



섹션 5. 책 요구사항 구현하기

이번 섹션의 목표

30강. 책 생성 API 개발하기

31강. 대출 기능 개발하기

32강. 반납 기능 개발하기

33강. 조금 더 객체지향적으로 개발할 수 있을까?

34강. JPA 연관관계에 대한 추가적인 기능들

35강. 책 대출/반납 기능 리팩토링과 지연 로딩

36강. Section 5 정리. 다음으로!

이번 섹션의 목표

1. 책 생성, 대출, 반납 API를 온전히 개발하며 지금까지 다루었던 모든 개념을 실습해 본다.
2. 객체지향적으로 설계하기 위한 연관관계를 이해하고, 연관관계의 다양한 옵션에 대해 이해한다.
3. JPA에서 연관관계를 매핑하는 방법을 이해하고, 연관관계를 사용해 개발할 때와 사용하지 않고 개발할 때의 차이점을 이해한다.

30강. 책 생성 API 개발하기

이번 시간에는 책 생성 API를 개발해 보자!! API 개발을 위해서는 지금까지 배웠던 스프링 빈과 계층형 아키텍처, JPA, 트랜잭션까지 모든 개념이 총동원된다! 😊 API 스펙까지만 함께 확인하고, 강의를 잠시 멈춰 먼저 개발해 보는 것도 정말 좋을 것 같다.

먼저 요구사항을 살펴보자.

- 도서관에 책을 등록할 수 있다.

다음으로 API 스펙을 확인해 보자.

- HTTP Method : POST

- HTTP Path : /book
- HTTP Body (JSON)

```
{
  "name": String // 책 이름
}
```

- 결과 반환 X (HTTP 상태 200 OK이면 충분하다)

매우 좋다~! 👍 머릿속에 해야 할 일이 떠오른다! `book` 테이블을 설계하고, `Book` 객체를 만들고, Repository, Service, Controller, DTO를 만들어 주면 된다. 꼭 이 순서로 진행해야 하는 것은 아니다. 작업하다 보면 익숙한 순서가 생길 것이다.

자 그럼 `book` 테이블을 고민해 보자. 책 정보는 현재 이름 하나를 가지고 있다. 그럼 다음과 같이 단순하게 테이블을 만들 수 있을 것이다.

```
create table book(
  id bigint auto_increment,
  name varchar(255),
  primary key (id)
);
```

책 이름은 최대 255자까지 들어갈 수 있도록 했다. 1) JPA를 사용할 때 `@Column`의 length 기본값이 255이기도 하고 2) 문자열 필드는 최적화를 해야 하는 경우가 아니면 조금 여유롭게 설정하는 것이 확장성이 좋아 이렇게 처리하였다.

다음으로 이 테이블과 호환되는 객체를 만들어보자! 패키지는 `book` 패키지에 계속해서 넣어 주도록 하겠다!

```
@Entity
public class Book {

  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id = null;

  @Column(nullable = false)
  private String name;
```

```
}
```

매우 좋다! 작업했던 User 객체와 비슷하게 처리하였다. 이어서 `BookRepository` 도 만들어주자.

```
public interface BookRepository extends JpaRepository<Book, Long> {  
  
}
```

깔끔하다~!! 😊

이제 `BookCreateRequest` DTO를 만들어주자! 이름만 받으면 된다.

```
public class BookCreateRequest {  
  
    private String name;  
  
    public String getName() {  
        return name;  
    }  
  
}
```

다음으로는 Controller와 Service이다. 주어진 API 스펙에 잘 맞추어 처리하면 된다! 👍

```
@RestController  
public class BookController {  
  
    private final BookService bookService;  
  
    public BookController(BookService bookService) {  
        this.bookService = bookService;  
    }  
  
    @PostMapping("/book")  
    public void saveBook(@RequestBody BookCreateRequest request) {  
        bookService.saveBook(request);  
    }  
  
}
```

```
@Service  
public class BookService {
```

```

private final BookRepository bookRepository;

public BookService(BookRepository bookRepository) {
    this.bookRepository = bookRepository;
}

@Transactional
public void saveBook(BookCreateRequest request) {
    bookRepository.save(new Book(request.getName()));
}
}

```

이 과정에서 필요한 Book의 생성자가 자연스럽게 생기게 된다!

```

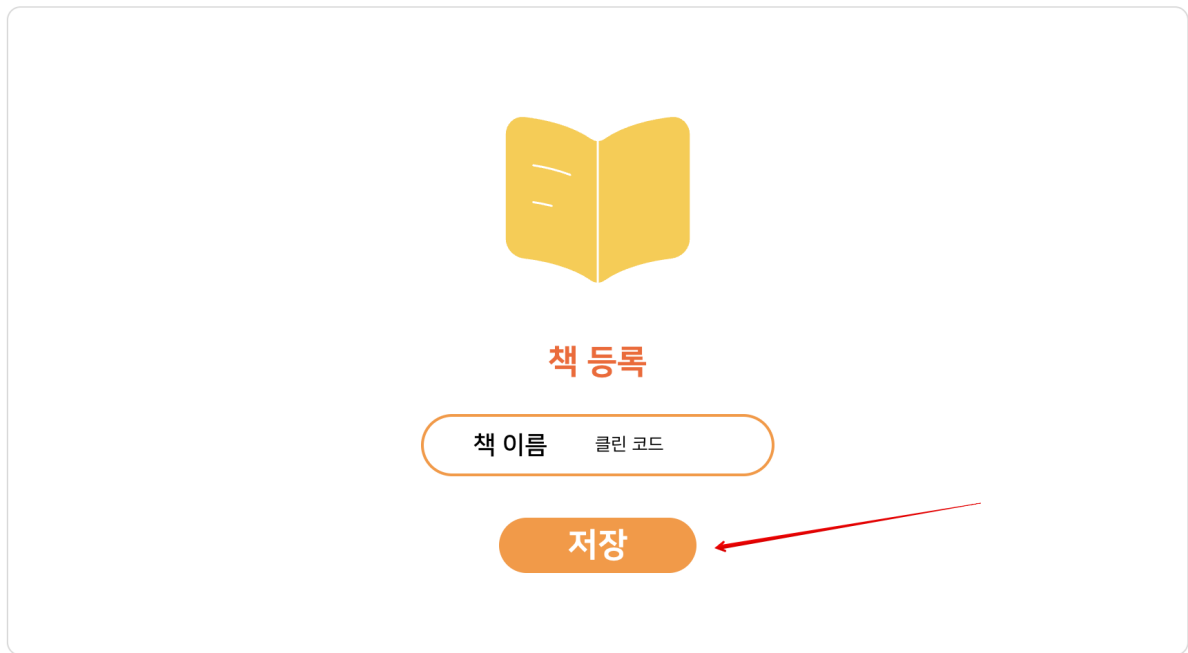
// Book.java 안에 추가된 로직
protected Book() { }

public Book(String name) {
    if (name == null || name.isBlank()) {
        throw new IllegalArgumentException(String.format("잘못된 name(%s)이 들어왔습니다", name));
    }
    this.name = name;
}
}

```

정말 좋다~!! 이제 간단한 API 정도는 쉽게 만들 수 있다! 그럼 우리가 작성한 기능이 잘 만들어졌는지 테스트를 해보도록 하자.

