실습**1:** **8**월**14**일**(**목**)** 제출

[실습**2-1]:** **8**월**18**일**(**월**)** 제출 **/ [**실습**2-2]: 8**월**20**일**(**수**)** 제출

[실습**2-3]:** **8**월**25**일**(**월**)** 제출

[실습**2-4]:** **8**월**27**일**(**수**)** 제출

예습 할 내용 8월13일(수)

SQL을 처음 사용하시는 분들은 Select, Insert, Update, Delete 이 무엇인지 이해하기

모두에게 적용되는 내용

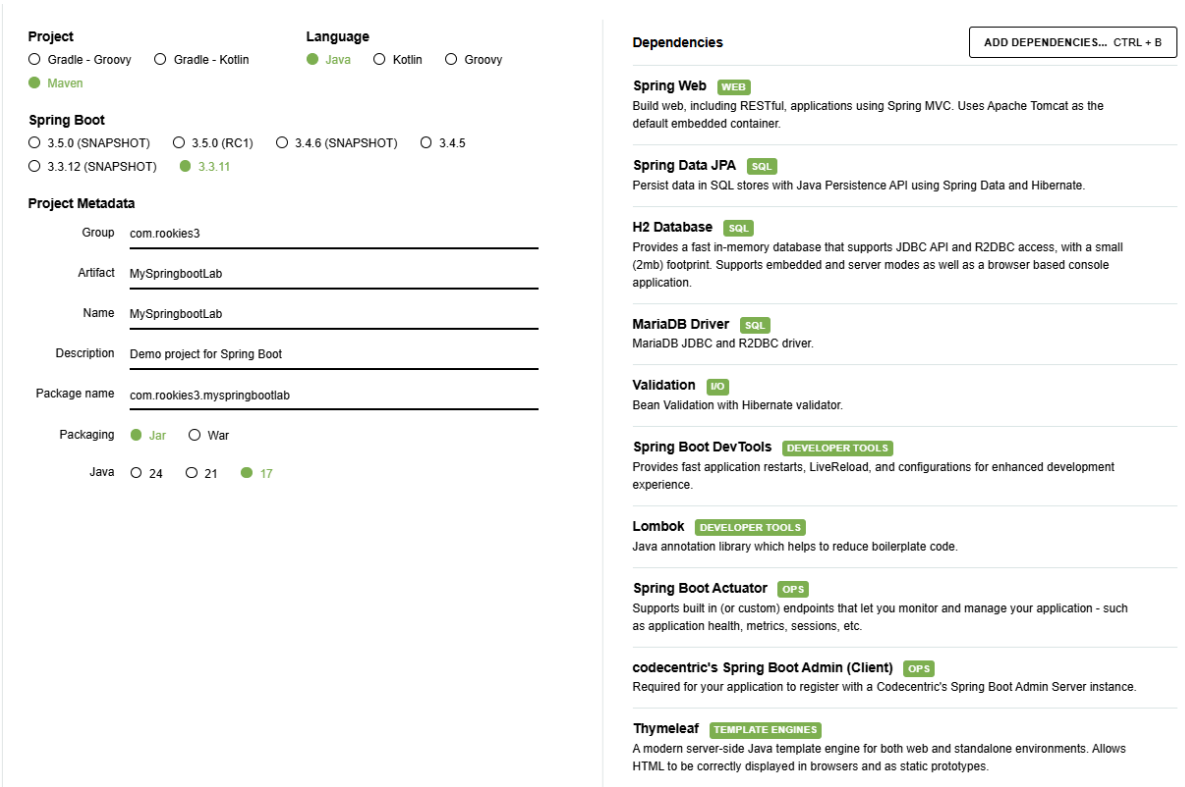
JPA(Java Persistence API) , Hibernate , Spring Data JPA 가 무엇인지 이해하고, 각각 어떤 차이점이 있는 미리 찾아보기

SpringBoot 연습문제

[SpringBoot 실습**1] Spring Boot** 프로젝트 작성하기

1-1) **Spring Initializer** 웹사이트에서 스프링 부트 프로젝트 작성하기

- **boot version**은 **3.4.8** 으로 선택해 주세요



Spring Initializer에서 **Spring Boot** 프로젝트를 생성할 때 추가 할 의존성

webmvc, data jpa, h2, mariadb, validation, devtools, lombok, actuator, spring boot admin client,thymeleaf

