

30강. 책 생성 API 개발하기

Section 5. 책 요구사항 구현하기

1. 책 생성, 대출, 반납 API를 온전히 개발하며 지금까지 다루었던 모든 개념을 실습해본다.
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.

지금까지 배웠던 개념들을 총동원해서
책 생성 API를 개발하자!

요구사항

도서관에 책을 등록할 수 있다.

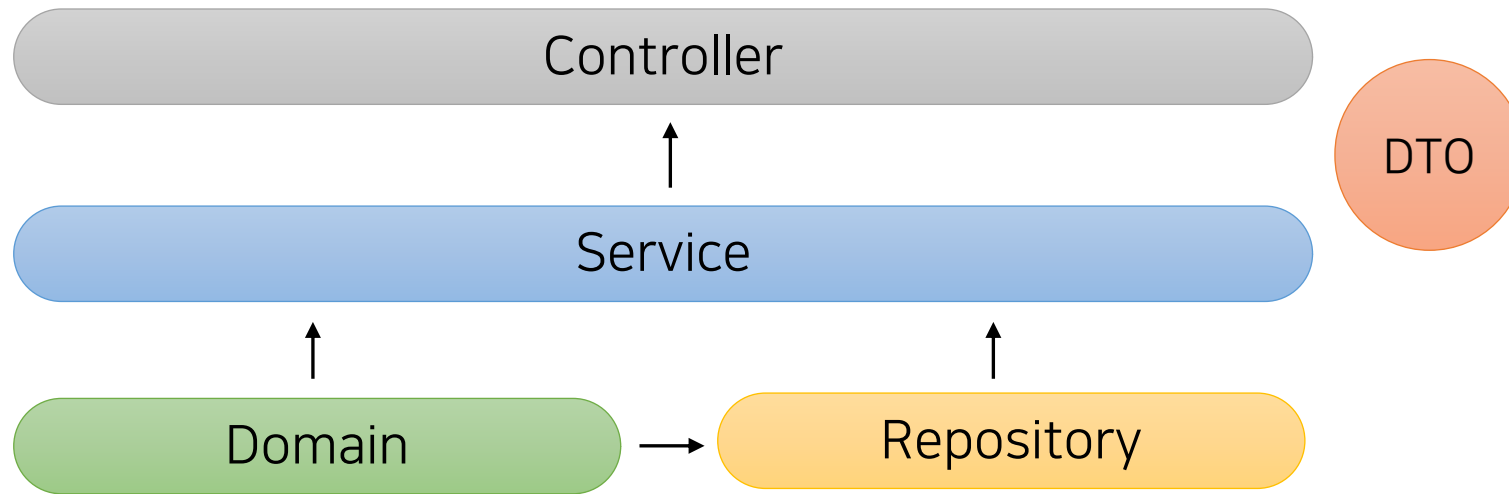
API 스펙

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

```
{  
  "name": String // 책 이름  
}
```

- 결과 반환 X (HTTP 상태 200 OK이면 충분하다)

할 일이 자연스럽게 떠오르시죠?!




book 테이블

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

id와 책 이름을 가지고 있는 book 테이블!

```
{  
  "name": String // 책 이름  
}
```



- 결과 반환 X (HTTP 상태 200 OK이면 충분하다)

book 테이블

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


book 테이블

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

1) @Column 의 length 기본값이 255

book 테이블

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

2) 문자열 필드는 최적화를 해야 하는 경우가 아닐때 조금 여유롭게 설정하는 것이 좋다!

Book 객체

```
@Entity
public class Book {

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

    @Column(nullable = false)
    private String name;

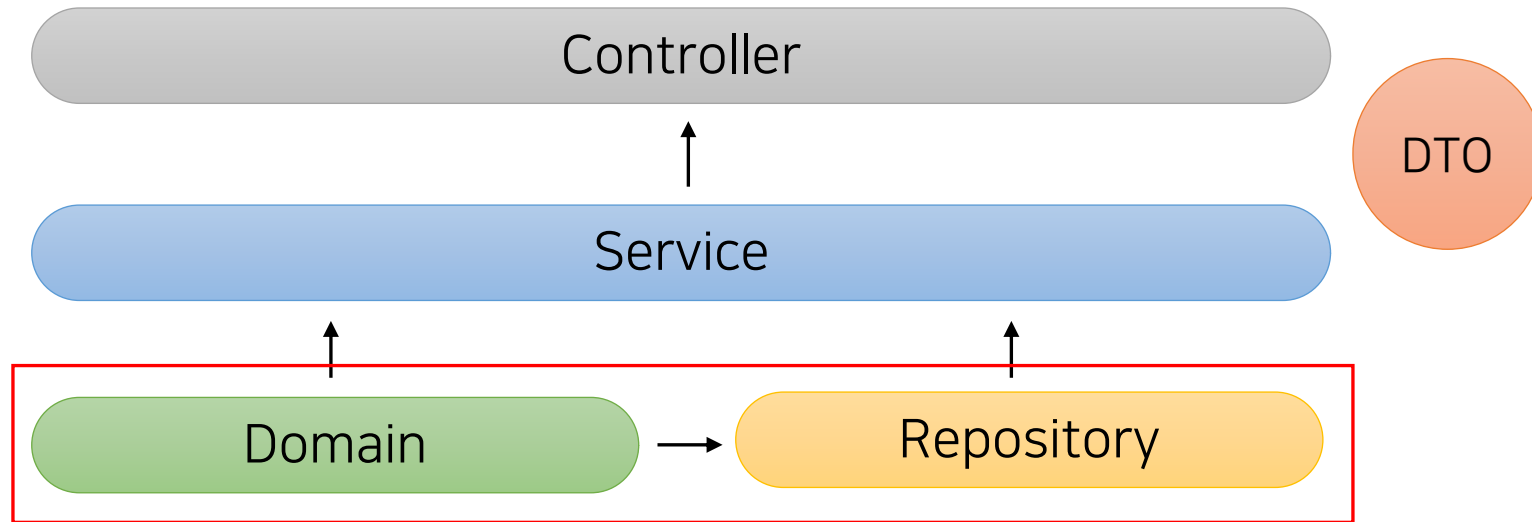
}
```

BookRepository

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

잠시 중간 점검!

여기까지 완성!



DTO를 만들어 주자!

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

다음으로는 Controller와 Service를 만들자!

다 완성했다면, 웹 UI와 DB를 통해 확인해보자!

31강. 대출 기능 개발하기

요구사항

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

API 스펙 확인

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

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

- 결과 반환 X (HTTP 상태 200 OK이면 충분하다)

요구사항을 보니 지금 테이블로는 충분하지 않다!

유저의 대출 기록을 저장하는 새로운 테이블이 필요하다!

user_loan_history 테이블 추가

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

user_loan_history 테이블 추가

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

어떤 유저가 책을 빌렸는지 알 수 있는 유저의 id

user_loan_history 테이블 추가

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

어떤 책을 빌렸는지 확인하기 위한 책 이름

user_loan_history 테이블 추가

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

현재 대출 중인지, 반납 완료했는지 확인

user_loan_history 테이블 추가

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

0이면 대출 중 / 1이면 반납한 것이다.

예를 하나 들어보자!

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

2번 유저는 2권의 책을 빌렸다.
클린 코드는 반납했고, 테스트 주도 개발은 대출중이다.

이제 UserLoanHistory 객체를 만들어 주자!

```
@Entity
public class UserLoanHistory {

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

    private long userId;

    private String bookName;

    private boolean isReturn;

}
```

이제 UserLoanHistory 객체를 만들어 주자!

boolean으로 처리하면,
tinyint에 잘 매핑된다!

```
@Entity
public class UserLoanHistory {

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

    private long userId;

    private String bookName;

    private boolean isReturn;

}
```



다음은 똑같다, DTO / Controller / Service 구현!

32강. 반납 기능 개발하기

요구사항 확인!

사용자가 책을 반납할 수 있다.

API 스펙 확인!

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

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

- 결과 반환 X (HTTP 상태 200 OK이면 충분하다)

API 스펙, HTTP Body 완전히 동일하다!

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

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

- 결과 반환 X (HTTP 상태 200 OK이면 충분하다)

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

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

- 결과 반환 X (HTTP 상태 200 OK이면 충분하다)

API 스펙, HTTP Body 완전히 동일하다!

이런 경우, DTO를 새로 만드는데 좋을까?
아니면 재사용하는게 좋을까?

API 스펙, HTTP Body 완전히 동일하다!

(개인적으로) 새로 만드는 것을 선호합니다!

API 스펙, HTTP Body 완전히 동일하다!

그래야 두 기능 중 한 기능에 변화가 생겼을 때
유연하고 side-effect 없이 대처할 수 있기 때문.

자 그러면 구현 한 번 해보겠습니다!

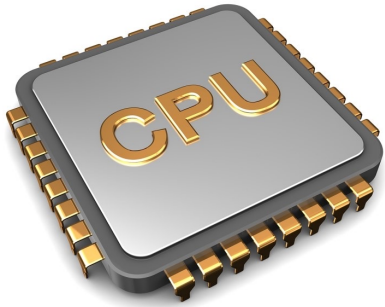
이번에도 마찬가지로 웹 UI와 DB를 열어 확인해보죠!

자 그런데, 한 가지 고민할만한 내용이 있습니다!

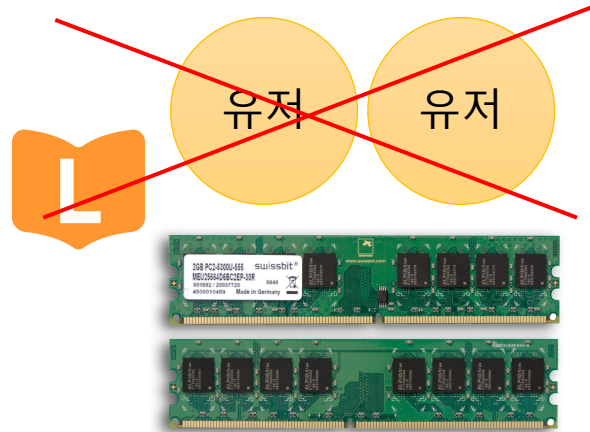
SQL 대신 ORM을 사용하게 된 이유 중 하나

“DB 테이블과 객체는 패러다임이 다르기 때문”

DB 테이블과 객체는 패러다임이 다르다!