서버를 시작 혹은 재시작한 다음, <http://localhost:8080/v1/index.html>에 들어가 웹 UI를 열고 책을 저장해 보자! 그다음 DB에 들어가 저장이 잘 되었는지 확인하면 된다!



A UI mockup for a book registration form. At the top is a yellow book icon. Below it is the title '책 등록' in red. There are two input fields: '책 이름' (Book Name) and '클린 코드' (Clean Code). Below the inputs is an orange '저장' (Save) button, which is pointed to by a red arrow from the right.

	id	name
1	1	클린 코드

매우 좋다~!! 😊 이제 다음 시간에는 이어서 대출 기능을 구현해 보자! 🔥

31강. 대출 기능 개발하기

이번 시간에는 대출 기능을 구현할 것이다. 요구사항과 API 스펙을 살펴보면 다음과 같다.

요구사항

- 사용자가 책을 빌릴 수 있다.
 - 다른 사람이 그 책을 진작 빌렸다면 빌릴 수 없다.

API 스펙

- HTTP Method : POST
- HTTP Path : /book/loan
- HTTP Body (JSON)

```
{
  "userName": String
  "bookName": String
}
```

- 결과 반환 X (HTTP 상태 200 OK이면 충분하다)

요구사항을 살펴보면, 지금 우리가 가지고 있는 `user`, `book` 2개의 테이블 만으로는 유저의 대출 기능을 만들 수 없다. 새로운 테이블이 필요하다는 의미이다!

어떤 유저가 어떤 책을 빌렸는지 반납했는지 확인할 수 있도록 추가적인 테이블을 구성하자.

테이블의 이름은 '유저의 대출 기록'이라는 뜻에서 `user_loan_history` 로 하면 될 것 같다! 이 테이블에는 어떤 필드들이 들어가야 할까?!

총 4가지 필드가 필요하다.

```
create table user_loan_history (
  id bigint auto_increment,
  user_id bigint,
  book_name varchar(255),
  is_return tinyint(1),
  primary key (id)
)
```

- `id` : `user` 테이블이나 `book` 테이블에도 존재하는 primary key, id이다.
- `user_id` : 어떤 유저가 빌렸는지 알 수 있도록, 유저의 id를 가지고 있도록 했다.
- `book_name` : 유저가 어떤 책을 빌렸는지 알 수 있도록 책의 이름을 가지고 있도록 했다.
- `is_return` : 유저가 빌린 책을 대출 중인지, 반납 완료했는지 확인하는 필드이다. 이 필드에 0이 들어가 있으면 대출 중인 것이고, 1이 들어가 있으면 반납한 것이다.

예를 하나 들어보자. 다음과 같은 데이터가 들어 있다.

user		user_loan_history			
id	name	id	user_id	book_name	is_return
1	A	1	2	클린 코드	1
2	B	2	2	테스트 주도 개발	0

이 데이터는 이렇게 해석할 수 있다.

2번 유저는 2권의 책을 빌렸다. '클린 코드'는 반납했고, '테스트 주도 개발'은 아직 대출 중이다.

이제 이 테이블에 대응되는 객체를 만들어보자! 패키지는 `user` 아래에 `loanhistory` 패키지를 만들어주자!

```
@Entity
public class UserLoanHistory {

    @Id
    @GeneratedValue(strategy = IDENTITY)
    private Long id;

    private long userId;

    private String bookName;

    private boolean isReturn;

}
```

매우 좋다!!! 👍 `is_return` 필드의 경우, tinyint가 들어 있는데 이를 boolean에 매핑하게 되면 true인 경우 1이, false인 경우 0이 저장된다.

이어서 `UserLoanHistoryRepository` 도 만들자!

```
public interface UserLoanHistoryRepository extends JpaRepository<UserLoanHistory, Long> {

}
```

다음으로, DTO와 Controller를 만들어주자! 도서 대출 역시 책 관련 기능이니까 Controller를 새로 만들지 않고, `BookController`에 넣어주도록 하겠다! DTO 역시 책 관련이니

```
// DTO
public class BookLoanRequest {

    private String userName;
    private String bookName;

    public String getUserName() {
        return userName;
    }

    public String getBookName() {
```

```

        return bookName;
    }
}

```

```

// Controller (BookController.java)
@PostMapping("/book/loan")
public void loanBook(@RequestBody BookLoanRequest request) {
    bookService.loanBook(request);
}

```

이제 대망의 Service 로직이다!! 🔥 Controller와 비슷하게, 책 관련 기능이니 **BookService**에 로직을 함께 작성해 주자.

```

@Transactional
public void loanBook(BookLoanRequest request) {

}

```

이제 이 서비스 로직에서 비즈니스 로직을 작성해 주어야 한다!

우선은 책 객체를 이름을 통해 가져오자. 만약 책이 없는 경우에는 예외를 던져주어야 한다. 이름을 기준으로 책을 가져오려면, **BookRepository**에 메소드 시그니처 작성도 필요하다.

```

// Repository
public interface BookRepository extends JpaRepository<Book, Long> {

    Optional<Book> findByName(String bookName);

}

// Service
@Transactional
public void loanBook(BookLoanRequest request) {
    Book book = bookRepository.findByName(request.getBookName())
        .orElseThrow(IllegalArgumentException::new);
}

```

Book 객체를 가져왔다면, DB에 존재하는 책이니 이제 이 책을 누군가 대출 중인지 확인하자. 이번에는 **UserLoanHistoryRepository**에 메소드 시그니처 작성이 필요하다.

```

public interface UserLoanHistoryRepository extends JpaRepository<UserLoanHistory, Long> {
}

```



```

    boolean existsByBookNameAndIsReturn(String bookName, boolean isReturn);
}

```

`existsByBookNameAndIsReturn`에 주어진 책 이름과 `false`를 넣은 값이 `true`가 나왔다는 의미는 현재 반납되지 않은 대출 기록이 있다는 의미이니 누군가 대출했다는 의미이다.