1-2) **Spring Boot** 배너 변경하기

1-3) **application.properties** 설정 파일에 환경변수 설정하기

myprop.username=springboot

myprop.port=${random.int(1,100)}

1-4) **MyPropRunner** 클래스 작성하기

@Value 어노테이션을 사용하여 application.properties 파일에 있는 환경변수를 Load 하여 출력하기

1-5) **MyPropProperties** 클래스를 작성하기

: application.properties에 있는 환경변수를 저장하고 조회하는 Properties 클래스 작성하기

: MyPropRunner에서 MyPropProperties 객체를 주입(Injection) 받아서 getter 메서드를 출력하기

: spring-boot-configuration-processor 의존성 설치하기

1-6) **ProdConfig** 클래스와 **DevConfig** 클래스 작성하기

MyEnvironment.java

```
public class MyEnvironment {
        String mode;
}
```

1. **ProdConfig** 클래스

@Profile("prod") 와 @Configuration 어노테이션 사용

MyEnvironment를 SpringBean으로 생성하고 mode값을 "운영환경" 으로 설정한다.

2. **TestConfig** 클래스

@Profile("test") 와 @Configuration 어노테이션 사용

MyEnvironment를 SpringBean으로 생성하고 mode값을 "개발환경" 으로 설정한다.

1-7) 프로파일용 **properties file** 작성하기

- application-prod.properties 에 base package의 로그 레벨을 INFO 설정하기
- application-test.properties 에 base package의 로그 레벨을 DEBUG 설정하기

MyPropsRunner 클래스 수정

System.out.println() 를 logger.debug()와 logger.info() 메서드로 변경합니다.

1-8) **Spring Boot** 프로젝트를 **jar** 파일로 생성하여 실행하기

[SpringBoot 실습2] Spring Boot와 JPA(Java Persistence API) 활용

MariaDB lab 계정생성 Script

도서 관리 시스템 구현하기

Book 클래스 다이어그램

[실습2-1] Entity, Repository, Test케이스 작성

1. **Book** 엔티티, **BookRepository** 인터페이스를 작성하세요.
    - **Book** 엔티티**:**
      id(Long), title(String), author(String), isbn(String),
      publishDate(LocalDate), price(Integer)
    - **BookRepository:**
      findByIsbn(String isbn), findByAuthor(String author) 메소드 추가
2. **BookRepositoryTest** 클래스에서 아래의 테스트 케이스를 구현하세요:
    - 도서 등록 테스트 (testCreateBook())
    - ISBN으로 도서 조회 테스트 (testFindByIsbn())
    - 저자명으로 도서 목록 조회 테스트 (testFindByAuthor())
    - 도서 정보 수정 테스트 (testUpdateBook())
    - 도서 삭제 테스트 (testDeleteBook())

title	author	isbn	price	publishDate
"스프링 부트 입문"	"홍길동"	"9788956746425"	30000	2025-05-07
"JPA 프로그래밍"	"박둘리"	"9788956746432"	35000	2025-04-30

[실습2-2] Rest컨트롤러 클래스 작성

3. **BookRestController** 클래스를 작성하세요. (BusinessException 과 Advice, ErrorObject 클래스는 그대로 사용하시면 됩니다.)

Postman으로 다음 API를 테스트하세요:

- POST /api/books : 새 도서 등록
- GET /api/books : 모든 도서 조회
- GET /api/books/{id} : ID로 특정 도서 조회
- GET /api/books/isbn/{isbn}/ : ISBN으로 도서 조회
- PUT /api/books/{id}: 도서 정보 수정
- DELETE /api/books/{id} : 도서 삭제

Book이 존재하지 않을때 (404) 처리하는 방법

1. getUserById() 메서드는 ResponseEntity<Book> Optional 클래스의 map() / orElse() 를 사용합니다.
   a. Optional 클래스의 map() / orElse() 를 사용하는 방법 대신에 BusinessException 과 ErrorObject / DefaultExceptionAdvice 를 사용하셔도 됩니다.

   : ResponseEntity<Book> getUserById(Long id)
2. getUserByIsbn() 메서드는 BusinessException 과 ErrorObject / DefaultExceptionAdvice 를 사용합니다.