CPU (연산담당)



RAM (메모리, 단기기억)



DISK (장기기록)

DB 테이블에 데이터를 저장하는 것은 필수이다!

자 그런데, 한 가지 고민할만한 내용이 있습니다!

하지만 Java 언어는 객체지향형 언어이고,

대규모 웹 애플리케이션을 다룰 때에도 절차지향적인 설계보다
객체지향적인 설계가 좋다!

자 그런데, 한 가지 고민할만한 내용이 있습니다!

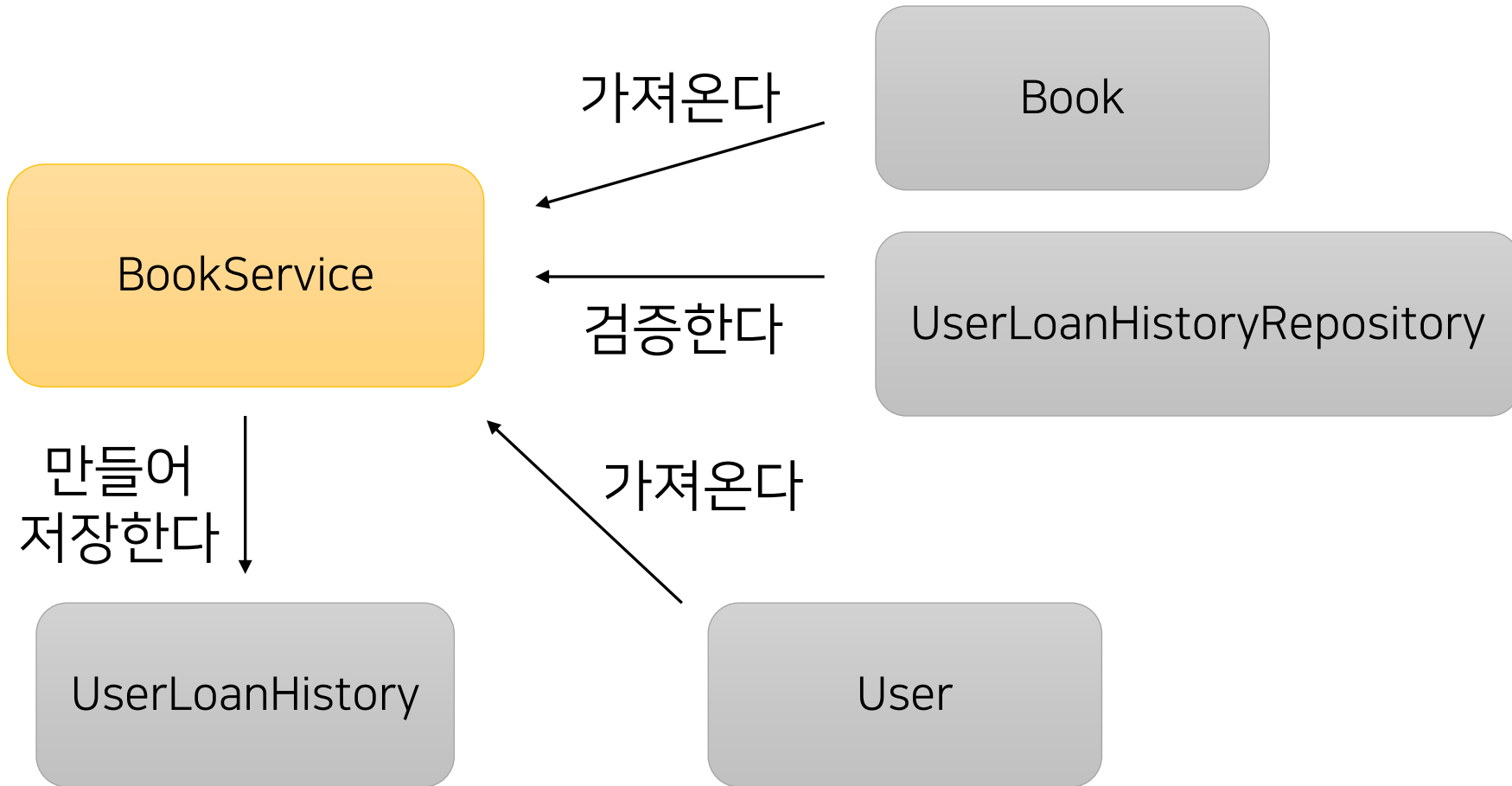
<20강. 스프링 컨테이너를 왜 사용할까?>
에서 살펴보았던 이유 역시
보다 객체지향적인 설계를 하기 위한 맥락에서 출발했다.

지금 코드를 조금 더 객체지향적으로 만들 수 없을까?

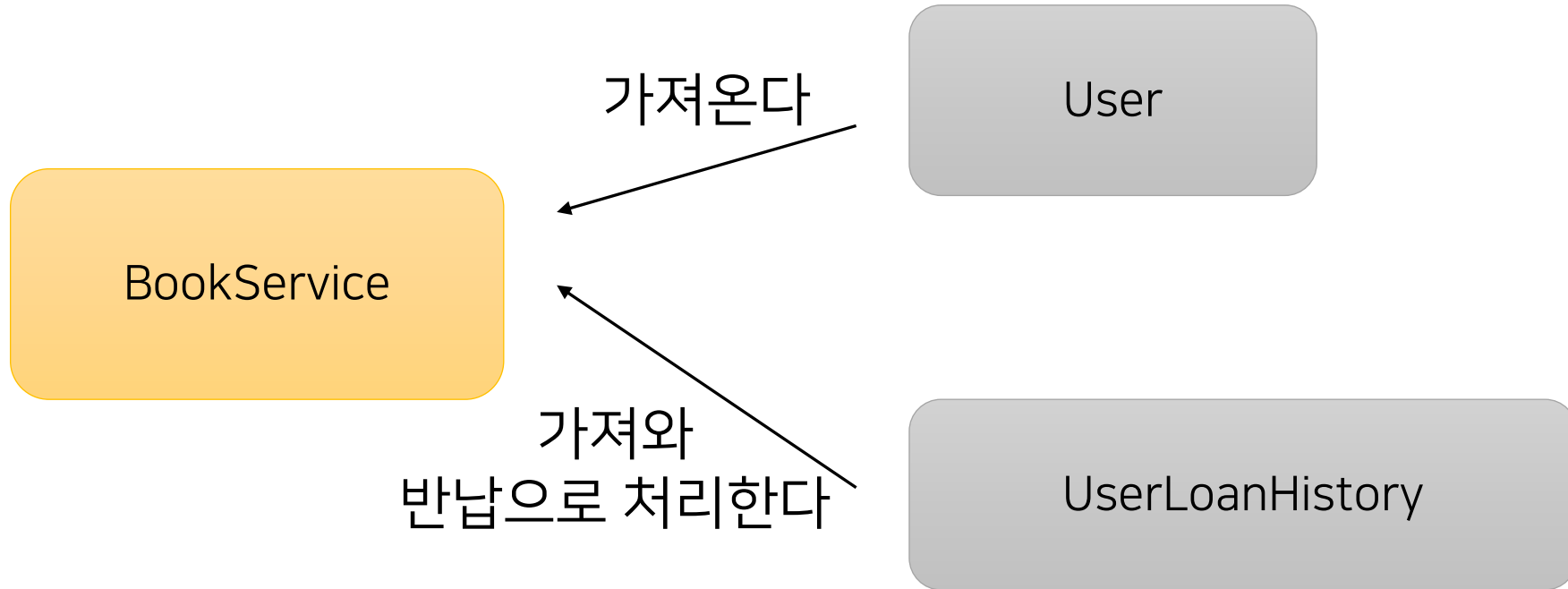
User와 UserLoanHistory가 직접 협업할 수 있게 처리할 수 없을까?

33강. 조금 더 객체지향적으로 개발할 수 있을까?

