섹션 3. 첫 번째 요구사항 추가하기 - 책의 분야

이번 섹션의 목표
21강. 책의 분야 추가하기
22강. Enum Class를 활용해 책의 분야 리팩토링 하기
23강. Boolean에도 Enum 활용하기 - 책 반납 로직 수정
24강. 첫 번째 요구사항 클리어!

이번 섹션의 목표

1. Type, Status 등을 서버에서 관리하는 방법들을 살펴보고 장단점을 이해한다.
2. Test Fixture의 필요성을 느끼고 구성하는 방법을 알아본다.
3. Kotlin에서 Enum + JPA + Spring Boot를 활용하는 방법을 알아본다.

21강. 책의 분야 추가하기

이번 시간에는 첫 번째 추가 요구사항인, “책 등록 요구사항"에 대해서 알아보고, 가장 간단한 방법으로 구현해보자. 요구사항은 다음과 같다.

책 등록 요구사항 추가

- 책을 등록할 때에 ‘분야'를 선택해야 한다.
  - 분야에는 5가지 분야가 있다 - 컴퓨터 / 경제 / 사회 / 언어 / 과학

이 요구사항을 어떻게 기존 구조에 반영할 수 있을까?! 도메인인 `Book` 객체에 type 필드를 추가하면 된다.

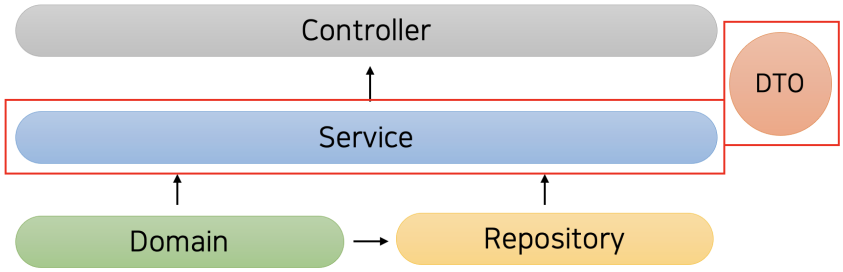
- null이 불가능하므로 `String?` 대신 `String` 타입으로 선언하자.
- 또한, 한 번 분야가 정해진 후 업데이트 하는 요구사항이 없으니 불변 (`val`)으로 선언하자.

```
@Entity
class Book constructor(
  val name: String,

  val type: String, // 추가되었다.

  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  val id: Long? = null,
)
```

좋다~ 이제 Book에 필드가 추가되어 영향 받는 로직들을 하나씩 체크하며 수정해주자.

Domain을 기준으로 상위 계층에 영향 범위가 전파된다고 생각하면 된다.

영향 범위 파악하기




```
// BookRequest.kt
data class BookRequest(
  val name: String,
  val type: String, // 추가되었다.
)
```

```
// BookService.kt
@Transactional
fun saveBook(request: BookRequest) {
  // request.type이 추가되었다.
  bookRepository.save(Book(request.name, request.type))
}
```

필요한 부분을 수정했다! 이것으로 과연 끝일까?!! 안타깝게도 끝이 아니다. 🥲

Book의 생성자는 테스트 코드에서도 사용해주고 있다. 때문에 **테스트 코드 역시 수정이 필요**하다.

테스트 코드를 바로 수정하기 전에, 잠깐 생각을 해보자.

테스트 코드 역시 우리가 유지보수 해야하는 대상이다. 프로덕션 코드에 약간의 변경이 있을 때 테스트 코드까지 수정이 필요하다면 유지보수 비용이 올라갈 것이다.

테스트 수가 지금은 적지만 프로덕트의 규모가 커질 수록 테스트 수의 개수는 수백, 수천개가 되고 이는 무시할 수준이 되지 못한다.

또한, 방금 추가된 Book의 type 같은 경우는 name과 다르게, 도서 대출 기능 / 도서 반납 기능에서 그렇게까지 중요한 의미를 갖지 않는다. 즉, type에 어떤 값이 들어가건 테스트의 성공, 실패에는 영향이 없다는 의미이다.

이런 배경에서, 객체에 필드가 하나 추가되더라도 객체를 사용하는 테스트 코드에 영향이 가지 않으면 좋을 것 같다. 어떻게 처리할 수 있을까?!!