   : Book getUserByIsbn(String isbn>


```
{
"title":"스프링 부트 입문",
"author":"홍길동",
"isbn":"9788956746425",
"price":30000,
"publishDate":"2025-05-07"
}
{
"title":"JPA 프로그래밍",
"author":"박둘리",
"isbn":"9788956746432",
"price":35000,
"publishDate":"2025-04-30"
}
```

[실습2-3] Spring Boot와 JPA(Java Persistence API) 활용

도서 관리 시스템 구현하기 (Service와 DTO 추가하기)

[실습2-3] 클래스 다이어그램

Book 과 Service 클래스 다이어그램

1. 계층 구조

Repository 계층

- `Book.java`: 데이터베이스 테이블과 매핑되는 JPA 엔티티입니다.
- `BookRepository.java`: JPA Repository 인터페이스로 데이터 액세스를 담당합니다.
- ISBN과 저자로 검색하는 메서드가 정의되어 있습니다.

DTO(Data Transfer Object) 계층

- `BookDTO.java`: 클라이언트와 서버 간의 데이터 전송을 위한 객체들을 모아놓은 클래스입니다.
- 내부 클래스로 세 가지 DTO를 정의함:
    - `BookCreateRequest`: 도서 생성 시 사용되는 DTO
    - `BookUpdateRequest`: 도서 정보 업데이트 시 사용되는 DTO
    - `BookResponse`: 클라이언트에게 반환되는 도서 정보 DTO
- 각 DTO는 검증 애노테이션을 포함하여 입력 데이터의 유효성을 검증합니다.

Service 계층

- `BookService.java`: 비즈니스 로직을 담당하는 서비스 클래스입니다.
- `@Transactional` 어노테이션을 사용해 트랜잭션 관리를 합니다.
- 기존에 컨트롤러에 있던 비즈니스 로직이 Service로 이동함
- 요청 검증, 데이터 변환, 저장소 호출 등의 작업을 수행합니다.

Controller 계층

- `BookController.java`: HTTP 요청을 처리하는 REST 컨트롤러입니다.
- 서비스 계층에 비즈니스 로직을 위임하고 HTTP 응답을 구성합니다.
- `@Valid` 애노테이션을 사용해 요청 본문의 유효성을 검증합니다.
- Update(수정) - 저자(author) 와 가격(price) , 제목(title), 출판일자(publishDate) 수정합니다.

예외 처리

- `BusinessException.java`: 비즈니스 로직 오류를 처리하는 예외 클래스입니다.

- `DefaultExceptionAdvice.java`: 애플리케이션 전체의 예외를 처리하는 클래스입니다.

2. 주요 개선 사항

1. 관심사 분리
    - 각 계층이 자신의 역할에만 집중하도록 구조화했습니다.
    - 컨트롤러는 요청 처리와 응답 반환에, 서비스는 비즈니스 로직에 집중합니다.
2. 데이터 검증 강화
    - DTO에 검증 애노테이션을 추가하여 입력 데이터의 유효성을 검증합니다.
    - 전역 예외 처리기를 통해 검증 오류에 대한 일관된 응답을 제공합니다.
3. 비즈니스 로직 캡슐화
    - 비즈니스 로직이 서비스 계층에 캡슐화되어 재사용성과 테스트 용이성이 향상되었습니다.
    - DTO와 엔티티 간 변환 로직이 명확하게 정의되었습니다.
4. 확장성 개선
    - 업데이트 요청에서 변경이 필요한 필드만 처리하도록 구현했습니다.
    - 새로운 기능이나 필드를 쉽게 추가할 수 있는 구조입니다.
5. 일관된 응답 형식
    - 모든 API 응답이 일관된 형식으로 반환됩니다.
    - 오류 응답도 통일된 형식으로 제공됩니다.

3. 추가 기능

1. 유효성 검증
    - 입력 데이터에 대한 유효성 검증이 강화되었습니다.
    - 잘못된 입력에 대한 친절한 오류 메시지를 제공합니다.
2. 예외 처리 개선
    - 비즈니스 예외와 검증 예외를 구분하여 처리합니다.
    - 전역 예외 핸들러를 통해 일관된 오류 응답 형식을 제공합니다

BookService의 updateUser() 메서드에서 입력값이 있는 경우에만 값을 변경하기

```
public BookDTO.BookResponse updateBook(Lond id, BookDTO.BookUpdateRequest request) {

    //변경이 필요한 필드만 업데이트

        if(request.getTitle() != null) {

      existBook.setTitle(request.getTitle());

        }

}
```