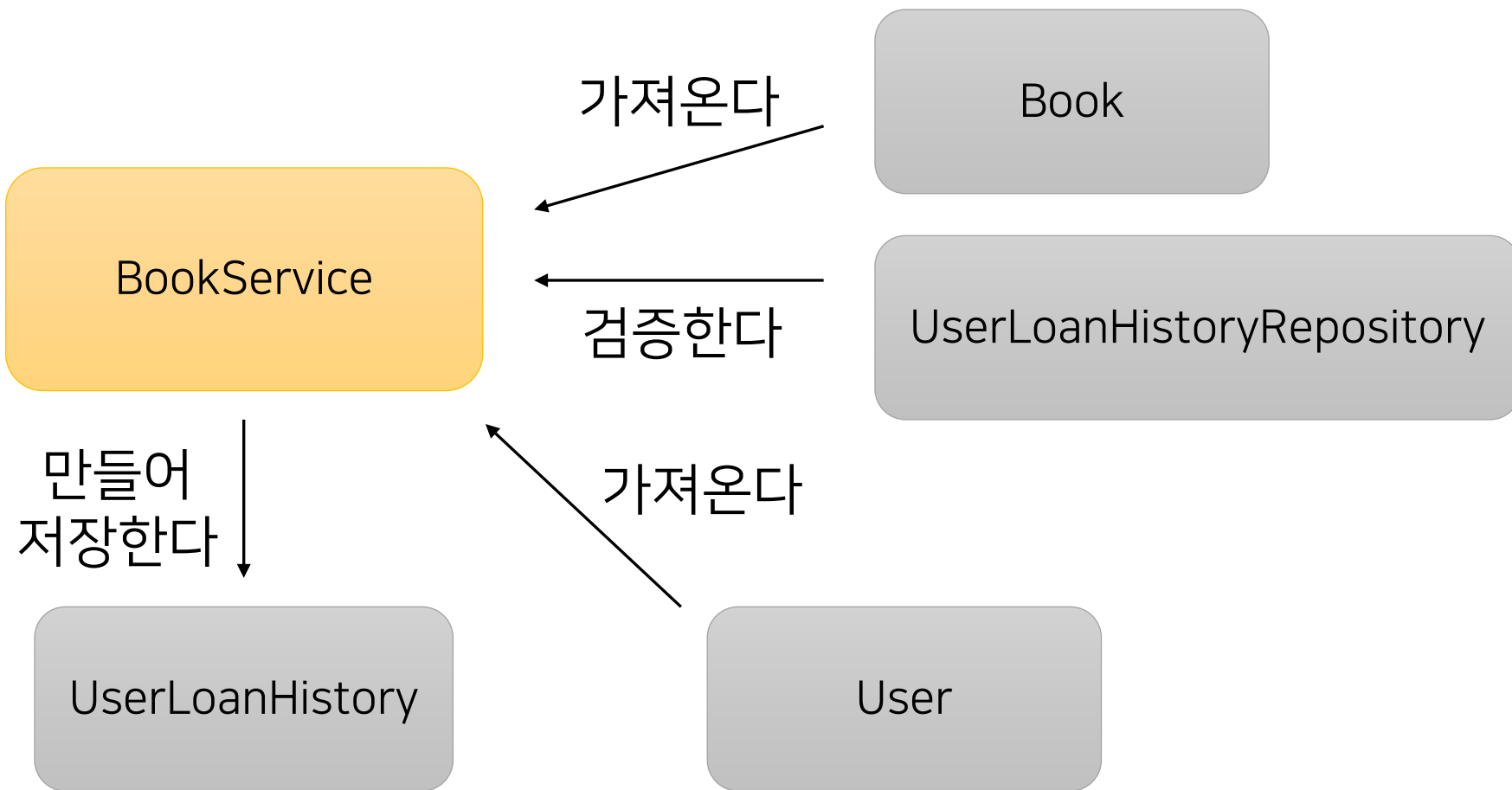
우선 현재의 대출 기능 관계를 살펴보자.



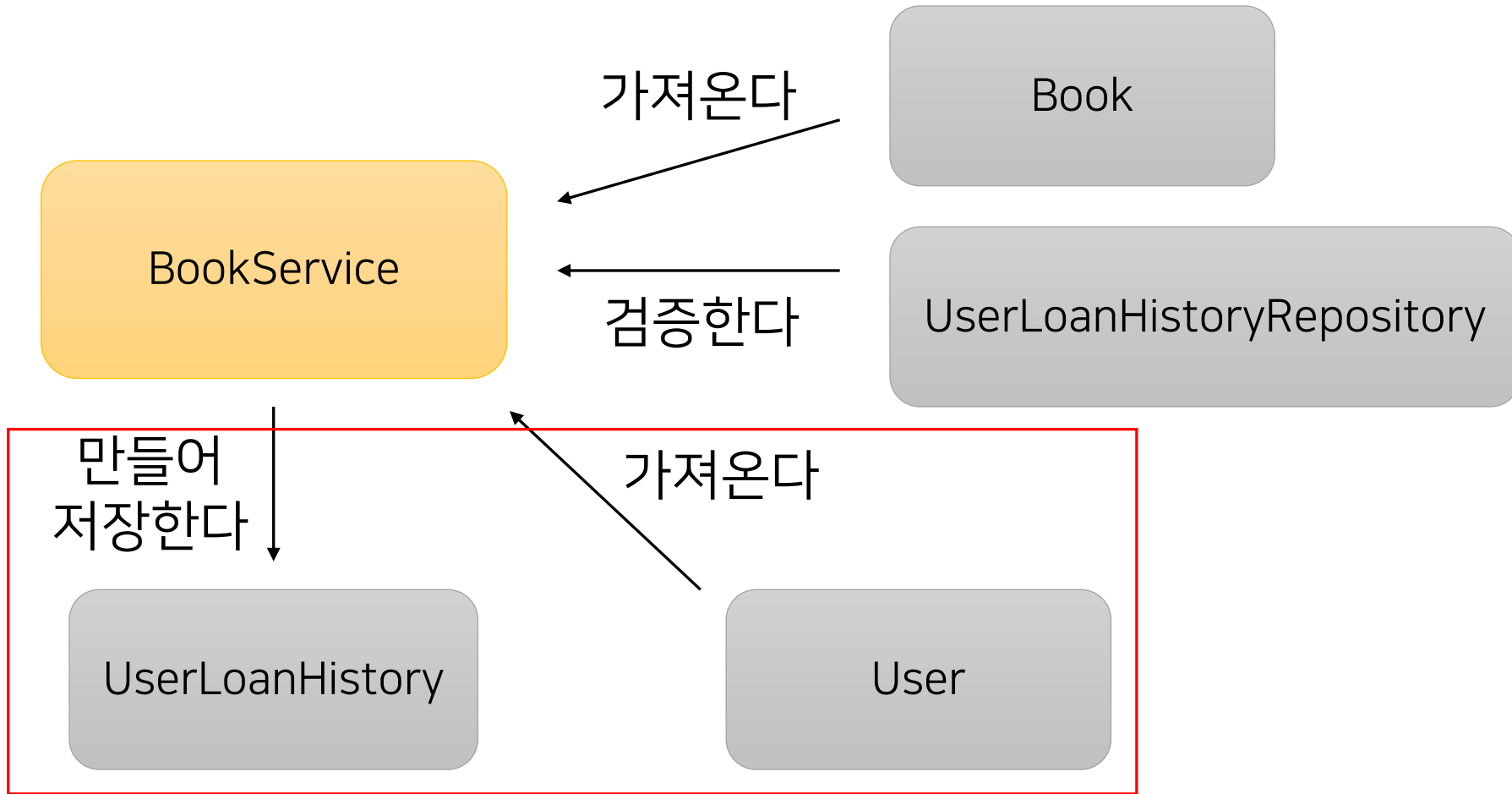
다음으로 현재의 반납 기능 관계를 살펴보자.



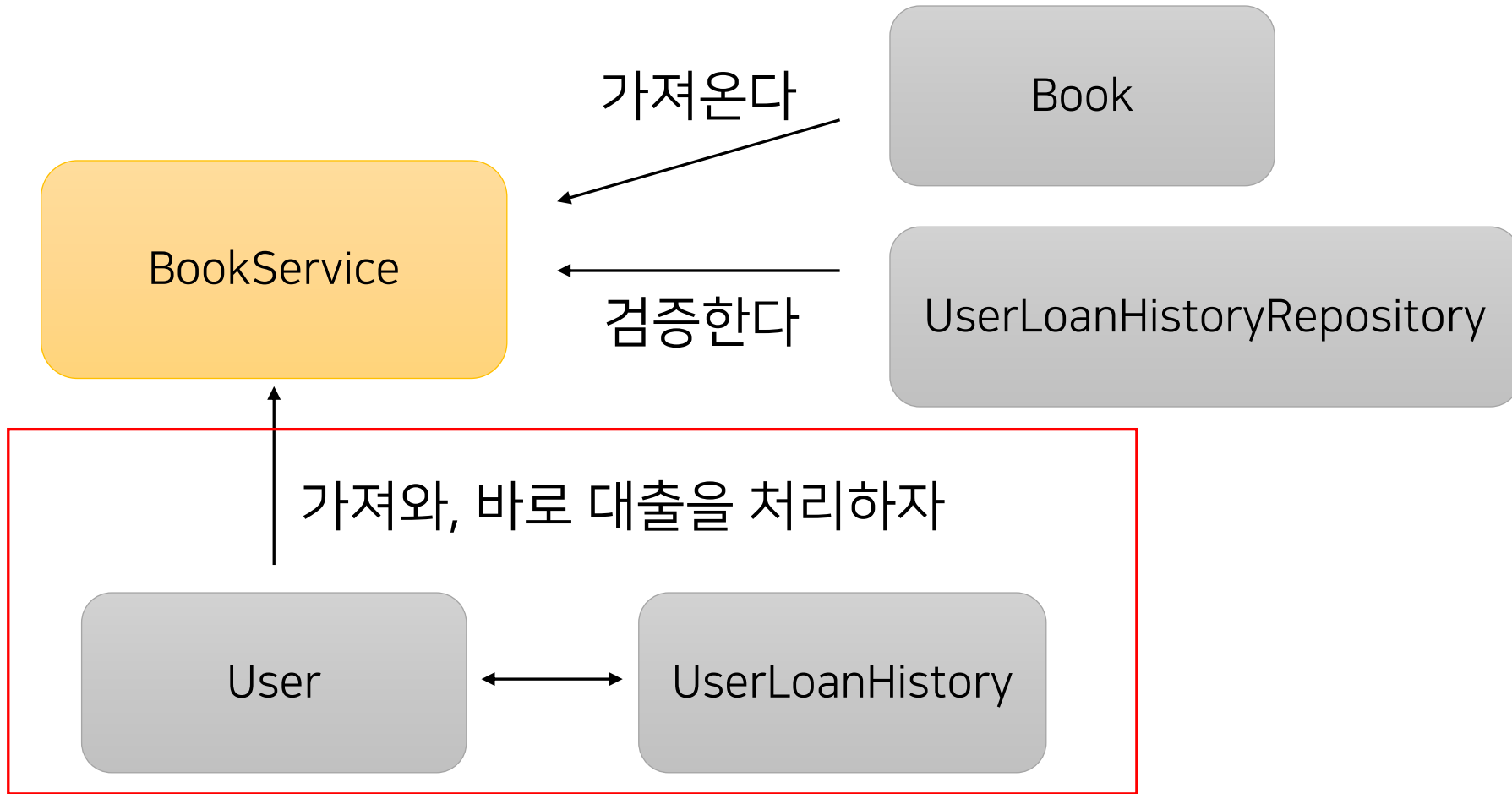
대출 기능은 이렇게 개선할 수 있다!