따라서 Service는 다음과 같이 변경된다.

```

@Service
public class BookService {

    private final BookRepository bookRepository;
    // UserLoanHistoryRepository에 접근해야 하니 의존성을 추가해주었다!
    private final UserLoanHistoryRepository userLoanHistoryRepository;

    // 생성자에서 스프링 컨테이너를 통해 주입받도록 하였다.
    public BookService(BookRepository bookRepository, UserLoanHistoryRepository userLoanHistoryRepository) {
        this.bookRepository = bookRepository;
        this.userLoanHistoryRepository = userLoanHistoryRepository;
    }

    // 저장 로직 생략

    @Transactional
    public void loanBook(BookLoanRequest request) {
        Book book = bookRepository.findByName(request.getBookName())
            .orElseThrow(IllegalArgumentException::new);

        // 추가된 로직, user_loan_history를 확인해 예외를 던져준다.
        if (userLoanHistoryRepository.existsByBookNameAndIsReturn(book.getName(), false))
        {
            throw new IllegalArgumentException("진작 대출되어 있는 책입니다");
        }
    }
}

```

if 문이 실행되지 않았다면, 대출되지 않은 책이라는 뜻이니 이제 대출 기록을 쌓아주면 된다. 이때 `userId`가 필요하니, 유저 객체를 가져온 후 `UserLoanHistory`를 저장해 주자.

`UserRepository`에 대한 의존성도 새로 필요하고, `UserRepository`의 로직도 변경이 필요하며, `UserLoanHistory`에 새로운 생성자도 필요하다! 최종적인 Service 코드는 다음과 같다.

```

@Service
public class BookService {

```

```

private final BookRepository bookRepository;
private final UserLoanHistoryRepository userLoanHistoryRepository;
private final UserRepository userRepository;

public BookService(
    BookRepository bookRepository,
    UserLoanHistoryRepository userLoanHistoryRepository,
    UserRepository userRepository
) {
    this.bookRepository = bookRepository;
    this.userLoanHistoryRepository = userLoanHistoryRepository;
    this.userRepository = userRepository;
}

// 저장 로직 생략

@Transactional
public void loanBook(BookLoanRequest request) {
    Book book = bookRepository.findByName(request.getBookName())
        .orElseThrow(IllegalArgumentException::new);

    if (userLoanHistoryRepository.existsByBookNameAndIsReturn(book.getName(), false))
    {
        throw new IllegalArgumentException("진작 대출되어 있는 책입니다");
    }

    User user = userRepository.findByName(request.getUserName())
        .orElseThrow(IllegalArgumentException::new);
    userLoanHistoryRepository.save(new UserLoanHistory(user.getId(), book.getName()));
}
}

```

정말 좋다~!! 😊 이제 웹 UI와 DB를 열어 기능이 잘 동작하는지 확인해 보면 끝이다!

다음 시간에는 마지막 요구사항인 반납 기능을 개발할 것이다.

32강. 반납 기능 개발하기

이번 시간에는~ 드디어 <도서 관리 애플리케이션>의 마지막 기능! 반납 기능을 구현할 것이다. 요구사항과 API 스펙을 빠르게 살펴보자!

요구사항

- 사용자가 책을 반납할 수 있다.

API 스펙

- HTTP Method : PUT

- HTTP Path : /book/return
- HTTP Body (JSON)

```
{
  "userName": String
  "bookName": String
}
```

- 결과 반환 X (HTTP 상태 200 OK이면 충분하다)

이 요구사항은 현재 존재하는 `user` `book` `user_loan_history` 3개의 테이블로 충분하다! 그런데 한 가지 고민이 생긴다. 대출 기능과 HTTP Body 스펙이 완전히 동일한데, 귀찮더라도 DTO를 새로 만들어야 할까, 아니면 함께 사용하는 것이 좋을까?!

명확한 정답이 있는 문제는 아니지만 개인적으로는 새로 만드는 것을 선호한다. 그래야 두 기능 중 한 기능에 변화가 생길 때, 더욱 유연하고 side-effect가 없이 대처할 수 있기 때문이다!

자 그러면 DTO와 Controller부터 바로 한 번 작업해 보자!

```
public class BookReturnRequest {

    private String userName;
    private String bookName;

    public String getUserName() {
        return userName;
    }

    public String getBookName() {
        return bookName;
    }

}
```

```
@PutMapping("/book/return")
public void returnBook(@RequestBody BookReturnRequest request) {
    bookService.returnBook(request);
}
```

다음은 Service 로직이다!

```
@Transactional
public void returnBook(BookReturnRequest request) {
    User user = userRepository.findByName(request.getUserName())
        .orElseThrow(IllegalArgumentException::new);
    UserLoanHistory history = userLoanHistoryRepository.findByUserIdAndBookName(user.getId(), request.getBookName())
        .orElseThrow(IllegalArgumentException::new);

    history.doReturn();
}
```

지금까지 다루었던 개념들을 모두 활용해 구현하였다. 도메인 객체와 Repository 역시 구현되어 있던 덕분에 조금의 코드만으로도 용이하게 기능 추가를 성공했다!! 😊

이제 마지막으로 웹 UI와 DB를 열어 잘 동작하는지 확인하자!

자 이렇게 모든 개발은 끝이 났다!! 하지만, 한 가지 고민할 만한 내용이 추가로 남아 있다.

23강에서, SQL 대신 ORM을 사용하게 된 이유 중 하나가 “데이터베이스의 테이블과 객체는 패러다임이 다르기 때문”이라고 언급했다. 조금 더 풀어 해석해 보자.

데이터베이스 테이블에 데이터를 저장하는 것은 필수적이다. 10강에서 살펴보았던 것처럼, 서버의 실행 / 종료와는 무관하게 데이터를 어딘가에 저장해야 하기 때문이다.

하지만 우리가 사용하는 Java 언어는 객체지향형 언어이고, 대규모 웹 애플리케이션을 다룰 때에도 절차 지향적인 설계보다는 객체지향적인 설계를 선호하게 된다. 사실 20강에서 살펴보았던 스프링 컨테이너를 사용하는 이유도, 보다 객체지향적인 설계를 하기 위한 맥락에서 출발했다.

그렇다면 이런 고민이 생긴다! 지금 코드를 조금 더 객체지향적으로 만들 수 없을까?! `User`와 `UserLoanHistory`가 직접 협업할 수 있게 처리하면 안 될까?!

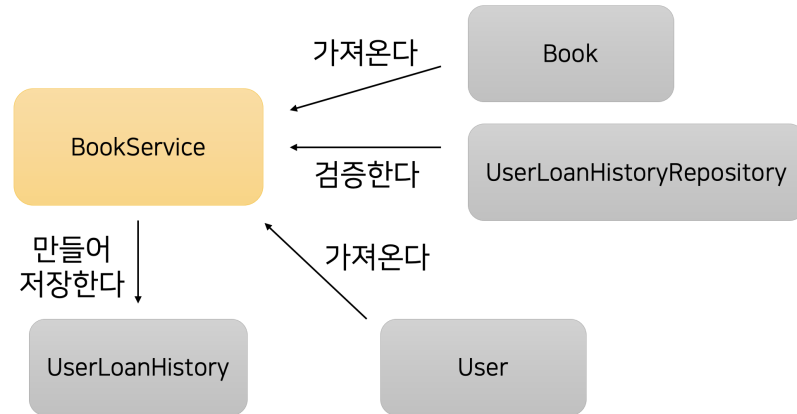
다음 시간에 그 방법을 알아보자!

