irian
Software Engineering Par Excellence

Part III

JPA Collections

JPQL

Projections

Pagind and Sorting

by Georgia Leustean

JPA Associations & Collections

Library Domain Examples & Join Strategies Explained

Overview

- Associations: OneToOne, ManyToOne, OneToMany, ManyToMany
- Collection of basic types: @ElementCollection
- Join strategies: @JoinColumn vs @JoinTable
- Cascade operations & orphanRemoval

@JoinColumn vs @JoinTable

- @JoinColumn: uses a foreign key column in the entity table
- @JoinTable: uses an intermediate link table to map associations
- Use JoinColumn for simple FK in child table
- Use JoinTable for many-to-many or unidirectional one-to-many

@ManyToOne (Book → Publisher)

```
@Entity
class Book {
  @ManyToOne(fetch = FetchType.LAZY)
  @JoinColumn(name = "publisher_id")
  private Publisher publisher;
}
```

@OneToOne (Book ↔ BookDetail)

```
@Entity
class Book {
  @OneToOne(cascade = CascadeType.ALL)
  @JoinColumn(name = "detail_id")
  private BookDetail detail;
}

@Entity
class BookDetail {
  @OneToOne(mappedBy = "detail")
  private Book book;
}
```

@OneToMany (Publisher → Books)

```
@Entity
class Publisher {
  @OneToMany(mappedBy = "publisher", cascade = CascadeType.PERSIST)
  private List<Book> books = new ArrayList<>();
}
```

@ManyToMany (Book ↔ Genre)

```
@Entity
class Book {
  @ManyToMany
  @JoinTable(
    name = "book_genre",
    joinColumns = @JoinColumn(name = "book_id"),
    inverseJoinColumns = @JoinColumn(name = "genre_id")
  )
  private Set<Genre> genres = new HashSet<>();
}
```

@ElementCollection (Member → Phone Numbers)

```
@Entity
class Member {
  @ElementCollection
  @CollectionTable(name = "member_phones",
                   joinColumns = @JoinColumn(name = "member_id"))
  @Column(name = "phone_number")
  private Set<String> phoneNumbers = new HashSet<>();
}
```

Cascade & Orphan Removal

- CascadeType: ALL, PERSIST, MERGE, REMOVE, REFRESH, DETACH
- orphanRemoval = true: deletes child when removed from collection
- Useful when parent fully controls child lifecycle

JPA @Embeddable & @Embedded

Defining reusable value types

Overview

- @Embeddable marks a class as a value type
- @Embedded includes an embeddable instance in an entity
- Use @AttributeOverride(s) to customize column mappings
- Support for nested and collection embeddables

Annotations

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

@Entity
public class Employee {
  @Id Long id;
  private String name;
  @Embedded
  private Address address;
}
```

@AttributeOverride Example

```
@Entity
class Employee {
  @Embedded
  @AttributeOverrides({
    @AttributeOverride(name="street", column=@Column(name="emp_street")),
    @AttributeOverride(name="zipCode", column=@Column(name="emp_zip"))
  })
  private Address address;
}
```

Nested Embeddables

```
@Embeddable
class Location {
  private double latitude;
  private double longitude;
}

@Embeddable
class Address {
  private String street;
  @Embedded
  private Location coords;
}
```

@ElementCollection of Embeddables

```
@Entity
class User {
  @ElementCollection
  @CollectionTable(name="user_addresses",
    joinColumns=@JoinColumn(name="user_id"))
  private List<Address> addresses = new ArrayList<>();
}
```

When to Use Embeddables

- Model value objects (e.g., money, period, address)
- Encapsulate reusable column groups
- Avoid separate entity overhead for simple types
- Combine with AttributeOverrides for flexibility

Ex 01

- Create entities for Order and Customer:

Customer:

- Name (required, max length 70)
- can have many Orders

Order

- Date
- Price – contains value and currency (RON, EUR etc)
- Can contain many Pizzas
- Status – enum (Created, In Progress, In delivery, Completed)

Create an endpoint to get all Orders

- Insert some Customers and Orders via script in data.sql
- Refactor Pizza entity to use a Embeddable for price

The @Transient Annotation

- Definition: Marks a field to be ignored by JPA persistence mapping
- Purpose: Exclude non-persistent fields
- Package: javax.persistence.Transient
- Use Cases: Computed or temporary data
  - Usage Example:
  - import javax.persistence.Transient;
  - @Transient private String tempLabel;

JPA EntityManager

Key Methods & Classical Repository Usage

What is EntityManager?

- Core interface to manage persistence context
- Provided by EntityManagerFactory
- Controls entity lifecycle operations and queries

Important Methods

- persist(entity) — Make a transient entity persistent
- find(Class<T>, id) — Retrieve entity by primary key
- merge(entity) — Merge state of detached into persistence context
- remove(entity) — Mark entity for removal
- createQuery(jpql) — Create dynamic JPQL query
- createNamedQuery(name) — Execute predefined JPQL query
- flush() — Synchronize persistence context to DB
- clear() — Detach all entities from context

Classical Repository Example

