

- It **recognizes you want pagination**
- No extra configuration is required

```
@GetMapping("/items")
public Page<Item> getItems(Pageable pageable) {
    return itemRepository.findAll(pageable);
}
```

Request Example

```
GET /items?page=2&size=25&sort=name,asc
```

Converted Automatically to:

```
PageRequest.of(2, 25, Sort.by("name").ascending());
```

Who Creates the `Pageable` Object?

Spring uses `PageableHandlerMethodArgumentResolver` to:

1. **Read request parameters** (`page`, `size`, `sort`)
2. **Create a** `PageRequest` object internally
3. Pass it into your service → repository

How Repository Uses Pageable

Spring Data JPA translates it to SQL automatically:

```
itemRepository.findAll(pageable);
```

Becomes database query with:

$\text{OFFSET} = \text{pageNumber} \times \text{pageSize}$

$\text{LIMIT} = \text{pageSize}$

```
SELECT * FROM items LIMIT 25 OFFSET 50; // Sample SQL query
```

Example (PageRequest.of(2, 25)):

OFFSET 50

LIMIT 25

This retrieves **page 3** (zero-based).

Pageable Return Types

Spring Data JPA supports **three** return types when using `Pageable`:

Return Type	Contains Metadata?	What You Get	Best Use Case
Page<T>	Yes (total pages, total elements)	Full pagination info + content	When UI needs page numbers / total count
Slice<T>	Partial (hasNext only)	Content + info if next page exists	For large tables where counting rows is expensive
List<T>	No metadata	Only content	When you just need results (e.g., Top N items)

Examples

Method Signature	Behavior
<code>Page<Item> findAll(Pageable pageable)</code>	Calculates total rows → more expensive
<code>Slice<Item> findAll(Pageable pageable)</code>	Does not count total rows → faster
<code>List<Item> findAll(Pageable pageable)</code>	Still applies LIMIT/OFFSET , but no metadata returned

What is Page<T>?

Page<T> represents the **paginated query result**.

It contains:

- getContent() → The actual list of data
- getTotalElements() → Total records in DB
- getTotalPages() → Total number of pages
- hasNext() / hasPrevious() → Navigation helpers

It gives both data + metadata.

Sorting in Spring Data JPA

Sorting determines the **order of query results** returned from the database.

Why Sorting Matters:

- Makes data easier to read in UI tables.
- Lets users see newest/oldest, highest/lowest, or custom order.
- Works seamlessly with pagination.

Sort Class

Spring Data JPA provides `Sort` **class** (`org.springframework.data.domain.Sort`) to define ordering.

Key features:

- Ascending or Descending order
- Multiple fields
- Can be used standalone or with `Pageable`

Example — Single Field

```
Sort sort = Sort.by("name").ascending();
```

Example — Descending

```
Sort sort = Sort.by("createdAt").descending();
```

Example — Multiple Fields

```
Sort sort = Sort.by("status").descending()  
                .and(Sort.by("createdAt").ascending());
```

How to apply sorting in Spring Data JPA

Using Sort Directly in Repository

You can simply **pass an instance of** `Sort` to the method:

```
List<User> users = userRepository.findAll(Sort.by("createdAt").descending());
```

However, what if we want to **both sort and page our data**?

We can do that by passing the sorting details into our *PageRequest* object itself:

```
Pageable pageable = PageRequest.of(0, 10, Sort.by("name").ascending());  
Page<User> users = userRepository.findAll(pageable);
```

Based on our sorting requirements, **we can specify the sort fields and the sort direction** while creating our *PageRequest* instance.

Projections in Spring Data JPA

- Projection allows fetching **only selected fields** from the database instead of the whole entity.
- Helps in:
 - Reducing memory usage
 - Improving query performance
 - Returning **custom DTOs** to APIs

Types of Projections (By Implementation Type)

Type	Description
Interface-based	Define a Java interface with getter methods for required fields. Spring auto-implements it.
Class-based (DTO)	Define a DTO class with constructor matching selected fields.
Dynamic projections	Return different projection types dynamically at runtime.

Interface-based Projection Example

- Fetch **only selected fields** from an entity.
- Define a Java **interface with getters** for required fields.
- Spring **automatically generates the implementation**.

```
public interface UserNameOnly {  
    String getFirstName();  
    String getLastName();  
}  
  
// Repository method  
List<UserNameOnly> findByActiveTrue();  
  
// JPQL  
@Query("SELECT u.firstName AS firstName, u.lastName AS lastName FROM User u WHERE u.active = true")  
List<UserNameOnly> findActiveUsersJPQL();  
  
// Native Query  
@Query(  
    value = "SELECT first_name AS firstName, last_name AS lastName FROM user WHERE active = true",  
    nativeQuery = true  
)  
List<UserNameOnly> findActiveUsersNative();
```

Class-based (DTO) Projection Example

```
public class UserDTO {  
    private String firstName;  
    private String lastName;  
  
    public UserDTO(String firstName, String lastName) {  
        this.firstName = firstName;  
        this.lastName = lastName;  
    }  
}  
  
// Repository  
@Query("SELECT new com.example.UserDTO(u.firstName, u.lastName) FROM User u")  
List<UserDTO> findActiveUsers();
```

Useful when you want **immutable DTOs** or **custom constructor logic**.