바로 **Book 객체를 만드는 함수를 미리 만들어 두는 것**이다. Book에 companion object를 만들고 fixture라는 이름을 가진 정적 팩토리 메소드를 만들어주자! 이때 default parameter를 활용해주면 더욱 좋다.

companion object 는 Kotlin Coding Convention 상 클래스의 가장 마지막 부분에 위치하게 된다.

```
companion object {
  fun fixture(
    name: String = "책 이름",
    type: String = "COMPUTER",
    id: Long? = null
  ): Book {
    return Book(
      name = name,
      type = type,
      id = id,
    )
  }
}
```

테스트에 이용할 객체를 만드는 함수를 어려운 말로 Object Mother라고 부르고, 이렇게 생겨난 테스트용 객체를 Test Fixture라고 부른다.

좋다~ 이제 테스트 코드를 수정하러 가자.

- `BookServiceTest.kt` 코드에서 `Book()`으로 Book 생성자를 호출했던 부분들을 모두 `Book.fixture()` 으로 변경해주자.
- 그리고 추가된 요구사항에 맞춰, 책 생성 테스트를 변경하자.

```
@Test
@DisplayName("책 등록이 정상 동작한다")
fun saveBookTest() {
  // given
  val request = BookRequest("이상한 나라의 엘리스", "SOCIETY") // 이 부분이 추가되었다.

  // when
  bookService.saveBook(request)

  // then
  val books = bookRepository.findAll()
  assertThat(books).hasSize(1)
  assertThat(books[0].name).isEqualTo("이상한 나라의 엘리스")
  assertThat(books[0].type).isEqualTo("SOCIETY") // 이 부분이 추가 되었다.
}
```

BookRequest 역시 fixture 함수를 만들어도 되지만, DTO의 경우 Entity보다 적게 사용되기 때문에 필요한 경우에만 선택적으로 만들어주는 편이다.

이제 BookServiceTest를 돌려보자. 전체 테스트가 통과했다면,
http://localhost:8080/v2/index.html 에 들어가 정상 동작하는지 확인해보자.
URL 중간 부분에 v1이 아니라, v2가 들어간다!

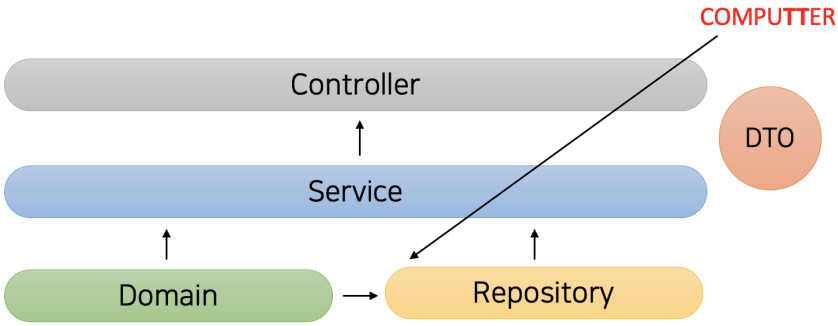
다음 시간에는, 현재 구조의 아쉬운 점을 살펴보고, 아쉬운 점들을 해결하는 방법을 소개한다.

22강. Enum Class를 활용해 책의 분야 리팩토링 하기

이전 시간에 우리는 `val type: String` 을 사용해서 새로운 기능을 추가했다.

하지만 이 방법은 몇 가지 단점이 있다.

1. 클라이언트에서 들어오는 type 필드에 무엇이 들어올지 모른다.
    a. 서버에서 검증해주는 로직을 **추가로 작성할 수 있긴 하지만, 코드를 추가 작성해야 한다는 부담이 있다.**



요청을 검증하고 있지 않다.

2. 코드만 보았을 때 Book 테이블의 type 필드에 어떤 값들이 있는지 알 수 없다.
3. type과 관련한 새로운 로직을 작성해야 할 때 분기 로직이 들어가게 된다.
    a. 예를 들어, 도서관에서 이벤트를 진행하는데 type에 따라 책 대출자에게 '점수'를 부여한다고 하자.
    b. 컴퓨터 : 10점 / 경제 : 8점 / 사회, 언어, 과학 : 5점이다

3번 단점을 Book 코드로 확인하면 다음과 같다.

```
fun getEventScore(): Int {
  return when (type) {
    "COMPUTER" -> 10
    "ECONOMY" -> 8
    "SOCIETY", "LANGUAGE", "SCIENCE" -> 5
    else -> throw IllegalArgumentException("잘못된 타입입니다")
  }
}
```

방금 보았던 단점을 깔끔하게 해결하기 위한 방법은 바로 Enum Class를 활용하는 것이다.