33강. 조금 더 객체지향적으로 개발할 수 없을까?

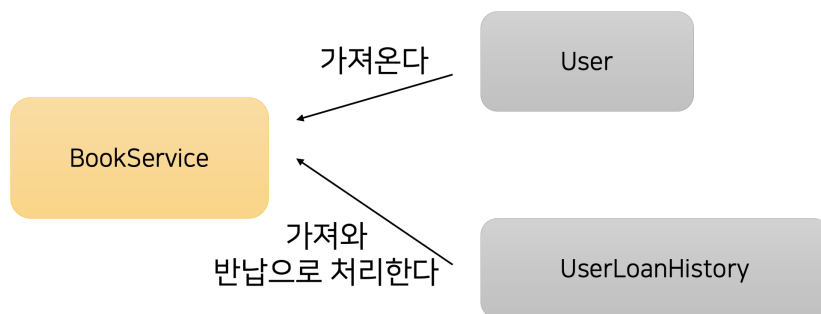
이번 시간에는 기존의 코드를 JPA 연관관계를 활용하는 방식으로 변경해 보자.

우리가 만들어 둔 테이블은 그대로 두고, 도메인 객체인 `User` 와 `UserLoanHistory` 만 변경해서, 이 둘이 직접 협업할 수 있게 변경할 것이다.

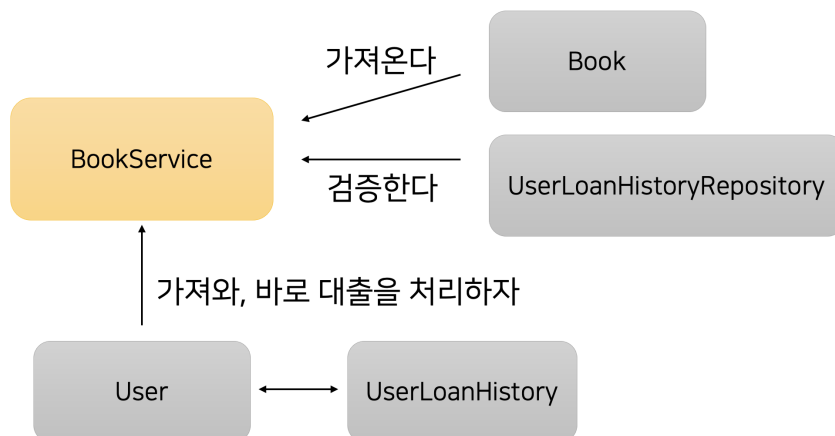
우선 현재의 대출 기능 관계를 살펴보자. 그림으로 살펴보면 다음과 같다.



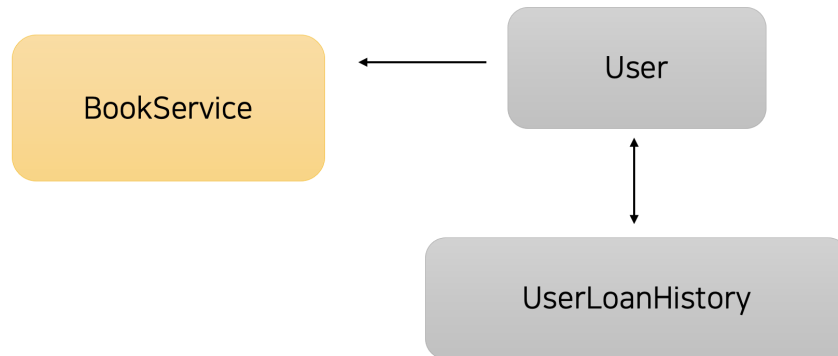
다음으로 반납 기능은 다음과 같다.



우리는 이런 이미지를 다음과 같이 바꾸고 싶다.



가져와 반납으로 처리한다



자 그렇다면, `UserLoanHistory` 와 `User` 가 서로를 직접 알고 있어야 한다.

먼저 `UserLoanHistory` 의 `userId` 를 `User` 로 변경해보자

```
@Entity
public class UserLoanHistory {

    private User user; // private long userId 였다!

    // 생략...

    public UserLoanHistory(User user, String bookName) {
        this.user = user; // User를 받아서 넣어준다
        this.bookName = bookName;
        this.isReturn = true;
    }

}
```

자 그랬더니, `User` 부분에서 빨간 줄이 뜬다. 당연하다! JPA는 `UserLoanHistory` 객체와 `user_loan_history` 테이블을 서로 매핑하려 하는데, `User` 필드는 어디에 매핑해야 할지 모르기 때문이다.

이때 사용해야 하는 어노테이션이 있다. 바로, `@ManyToOne` 이다!

```
@ManyToOne
private User user;
```

`@ManyToOne`의 의미는 내가 다수이고, 네가 1개라는 의미이다. 즉, `UserLoanHistory`가 다수이고 `User`가 1개라는 뜻이다. 이것을 N : 1 관계라고 한다.

교실과 학생의 예를 들어보자. 1개의 교실에는 여러 명(=N명)의 학생이 들어갈 수 있다. 하지만 한 학생은 동시에 여러 교실에 들어가지 못한다.

학생 입장에서는 학생이 다수이고, 교실이 1개라는 의미이다. 즉 N : 1 관계가 된다.

비슷하게 도서 대출 기록 역시, 1명의 유저 입장에서 여러 책을 빌리거나 반납할 수 있기 때문에 대출 기록 : 유저는 N : 1의 관계가 된다.

이를 표현하기 위해 `@ManyToOne` 어노테이션을 붙여주었다.

좋다~! 😊 여기까지는 어렵지 않다. 다음으로 `User` 쪽에서 `UserLoanHistory`를 알고 있게 처리해 주자.

1명의 유저는 N개의 `UserLoanHistory`를 가지고 있을 수 있기 때문에, `UserLoanHistory`를 List로 들고 있어야 한다.

```
@Entity
public class User {

    // 생략 ...

    private List<UserLoanHistory> userLoanHistories = new ArrayList<>();

    // 생략...
}
```

이때도 마찬가지로 빨간 줄이 나오는데, 이를 해결하기 위해서는 `@ManyToOne`의 반대인, `@OneToMany`를 붙여주어야 한다!!

```
@OneToMany
private List<UserLoanHistory> userLoanHistories = new ArrayList<>();
```

`User` 입장에서는 `UserLoanHistory`와의 관계가 1 : N이기 때문이다.

자 다음으로 Service에서 나오는 빨간 줄을 해결해 줄 건데, 그전에 딱 한 가지 처리해 주어야 할 게 있다!


