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

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

VOs vs DTOs & CQRS

VOs as Read Models

- Represent data retrieved from queries
- Populate directly from database results
- Immutable, tailored to read-only use
- Shape fits query requirements (projections)

DTOs as Write Models

- Receive data from clients for persistence
- Map to domain entities to save changes
- Often mutable to support binding
- Include validation and transformation logic

VO vs DTO: Key Differences

- Role – VO: Read-only representation vs DTO: Write/input model
- Source – Populated from queries/projections vs Received from client requests
- Mutability – Immutable vs Mutable
- Use case – Query side vs Command side
- Validation – Assumed correct vs Explicit validation

Aligning with CQRS

- Query Model: Use VOs for read operations
- Command Model: Use DTOs for write operations
- Separate handlers/services for reads and writes
- Optimize models independently

Best Practices

- Keep VOs immutable and focused on read needs
- Design DTOs with clear validation rules
- Maintain separate query/write pipelines
- Document data contracts for each role

Writing POST & PUT Methods in Spring

HTTP POST vs PUT

- **POST**: Create new resource, not idempotent
- **PUT**: Update/replace resource, idempotent

POST Example

~~~
```java
@PostMapping
public ResponseEntity<ItemDto> create(@Valid @RequestBody ItemDto dto) {
    ItemDto created = service.create(dto);
    URI location = URI.create("/api/items/" + created.getId());
    return ResponseEntity.created(location).body(created);
}```
~~~

PUT Example

```java
@PutMapping("/{id}")
public ResponseEntity<ItemDto> update(@PathVariable Long id,
    @Valid @RequestBody ItemDto dto) {
    ItemDto updated = service.update(id, dto);
    return ResponseEntity.ok(updated);
}
```

What is @RequestBody?

- Annotation to bind HTTP request body to a Java object
- Part of Spring MVC @Controller and @RestController
- Uses HttpMessageConverters (e.g., Jackson)
- Supports JSON, XML, and other formats

Basic Usage

~~~
```java
@PostMapping("/users")
public ResponseEntity<UserDto> createUser(
    @RequestBody UserDto userDto) {
    UserDto created = userService.create(userDto);
    return ResponseEntity.status(HttpStatus.CREATED).body(created);
}```
~~~

JSON Mapping

- Ensure `Content-Type: application/json` header
- Spring uses Jackson by default to deserialize JSON
- Unknown properties: configure FAIL_ON_UNKNOWN_PROPERTIES
- Customize with @JsonProperty, @JsonIgnore, etc.

Advanced Usage

- Use `@RequestBody(required = false)` for optional bodies
- Consume different media types with `consumes` attribute:
- `@PostMapping(consumes = MediaType.APPLICATION_XML_VALUE)`
- Implement custom HttpMessageConverter for new formats

Best Practices

- Use DTOs to decouple API from domain models
- Validate input early and clearly
- Handle missing or malformed bodies gracefully
- Document API with OpenAPI/Swagger annotations
- Limit body size to prevent abuse

Transactions & Flushing

- EntityManager requires active transaction for write operations
- em.flush(): pushes changes to the database without commit
- Commit automatically triggers flush
- em.clear(): detaches all entities

Transactions & @Transactional in Spring

What is a Transaction?

- A unit of work that is atomic, consistent, isolated, durable (ACID).
- Ensures all operations succeed or none take effect.
- Critical for data integrity in databases.

ACID Properties

- Atomicity – All-or-nothing execution.
- Consistency – Transition from one valid state to another.
- Isolation – Concurrent transactions do not interfere.
- Durability – Once committed, results are permanent.

Declarative vs Programmatic

- Declarative: Use @Transactional annotation at class/method level.
- Programmatic: Use TransactionTemplate or PlatformTransactionManager.
- Declarative preferred for simplicity and readability.

@Transactional Annotation

```java
@Service
public class OrderService {
    @Transactional
    public void placeOrder(OrderDTO orderDto) {
        // business logic
    }
}
```

Propagation Behaviors

- REQUIRED: join existing or create new.
- REQUIRES_NEW: suspend current, start new.
- MANDATORY: must run within existing.
- SUPPORTS: join if exists, else run non-transactional.

Isolation Levels

- READ_UNCOMMITTED: dirty reads allowed.
- READ_COMMITTED: prevents dirty reads.
- REPEATABLE_READ: prevents non-repeatable reads.
- SERIALIZABLE: full isolation, lowest throughput.