Enum Class를 활용하기 위해 먼저 BookType 이라는 Enum Class를 만들어주자.

```
enum class BookType {
  COMPUTER,
  ECONOMY,
  SOCIETY,
  LANGUAGE,
  SCIENCE,
  ;
}
```

enum class BookType의 위치는 3가지 후보가 있다.

1. Book.kt 파일에 Book class와 함께 두기
2. `com.group.libraryapp.domain.book` 패키지에 두기
3. `com.group.libraryapp.type.book` 패키지에 두기

셋 모두 크게 장단점이 있지는 않아, 팀 간의 컨벤션을 잘 맞추면 된다. 본 강의에서는 2번 방식으로 진행할 예정이다.

이제 `val type: String` 대신 `val type: BookType` 을 사용하자!

```
@Entity
class Book constructor(
  val name: String,

  val type: BookType,

  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  val id: Long? = null,
) {
  // 생략...
}
```

윗 부분을 수정하고 나니, 아랫 부분에서 빨간 줄이 나온다. companion object 안에 있는 fixture 함수이다. 이 함수에서 빨간 줄이 나온 덕분에 바로 수정할 수 있었고, 이 부분을 수정하면 테스트에서 `Book.fixture()` 를 호출하는 모든 부분을 일일이 수정하지 않아도 된다.

```
// fixture 함수
fun fixture(
  name: String = "책 이름",
  type: BookType = BookType.COMPUTER,
  id: Long? = null
): Book {
  return Book(
    name = name,
    type = type,
    id = id,
  )
}
```

이제 다른 코드들도 수정해주자.

- 요청 DTO
- 생성 테스트

를 수정하면 된다.

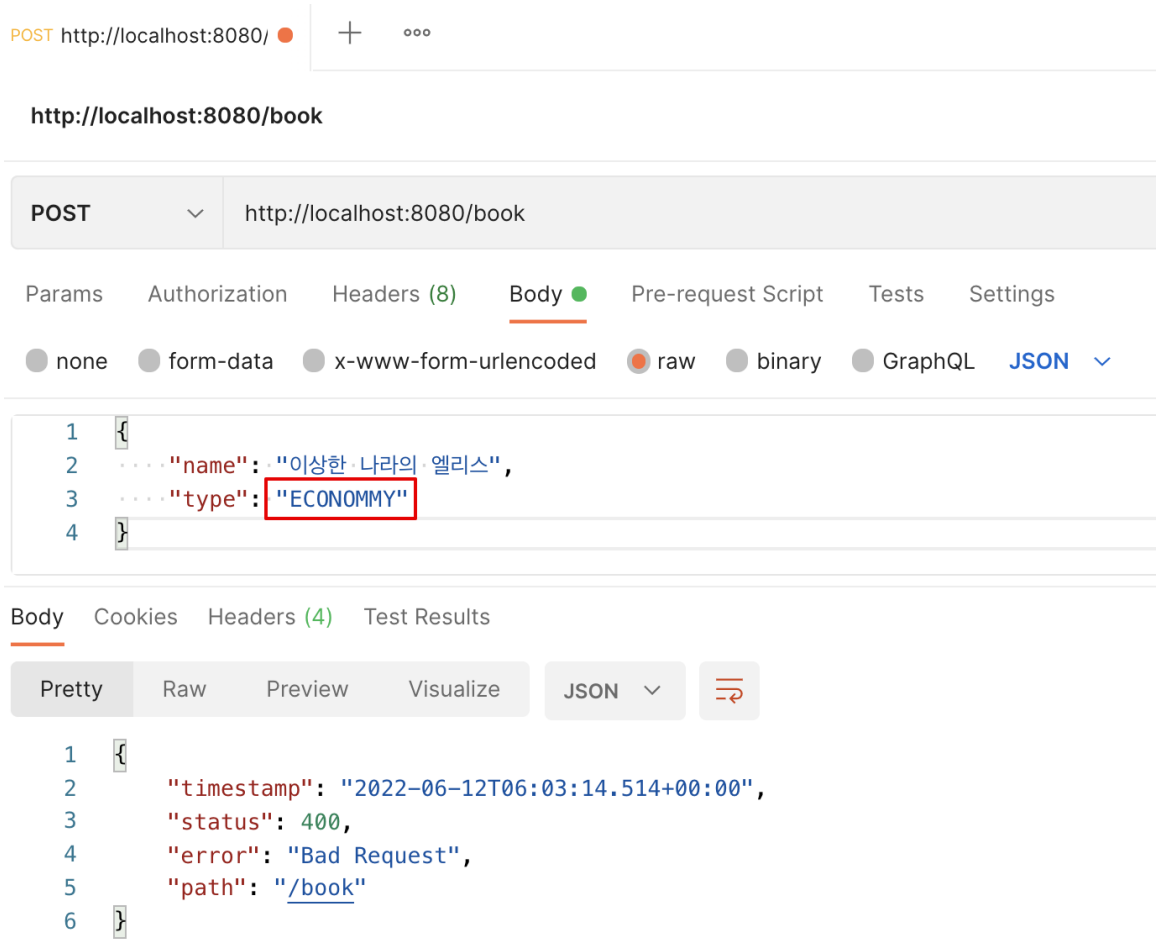
```
data class BookRequest(
  val name: String,
  val type: BookType,
)
```

```
@Test
@DisplayName("책 등록이 정상 동작한다")
fun saveBookTest() {
  // given
  val request = BookRequest("이상한 나라의 엘리스", BookType.SOCIETY) // Enum으로 변경

  // when
  bookService.saveBook(request)

  // then
  val books = bookRepository.findAll()
  assertThat(books).hasSize(1)
  assertThat(books[0].name).isEqualTo("이상한 나라의 엘리스")
  assertThat(books[0].type).isEqualTo(BookType.SOCIETY) // Enum으로 변경
}
```