바로 연관관계의 주인이다. 아니 연관관계의 주인이라니, 말이 참 어렵다!

하지만 이 연관관계의 주인은 우리가 알고 있는 개념이다. Table을 바라보았을 때 누가 관계의 주도권을 가지고 있는지를 의미한다.

주인!

```
create table user
(
    id    bigint auto_increment,
    name  varchar(25),
    age   int,
    primary key (id)
);
```

```
create table user_loan_history
(
    id          bigint auto_increment,
    user_id     bigint,
    book_name   varchar(255),
    is_return   tinyint(1),
    primary key (id)
);
```



현재 `user` 테이블과 `user_loan_history` 테이블을 보면, `user_loan_history`는 `user`를 알고 있다. 반면 `user`는 `user_loan_history`를 알지 못한다! 관계의 주도권을 `user_loan_history`가 가지고 있는 것이다!!! 🏰

이 사실을 JPA에도 알려주어야 한다. 알려주는 방법은, 주도권이 없는 쪽에서 `@OneToMany`와 같은 연관관계 어노테이션에 `mappedBy` 옵션을 달아주는 것이다.

현재 `User`가 주도권이 없으니 다음과 같이 작성해 주자.

```
@OneToMany(mappedBy = "user")
private List<UserLoanHistory> userLoanHistories = new ArrayList<>();
```

`mappedBy = "user"`라고 작성하게 되면, `UserLoanHistory`의 `user` 필드를 JPA가 연관관계의 주인으로 인식하게 된다.

그렇다면 연관관계의 주인 효과는 무엇일까?! 연관관계의 주인의 값이 설정되어야 진정한 데이터가 저장된다. 말이 조금 어렵다! 이 부분은 다음 시간에 자세히 알아볼 예정이다.

마지막으로 Service 코드에서 나고 있는 빨간 줄을 해결해 주자. `userId` 대신 `user`를 넣어 주면 된다.

```
// 원래 코드
userLoanHistoryRepository.save(new UserLoanHistory(userId, book.getName()));

// 변경된 코드
userLoanHistoryRepository.save(new UserLoanHistory(user, book.getName()));
```


매우 좋다~ 🍌 이렇게 우리는 `User` 와 `UserLoanHistory` 가 서로를 알아보고 있도록 처리하였다!

하지만 아직, 아까처럼 원했던 그림은 달성하지 못했다. 여전히 `BookService` 는 `User` 와 `UserLoanHistory` 둘 모두를 각자 다루고 있다. `User` 와 `UserLoanHistory` 가 온전히 협력하지 못하는 것이다.

자, 우선은 다음 시간에 이어 JPA 연관관계에 대한 추가적인 어노테이션과 설정들을 알아보도록 하자.

34강. JPA 연관관계에 대한 추가적인 기능들

지난 시간에는 교실과 학생처럼 1 : N 관계에 있는 테이블을 JPA를 활용해 객체로 매핑하는 간단한 방법을 다루었다. 이번 시간에는 JPA 연관관계에 대한 추가적인 내용을 다룰 예정이다.

가장 먼저 1 : 1 관계부터 살펴보자!

1 : 1 관계를 이해하기 위해 예시를 들어보자. 여기 사람이 있다. 이 사람이 실제 거주하고 있는 주소 정보 역시 존재한다. 한 사람은 한 곳에서만 거주할 수 있으니 사람 정보 1개는 주소 정보를 단 1개만 가지고 있다.

이를 JPA가 관리하는 Entity 객체로 표현하면 다음과 같다.

```
@Entity
public class Person {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id = null;

    private String name;

    private Address address;

}
```

```
@Entity
public class Address {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
```

```

private Long id = null;

private String city;

private String street;

private Person person;

}

```

코드에서 알 수 있듯이, Person은 Address를 가지고 있고, Address는 Person을 가지고 있다!

자 그런데 이를 MySQL 테이블로 옮겨 생각해 보자. 테이블에서는 둘의 연결을 표현하기 위해 한 테이블이 다른 테이블의 id를 가지고 있으면 된다.

즉, **person** 이 **address** 의 id를 가지고 있을 수도 있고 **address** 가 **person** 의 id를 가지고 있을 수도 있다. 객체는 서로 연결이 되어 있는데 말이다!

여기서 바로 지난 강의에서 잠깐 언급한, **연관관계의 주인**이 등장한다.

자 우선, 1 : 1 관계를 표현하기 위해 **@OneToOne** 어노테이션을 각각 붙여 주자.

```

// Person.java
@OneToOne
private Address address;

// Address.java
@OneToOne
private Person person;

```

이 상황에서 테이블을 다음과 같이 구성했다고 생각해 보자.

```

// person 테이블
create table person
(
    id    bigint auto_increment,
    name  varchar(255),
    address_id bigint,
    primary key (id)
);

// address 테이블
create table address
(
    id    bigint auto_increment,

```

```

    city varchar(255),
    street varchar(255),
    primary key (id)
);

```

이 경우는 `person` 테이블이 `address` 테이블을 가리키고 있다. 즉, `person` 이 두 관계의 주도권을 가지고 있으며, 연관관계의 주인인 것이다!

이를 JPA에 알려주려면 연관관계의 주인이 아닌 쪽에 `mappedBy` 옵션을 사용해 주어야 한다. `mappedBy` 라는 단어 뜻 자체가 어딘가에 매여 있다는 뜻이다. 즉 연관관계의 주인이 아니면, 주인에게 매여 있는 것이다.

```

// Person.java
@OneToOne
private Address address;

// Address.java
@OneToOne(mappedBy = "address")
private Person person;

```

자 그렇다면 **연관관계의 주인 효과**는 무엇일까?! 바로 **객체가 연결되는 기준**이 된다. 어떤 의미인지 다음 코드와 함께 살펴보자. 원래 setter 사용은 피해야 하지만, 상황을 설명하기 위해 setter를 노출시켰다.

```

@Service
public class PersonService {

    private final AddressRepository addressRepository;
    private final PersonRepository personRepository;

    public PersonService(AddressRepository addressRepository, PersonRepository personRepository) {
        this.addressRepository = addressRepository;
        this.personRepository = personRepository;
    }

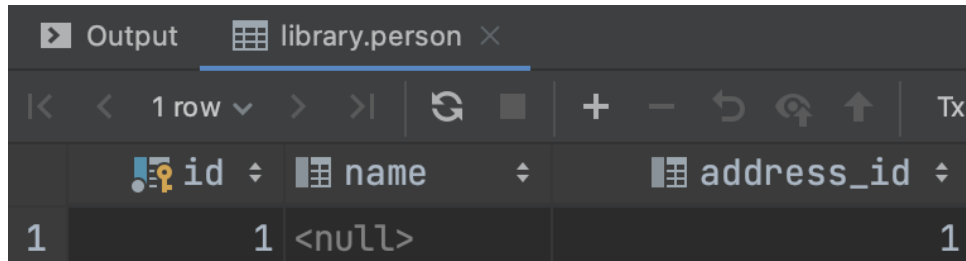
    @Transactional
    public void savePerson() {
        Person person = personRepository.save(new Person());
        Address address = addressRepository.save(new Address());
        person.setAddress(address);
    }
}

```

위 코드의 핵심은 `savePerson` 이다. 이 함수는 `Person` 객체를 만들어 저장하고 `Address` 객체를 만들어 저장한다. 그리고 `Person` 의 `setAddress` 를 이용해 `Person` 과 `Address` 를 이어준다.