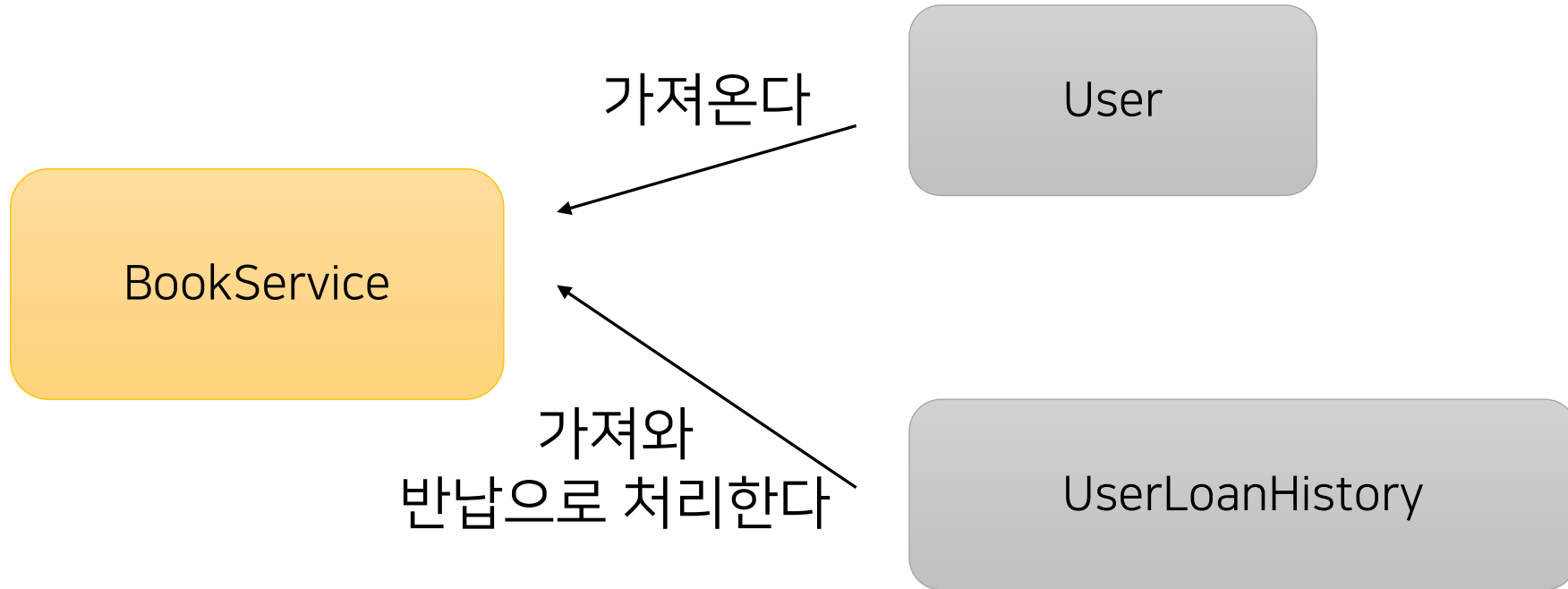
대출 기능은 이렇게 개선할 수 있다!



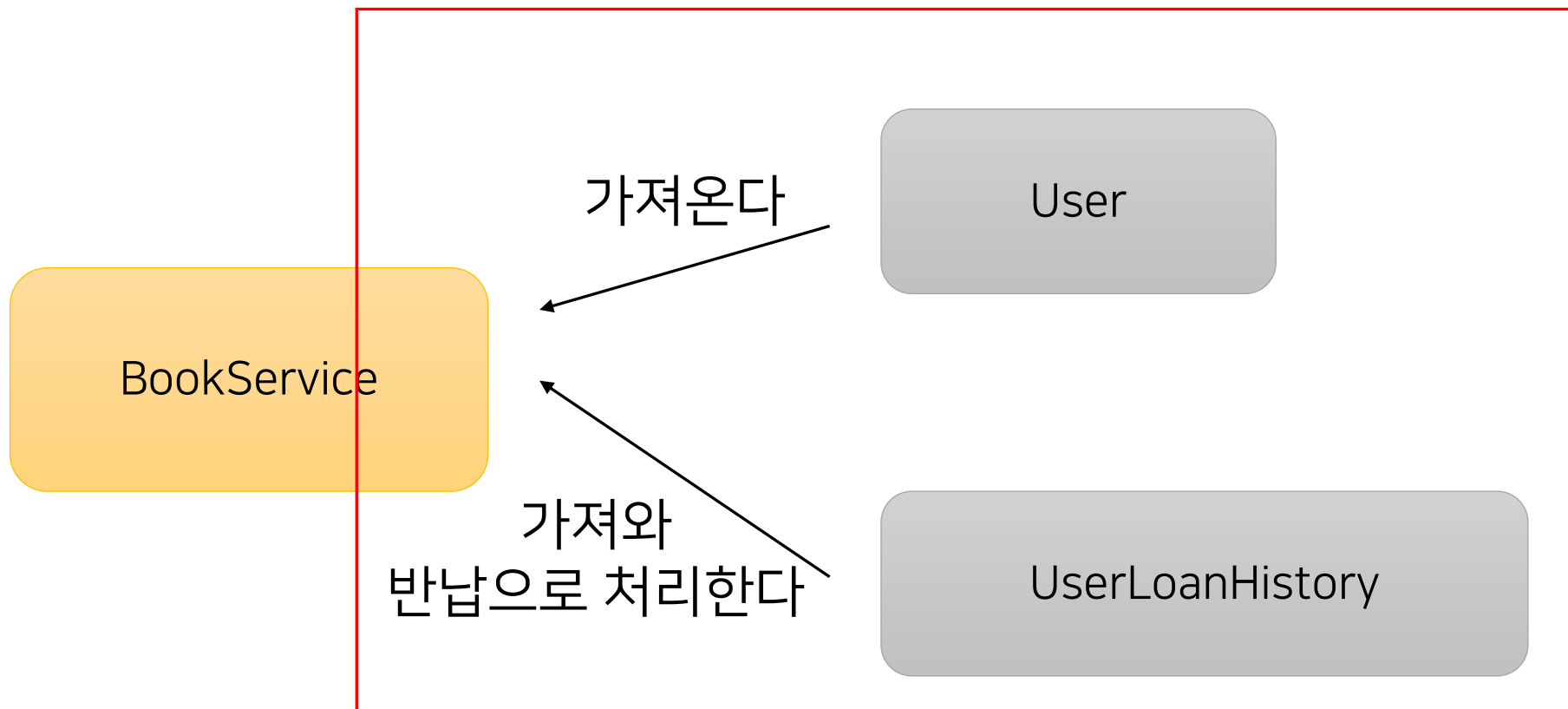
대출 기능은 이렇게 개선할 수 있다!



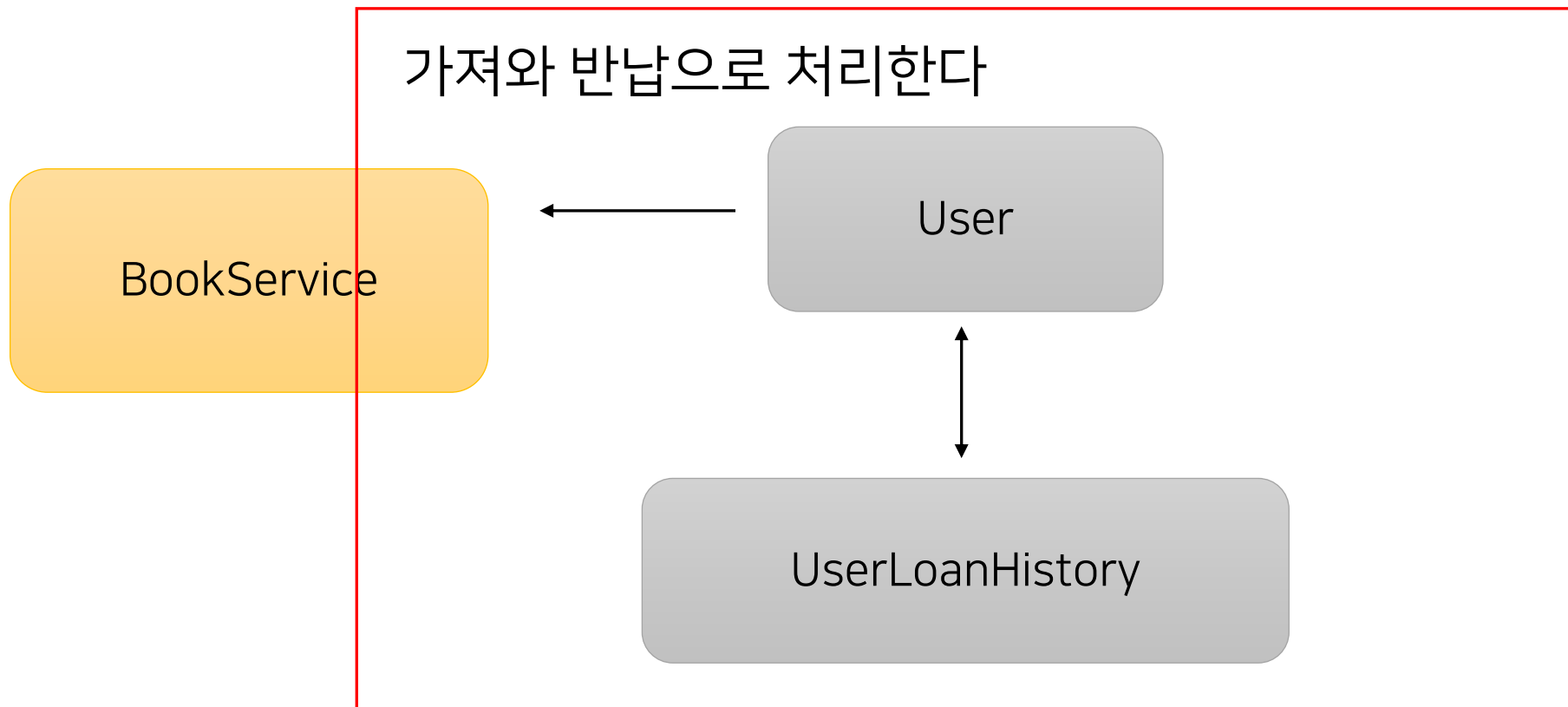
반납 기능은 이렇게 개선할 수 있다!