좋다! 이제 type: String을 사용했을 때의 단점 3가지는 모두 해결되었다. 간단하게 확인해보자.

[기존 문제 1] 클라이언트에서 들어오는 type 필드에 무엇이 들어올지 모른다.

먼저 요청이다. post man과 같은 API 테스트 도구로 존재하지 않는 Enum 값을 넣어 요청을 보내보자.



그러면 위와 같이 막히는 것을 확인할 수 있다. 정상적인 값은 `ECONOMY` 인데 `ECONOMMY` 라고 `M` 을 2번 타이핑 했음에도 서버에서 Enum을 확인해 자동으로 막아준 것이다.

[기존 문제 2] 코드만 보았을 때 Book 테이블의 type 필드에 어떤 값들이 있는지 알 수 없다.

이제 Enum만 보면 바로 알 수 있다. 만약 사회 분야의 책이 다시 등록될 일은 없다면, 다음과 같이 주석을 남겨, 코드의 유지보수를 용이하게 만들 수 있다.

```
enum class BookType {
  COMPUTER,
  ECONOMY,

  // 사회 분야는 2022-12-31 이후로 입고되지 않지만, DB에는 과거 기록이 있다.
  SOCIETY,
  LANGUAGE,
  SCIENCE,
  ;
}
```

[기존 문제 3] type과 관련한 새로운 로직을 작성해야 할 때 분기 로직이 들어가게 된다.

아까 보았던 `getEventScore()` 라는 예시 함수는 다음과 같이 변경할 수 있다.

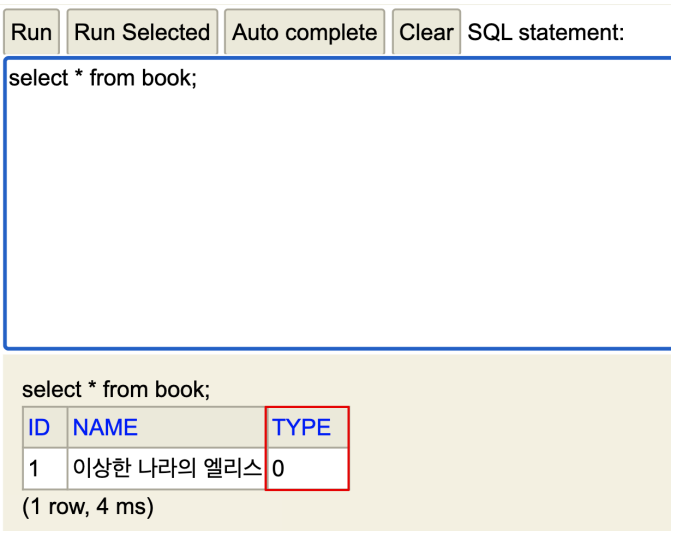
```
enum class BookType(val score: Int) {
  COMPUTER(10),
  ECONOMY(8),
  SOCIETY(5),
  LANGUAGE(5),
  SCIENCE(5)
}
```