[수업중 설명 **2-4**] **Spring Boot**와 **JPA(Java Persistence API)** 활용

학생 관리 시스템 구현하기 **(1:1** 연관관계 **)**

[수업중 실습2-4] 제공하는 Source code

Student 와 Student Detail Source Code (StudentService는 ErrorCode를 사용하지 않음)

ErrorCode 와 변경된 BusinessException (StudentService는 ErrorCode를 사용함)

1. 엔티티 클래스

Student 엔티티

- 기본 정보: id, name, studentNumber 필드를 포함
- `@OneToOne` 관계를 통해 StudentDetail과 연결
- `mappedBy="student"`로 지정하여 Student가 관계의 주인이 아님을 표시

StudentDetail 엔티티

- 추가 정보: address, phoneNumber, email, dateOfBirth 필드를 포함
- `@OneToOne(fetch = FetchType.LAZY)`로 Student 엔티티와 지연 로딩 관계 설정
- `@JoinColumn(name = "student_id", unique = true)`로 외래 키 설정 및 유니크 제약조건 추가
- 이 엔티티가 관계의 주인이 됩니다 (외래 키를 소유)

JPA(Java Persistence API)의 `FetchType.LAZY`와 `FetchType.EAGER`는 연관 관계에 있는 엔티티를 언제 데이터베이스에서 로드 할지를 결정하는 전략

FetchType.LAZY (지연 로딩**)**

- 연관된 엔티티를 실제로 사용할 때까지 로딩을 지연시킵니다
- 프록시 객체를 생성하여 실제 데이터가 필요한 순간에 DB 쿼리를 실행합니다
- 메모리 효율적이고 초기 로딩 시간이 빠릅니다

FetchType.EAGER (즉시 로딩**) -** 성능에 좋지 않음

- 엔티티를 로딩할 때 연관된 엔티티도 함께 즉시 로딩합니다
- 한 번의 쿼리로 모든 연관 데이터를 가져옵니다 (JOIN 활용)
- 즉시 모든 데이터를 메모리에 로드하므로 메모리 사용량이 높을 수 있습니다.

기본 개념

JPA에서 1:1(일대일) 연관관계는 한 엔티티가 다른 하나의 엔티티와만 연결되는

관계입니다.

- 학생(Student)과 학생상세정보(StudentDetail)
- 사용자(User)와 사용자프로필(UserProfile)

주인(Owner) 개념

- 주인**(Owner)**: 외래키를 직접 관리하는 쪽이 관계의 주인입니다.
- 주인이 아닌 쪽**(mappedBy)**: 관계를 참조만 하는 쪽입니다.

누가 주인 인가요?

- **StudentDetail**이 주인**(Owner)**이 맞습니다**!**
  - `@JoinColumn`이 StudentDetail 쪽에 있기 때문입니다.
  - Student 엔티티는 `mappedBy`로 표시되어 관계를 참조만 합니다.
  -
- 외래키 위치: `student_id`는 StudentDetail 테이블에 생성됩니다.
- **LAZY** 로딩: 양쪽 모두 지연 로딩으로 설정되어 있어 성능 최적화됨
- **CASCADE**: Student를 저장할 때 StudentDetail도 자동으로 저장됩니다.
- **unique=true**: 한 Student는 하나의 StudentDetail만 가질 수 있음

Fetch Join이란?

Fetch Join은 JPA에서 성능 최적화를 위해 제공하는 특별한 JOIN 기능으로, 연관된

엔티티나 컬렉션을 한 번의 SQL 쿼리로 함께 조회하는 기능입니다.

주요 기능

1. 즉시 로딩 최적화: Fetch Join을 사용하면 지연 로딩(Lazy Loading)으로 설정된 연관 관계도 한 번에 조회 가능
2. **N+1** 문제 해결: 연관된 엔티티를 조회할 때 발생하는 N+1 쿼리 문제를 방지
3. 단일 쿼리 실행: 여러 엔티티를 조인하여 하나의 쿼리로 가져옴

필요성

1. 성능 최적화:
   - 일반적인 지연 로딩은 처음 엔티티 조회 시 1번, 연관 엔티티 접근 시 N번의 쿼리가 발생(N+1 문제)
   - Fetch Join은 처음부터 연관 데이터를 함께 가져오므로 추가 쿼리 발생 없음
2. 불필요한 쿼리 감소:
3. 컬렉션 조회 시 유용:
   - 일대다 관계의 컬렉션을 조회할 때 특히 효과적