반납 기능은 이렇게 개선할 수 있다!

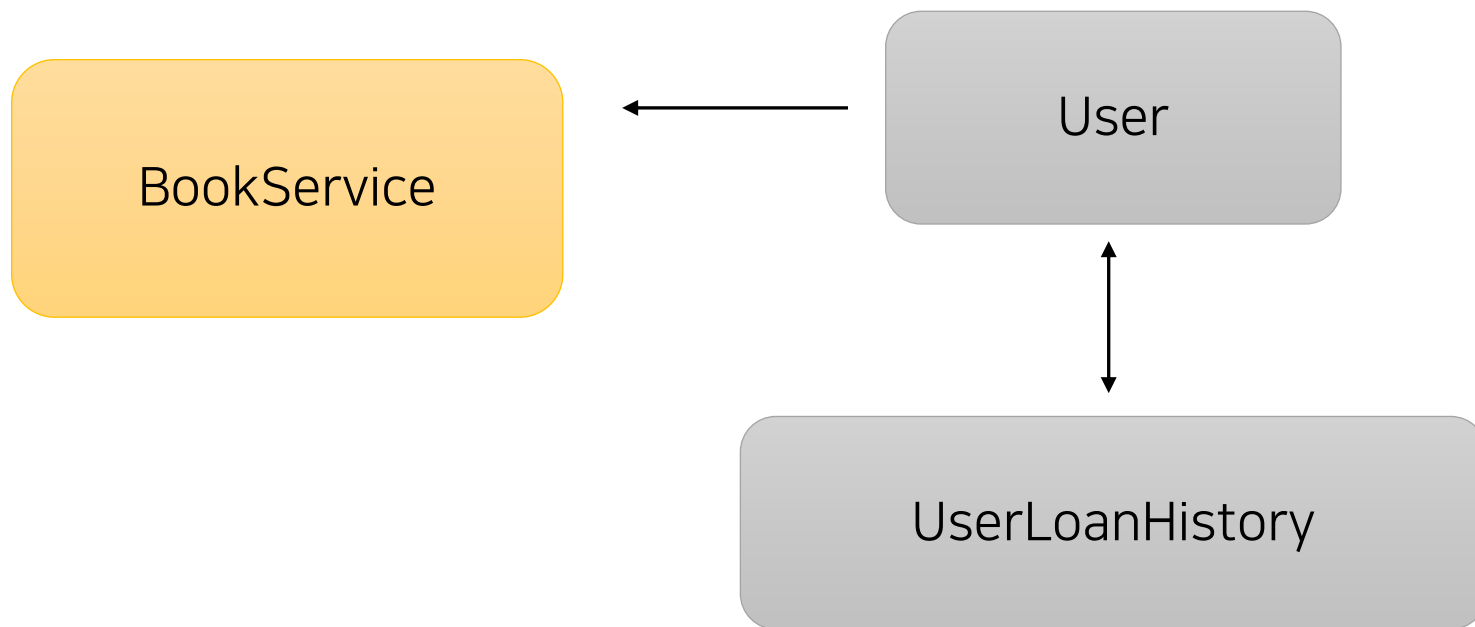


반납 기능은 이렇게 개선할 수 있다!



반납 기능은 이렇게 개선할 수 있다!

가져와 반납으로 처리한다



선행 조건

User와 UserLoanHistory가 서로를 알아야 한다.

코드를 한 번 변경해보자!

N : 1 관계란?!



학생



교실

학생 여러명이 교실에 들어갈 수 있다!

학생 N : 교실 1

연관관계의 주인

Table을 보았을 때 누가 관계의 주도권을 가지고 있는가

연관관계의 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)
);
```

연관관계의 주인

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)
);
```



연관관계의 주인

연관관계의 주인이 아닌 쪽에 mappedBy 옵션을 달아 주어야 한다.

연관관계의 주인

연관관계의 주인의 값이 설정되어야만 진정한 데이터가 저장된다.

34강. JPA 연관관계에 대한 추가적인 기능들

1 : 1 관계



사람



실거주 주소

한 사람은 한 개의 실거주 주소만을 가지고 있다!

1 : 1 관계



사람



실거주 주소

이를 객체로 표현해보자!

1 : 1 관계

```
@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;

}
```



Person과 Address를 이제 Table로 변경하자

잠깐! person 테이블이 address 테이블의 id를 가질 수도 있고,
address 테이블이 person 테이블의 id를 가질 수도 있다!

Person과 Address를 이제 Table로 변경하자

예를 들어 테이블을 다음과 같이 구성했다고 해보자!

Person과 Address를 이제 Table로 변경하자


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

```
create table address
(
    id    bigint auto_increment,
    city  varchar(255),
    street varchar(255),
    primary key (id)
);
```

Person과 Address를 이제 Table로 변경하자

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

```
create table address
(  
    id    bigint auto_increment,  
    city  varchar(255),  
    street varchar(255),  
    primary key (id)  
);
```



Person과 Address를 이제 Table로 변경하자

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


```
create table address
(  
    id    bigint auto_increment,  
    city  varchar(255),  
    street varchar(255),  
    primary key (id)  
);
```

여기서 연관관계의 주인이 등장한다!

Person과 Address를 이제 Table로 변경하자

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

```
create table address
(  
    id    bigint auto_increment,  
    city  varchar(255),  
    street varchar(255),  
    primary key (id)  
);
```




person이 주도권을 가지고 있다!

Person과 Address를 이제 Table로 변경하자

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

```
create table address
(  
    id    bigint auto_increment,  
    city  varchar(255),  
    street varchar(255),  
    primary key (id)  
);
```



즉, Person이 연관관계의 주인이다!

1 : 1 관계의 @OneToOne 사용

```
@Entity
public class Person {

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

    private String name;

    @OneToOne
    private Address address;
```

```
@Entity
public class Address {

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

    private String city;

    private String street;

    @OneToOne(mappedBy = "address")
    private Person person;
```

1 : 1 관계의 @OneToOne 사용

```
@Entity
public class Person {

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

    private String name;

    @OneToOne
    private Address address;
```

```
@Entity
public class Address {

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

    private String city;

    private String street;

    @OneToOne(mappedBy = "address")
    private Person person;
```

연관관계의 주인이 아닌 쪽에 mappedBy 사용

연관관계의 주인 효과

객체가 연결되는 기준이 된다!

연관관계의 주인 효과

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

이 코드를 실행하면 DB에서 정상으로 테이블이 연결된다!

연관관계의 주인 효과

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

이 코드를 실행하면 DB에서 정상으로 테이블이 연결된다!
연관관계의 주인(Person)을 통해 객체가 연결 되었기 때문!

연관관계의 주인 효과

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

그렇다면 이 코드는 어떻게 될까?
실행시키면, 테이블 간의 연결은 되어 있지 않다!

연관관계의 주인 효과

- 1) 상대 테이블을 참조하고 있으면 연관관계의 주인
- 2) 연관관계의 주인이 아니면 mappedBy를 사용
- 3) 연관관계의 주인의 setter가 사용되어야만 테이블 연결

연관관계의 사용시 한 가지 주의해야 할 점

```
@Transactional
public void savePerson() {
    Person person = personRepository.save(new Person());
    Address address = addressRepository.save(new Address());
    person.setAddress(address);
    System.out.println(address.getPerson()); // null이 될 것이다!!
}
```

트랜잭션이 끝나지 않았을 때,
한 쪽만 연결해두면 반대 쪽은 알 수 없다!

해결책 : setter 한 번에 둘을 같이 이어주자!

N : 1 관계 - @ManyToOne과 @OneToMany

@ManyToOne를 단방향으로만 사용할 수도 있다!

@JoinColumn

연관관계의 주인이 활용할 수 있는 어노테이션.

필드의 이름이나 null 여부, 유일성 여부, 업데이트 여부 등을 지정

N : M 관계 - @ManyToMany

구조가 복잡하고, 테이블이 직관적으로 매핑되지 않아
사용하지 않는 것을 추천합니다!!