위와 같이 Enum에 값을 넣어주고, `getEventScore()` 함수에서 아래와 같이 type의 score를 가져오면 된다.

```
fun getEventScore(): Int {
  return this.type.score
}
```

이렇게 되면, 이전 코드와 다르게 새로운 type이 추가되더라도 `score` 를 빠뜨리지 않고 추가할 수 있게 된다. 이전 코드는 when과 문자열, else의 조합이기 때문에 빠뜨리기 쉬웠다.

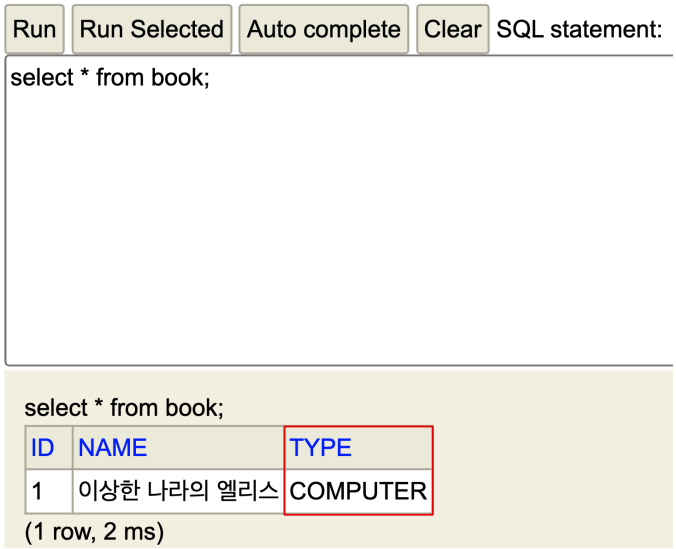
아직 추가로 개선할 부분이 있다!! 🤔 바로 DB에 저장되는 값이다. 현재 DB Book 테이블에 있는 type 필드에는 숫자가 저장된다. 이 숫자는 Enum의 순서에 따라 저장되는 숫자이고, 0부터 시작한다.



이렇게 숫자를 방식은 Enum의 순서가 변경되거나 새로운 Enum type이 추가되었을 때 대응이 어렵다는 단점이 있다. 때문에 `type` 필드에 JPA의 어노테이션인 `@Enumerated(STRING)` 붙여 주어야 한다.

```
@Entity
class Book constructor(
  val name: String,

  @Enumerated(EnumType.STRING) // 이 부분이 추가되었다.
  val type: BookType,

  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  val id: Long? = null,
) {
  // 생략...
}
```

이 어노테이션이 붙으면 DB에 아까와 같은 숫자 대신, Enum에서 지정한 문자열이 들어가게 된다.



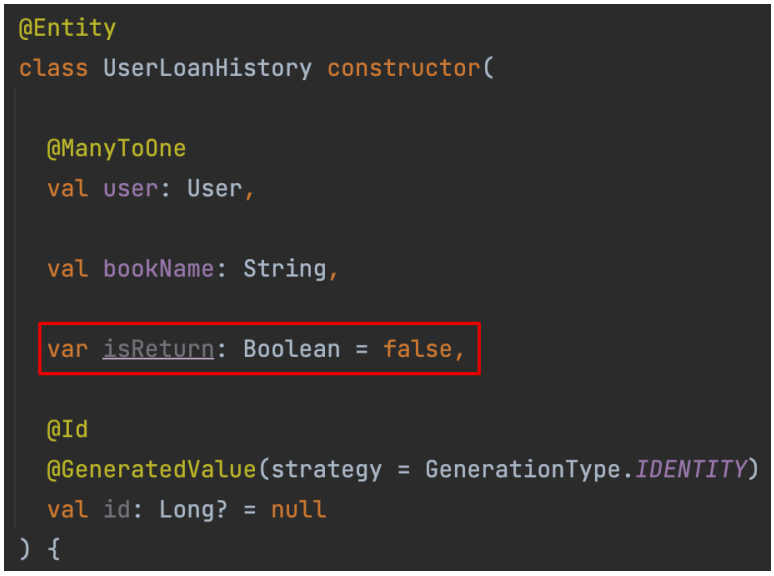
매우 좋다~ 이번 시간에 우리는 책의 유형으로 문자열을 사용했을 때 문제점을 알아보고, Enum을 활용해 해결하였다. 이런 경우처럼 어떤 객체의 상태나 유형 등을 표현할 때는 Enum을 잘 활용하면 좋은 구조를 가져갈 수 있다.