```
@Repository
public class CustomerRepository {
   @PersistenceContext
   private EntityManager em;

   public Customer findById(Long id) {
      return em.find(Customer.class, id);
   }

   public Customer save(Customer c) {
      if (c.getId() == null) em.persist(c);
      else c = em.merge(c);
      return c;
   }

   public void delete(Customer c) {
      Customer managed = em.contains(c) ? c : em.merge(c);
      em.remove(managed);
   }

   public List<Customer> findAll() {
      return em.createQuery(
         "SELECT c FROM Customer c", Customer.class)
         .getResultList();
   }
}
```

Entity Lifecycle Callbacks

- @PrePersist, @PostPersist
- @PreUpdate, @PostUpdate
- @PreRemove, @PostRemove
- @PostLoad

JPQL & Spring Data @Query

Writing Custom Queries in Repositories

What is JPQL?

- Java Persistence Query Language – object-oriented SQL
- Operates on entity objects and their properties
- Portable across JPA providers (e.g. Hibernate, EclipseLink)
- Supports SELECT, UPDATE, DELETE, JOIN, aggregation

JPQL Syntax Basics

- SELECT e FROM Entity e WHERE e.property = :value
- JOIN FETCH associations: SELECT o FROM Order o JOIN FETCH o.items
- Aggregations: SELECT COUNT(c) FROM Customer c
- ORDER BY: SELECT p FROM Product p ORDER BY p.price DESC

Basic JPQL Example

```
// Using EntityManager:
String jpql = "SELECT c FROM Customer c WHERE c.status = 'ACTIVE'";
List<Customer> active = em.createQuery(jpql, Customer.class)
                          .getResultList();
```

@Query Annotation

```
public interface CustomerRepo extends JpaRepository<Customer, Long> {
    @Query("SELECT c FROM Customer c WHERE c.status = 'ACTIVE'")
    List<Customer> findActive();
}
```

@Query with Named Parameters

```
@Query("SELECT o FROM Order o WHERE o.total >= :minTotal AND o.date <= :endDate")
List<Order> findByTotalAndDate(@Param("minTotal") BigDecimal minTotal,
                               @Param("endDate") LocalDate endDate);
```

@Query with Positional Parameters

```
@Query("SELECT p FROM Product p WHERE p.category = ?1 AND p.available = ?2")
List<Product> findByCategoryAndAvailability(String category, boolean available);
```

@Modifying @Query

```
@Modifying
@Query("UPDATE Account a SET a.status = 'SUSPENDED' WHERE a.lastLogin < :cutoff")
int suspendInactive(@Param("cutoff") LocalDate cutoff);

@Modifying
@Query("DELETE FROM Session s WHERE s.expired = true")
int deleteExpiredSessions();
```

Native Queries

- `@Query(value = "SELECT * FROM users WHERE role = :role", nativeQuery = true)`
- `List<User> findByRoleNative(@Param("role") String role);`

Ex 02

- Write a JPQL query to retrieve all Orders which contain a certain Pizza (filter by Pizza name parameter)
- Write the corresponding endpoint and service which receives a Pizza name and returns all Orders containing that Pizza (if no pizza name is received then the endpoint should return all Orders)

Spring Data Projections & Value Objects

Efficient Data Retrieval & Immutable Models

Interface-based Projections

```
// Define projection interface
public interface UserNameOnly {
   String getFirstName();
   String getLastName();
}

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

Class-based (DTO) Projections

```
// DTO/VO class
public class UserInfo {
   private final String email;
   private final int age;
   public UserInfo(String email, int age) {
      this.email = email; this.age = age; }
   // getters...
}

// Repository method with constructor expression
@Query("SELECT new com.example.dto.UserInfo(u.email, u.age)
      FROM User u WHERE u.active = true")
List<UserInfo> findActiveUserInfo();
```

Dynamic Projections

```
// Generic method signature
<T> List<T> findByLastName(String name, Class<T> type);

// Usage examples:
repo.findByLastName("Smith", UserNameOnly.class);
repo.findByLastName("Smith", UserInfo.class);
```

Standard Repository (No Projection)

```
// Repository
List<User> findAll();

// JPQL: SELECT u FROM User u
// SQL: SELECT * FROM users
```

Interface-based Projection

```
// Projection Interface
public interface UserNameOnly {
  String getFirstName();
  String getLastName();
}

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

// JPQL: SELECT u.firstName, u.lastName FROM User u WHERE u.active = true
// SQL: SELECT first_name, last_name FROM users WHERE active = true
```