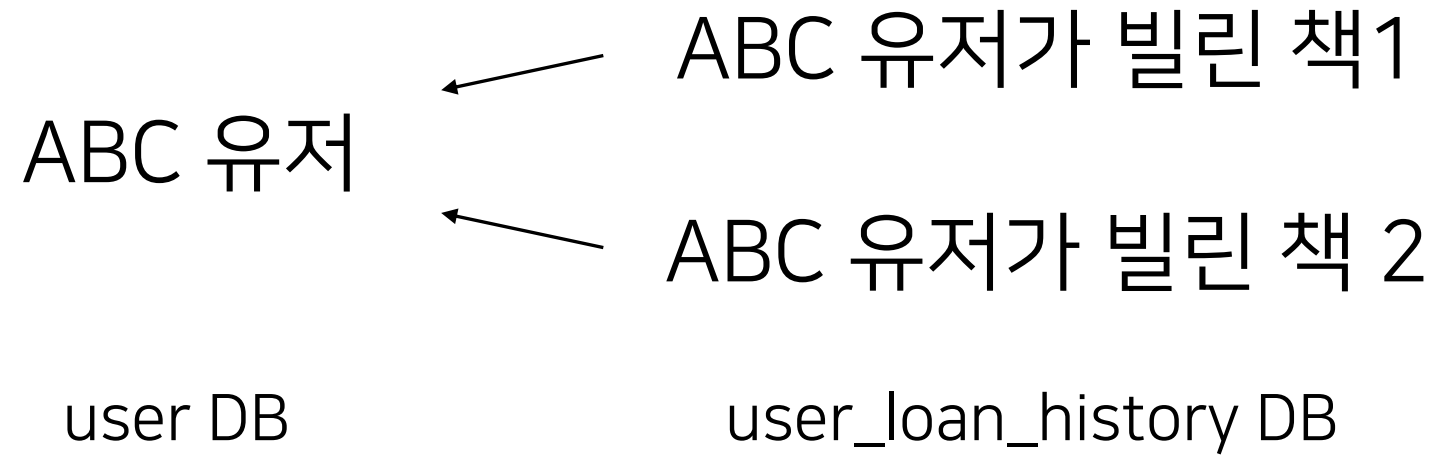
cascade 옵션

cascade : 폭포처럼 흐르다

cascade 옵션

한 객체가 저장되거나 삭제될 때,
그 변경이 폭포처럼 흘러
연결되어 있는 객체도 함께 저장되거나 삭제되는 기능

cascade 옵션



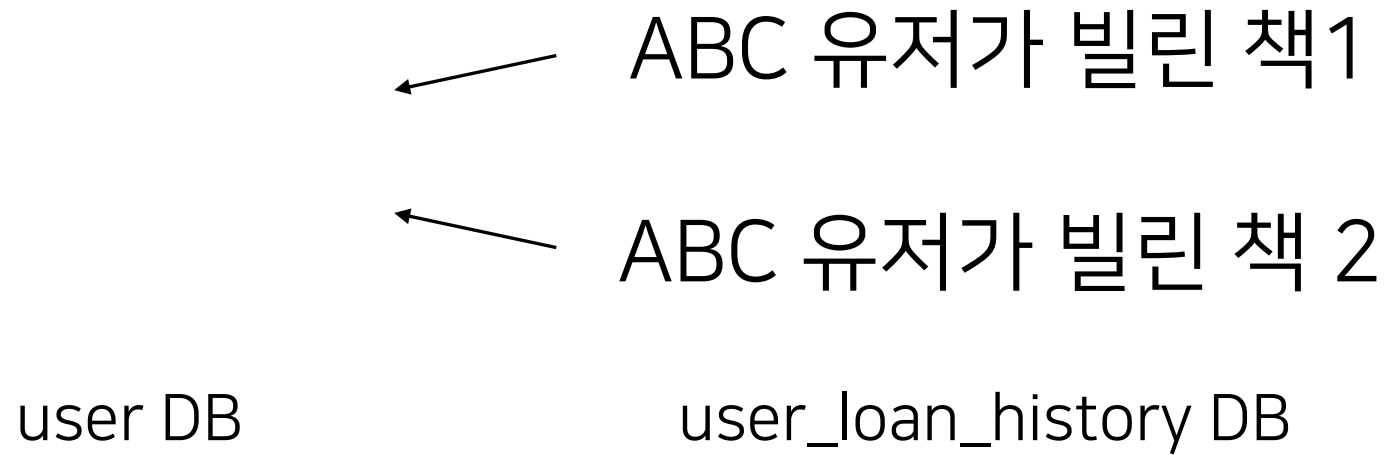
cascade 옵션

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

    userRepository.delete(user);
}
```

ABC 유저를 삭제하면 DB에서는 어떤 데이터가 삭제될까?!

cascade 옵션



cascade 옵션

유저가 빌린 책 기록은 그대로 남아 있는데
유저만 쏙 사라지는게 매우 이상하다!!

cascade 옵션

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

cascade 옵션

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

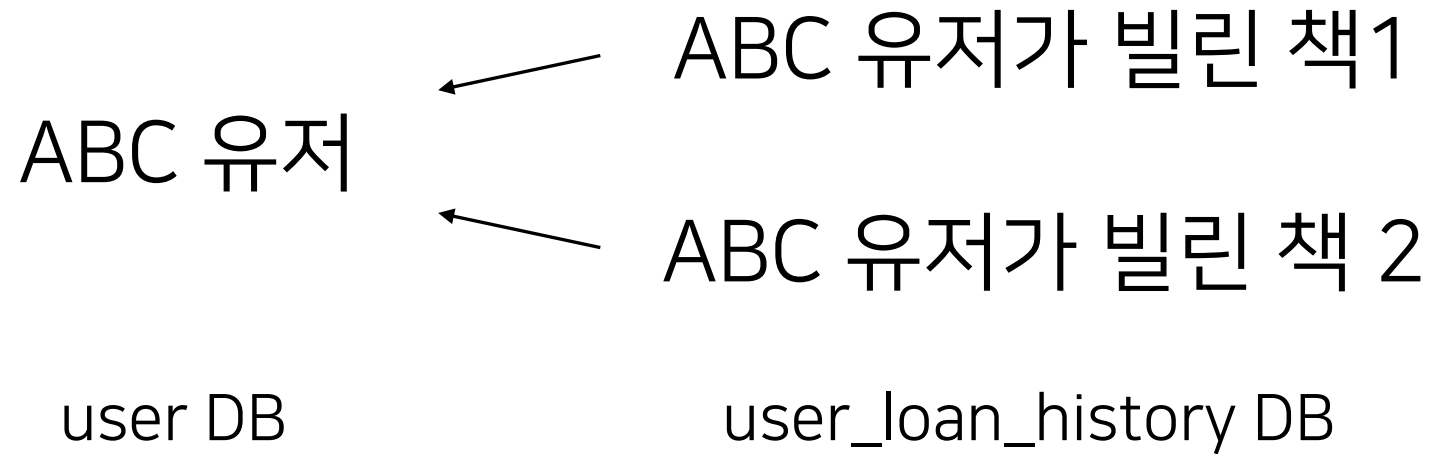
    userRepository.delete(user);
}
```

cascade 옵션을 붙이고 다시 실행해보자!

cascade 옵션

User가 삭제될 때 User와
연결된 UserLoanHistory까지
한 번에 사라지게 된다!

orphanRemoval 옵션



orphanRemoval 옵션

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

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


orphanRemoval 옵션

이때 DB는 어떻게 변할까요?!

orphanRemoval 옵션

아무런 변화가 없습니다!!

orphanRemoval 옵션

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

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

orphanRemoval 옵션

이럴 때 바로 orphanRemoval 옵션을 쓸 수 있다!

orphanRemoval 옵션

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

orphanRemoval 옵션

객체간의 관계가 끊어진 데이터를 자동으로 제거하는 옵션

관계가 끊어진 데이터 = orphan (고아)
제거 = removal

오늘 내용 정리!

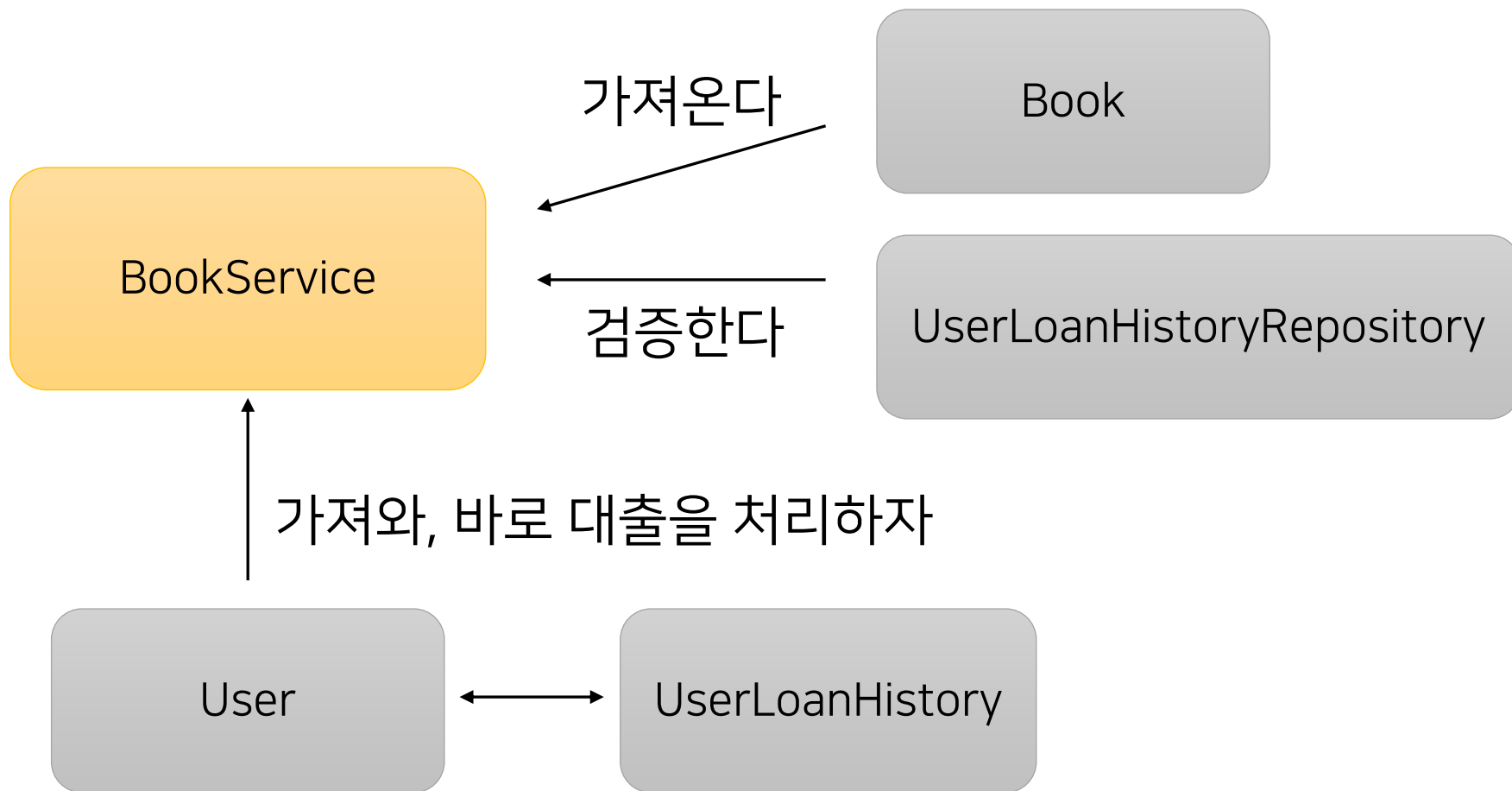
1. 상대 테이블을 가리키는 테이블이 연관관계의 주인이다.
연관관계의 주인이 아닌 객체는 mappedBy를 통해 주인에게 매여 있음을 표시해 주어야 한다.
2. 양쪽 모두 연관관계를 갖고 있을 때는 양쪽 모두 한 번에 맺어주는게 좋다.