다음 시간에는 다른 테이블에도 Enum을 사용할만한 곳이 있는지 확인해보자.

23강. Boolean에도 Enum 활용하기 - 책 반납 로직 수정

이번 시간에는 현재 존재하는 테이블에서도 Enum을 활용할 수 있는지 확인해보고, 개선해 보자.

개인적으로는 UserLoanHistory에 있는 이 Boolean을 Enum으로 바꿀 수 있다고 생각한다.



```
@Entity
class UserLoanHistory constructor(

  @ManyToOne
  val user: User,

  val bookName: String,

  var isReturn: Boolean = false,

  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  val id: Long? = null
) {
```

그 이유는 다음과 같다.

예를 들어, 유저 테이블이 있다고 해보자. 이 유저 테이블은 원래 이름만 존재했다. 그런데 어느날 휴면 여부를 관리해야 한다고 한다.

이때 가장 먼저 떠오르는 방법은 `isActive` 와 같은 `Boolean` 타입의 필드를 추가하는 것이다.

```
@Entity
class User(
  val name: String,

  val isActive: Boolean,

  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  val id: Long? = null,
)
```

`isActive` 가 true이면 이 유저는 활성화된 - 즉 휴면이 아닌 유저이고, false이면 비활성화된 유저이다.

이런 Boolean 필드를 FLAG라고도 한다.

여기까진 괜찮다. 자 그런데 한 달이 지나자 '유저의 탈퇴 여부도 soft 하게 기록해주세요~'라는 추가적인 요구사항이 들어온다. 탈퇴의 경우는, 휴면을 해제하고 로그인을 해야 가능하다고 한다. 여기서 말하는 soft delete란 실제 DB에서 완전히 삭제하는 hard delete와 다르게 DB에는 존재하지만 삭제된 것처럼 간주해야 하는 경우를 의미한다.

이 상황에서 `isDeleted` 라는 Boolean을 추가로 사용했다고 해보자. 그러면 테이블은 다음과 같이 변하게 된다.

```
@Entity
class User(
  val name: String,

  val isActive: Boolean,

  val isDeleted: Boolean,

  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  val id: Long? = null,
)
```

이렇게 되면, 문제가 발생한다.

[문제 1] Boolean이 2개 존재함으로써 코드의 이해가 어려워진다.

- 한 객체가 여러 상태를 표현할 수록 코드를 이해하기가 어려워지는데, 현재는 2^2 이므로 4가지나 된다.
- 지금은 그나마 4가지 경우이지만, Boolean이 1개 더 늘어나면 8가지 경우의 수 / 2개 더 늘어나게 된다면 16가지 경우의 수가 된다.

[문제 2] Boolean 2개로 표현할 수 있는 4가지 상태가 모두 유의미하지 않다.

- (isActive, isDeleted)는 총 4가지 경우가 있고 각 의미는 다음과 같다.
  - (false, false) - 휴면 상태인 유저이다.
  - (false, true) - 휴면이면서 탈퇴한 유저일 수는 없다. 사실상 DB에는 존재할 수 없는 조합이다.

    - (true, false) - 활성화된 유저이다.
    - (true, true) - 탈퇴한 유저이다.
- 여기서 2번째 경우는 DB에 존재할 수 없는 조합이고, 이런 경우가 '코드적으로' 가능한 것은 유지보수를 어렵게 만든다.
- 만약 Boolean이 더 늘어나면 유의미하지 않은 조합이 더 늘어날 것이다.

이런 문제는 Enum을 도입함으로써 해결할 수 있다. 다시 처음으로 돌아가, 휴면 유저를 구분하는 기능을 구현할 때 Enum을 사용했다고 해보자

```
@Entity
class User(
  val name: String,

  @Enumerated(STRING)
  val status: UserStatus,

  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  val id: Long? = null,
)
```

```
enum class UserStatus {
  ACTIVE,
  IN_ACTIVE,
}
```

이후 탈퇴 유저에 대한 요구사항은 다음과 같이 구현된다.

```
enum class UserStatus {
  ACTIVE,
  IN_ACTIVE,
  DELETED
}
```

이렇게 Enum을 사용하면

1. 필드 1개로 여러 상태를 표현할 수 있기 때문에 코드의 이해가 쉬워지고
2. 정확하게 유의미한 상태만 나타낼 수 있기 때문에 코드의 유지보수가 용이해진다.