주의사항

1. 페이징 처리 문제: 컬렉션 Fetch Join에 페이징 적용 시 메모리에서 처리되므로 주의 필요
2. 중복 데이터: 일대다 조인 시 결과에 중복이 발생할 수 있어 DISTINCT 사용 필요

2. Repository 인터페이스

StudentRepository

- 기본적인 CRUD 기능 제공
- `findByIdWithStudentDetail` 메서드로 Student와 StudentDetail을 함께 로드

StudentDetailRepository

- `findByStudentId`로 특정 Student의 Detail 조회
- 중복 이메일/전화번호 체크를 위한 exists 메서드 제공

3. DTO 클래스

StudentDTO

- Request: 학생 생성/수정 시 사용
- StudentDetailDTO: 중첩 클래스로 학생 상세 정보 관리
- Response: API 응답용 DTO
- StudentDetailResponse: 상세 정보 응답용 중첩 DTO

4. Service 클래스

StudentService에서는:

- 모든 학생 조회
- ID 또는 학번으로 학생 조회
- 학생 생성/수정/삭제 기능
- 중복 확인 로직 (학번, 이메일, 전화번호)
- 양방향 관계 설정 로직

5. Controller 클래스

StudentController에서는:

- RESTful API 인터페이스 제공
- 표준 HTTP 메서드(GET, POST, PUT, DELETE) 및 응답 코드 사용

1. 등록 (post)

http://localhost:8080/api/students

```
{
  "name": "홍길동",
  "studentNumber": "S12345",
  "detailRequest": {
    "address": "서울시 강남구 테헤란로 123",
    "phoneNumber": "010-1234-5678",
    "email": "hong@example.com",
    "dateOfBirth": "1995-05-15"
  }
}

{
  "name": "김철수",
  "studentNumber": "S67890",
  "detailRequest": {
    "address": "서울시 마포구 홍대로 456",
    "phoneNumber": "010-9876-5432",
    "email": "kim@example.com",
    "dateOfBirth": "1996-08-21"
  }
}

{
  "name": "이영희",
  "studentNumber": "S13579",
  "detailRequest": {
    "address": "서울시 서초구 서초대로 789",
    "phoneNumber": "010-5555-7777",
    "email": "lee@example.com",
    "dateOfBirth": "1997-11-03"
  }
}
```

2. 목록 조회 (get)

http://localhost:8080/api/students

3. ID로 조회(get) - FETCH Join

http://localhost:8080/api/students/1

JPQL

```
select s1_0.student_id,s1_0.name,sd1_0.student_detail_id,sd1_0.address,
sd1_0.date_of_birth,sd1_0.email,sd1_0.phone_number,s1_0.student_number
from students s1_0 join student_details sd1_0 on s1_0.student_id=sd1_0.student_id
where s1_0.student_id=?
```

Navtive Query

```
select s.name, s.student_number,sd.address,sd.email
  from students s, student_details sd
  where s.student_id = sd.student_id;
```

4. 학번으로 조회 - Query Method

http://localhost:8080/api/students/number/S12345

JPQL

```
Hibernate: select s1_0.student_id,s1_0.name,s1_0.student_number from students s1_0 where s1_0.student_number=?
Hibernate: select sd1_0.student_detail_id,sd1_0.address,sd1_0.date_of_birth,sd1_0.email,sd1_0.phone_number,sd1_0.student_id
from student_details sd1_0 where sd1_0.student_id=?
```

5. 수정(put)

http://localhost:8080/api/students/1

```
{
  "name": "홍길동",
  "studentNumber": "S12345",
  "detailRequest": {
    "address": "서울시 송파구 올림픽로 123",
    "phoneNumber": "010-1234-5678",
    "email": "hong.updated@example.com",
    "dateOfBirth": "1995-05-15"
  }
}
```

JPQL (address, email 수정)

```
Hibernate: select s1_0.student_id,s1_0.name,s1_0.student_number from students s1_0 where s1_0.student_id=?
Hibernate: select sd1_0.student_detail_id,sd1_0.address,sd1_0.date_of_birth,sd1_0.email,sd1_0.phone_number,sd1_0.student_id from student_details sd1_0 where sd1_0.student_id=?
Hibernate: select sd1_0.student_detail_id from student_details sd1_0 where sd1_0.email=? limit ?
```

Hibernate: update student_details set address=?,date_of_birth=?,email=?,phone_number=?,student_id=? where student_detail_id=?