Rollback Rules

- Default: rollback on unchecked exceptions (RuntimeException).
- use rollbackFor / noRollbackFor to customize.
- `@Transactional(rollbackFor = Exception.class)`

Best Practices

- Keep transactions short to avoid locks and contention.
- Avoid database calls in loops within transactions.
- Use readOnly=true for read-only operations.
- Document transaction boundaries and behaviors.

Pitfalls of Missing @Transactional at Service Layer

Over-reliance on Spring Data JPA repository transactions

LazyInitializationException

- Occurs when accessing lazy-loaded associations outside a transaction
  - Example: Fetch entity, close repository scope, then access a collection
  - Results in org.hibernate.LazyInitializationException

Partial Persistence / Inconsistent State

- Multiple repository calls are not atomic
  - Service method calls save() on RepositoryA then RepositoryB
  - If second call fails, first change remains persisted
  - Leads to data inconsistency

Lack of Proper Rollback

- Without @Transactional, exceptions won't rollback multiple operations
  - No global rollback is applied
  - Requires manual compensation logic

Propagation & Isolation Issues

- Nested repository calls lack clear propagation
  - @Transactional allows setting propagation and isolation
  - Repository-level defaults may not suit complex flows

Code Example: Without @Transactional

```
public class PizzaService {
    private final PizzaRepository pizzaRepo;
    private final OrderRepository orderRepo;

    public PizzaService(PizzaRepository pizzaRepo, OrderRepository orderRepo) {
        this.pizzaRepo = pizzaRepo;
        this.orderRepo = orderRepo;
    }

    public void createOrder(OrderDto dto) {
        // Save pizza
        Pizza pizza = new Pizza(dto.getName(), dto.getSize());
        pizzaRepo.save(pizza);

        // If an exception occurs here, pizza remains persisted, order not

        // Save order
        Order order = new Order(dto.getCustomer(), pizza);
        orderRepo.save(order);

}
```

Code Example: With @Transactional

```
@Service
public class PizzaService {
    private final PizzaRepository pizzaRepo;
    private final OrderRepository orderRepo;

    public PizzaService(PizzaRepository pizzaRepo, OrderRepository orderRepo) {
        this.pizzaRepo = pizzaRepo;
        this.orderRepo = orderRepo;
    }

@Transactional
public void createOrder(OrderDto dto) {
        // Save pizza and order within one transaction
        Pizza pizza = new Pizza(dto.getName(), dto.getSize());
        pizzaRepo.save(pizza);
        Order order = new Order(dto.getCustomer(), pizza);
        orderRepo.save(order);

        // Exception here triggers full rollback of both operations
    }
```

Ex 05

- Create PizzaDTO, create save methods in Controller + Service and save new Pizzas to the database (using Postman)

OBS: If using data inserted via data.sql you will need to specify the next id to be generated like this:

ALTER TABLE pizza ALTER COLUMN id RESTART WITH 10;

Use the next value after the ones you inserted

Using ResponseEntity in Spring

What is ResponseEntity?

- Wrapper around HTTP response, including status, headers, and body
- Part of Spring MVC – org.springframework.http.ResponseEntity<T>
- Used in @RestController methods to craft full responses
- Provides fluent builders for flexibility

Creating ResponseEntity

```
new ResponseEntity<>(body, status)
```

Or use static builders:

```
ResponseEntity.ok(body)
ResponseEntity.status(HttpStatus.CREATED).body(body)
```

Common HTTP Status Codes

- 200 OK – Successful GET/PUT requests
- 201 Created – Successful POST with resource creation
- 204 No Content – Successful DELETE or no body
- 400 Bad Request – Validation or client errors
- 404 Not Found – Resource missing
- 500 Internal Server Error – Unhandled exceptions

Setting Headers

```
ResponseEntity.ok().header("X-Custom-Header", "value").body(body)
ResponseEntity.created(uri).headers(headers).body(body)
```