즉 앞서 살펴보았던 2가지 문제가 모두 해결되는 것이다.

UserLoanHistory 역시 위의 예시와 비슷하게 Enum을 활용하면 추후 확장이 용이할 것이다.

현재는 유저가 특정한 책을 단순대출 / 반납 하지만, 나중에는 장기대출, 임시 반납 등 여러 상태가 생길 수 있기 때문이다. (물론 본 강의에서 다루지는 않을 것이지만, 실제 서비스라면 그렇게 흘러갈 확률이 높다! 🏹)

예상되므로 연습하는 차원에서 Enum으로 변경해보자!

다음과 같은 코드들이 변경되어야 한다!

- 새로운 Enum 생성
- UserLoanHistory 변경
- UserLoanHistory를 변경했을 때 빨간 줄이 나오는 영역 (테스트 코드)

```
enum class UserLoanStatus {

  RETURNED, // 반납 완료
  LOANED, // 대출 중인 상태
  ;

}
```

```
@Entity
class UserLoanHistory constructor(

  @ManyToOne
  val user: User,

  val bookName: String,

  @Enumerated(EnumType.STRING)
  var status: UserLoanStatus = UserLoanStatus.LOANED,

  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  val id: Long? = null
) {

  fun doReturn() {
    this.status = UserLoanStatus.RETURNED
  }

}
```

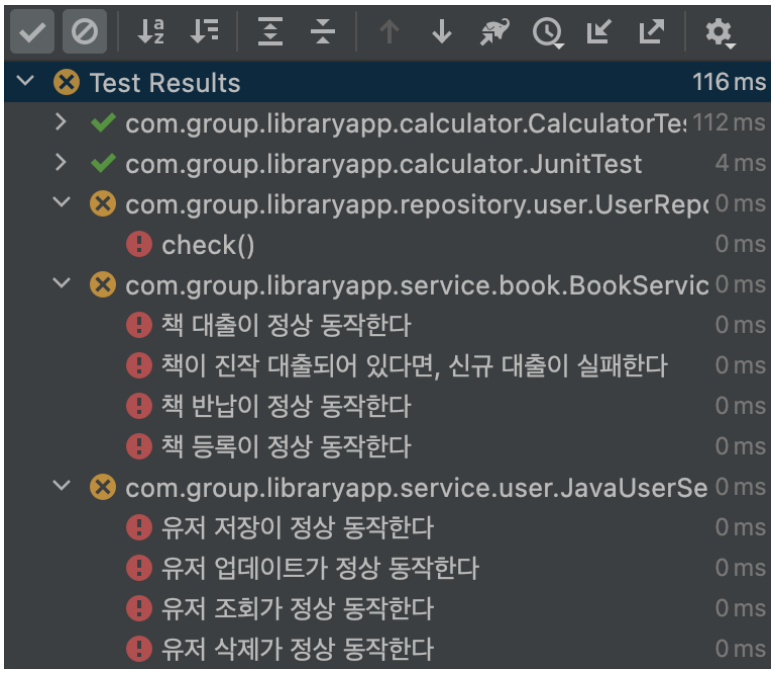
UserLoanHistory는 아직 Fixture가 없어 Test 쪽도 변경해주어야 한다. fixture를 활용해보자.

아래와 같이 fixture 메소드를 만들어 주고, 테스트 코드도 변경해보았다.

```
companion object {
  fun fixture(
    user: User,
    bookName: String = "이상한 나라의 엘리스",
    status: UserLoanStatus = UserLoanStatus.LOANED,
  ): UserLoanHistory {
    return UserLoanHistory(
      user = user,
      bookName = bookName,
      status = status,
    )
  }
}
```

테스트 코드의 경우는 Fixture를 사용하는 부분, 그리고 코드를 검증하는 부분 모두 수정해주어야 한다. 자 이제 모든 빨간줄이 제거되었으니 테스트를 돌려보자!!

테스트를 돌려보니 Spring Context를 이용하는 모든 테스트가 실패한다!!