삭제(delete)

http://localhost:8080/api/students/1

JPQL

Hibernate: select count(*) from students s1_0 where s1_0.student_id=?
Hibernate: select s1_0.student_id,s1_0.name,s1_0.student_number from students s1_0 where s1_0.student_id=?
Hibernate: select sd1_0.student_detail_id,sd1_0.address,sd1_0.date_of_birth,sd1_0.email,sd1_0.phone_number,sd1_0.student_id from student_details sd1_0 where sd1_0.student_id=?
Hibernate: delete from student_details where student_detail_id=?
Hibernate: delete from students where student_id=?

[실습2-4] Spring Boot와 JPA(Java Persistence API) 활용

도서 관리 시스템 구현하기 (1:1 연관관계)

[실습2-4] 클래스 다이어그램

Book과 BookDetail 클래스 다이어그램

1. 엔티티 클래스

Book.java

- 책에 관한 기본 정보 (제목, 저자, ISBN, 가격, 출판일)를 저장합니다.
- `@OneToOne` 관계를 통해 BookDetail과 연결되어 있습니다.
- `mappedBy="book"`으로 지정하여 Book이 관계의 주인이 아님을 표시합니다.

BookDetail.java

- 책에 대한 상세 정보 (설명, 언어, 페이지 수, 출판사, 표지 이미지 URL, 에디션)을 저장합니다.
- `@OneToOne(fetch = FetchType.LAZY)`로 Book 엔티티와 지연 로딩 관계를 설정합니다.
- `@JoinColumn(name = "book_id", unique = true)`로 외래 키 설정 및 유니크 제약조건을 추가합니다.
- 이 엔티티가 관계의 주인입니다 (외래 키를 소유).

1:1 관계: Book과 BookDetail은 일대일 관계를 가지며, 각 책은 하나의 상세 정보만 가질 수 있습니다.

지연 로딩: `FetchType.LAZY`를 사용하여 필요할 때만 연관된 데이터를 로드합니다.

영속성 컨텍스트: `CascadeType.ALL`을 사용하여 Book을 저장/삭제할 때 BookDetail도 함께 처리됩니다.

2. 레포지토리 인터페이스

BookRepository.java

- 책을 ID, ISBN, 저자, 제목 등으로 검색하는 메서드를 제공합니다.
- `findByIdWithBookDetail`과 `findByIsbnWithBookDetail` 메서드로 Book과 BookDetail을 함께 로드합니다.
- 중복 ISBN 확인을 위한 `existsByIsbn` 메서드를 제공합니다.

BookDetailRepository.java

- `findByBookId`로 특정 책의 상세 정보를 조회합니다.
- `findByIdWithBook`로 상세 정보와 함께 책 정보를 로드합니다.
- `findByPublisher`로 특정 출판사의 책 상세 정보를 조회합니다.

3. 레포지토리 테스트

BookRepository 테스트 클래스 Source

구현하지 않고 위의 링크에서 복사해서 실행만 해보시면 됩니다.

BookRepositoryTest.java

- 책과 책 상세 정보의 생성, 조회 기능을 테스트합니다.
- 다양한 검색 조건(ISBN, 저자)에 따른 조회 기능을 검증합니다.
- 책과 책 상세 정보 간의 1:1 관계가 올바르게 작동하는지 확인합니다.

4. DTO 클래스

BookDTO 클래스 Source

구현하지 않고 위의 링크에서 복사해서 사용하시면 됩니다.

BookDTO.java

- **Request**: 책 생성/수정 시 사용하는 DTO입니다.
- **BookDetailDTO**: 책 상세 정보 요청용 중첩 DTO입니다.
- **Response**: API 응답용 DTO입니다.
- **BookDetailResponse**: 책 상세 정보 응답용 중첩 DTO입니다.
- 유효성 검사: ISBN 패턴 검사, 가격 음수 방지, 출간일 과거 날짜 제한 등의 검증 규칙이 포함되어 있습니다.