이는 영속성 컨텍스트의 변경 감지에 의해 저장될 것이다!

자 임시로 Controller를 만들어 위의 코드를 실행시켜 보면, DB에 정상적으로 연결된 것을 확인할 수 있다. (아래는 `person` 테이블이다.)



The screenshot shows a database viewer window titled 'library.person'. It displays a table with three columns: 'id', 'name', and 'address_id'. There is one row with the values '1', '<null>', and '1'.

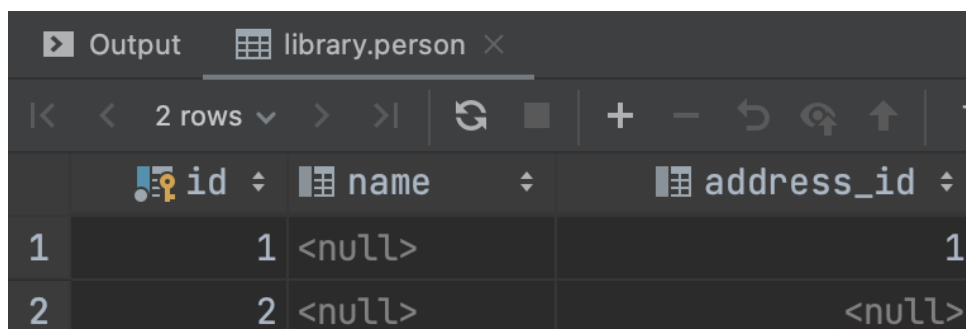
	id	name	address_id
1	1	<null>	1

자 그런데, 이번에는 `savePerson` 로직을 딱 하나만 변경해 보자!

```
@Transactional
public void savePerson() {
    Person person = personRepository.save(new Person());
    Address address = addressRepository.save(new Address());
    address.setPerson(person); // 바뀐 지점
}
```

아까는 `Person` 객체 안에 있는 `Address` 를 변경해 주었다면, 이번에는 `Address` 객체 안에 있는 `Person` 을 변경한 것이다.

그리고 다시 코드를 실행시켜 보면..!



The screenshot shows the same database viewer window, but now it displays two rows. The first row is the same as before. The second row has the values '2', '<null>', and '<null>'. The '<null>' in the third column is underlined in red.

	id	name	address_id
1	1	<null>	1
2	2	<null>	<null>

아까와 다르게 `person` 테이블과 `address` 테이블이 연결되지 않았다!!! 도대체 왜 이런 일이 발생했을까??

이것이 바로 연관관계의 주인 효과이다.

상대 테이블을 참조하고 있는, `mappedBy` 옵션을 가지고 있지 않은, 연관관계의 주인을 기준으로 테이블이 연결되는 것이다.

1 : 1 관계에서는 반대로도 가능하다. 만약 `address` 테이블이 `person_id` 를 가지고 있었다면, `Address` 가 연관관계의 주인이 될 것이고, `mappedBy` 속성 역시 `Person` 쪽에 붙게 되며, `address.setPerson()` 을 사용해야지만, 테이블 간의 매핑이 정상적으로 이루어진다.

연관관계를 다룰 때, 또 하나 주의해야 할 점이 있다. 연관관계의 주인을 통해 객체를 이어준다 하더라도, 그 즉시 반대쪽도 이어지지 않는다는 점이다. 이 역시 코드를 통해 바로 알아보자.

```
@Transactional
public void savePerson() {
    Person person = personRepository.save(new Person());
    Address address = addressRepository.save(new Address());
    person.setAddress(address);
}
```

`Person` 이 연관관계의 주인인 상황에서의 코드를 가져왔다. 이때 `Person` 은 `setAddress` 를 통해 `Address` 가 연결되어 있고, 트랜잭션이 커밋될 때 정상적으로 테이블도 연결될 것이다.

그러나! 아직 트랜잭션이 끝나지 않은 시점에 `Address` 는 `Person` 을 알고 있지 않다.

```
@Transactional
public void savePerson() {
    Person person = personRepository.save(new Person());
    Address address = addressRepository.save(new Address());
    person.setAddress(address);

    address.getPerson(); // null이 될 것이다!!
}
```

생각해 보면 당연하다. `Person` 한테는 set을 해주었지만, `Address` 한테는 set을 해주지 않았기 때문이다. 하지만 한 편으로는 또 이상하다. 이 함수가 끝나면 테이블은 잘 매핑되어 있을 것이기 때문이다.

이런 이상한 상황을 해결하기 위한 방법은, 하나의 setter 안에서 객체끼리 완전히 연결시켜 주는 것이다! 예를 들어, `Person` 의 `setAddress` 를 다음과 같이 바꿀 수 있다.

```
public void setAddress(Address address) {
    this.address = address;
    this.address.setPerson(this);
}
```

이렇게 되면, `Person`은 `Address`를 설정하고, 다시 이 `address`는 `Person`을 설정해 주었기 때문에 테이블끼리도 객체끼리도 완전하게 연결된다! 😊

매우 좋다~!! 👍 우리는 1 : 1 매핑과 연관관계의 주인이라는 개념을 익혔다.

이 연관관계의 주인은 우리가 지난 시간에 사용했던 `@OneToMany` + `@ManyToOne` 조합에서도 사용되었다.

이 어노테이션들은 1 : N 관계에서 활용되었다.

```
// User.java
@OneToMany(mappedBy = "user")
private List<UserLoanHistory> userLoanHistories = new ArrayList<>();

// UserLoanHistory.java
@ManyToOne
private User user;
```

여기서도 연관관계의 주인을 찾을 수 있다. `user_loan_history` 테이블이 `user` 테이블을 가리키고 있기 때문에 `UserLoanHistory`가 연관관계의 주인이고 때문에 `User` 쪽에 `mappedBy` 옵션이 붙게 된 것이다.

`@ManyToOne`을 단방향으로만 활용할 수도 있다. 예를 들어, `User`가 `List<UserLoanHistory>`를 아예 가지고 있지 않은 것이다! 이렇게 되면 `UserLoanHistory`를 `User`에 접근할 수 있지만, `User`는 `UserLoanHistory`에 접근할 수 없게 된다.