오늘 내용 정리!

- 3. cascade 옵션을 활용하면, 저장이나 삭제를 할 때
연관관계에 놓인 테이블까지 함께 저장 또는 삭제가 이루어진다.
- 4. orphanRemoval 옵션을 활용하면, 연관관계가 끊어진 데이터를
자동으로 제거해준다.

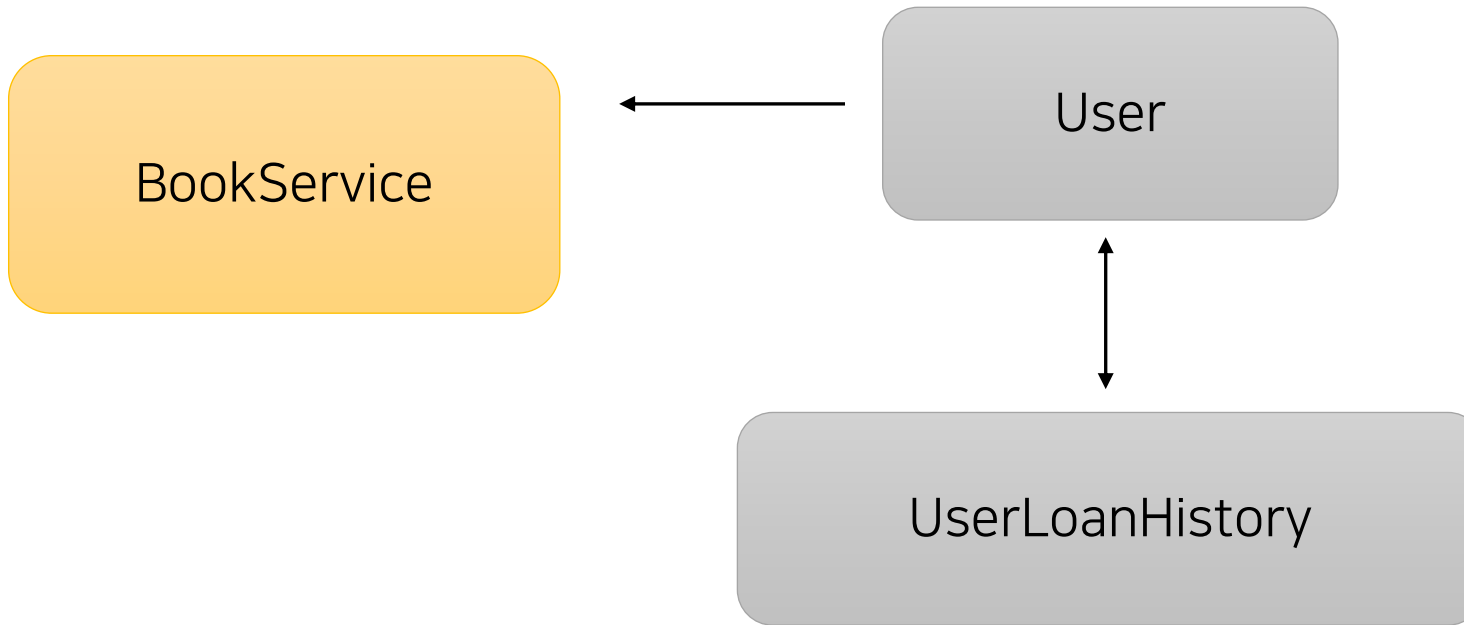
35강. 책 대출/반납 기능 리팩토링과 지연로딩

대출기능을 리팩토링해보자!



반납기능을 리팩토링해보자!

가져와 반납으로 처리한다



반납 기능 로직

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

함수형 프로그래밍을 할 수 있게, stream을 시작한다.

반납 기능 로직

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

들어오는 객체들 중에 다음 조건을 충족하는 것만 필터링 한다.

반납 기능 로직

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

UserLoanHistory 중 책 이름이 bookName과 같은 것!

반납 기능 로직

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

첫 번째로 해당하는 UserLoanHistory를 찾는다.

반납 기능 로직

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

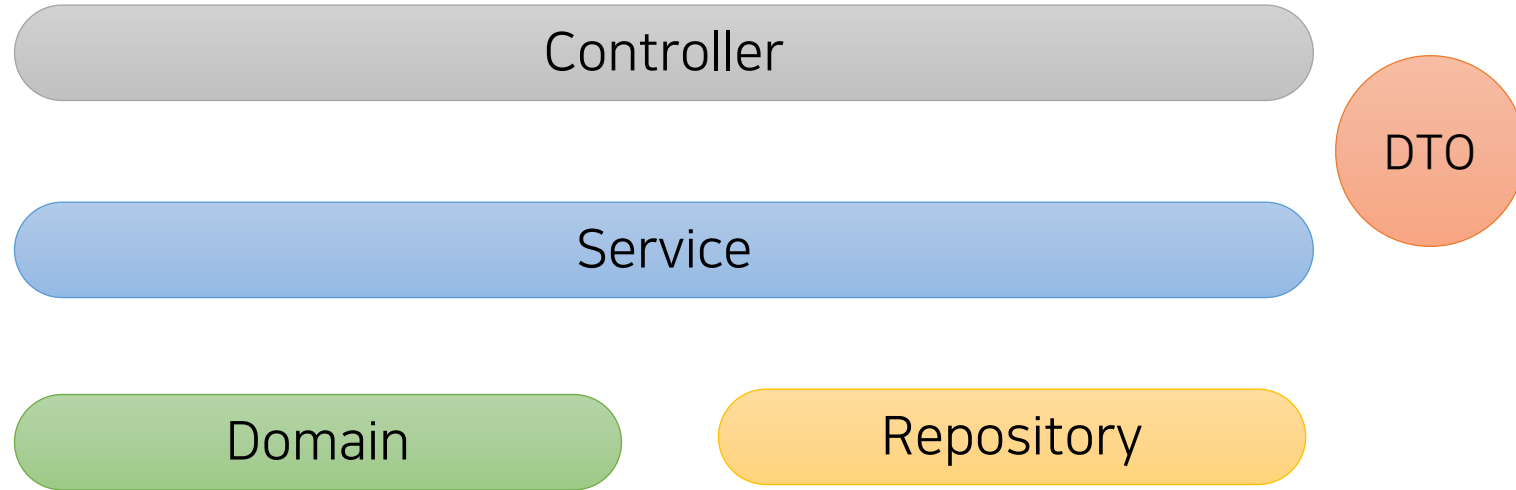
Optional을 제거하기 위해 없으면 예외를 던진다.

반납 기능 로직

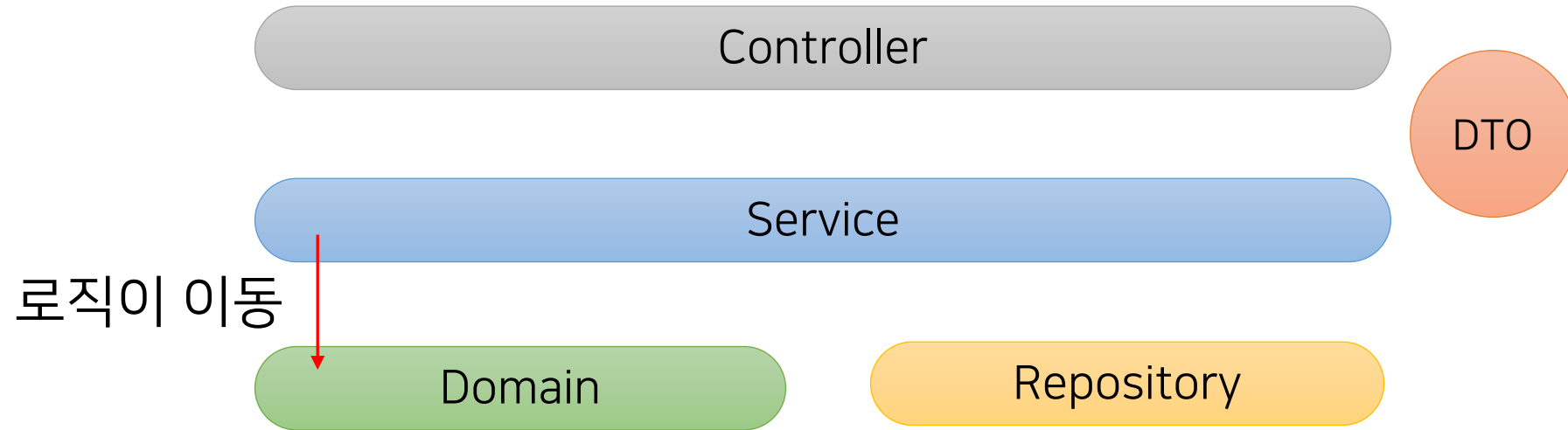
```
public void returnBook(String bookName) {  
    UserLoanHistory targetHistory = this.userLoanHistories.stream() Stream<UserLoanHistory>  
        .filter(history -> history.getBookName().equals(bookName))  
        .findFirst() Optional<UserLoanHistory>  
        .orElseThrow();  
    targetHistory.doReturn();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

그렇게 찾은 UserLoanHistory를 반납처리 한다.

도메인 계층에 비즈니스 로직이 들어갔다.



도메인 계층에 비즈니스 로직이 들어갔다.



영속성 컨텍스트의 4번째 능력!

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

User를 가져오는 부분과,
도메인 로직 실행 중간에 출력을 해보자!

영속성 컨텍스트의 4번째 능력!