5. 서비스 클래스

BookService.java

- 비즈니스 로직을 처리하는 서비스 계층입니다.
- 주요 기능:
    - 모든 책 조회
    - ID 또는 ISBN으로 특정 책 조회
    - 저자나 제목으로 책 검색
    - 책 생성/수정/삭제
    - ISBN 중복 검사

6. 컨트롤러 클래스

BookController.java

- REST API 엔드포인트를 제공합니다.
- 주요 엔드포인트:
    - `GET /api/books` - 모든 책 조회
    - `GET /api/books/{id}` - ID로 책 조회
    - `GET /api/books/isbn/{isbn}` - ISBN으로 책 조회
    - `GET /api/books/search/author?author={author}` - 저자로 책 검색
    - `GET /api/books/search/title?title={title}` - 제목으로 책 검색
    - `POST /api/books` - 책 생성
    - `PUT /api/books/{id}` - 책 수정
    - `DELETE /api/books/{id}` - 책 삭제

1. 등록 (post)

http://localhost:8080/api/books

```
{
  "title": "Clean Code",
  "author": "Robert C. Martin",
  "isbn": "9780132350884",
  "price": 45,
  "publishDate": "2008-08-01",
  "detailRequest": {
    "description": "A handbook of agile software craftsmanship",
    "language": "English",
    "pageCount": 464,
    "publisher": "Prentice Hall",
    "coverImageUrl": "https://example.com/cleancode.jpg",
    "edition": "1st"
  }
}
```

```
{
  "title": "Effective Java",
  "author": "Joshua Bloch",
  "isbn": "9780134685991",
  "price": 55,
  "publishDate": "2017-12-27",
  "detailRequest": {
    "description": "The definitive guide to Java programming language",
    "language": "English",
    "pageCount": 416,
    "publisher": "Addison-Wesley",
    "coverImageUrl": "https://example.com/effectivejava.jpg",
    "edition": "3rd"
  }
}
```

2. ISBN으로 책 조회 (GET)

http://localhost:8080/api/books/isbn/9780132350884

3. 저자로 책 검색

http://localhost:8080/api/books/search/author?author=Robert

도서 관리 시스템 구현하기 **(1:1** 연관관계 **) -** 부분수정**(Patch)** 기능

주요 개선사항**:**

1. PATCH 메서드 추가

- **`PATCH /api/books/{id}`**: Book의 일부 필드만 수정
- **`PATCH /api/books/{id}/detail`**: BookDetail의 일부 필드만 수정

2. 새로운 **DTO** 클래스

- **`PatchRequest`**: Book 부분 수정용 (모든 필드가 Optional)
- **`BookDetailPatchRequest`**: BookDetail 부분 수정용

3. 부분 업데이트 로직

- null 체크를 통해 제공된 필드만 업데이트
- 제공되지 않은 필드는 기존 값 유지

```
if (request.getTitle() != null) {
        book.setTitle(request.getTitle());
}
```

4. ISBN 체크 로직 설명

```
if (!book.getIsbn().equals(request.getIsbn()) &&
   bookRepository.existsByIsbn(request.getIsbn())) {
   throw new BusinessException(ErrorCode.ISBN_DUPLICATE, request.getIsbn());
}
```

이 로직이 필요한 이유**:**

- 시나리오 **1**: ISBN을 변경하지 않는 경우 → 체크 불필요
- 시나리오 **2**: 새로운 ISBN으로 변경하는데 이미 다른 책이 사용 중 → 오류 발생해야 함
- 시나리오 **3**: 새로운 고유한 ISBN으로 변경 → 정상 처리

4. 수정 **(put) -** 전체수정

http://localhost:8080/api/books/1

```
{
 "title": "Clean Code",
 "author": "Robert C. Martin",
 "isbn": "9780132350884",
 "price": 45,
 "publishDate": "2008-08-01",
 "detailRequest": {
```

```
    "description": "A handbook of agile software craftsmanship",
    "language": "English",
    "pageCount": 464,
    "publisher": "Prentice Hall",
    "coverImageUrl": "https://example.com/cleancode.jpg",
    "edition": "1st"
  }
}
```

4-1. 수정 (patch)

(제목만 수정) PATCH

http://localhost:8080/api/books/1

```
{
  "title": "새로운 제목"
}
```

// 가격과 언어만 수정 PATCH

http://localhost:8080/api/books/1

```
{
  "price": 15000,
  "detailRequest": {
    "language": "Korean"
  }
}
```

// BookDetail의 설명만 수정 PATCH

http://localhost:8080/api/books/1/detail

```
{
  "description": "새로운 책 설명"
}
```