에러를 읽어보자.

```
Caused by: org.springframework.data.mapping.PropertyReferenceException: No property 'i
sReturn' found for type 'UserLoanHistory'!
```

그렇다. **사실은 Repository 메소드 이름과 그 메소드를 사용하는 Service도 수정해 주어야 한다**. IDE에서 빨간 줄이 나오지 않았지만, Domain 필드의 이름이 변경되면서 context가 뜰 수 없게 되는 것이다.

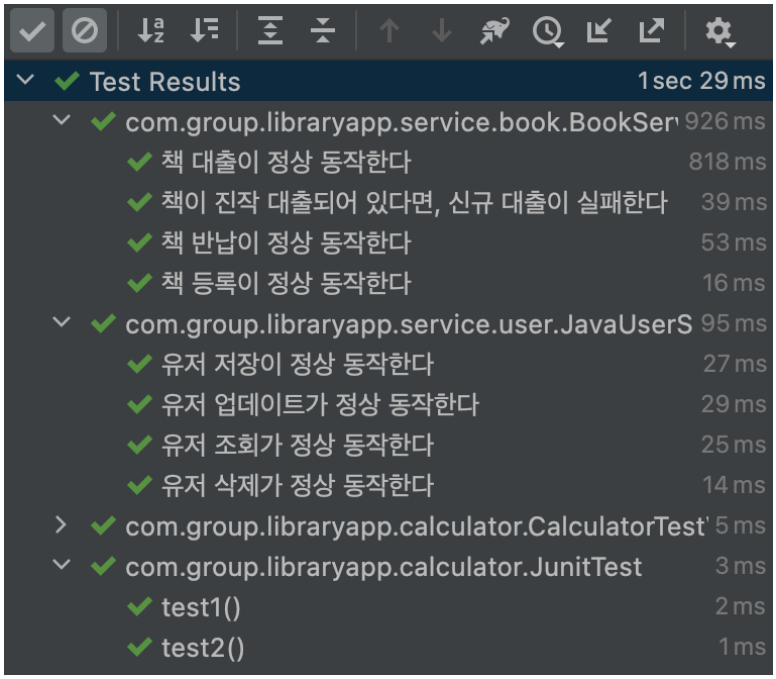
이것이 Spring Data JPA 혹은 JPQL을 사용하는 것의 단점인데 이 문제점은 <Section 6>에서 querydsl을 적용하여 해결할 것이다.

Repository와 Service 쪽도 다음과 같이 수정해주자.

```
interface UserLoanHistoryRepository : JpaRepository<UserLoanHistory, Long> {

  fun findByBookNameAndStatus(bookName: String, status: UserLoanStatus): UserLoanHisto
ry?

}
```

```
@Transactional
  fun loanBook(request: BookLoanRequest) {
    if (userLoanHistoryRepository.findByBookNameAndStatus(request.bookName, UserLoanSt
atus.LOANED) != null) {
      throw IllegalArgumentException("진작 대출되어 있는 책입니다")
    }

    val book = bookRepository.findByName(request.bookName) ?: fail()
    val user = userRepository.findByName(request.userName) ?: fail()
    user.loanBook(book)
  }
```

이제 다시 한 번 테스트를 돌려보자.



좋다~ 🎉🎉 드디어 전체 테스트가 잘 동작한다!! 우리는 테스트 덕분에 내부 구조를 더 좋은 구조로 변경했음에도, 안정성을 보장할 수 있다.

이제 다음 시간에는 첫 번째 요구사항에 대해 간략하게 정리해보도록 하자.

24강. 첫 번째 요구사항 클리어!

우리는 아래와 같은 첫 번째 요구사항을 완벽하게 구현하였다. 🎉

책 등록 요구사항 추가

- 책을 등록할 때에 '분야'를 선택해야 한다.
  - 분야에는 5가지 분야가 있다 - 컴퓨터 / 경제 / 사회 / 언어 / 과학

덕분에 이번 `Section 3. 첫 번째 요구사항 추가하기 - 책의 분야` 를 통해 다음과 같은 내용을 배울 수 있었다.

1. Type, Status 등을 서버에서 관리하는 방법
   a. Boolean, Enum의 장단점
2. Test Fixture의 필요성과 구성 방법
3. Kotlin에서 Enum + JPA + Spring Boot를 활용할 수 있는 방법

이제 다음 추가 기능을 구현하러 가보자!! 🏃‍♀️🔥