04

유효성 검증

01. 유효성 검사/데이터 검증

- 서비스의 비즈니스 로직을 올바르게 동작하게 하기 위해 사용되는 데이터에 대한 사전 검증을 하는 작업
- 유효성 검사 또는 데이터 검증(Validation)이라고 함
- 데이터 검증은 여러 Layer에서 발생
- 들어오는 데이터에 대해 의도한 형식의 값이 제대로 들어오는지 체크하는 과정
- Dependency를 추가해야함
- implementation 'org.springframework.boot:spring-boot-starter-validation'

02. Validation Annotaion

- @Size : 문자의 길이 조건
  - @Size(min=10, max=20)
- @NotNull : Null 값 불가
- @NotEmpty : @NotNull + “” 불가
- @NotBlank : @NotEmpty + “ “불가
- @Max : 최대값 조건 설정
  - @Max(value=100)
- @Min : 최소값 조건 설정
- @Past /@PastOrPresent : 과거 날짜/ 현재포함과거날짜
- @Future/@FutureOrPresent

03. DTO 클래스에 검증 Annotation 설정

```
@Data
@AllArgsConstructor
@NoArgsConstructor
@Builder
public class ProductDTO {
    private Integer id;
    @NotEmpty(message="이름은  비어있을수  없습니다")
    private String title;
    private String imgsrc;
    private int price;
}
```

04. Controller에서 검증 설정

```
@PostMapping(value="/new-product")
public ResponseEntity<ProductDTO>
newProduct(@Valid  @RequestBody ProductDTO
productDTO){
    ProductDTO
product=this.productService.saveProduct(productDTO);
    return
ResponseEntity.status(HttpStatus.OK).body(product);
}
```

05

예외처리

01. ResponseEntity<T>

REST API 설계 시 상태 코드와 메시지를 Client에 전송

상태 코드

- Client Error : 400번대
- Server Error : 500번대
- 정상 : 200번대

```
@GetMapping("/detail/{id}")
ResponseEntity<String> detail() {
    try {
        throw new Exception("이런저런에러");
    } catch(Exception e){
        return ResponseEntity.status(에러코드).body("에러이유");
    }
}
```

ResponseEntity.status(HttpStatus.NOT_FOUND).body("에러남");

02. @ControllerAdvice & @RestControllerAdvice

@ControllerAdvice

- 전역 예외 처리를 구현
- 컨트롤러에서 발생하는 예외를 한 곳에서 처리
- 특정 Controller를 지정할 수도 있음
  - @ControllerAdvice(basePackages = "com.example.controller")
- @ResponseBody와 함께 사용하여 Json데이터를 오류데이터로 전달 가능

@RestControllerAdvice

- @ControllerAdvice 와 @ResponseBody를 합친 annotaion

02. @ControllerAdvice & @RestControllerAdvice

```
@RestControllerAdvice
public class MyExceptionHandler {
    @ExceptionHandler(Exception.class)
    public ResponseEntity<String> handleException(Exception e) {
        return ResponseEntity.status(HttpStatus.NOT_FOUND).body("에러남");
    }

    @ExceptionHandler(MethodArgumentTypeMismatchException.class)
    public ResponseEntity<String> handleException3(Exception e) {
        return ResponseEntity.status(HttpStatus.NOT_FOUND).body("에러남");
    }
}
```

03. MethodArgumentNotValidException

- 유효성 검사 실패 시 발생하는 예외
  - @Valid를 사용하여 요청 데이터의 유효성을 검사할 때 유효하지 않은 값이 전달되면 발생.
- 주로 요청 데이터의 필드 오류를 반환

```
@ExceptionHandler(MethodArgumentNotValidException.class)
public ResponseEntity<Map<String, String>>
handleValidationExceptions(MethodArgumentNotValidException ex) {
    Map<String, String> errors = new HashMap<>();
    ex.getBindingResult().getFieldErrors().forEach(error ->
            errors.put(error.getField(), error.getDefaultMessage())
    );
    return
ResponseEntity.status(HttpStatus.BAD_REQUEST).body(errors);
}
```

04. HttpMessageNotReadableException

- **JSON 형식 오류나 요청 본문 파싱 오류를 처리**

```
@ExceptionHandler(HttpMessageNotReadableException.class)
public ResponseEntity<String>
handleHttpMessageNotReadableException(HttpMessageNotReadab
leException ex) {
    return ResponseEntity.status(HttpStatus.BAD_REQUEST)
            .body("Invalid request body: " + ex.getMessage());
}
```

05. EntityNotFoundException

- JPA 엔티티를 찾지 못했을 때 발생하는 예외를 처리

```
@ExceptionHandler(EntityNotFoundException.class)
public ResponseEntity<String>
handleEntityNotFoundException(EntityNotFoundException ex) {
    return ResponseEntity.status(HttpStatus.NOT_FOUND)
            .body("Resource not found: " + ex.getMessage());
}
```

- JPA의 find(), findById(), deleteById() 메소드는 기본적으로 데이터가 없을 경우 ***발생하는 것이 아님***
- repository.findById().get()을 했을때 repository.findById()가 Entity객체를 찾지 못했는데 get()을 호출했을 때 발생

06. MethodArgumentTypeMismatchException

- 컨트롤러 메소드의 매개변수 타입과 클라이언트가 전달한 요청 값의 타입이 일치하지 않을 때 발생

```
@ExceptionHandler(MethodArgumentTypeMismatchException.class)
public ResponseEntity<String>
handleMethodArgumentTypeMismatch(MethodArgumentTypeMismatchEx
ception ex) {
    String errorMessage = String.format(
            "Invalid argument: '%s'. Expected type: '%s'.",
            ex.getValue(),
            ex.getRequiredType().getSimpleName()
    );
    return
ResponseEntity.status(HttpStatus.BAD_REQUEST).body(errorMessage);
}
```