```
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=?
```

...

영속성 컨텍스트의 4번째 능력!

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=?
```

...

영속성 컨텍스트의 4번째 능력!

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=?
```

...

영속성 컨텍스트의 4번째 능력!

```
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=?
```

...

User를 가장 먼저 가져오고

Hello 출력이 된다음

UserLoanHistory를 가져온다!

영속성 컨텍스트의 4번째 능력!

```
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=?
```

...

꼭 필요한 순간에 데이터를 로딩한다!!

지연 로딩 (Lazy Loading)

영속성 컨텍스트의 4번째 능력! - 지연 로딩

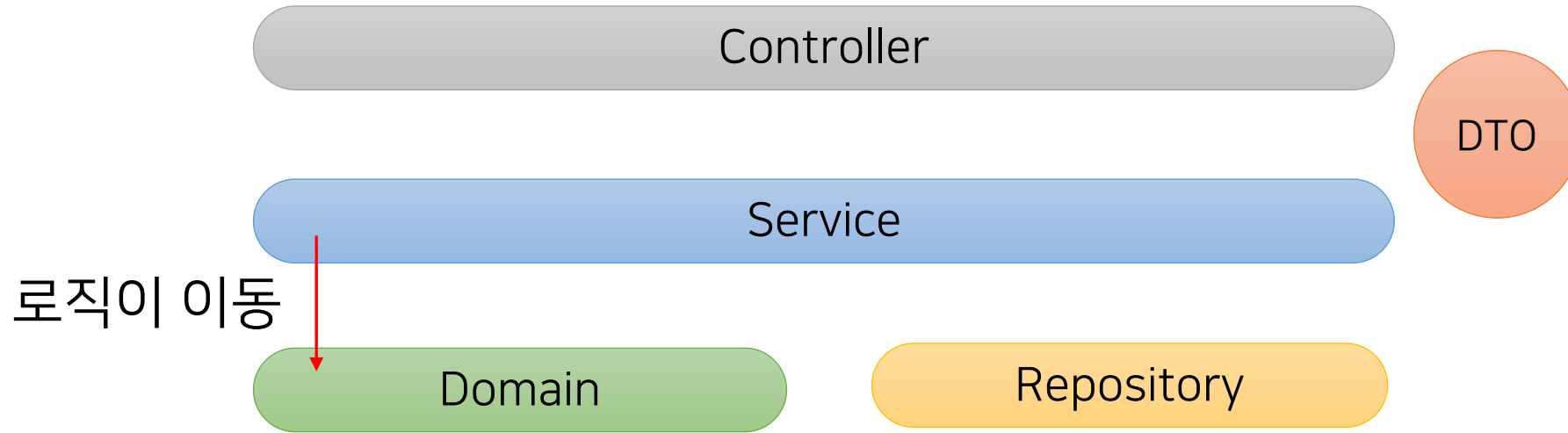
@OneToMany의 fetch 옵션

영속성 컨텍스트의 4번째 능력! - 지연 로딩

지연 로딩을 사용하게 되면,
연결되어 있는 객체를 꼭 필요한 순간에만 가져온다

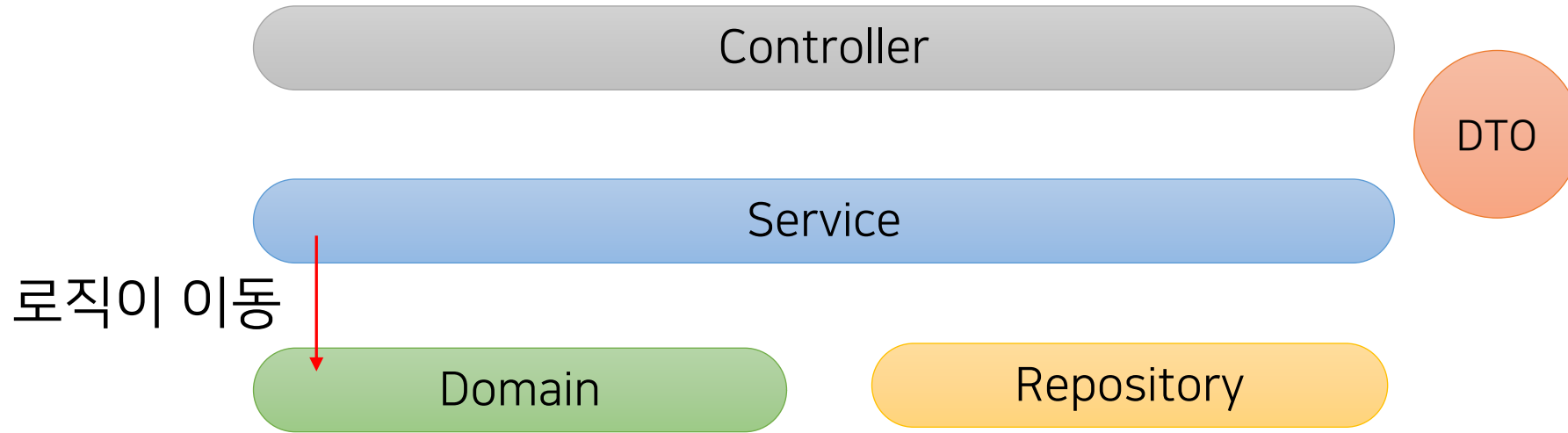
두 가지 추가적으로 생각해볼거리

[1] 연관관계를 사용하면 무엇이 좋을까?



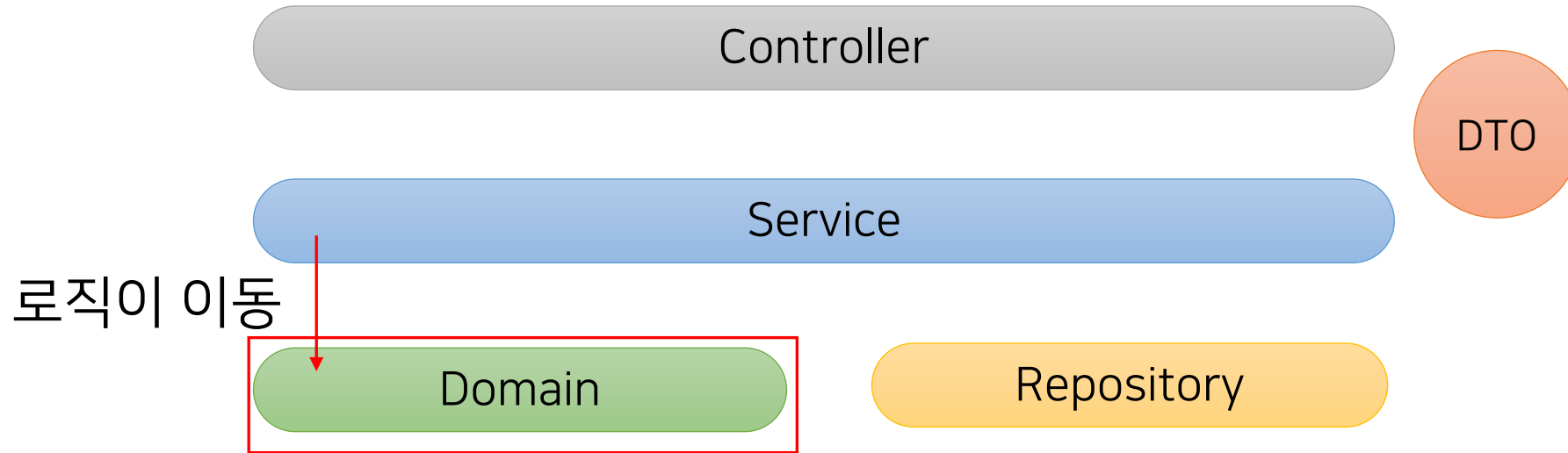
1) 각자의 역할에 집중하게 된다! (= 응집성)

[1] 연관관계를 사용하면 무엇이 좋을까?



- 1) 각자의 역할에 집중하게 된다! (= 응집성)
- 2) 새로운 개발자가 코드를 읽을 때 이해하기 쉬워진다.

[1] 연관관계를 사용하면 무엇이 좋을까?



- 1) 각자의 역할에 집중하게 된다! (= 응집성)
- 2) 새로운 개발자가 코드를 읽을 때 이해하기 쉬워진다.
- 3) 테스트 코드 작성이 쉬워진다.

[2] 연관관계를 사용하는 것이 항상 좋을까?

그렇지는 않다!

[2] 연관관계를 사용하는 것이 항상 좋을까?

지나치게 사용하면, 성능상의 문제가 생길 수도 있고
도메인 간의 복잡한 연결로 인해 시스템을 파악하기 어려워질 수 있다.

[2] 연관관계를 사용하는 것이 항상 좋을까?

또한 너무 얽혀 있으면,
A를 수정했을 때 B C D 까지 영향이 퍼지게 된다.

[2] 연관관계를 사용하는 것이 항상 좋을까?

비즈니스 요구사항, 기술적인 요구사항, 도메인 아키텍처 등
여러 부분을 고민해서 연관관계 사용을 선택해야 한다.

36강. Section 5 정리. 다음으로!

책 요구사항 구현하기

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 매핑하기 위한 방법을 이해하고, 연관관계를 사용해 개발할 때와 사용하지 않고 개발할 때의 차이점을 이해한다.

다음 Section에서는...!

배포

감사합니다