Use HttpHeaders for multiple headers

Example Code

~~~
```java
@PostMapping("/items")
public ResponseEntity<ItemDto> createItem(@RequestBody ItemDto dto) {
    ItemDto created = service.create(dto);
    URI location = URI.create("/api/items/" + created.getId());
    return ResponseEntity.created(location)
                         .header("X-Trace-Id", "12345")
                         .body(created);
}```
~~~

Best Practices

- Always return appropriate status codes
- Include Location header for newly created resources
- Handle errors and return meaningful messages
- Avoid exposing internal details in responses
- Use generics for type safety

What is @Valid and Bean Validation

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

- @Valid triggers JSR-303/JSR-380 validation on method arguments
- Supported via Hibernate Validator by default in Spring Boot
- Integrates with HttpMessageConverters for request bodies
- Ensures data integrity and reduces manual checks

Predefined Constraint Annotations

- @NotNull – Field must not be null
- @NotEmpty – String/List must not be empty
- @NotBlank – String must contain non-whitespace
- @Size(min, max) – Size constraints for String, Collection
- @Email – Valid email format
- @Pattern(regexp) – Matches regex pattern
- @Min/@Max – Numeric range constraints
- @Positive/@Negative – Numeric sign constraints

Annotating DTO Fields

```java
public class UserDto {
    @NotBlank
    private String username;

    @Email
    @NotNull
    private String email;

    @Size(min = 8, max = 20)
    private String password;
}
```

Using @Valid in Controller

```java
@RestController
@RequestMapping("/api/users")
public class UserController {

    @PostMapping
    public ResponseEntity<UserDto> createUser(
        @Valid @RequestBody UserDto userDto,
        BindingResult result) {
      if (result.hasErrors()) {
        // handle errors
      }
      // service call
    }
```

Ex 06

- Add validation on PizzaDTO to make sure name has only letter characters and space and the price is minimum 10 and maximum 100 and all fields are required
- Use Response Entity to return appropriate statuses

For ease of use:

ResponseEntity.created(location).build();

ResponseEntity.badRequest().body(result.getAllErrors());

Creating Custom Validation Annotations in Spring

1. Defining the Annotation

```java
@Documented
@Constraint(validatedBy = NameValidator.class)
@Target({ElementType.FIELD, ElementType.PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
public @interface ValidName {
    String message() default "Invalid name";
    Class<?>[] groups() default {};
    Class<? extends Payload>[] payload() default {};
}
```

2. Implementing the Validator

```java
public class NameValidator implements ConstraintValidator<ValidName, String> {
    @Override
    public void initialize(ValidName constraint) { }

    @Override
    public boolean isValid(String value, ConstraintValidatorContext ctx) {
        return value != null && value.matches("[A-Za-z ]+");
    }
}
```

3. Wiring with @Constraint

- - The `validatedBy` attribute links to your validator class
- - Spring Boot auto-detects ConstraintValidator implementations
- - Ensure `hibernate-validator` is on the classpath

4. Using the Annotation

```java
public class UserDto {
    @ValidName
    private String fullName;

    // other fields/getters/setters
}
```

5. Customizing Messages

- - Use `message` attribute in the annotation
- - Reference messages in `ValidationMessages.properties`:
- `validname.invalid=Name must contain only letters and spaces`
- - Support i18n by locale-specific files

Global Validation Error Handling with @RestControllerAdvice

Manual Validation Drawbacks

- Controllers cluttered with validation logic
- Repetitive error checking across endpoints
- Inconsistent error response formats
- Harder to maintain and evolve

@RestControllerAdvice for Global Handling

- Centralizes exception handling logic
- Applies across all @RestController endpoints
- Keeps controllers focused on business logic
- Consistent response format for errors

Example: UniversalExceptionHandler

```java
@RestControllerAdvice
public class UniversalExceptionHandler {

    @ResponseStatus(HttpStatus.BAD_REQUEST)
    @ExceptionHandler(MethodArgumentNotValidException.class)
    public Map<String, String> handleValidationExceptions(
            MethodArgumentNotValidException ex) {
        Map<String, String> errors = new HashMap<>();
        ex.getBindingResult()
          .getAllErrors()
          .forEach(error -> {
              String field = ((FieldError) error).getField();
              String msg = error.getDefaultMessage();
              errors.put(field, msg);
          });
        return errors;
    }
}
```

Benefits of Global Handling

- DRY: No duplication across controllers
- Consistency: Uniform error structure
- Maintainability: Single place to update
- Cleaner Controllers: Focus on core logic

Best Practices

- Define a standard error response DTO
- Include error codes and user-friendly messages
- Support internationalization (i18n)
- Log exceptions appropriately
- Handle other exceptions (e.g., NotFound, AccessDenied)

Ex 07

- Add custom validation to check there are no duplicate pizza names

You can use existsBy keyword at repository level

- Add Global exception handling