06

JPA연관성

01. JPA 연관 관계 매핑

- 데이터베이스 테이블 간의 관계를 엔티티 객체 간의 관계로 표현
- 객체 지향 프로그래밍과 관계형 데이터베이스의 간극을 해소
- 주요 연관 관계:
  - 일대일, 다대일, 일대다, 다대다

02. 일대일 관계 (@OneToOne)

- 한 엔티티가 다른 엔티티와 1:1로 매핑
- 주로 상세 정보를 별도 테이블로 관리할 때 사용
- 예시: 사용자와 사용자 프로필 관계

```
@Entity
public class User {
    @Id @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Long id;
    private String name;
    @OneToOne
    @JoinColumn(name = "profile_id") // 외래 키 지정
    private Profile profile;
}
@Entity
public class Profile {
    @Id @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Long id;
    private String bio;
}
```

03. Many-to-One (N:1 관계) : 단방향 관계

- 여러 개의 엔티티가 하나의 엔티티와 매핑
- 예: 게시글과 작성자
  - 여러 게시글이 동일한 작성자를 가질 수 있음.
  - One-to-Many와 반대 방향에서 바라본 관계

```
@Entity
public class Post {
    @Id @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Long id;
    private String title;
    @ManyToOne
    @JoinColumn(name = "user_id")
    private User user;
}

@Entity
public class User {
    @Id @GeneratedValue
    private Long id;
    private String name;
}
```

04. One-to-Many (1:N 관계) : 양방향 관계

- 하나의 엔티티가 여러 개의 다른 엔티티와 매핑
- 예: 사용자와 게시글
  - 한 사용자는 여러 개의 게시글을 가질 수 있음.

```java
@Entity
public class User {
    @Id @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Long id;
    private String name;
    @OneToMany(mappedBy = "user") // 게시글에서 user 필드와 매핑
    private List<Post> posts = new ArrayList<>(); // 실제 테이블에는 존재하지 않음
}

@Entity
public class Post {
    @Id @GeneratedValue
    private Long id;
    private String title;
    @ManyToOne
    @JoinColumn(name = "user_id") // 외래 키 지정
    private User user;
}
```

04. One-to-Many (1:N 관계) : 양방향 관계

- 서로의 정보를 가지고 있기 때문에 직렬화시 순환참조의 문제가 있음
- @JsonManagedReference : 직렬화에 포함되는 쪽
- @JsonBackReference : 직렬화에 포함되지 않는 쪽
- 양방향이므로 어느쪽이 주인이 되는지 반드시 지정해야 함
  - mappedBy : 주인이 되는 쪽을 지정
  - 비주인이 되는 쪽은 데이터베이스에 영향을 주지 않음
  - 주인이 되는 쪽이 외래키를 관리함

05. Many-to-Many (N:M 관계)

- 여러 개의 엔티티가 서로 여러 개와 매핑
- 예: 학생과 강의
  - 한 학생은 여러 강의를 수강할 수 있고, 한 강의는 여러 학생이 수강할 수 있음
- @ManyToMany를 사용하며, 중간 테이블이 필요
  - Jpa가 자동으로 생성
  - 중간테이블에 양쪽 테이블과 연결되는 필드 이외에 필드가 필요하다면 직접 생성할 수도 있음

05. Many-to-Many (N:M 관계)

```java
@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 = new ArrayList<>();
}
```

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

    private String title;

    @ManyToMany(mappedBy = "courses")
    private List<Student> students = new ArrayList<>();
}
```

06. 연관성 매핑 시 주의점

- **mappedBy 속성**
  - 관계의 주인을 정의
  - 주인은 외래 키를 관리하며, 반대쪽은 읽기 전용
  - 일반적으로 @OneToMany, @ManyToMany에서 사용
- **Fetch Type (지연 로딩 vs 즉시 로딩)**
  - Eager Fetch (즉시 로딩)
    - 연관된 엔티티를 즉시 조회
    - 기본값: @OneToOne, @ManyToOne
  - Lazy Fetch (지연 로딩)
    - 연관된 엔티티를 실제로 사용할 때 조회
    - 기본값: @OneToMany, @ManyToMany

```
@OneToMany(fetch = FetchType.LAZY)
private List<Post> posts;
```

07. 연관성 매핑 활용방법

- 연관 관계의 주인 명확히 설정
  - mappedBy를 사용해 주인을 설정하고, 외래 키를 관리
- 지연 로딩(Lazy)을 기본으로 설정
  - 성능 최적화를 위해 필요할 때만 데이터를 로드
- Cascade 옵션 활용
  - 연관된 엔티티를 자동으로 저장, 삭제하려면 cascade 속성을 설정

```
@OneToMany(cascade = CascadeType.ALL,
                        orphanRemoval = true)
private List<Post> posts;
```

08. Native SQL 을 이용한 사용자 정의 Query를 실행

```
@Query(value= "select * from usertbl", nativeQuery=true)
List<UserEntity> allUserInfo();

@Query(value= "select * from usertbl where addr=:addr",
nativeQuery=true)
List<UserEntity> searchUserInfo(@Param("addr") String addr);

@Query(value= “select * from usertbl where birthyear=:birthyear",
nativeQuery=true)
List<UserEntity> searchUserInfo(@Param("birthyear") Integer birthYear);

 @Query(value="select * from usertbl where addr=:addr and
              birthyear=:birthyear", nativeQuery = true)
   List<UserEntity> searchUserInfo(@Param("addr") String addr,
                                   @Param("birthyear") Integer
birthYear);
```