Class-based (DTO) Projection

```
// DTO Class
public class UserInfo {
  private final String email;
  private final int age;
  public UserInfo(String email, int age) { this.email = email; this.age = age; }
}

// Repository Method
@Query("SELECT new com.example.dto.UserInfo(u.email, u.age) FROM User u WHERE u.active = true")
List<UserInfo> findActiveUserInfo();

// JPQL: SELECT new com.example.dto.UserInfo(u.email, u.age) ...
// SQL: SELECT email, age FROM users WHERE active = true
```

Query Comparison

- **No Projection**: SELECT * FROM users
- **Interface Projection**: SELECT first_name, last_name FROM users WHERE active = true
- **DTO Projection**: SELECT email, age FROM users WHERE active = true

Performance Implications

- Projections reduce data transfer and memory usage
- Interface projections are simpler and require no DTO classes
- DTO projections allow complex mappings and computed fields
- Use projections when only a subset of data is needed

Spring Data Projections: Nested Interfaces & SpEL

Employee-Department Example

Overview

- Use interface-based projections to fetch only needed data
- Nested interfaces for type-safe access to related entities
- SpEL (@Value) for flat projections without nested types
- Improved performance & cleaner API

Nested Interface Projection

```
// Projection interface
public interface EmployeeSummary {
    Long getId();
    String getName();

    DepartmentInfo getDepartment();

    interface DepartmentInfo {
        String getName();
        String getLocation();
    }
}

// Repository method
List<EmployeeSummary> findByActiveTrue();
```

SpEL-based Flat Projection

```
// Flat projection with SpEL
public interface EmployeeFlat {
    Long getId();
    String getName();

    @Value("#{target.department.name}")
    String getDepartmentName();

    @Value("#{target.department.location}")
    String getDepartmentLocation();
}

// Repository method
List<EmployeeFlat> findByDepartmentLocation(String location);
```

Generated SQL Comparison

Nested Interface:

SELECT e.id, e.name, d.name, d.location

FROM employee e

LEFT JOIN department d ON e.department_id = d.id

SpEL Flat:

SELECT e.id, e.name, d.name AS departmentName, d.location AS departmentLocation

FROM employee e

LEFT JOIN department d ON e.department_id = d.id

Best Practices

Use nested interfaces for clear type-safe nested access

Prefer SpEL flat projections for simple, flattened views

Mind performance: both generate tailored SELECTs

Avoid SpEL for complex logic—keep projections simple

Leverage repository method naming to derive queries

Ex 03

1. Write a Projection method to retrieve a list of OrderVO objects, where each object has:

- Order Number
- Order Date
- Customer Name

```
List<OrderVO> findAllProjectedBy();
```

2. Write a Projection Query to retrieva a list of OrderCountVO objects where each object has:

- Order Number
- Number of Pizzas

Paging & Sorting in Spring Data Repositories

Why Paging & Sorting?

- Avoid loading huge result sets into memory
- Improve REST API performance (limit payload)
- Let clients request only what they need
- Sort by one or more fields

Enabling Paging & Sorting in Repos

```
public interface PizzaRepository extends JpaRepository<Pizza, Long> {
    // findAll(Pageable pageable): Page<Pizza>
    // findAll(Sort sort): List<Pizza>
}
```

Constructing Requests

- GET /api/pizzas?sort=price,desc
- GET /api/pizzas?page=2&size=5&sort=name,asc&sort=price,desc

Raw Response (no pagination)

```
[
  { "id": 1, "name": "Margherita", "price": 7.5 },
  { "id": 2, "name": "Funghi",    "price": 8.0 },
  { "id": 3, "name": "Diavola",   "price": 9.0 },
  …
]
```

Paginated Response

```
{
  "content": [
    { "id": 11, "name": "Quattro Stagioni", "price": 10.5 },
    { "id": 12, "name": "Capricciosa",      "price": 11.0 }
    …
  ],
  "totalPages": 4,
  "totalElements": 20,
  "number": 2
}
```

Controller Snippet

```
@RestController
@RequestMapping("/api/pizzas")
public class PizzaController {
    @Autowired private PizzaRepository repo;
    @GetMapping
    public Page<Pizza> list(Pageable pageable) {
        return repo.findAll(pageable);
    }
}
```

Summary & Best Practices

- Support sensible defaults (e.g. page=0, size=20)
- Validate/max-limit size to avoid abuse
- Expose only needed metadata
- Use DTOs for custom field names
- Consider HATEOAS links for navigation

Ex 04

- Implement Paging and Sorting in the Pizza Repository, Service and Controller
- Test retrieving pages 1 and 3 with page sizes of 7 and 5 (insert more pizzas in data.sql if needed)
- Test sorting by name ascending and by price descending