다음으로는 `@JoinColumn`이라는 어노테이션이 있다. 이 어노테이션은 연관관계의 주인에게 활용할 수 있는 어노테이션이다. 연관관계의 주인이 가지고 있는 다른 테이블을 가리키는 필드의 이름이나 null 여부, 유일성 여부, 업데이트 가능 여부 등을 정해줄 수 있다.

```
@JoinColumn(nullable = false)
@ManyToOne
private User user;
```

보통은 이름을 특별히 변경해 주어야 할 때 활용되며, `user`라는 필드 이름을 가지고 있다면, 기본적으로 `user_id`에 매핑된다.

다음으로 N : M 연관관계를 의미하는 `@ManyToMany` 어노테이션도 존재한다. 하지만 N : M 연관관계는 구조가 복잡하고, 테이블 역시 직관적으로 매핑되지 않기 때문에 사용하지 않는 것

이 좋다.

다음은, 조금 심화 내용으로 `cascade` 옵션과 `orphanRemoval` 옵션에 대해 알아보자. 지금 당장 디테일한 모든 내용을 알기보다는 어떤 옵션인지를 이해해 보자.

먼저 `cascade` 옵션을 살펴보자. `cascade` 는 ‘폭포처럼 흐르다’라는 의미를 가지고 있다. 즉 한 객체가 저장되거나 삭제될 때, 그 변경이 폭포처럼 흘러 연결되어 있는 객체도 함께 저장되거나 삭제되는 기능이다.

예를 들어 DB에 다음과 같은 데이터가 존재한다.

- ABC 유저
 - ABC 유저가 빌린 책1 기록
 - ABC 유저가 빌린 책2 기록

이 상황에서, 다음과 같은 코드를 ABC 유저에 대해 실행시키면 무슨 일이 일어날까?

```
@Transactional
public void deleteUser(String name) {
    User user = userRepository.findByName(name)
        .orElseThrow(IllegalArgumentException::new);

    userRepository.delete(user);
}
```

잠깐 상상해 보자.

정답은~ `user` 데이터만 쏙 사라진다!!! `User` 객체를 삭제했는데 연결되어 있던 `UserLoanHistory` 는 여전히 DB에 남아 있는 것이다.

이런 상황과 같이, `User` 가 삭제될 때 혹은 저장될 때 연결되어 있는 `UserLoanHistory` 도 똑같이 처리해 주고 싶다면, `cascade` 옵션을 사용할 수 있다.

```
@OneToMany(mappedBy = "user", cascade = CascadeType.ALL)
private List<UserLoanHistory> userLoanHistories = new ArrayList<>();
```

이렇게 코드를 변경하고 다시 데이터를 준비해서 `deleteUser` 를 실행시킨다면, 이번에는 `user` 데이터와 `user_loan_history` 데이터까지 모두 사라지게 된다!

다음은 `orphanRemoval` 옵션이다. `orphan` 은 고아라는 뜻이고, `removal`은 제거라는 뜻으로 `orphanRemoval` 만 보면 고아 제거라는 옵션이다. 도대체 무슨 의미일까? 이 역시 코드를 통해 살펴보자!

이번에도 다음과 같은 데이터가 존재한다.

- ABC 유저
 - ABC 유저가 빌린 책1 기록
 - ABC 유저가 빌린 책2 기록

이때 우리는 아래와 같은 코드를 실행시켜보자. `User` 객체에 연결되어 있던 책1의 대출 기록을 `List` 에서 제거하는 코드이다.

```
@Transactional
public void deleteUserHistory() {
    User user = userRepository.findByName("ABC")
        .orElseThrow(IllegalArgumentException::new);
    user.removeOneHistory();
}

// User.java
public void removeOneHistory() {
    userLoanHistories.removeIf(history -> "책1".equals(history.getBookName()));
}
```

자 이 상황에서 코드가 실행되면 무슨 일이 일어날지 잠시 생각해 보자. 정답은, 데이터베이스에 변화가 없다는 것이다!

만약 `UserLoanHistory` 를 데이터베이스에서 제거하고 싶다면, `User` 에 연결된 `List` 에서 제거하는 게 아니라 직접 `UserLoanHistoryRepository.delete()` 를 사용해 주어야 할 것이다.

하지만! 이럴 때 바로 `orphanRemoval` 옵션을 사용할 수 있다. 이 옵션은 연결이 끊어지면 그 데이터를 삭제하는 옵션이다. 지금처럼 책1이 `List` 에서 제거되는 상황에 딱 맞게 사용할 수 있는 옵션인 것이다.

```
@OneToMany(mappedBy = "user", cascade = CascadeType.ALL, orphanRemoval = true)
private List<UserLoanHistory> userLoanHistories = new ArrayList<>();
```

다시 한번 옵션을 적용하고 위의 코드를 실행시켜보자. 그랬더니, 실제 DB에서 책1에 대한 기록이 사라진 것을 확인할 수 있다!

오늘 나왔던 내용을 간단히 정리해 보자!

- 상대 테이블을 가리키는 테이블이 연관관계의 주인이다. 연관관계의 주인이 아닌 객체는 `mappedBy`를 통해 주인에게 매여 있음을 표시해 주어야 한다.
- 양쪽 모두 연관관계를 갖고 있을 때는 양쪽 모두 한 번에 맺어주는 게 좋다.
- `cascade` 옵션을 활용하면, 저장이나 삭제를 할 때 연관관계에 놓인 테이블까지 함께 저장 또는 삭제가 이루어진다.
- `orphanRemoval` 옵션을 활용하면, 연관관계가 끊어진 데이터를 자동으로 제거해 준다.

매우 좋다~! 😊 이제 다음 시간에는 연관관계를 제대로 활용할 수 있도록 책 대출/반납 기능을 리팩토링할 것이다!

35강. 책 대출/반납 기능 리팩토링과 지연 로딩

이번 시간에는 이전에 만들었던 책 대출/반납 기능을 다음 그림과 같이 리팩토링해보자!

먼저, 유저 대출 기능이다. 우리는 `cascade` 옵션을 넣어둔 덕분에 `UserLoanHistory`를 직접 Service에서 만들어 저장하는 것이 아니라 `User` 객체에 책을 대출한다는 함수를 호출시키고, 그 함수 안에서 `UserLoanHistory`를 List에 넣어주기만 하면 된다.