Dynamic Projections

- Same repository method, different projection return types.

```
<T> List<T> findByActiveTrue(Class<T> type);
```

```
// Usage
```

```
List<UserNameOnly> users = userRepository.findByActiveTrue(UserNameOnly.class);
```

```
List<UserDTO> usersDTO = userRepository.findByActiveTrue(UserDTO.class);
```

Closed vs Open Projections

Closed Projection

- Only exposes **fields defined in the projection interface or DTO**.
- No computed or derived fields allowed.
- Fetches only **exact database columns** mapped to getters.

Example — Closed Projection

```
// Repository
List<UserNameOnly> findByActiveTrue();
```

- Only `firstName` and `lastName` are available.
- Accessing any other field will fail.

Open Projection

- Allows **computed fields** or **SpEL expressions**.
- Can include derived properties that are **not directly stored in the database**.

```
public interface UserNameFull {
    String getFirstName();
    String getLastName();

    // Computed property
    @Value("#{target.firstName + ' ' + target.lastName}")
    String getFullName();
}

// Repository
List<UserNameFull> findByActiveTrue();
```

- `getFullName()` is **computed at runtime**.
- Still fetches only `firstName` and `lastName` from DB.

Spring Data JPA Specifications

What Are Specifications?

- A **type-safe, dynamic query building** mechanism in Spring Data JPA
- Based on the **JPA Criteria API**
- Allows combining conditions at runtime

Why Do We Need Specifications?

Sometimes queries depend on:

- Multiple optional filters
- User-input criteria
- Runtime conditions

Static repository methods like:

```
findByNameAndCategoryAndPriceLessThan(...)
```

→ **Become unmanageable** as filters grow.

Specifications solve this by enabling **dynamic + composable queries**.

Specification Interface

```
public interface Specification<T> {  
    Predicate toPredicate(  
        Root<T> root,  
        CriteriaQuery<?> query,  
        CriteriaBuilder cb  
    );  
}
```

Key Concepts

Term	Purpose
Root	Represents the entity/table
CriteriaQuery	Represents the query structure
CriteriaBuilder	Used to build conditions (=Predicate)

How to create JPA Specifications

Example Entity

```
@Entity
public class Product {
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private String category;
    private double price;
}
```

Creating Specification Filters

```
public class ProductSpecification {

    public static Specification<Product> hasNameDisntinct(String name) {
        return (root, query, cb) -> {
            query.distinct(true);
            return cb.equal(root.get("name"), name);
        };
    }

    public static Specification<Product> hasCategory(String category) {
        return (root, query, cb) -> cb.equal(root.get("category"), category);
    }

    public static Specification<Product> priceGreaterThan(double price) {
        return (root, query, cb) -> cb.greaterThan(root.get("price"), price);
    }

    public static Specification<Product> priceLessThan(double price) {
        return (root, query, cb) -> cb.lessThan(root.get("price"), price);
    }
}
```

How to use JPA Specifications

Repository Setup

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

You **must** extend `JpaSpecificationExecutor` to run Specifications.

Example Usage

```
Specification<Product> spec = Specification
    .where(ProductSpecification.hasName("Samsung"))
    .and(ProductSpecification.priceGreaterThan(500));

List<Product> results = productRepository.findAll(spec);
```

JPA Auditing

Purpose:

Automatically track **creation** and **modification** metadata of entities without manual intervention.

How to Enable Auditing:

- **Enable JPA Auditing in Spring Boot:**

```
@SpringBootApplication
@EnableJpaAuditing
public class MyApplication { }
```

- **Add Auditing Listener to Entity:**

```
@Entity
@EntityListeners(AuditingEntityListener.class)
public class User {

    @CreatedDate
    private LocalDateTime createdAt;

    @LastModifiedDate
    private LocalDateTime updatedAt;
}
```

What Happens Behind the Scenes:

- AuditingEntityListener listens to **entity lifecycle events**:
 - @PrePersist → sets @CreatedDate
 - @PreUpdate → sets @LastModifiedDate

Database Table Example:

Column Name	Type	Description
created_at	TIMESTAMP	Auto-set on entity creation
updated_at	TIMESTAMP	Auto-set on entity update

Running Sample Project

- **Clone the Repository**

If you haven't cloned the repository yet, run the following command (ensure `git` is installed):

```
git clone https://github.com/PialKanti/Ostad-SpringBoot-Course.git
```

Then switch to the correct branch for today's class (replace with the actual branch name, e.g., `class-24-jpa-relationship`):

```
git fetch
git switch class-26-pagination-sorting
```

Or,

If You Already Have the Repository Cloned, simply open your existing project folder and switch (or checkout) to the appropriate branch:

```
git fetch
git switch class-26-pagination-sorting
```

- **Set Up and Run PostgreSQL Database**

You can run PostgreSQL **either via Docker** or a **desktop installation**.

Option 1: Run via Docker

A `compose.yml` file is available in the root of the repository.

Run the following command from the project root:

```
docker compose up -d
```

This will start a PostgreSQL container automatically.

Or,

Option 2: Run via PostgreSQL Desktop (Manual Setup)

If you already have PostgreSQL installed locally:

1. Start your PostgreSQL server.
2. Create a new database named `crud_db` if not exists.

- **Open the Project in IntelliJ IDEA**

1. Open IntelliJ IDEA.
2. Click **File** → **Open** and select the `crud-sample` folder inside the repository.
3. Let IntelliJ import Maven/Gradle dependencies automatically.