유저에 다음과 같은 함수를 만들자.

```
public void loanBook(String bookName) {
    this.userLoanHistories.add(new UserLoanHistory(this, bookName));
}
```

그다음 Service 코드는 `UserLoanHistory`를 직접 사용하지 않고 `User`를 통해 대출 기록을 저장하도록 변경하자.

```
@Transactional
public void loanBook(BookLoanRequest request) {
    Book book = bookRepository.findByName(request.getBookName())
        .orElseThrow(IllegalArgumentException::new);

    if (userLoanHistoryRepository.existsByBookNameAndIsReturn(book.getName(), false)) {
        throw new IllegalArgumentException("진작 대출되어 있는 책입니다");
    }

    User user = userRepository.findByName(request.getUserName())
        .orElseThrow(IllegalArgumentException::new);
    user.loanBook(book.getName()); // 바뀐 코드
}
```

매우 좋다~! 😊 우리가 원하는 것처럼 `BookService` 는 `UserLoanHistory` 객체를 직접 사용하지 않게 변경되었다.

또한 ‘대출’이라는 행위는 `User` 와 `UserLoanHistory` , 2개의 객체가 서로 협력해서 이루어내도록 변경되었다. 이를 가리켜 ‘도메인 계층에 비즈니스 로직이 들어갔다’라고 표현한다.

다음으로 이어서 반납 기능도 개선해 보자! 대출과 비슷하게 책 이름을 받는 함수를 만들 수 있다!

```
public void returnBook(String bookName) {
    UserLoanHistory targetHistory = this.userLoanHistories.stream()
        .filter(history -> history.getBookName().equals(bookName))
        .findFirst()
        .orElseThrow(IllegalArgumentException::new);
    targetHistory.doReturn();
}
```

Java8에 나오는 문법인 스트림과 람다를 적극 활용했다. `User` 객체가 가지고 있는 `List<UserLoanHistory>` 에서 반납이 들어온 책 기록을 찾아 반납 처리를 해주는 로직이.

이제 Service 로직 역시 바뀌게 된다!

```
@Transactional
public void returnBook(BookReturnRequest request) {
    User user = userRepository.findByName(request.getUserName())
        .orElseThrow(IllegalArgumentException::new);
    user.returnBook(request.getBookName());
}
```

이전에 비해 크게 간소화된 것을 확인할 수 있다! 👍

자 그런데 바로 여기서 영속성 컨텍스트의 4번째 능력이 동작하게 된다. `User` 를 가져오는 코드 아래에 `System.out.println("Hello")` 를 두고 API를 호출해 자동으로 날아가는 SQL 로 그를 확인해 보자.

그러면 다음과 같은 로그를 확인할 수 있다!

```
Hibernate:
select
  user0_.id as id1_3_,
  user0_.age as age2_3_,
  user0_.name as name3_3_
from
  user user0_
```

```

        where
            user0_.name=?
Hello
Hibernate:
    select
        userloanhi0_.user_id as user_id4_4_0_,
        userloanhi0_.id as id1_4_0_,
        userloanhi0_.id as id1_4_1_,
        userloanhi0_.book_name as book_nam2_4_1_,
        userloanhi0_.is_return as is_retur3_4_1_,
        userloanhi0_.user_id as user_id4_4_1_
    from
        user_loan_history userloanhi0_
    where
        userloanhi0_.user_id=?
...

```

무언가 느껴지지 않는가?! 그렇다! `User` 를 먼저 가져온 다음, 필요한 순간에 `List<UserLoanHistory>` 를 가져오고 있는 모습을 확인할 수 있다! 만약, `List<UserLoanHistory>` 가 필요하지 않다면 영원히 가져오지 않을 것이다. 이를 어려운 말로 '지연 로딩'이라고 부르며 영속성 컨텍스트의 4번째 능력이다.

이렇게 연결되어 있는 객체를 지연 로딩한 이유는 fetch 옵션 때문이다. `@OneToMany` 코드를 보면, `FetchType` 을 사용하는 `fetch` 옵션이 있는데, 이 옵션은 2가지 종류가 있다.

지금처럼 꼭 필요할 때 가져오는 `LAZY` 옵션과, 처음 데이터를 로딩할 때 바로 가져오는 `EAGER` 옵션이다! `@OneToMany` 는 기본적으로 `LAZY` 옵션을 가지고 있기 때문에 우리가 확인한 것처럼 `User` 를 먼저 가져오고 꼭 필요한 순간에 `UserLoanHistory` 를 다시 가져온 것이다.

이렇게 대출과 반납 기능 모두 리팩토링해보고, 영속성 컨텍스트의 4번째 능력까지 살펴봤다.

추가적으로, 2가지 생각해 볼 거리에 대해 간단히 이야기해보자.

[1] 연관관계를 사용하면 무엇이 좋을까?

연관관계를 사용하기 전후의 객체 관계도를 비교해 보자. 연관관계를 사용하게 되면서 가장 큰 변화는 `User` 와 `UserLoanHistory` 가 직접 협업하게 되었다는 점이다.

덕분에 Service 코드가 간결해졌고, 우리의 비즈니스 로직이 도메인 계층으로 내려가게 되었다. 각각의 역할에 더 집중할 수 있게 된 것이다. 이렇게 되면 새로운 개발자가 왔을 때 코드를 조금 더 이해하기 쉬워지고, 이번 강의에서는 작성하지 않지만 테스트 역시 쉬워진다.

[2] 그렇다면 연관관계를 사용하는 것이 항상 좋을까?

꼭 그렇지는 않다. 연관관계를 너무 지나치게 사용하면, 성능상의 문제나 도메인 간의 복잡한 연결로 인해 시스템을 파악하기 어려워지고, 한 군데의 수정이 일어날 경우 다른 곳까지 영향을 주게 된다.

그럼 언제 연관관계를 쓰고, 언제 연관관계를 쓰지 말아야 할까? 참 정답이 없고 어려운 문제이다. 비즈니스 요구사항, 기술적인 요구사항, 도메인 아키텍처 등 여러 부분을 고민해야 된다. 그래서 설계의 재미가 있는 것 같다.

매우 좋다~!! 😊 우리는 연관관계를 활용하여 도서 대출 기능과 반납 기능까지 완전히 리팩토링했다~! 이제 다음 시간에 이번 Section을 정리해 보자!

36강. Section 5 정리. 다음으로!

드디어 우리의 도서관리 애플리케이션은 모두 끝이 났다!! 🎉🎉

특히 이번 Section에서는 다음과 같은 내용을 다룰 수 있었다. 👍

1. 책 생성, 대출, 반납 API를 온전히 개발하며 지금까지 다루었던 모든 개념을 실습해 본다.
2. 객체지향적으로 설계하기 위한 연관관계를 이해하고, 연관관계의 다양한 옵션에 대해 이해한다.
3. JPA에서 연관관계를 매핑하는 방법을 이해하고, 연관관계를 사용해 개발할 때와 사용하지 않고 개발할 때의 차이점을 이해한다.

자 하지만, 강의는 끝나지 않았다! 이제 우리는 ‘개발’을 끝냈을 뿐, ‘배포’를 해야 한다. 다음 Section에서는 배포가 무엇인지, 배포를 하기 위해 알아야 할 지식과 준비해야 할 세팅은 무엇인지 하나하나 살펴보기로 하자!