



# 섹션 4. 두 번째 요구사항 추가하기 - 도서 대출 현황

## 이번 섹션의 목표

25강. 유저 대출 현황 보여주기 - 프로덕션 코드 개발

26강. 유저 대출 현황 보여주기 - 테스트 코드 개발

27강. N + 1 문제와 N + 1 문제가 발생하는 이유

28강. SQL join에 대해 알아보자

29강. N + 1 문제를 해결하는 방법! fetch join

30강. 조금 더 깔끔한 코드로 변경하기

31강. 두 번째 요구사항 클리어!

## 부록

## 이번 섹션의 목표

1. join 쿼리의 종류와 차이점을 이해한다.
2. JPA N + 1 문제가 무엇이고 발생하는 원인을 이해한다.
3. N + 1 문제를 해결하기 위한 방법을 이해하고 활용할 수 있다.
4. 새로운 API를 만들 때 생길 수 있는 고민 포인트를 이해하고 적절한 감을 잡을 수 있다.

## 25강. 유저 대출 현황 보여주기 - 프로덕션 코드 개발

이번 시간에는 두 번째 추가 요구사항인 “유저 대출 현황 화면”에 대해서 알아보고, 가장 간단한 방법을 활용해 API를 구현해보자.

### 유저 대출 현황 화면

- 유저 대출 현황을 보여준다.

사용자 이름	책 이름	대여 상태
번개맨	엘리스를 찾아서	반납 완료
아이언맨	-	-

- 과거에 대출했던 기록과 현재 대출 중인 기록을 보여준다.
- 아무런 기록이 없는 유저도 화면에 보여져야 한다.

이 요구사항을 달성하기 위한 클라이언트는 먼저 개발이 끝나 있는 상황이다. 이제 우리가 해야 할 것은 클라이언트가 원하는 스펙에 맞춰 API를 만들어 주는 것이다. 클라이언트는 다음과 같은 스펙을 요구하였다.

### GET /user/loan

요청 : 파라미터 없음

#### 응답

```
[{
  "name": String,
  "books": [
    "name": String,
    "isReturn": Boolean
  ]
}, ...]
```

이 스펙에 맞춰 API를 구현해보자!

우선 DTO부터 작성해주겠다. `com.group.libraryapp.dto.user.response` 에 UserLoanHistoryResponse Class를 만들고, `books` 를 표현하기 위해 같은 파일 내에 BookHistoryResponse를 만들어 주었다.

```
package com.group.libraryapp.dto.user.response

data class UserLoanHistoryResponse(
```

```
    val name: String, // 유저 이름
    val books: List<BookHistoryResponse>
)

data class BookHistoryResponse(
    val name: String, // 책의 이름
    val isReturn: Boolean
)
```

코틀린에서는 위와 같이 한 파일에 여러 클래스를 만드는 것이 가능하다. 또한, 그 클래스들이 비슷한 특성을 가지고 있다면 권장되는 경우도 많다.



영상과 PPT를 함께 보시면 더욱 좋습니다 😊

이제 Controller부터 구현을 시작해주자. 자 그런데 한 가지 고민이 생긴다.

1. 새로운 Controller를 만들어야 할까?!
2. 아니면 기존의 Controller에 추가해야 할까?
  - a. 만약 기존 Controller에 추가 한다면 어떤 Controller에 추가해야 할까?!

Controller를 나누는 기준은 여러가지가 있다.

어떤 경우는 화면에서 사용되는 API 끼리 모아두는 경우도 있고, 화면과는 무관하게 같은 도메인을 대상으로 하는 API 끼리 모아두기도 한다. 또는 간혹 한 Controller는 하나의 API만을 가지고 있게 만들기도 한다.

각각의 방식에 대한 장단점을 알아보자.

## 화면에서 사용되는 API끼리 모아 둔다

### 장점

- 화면에서 어떤 API가 사용되는 한 눈에 알기 용이하다.

### 단점

- 한 API가 여러 화면에서 사용되면 위치가 애매하다.
- 서버 코드가 화면에 종속적이다.

## **동일한 도메인끼리 API를 모아 둔다**

### **장점**

- 화면 위치와 무관하게 서버 코드는 변경되지 않아도 된다.
- 비슷한 API끼리 모이게 되며 코드의 위치를 예측할 수 있다.

### **단점**

- 이 API가 어디서 사용되는지 서버 코드만 보고 알기는 어렵다.

## **1 API 1 Controller를 사용한다**

### **장점**

- 화면 위치와 무관하게 서버 코드는 변경되지 않아도 된다.

### **단점**

- 이 API가 어디서 사용되는지 서버 코드만 보고 알기는 어렵다.

지금까지 Controller에 API를 배치하는 여러 방법을 살펴보았다. 하지만 사실 근본적으로 중요한 것은 “프로젝트가 낯선 사람 입장에서 어떤 API가 어떤 Controller에 있는지 찾는 것”이다. Controller에 API가 어떤 식으로 분배되어 있건, Controller를 찾기만 하면 로직을 파고 들어갈 수 있기 때문이다.

이런 근본적인 문제를 해소하기 위해서 크게 3가지 방법을 사용할 수 있다.

1. IntelliJ 전체 검색
2. API full URL을 모아두는 Kotlin File 사용
3. IntelliJ 유료 버전의 Endpoints 사용

### [전체 검색 단축키]

- MAC : Command + Shift + F
- Windows / Linux : Ctrl + Shift + F

지금까지 Controller 작성을 위한 많은 고민들을 살펴보았다. 현재 API는 도메인 단위로 묶여 있으므로 우리는 UserController에 API를 추가할 것이다!

```
@GetMapping("/user/loan")
fun getUserLoanHistories(): List<UserLoanHistoryResponse> {
    return userService.getUserLoanHistories()
}
```

다음으로는 Service를 만들어 주자.

```
@Transactional(readOnly = true)
fun getUserLoanHistories(): List<UserLoanHistoryResponse> {
    return userRepository.findAll().map { user ->
        UserLoanHistoryResponse(
            name = user.name,
            books = user.userLoanHistories.map { history ->
                BookHistoryResponse(
                    name = history.bookName,
                    isReturn = history.status == UserLoanStatus.RETURNED
                )
            }
        )
    }
}
```

좋다~ 이제 프로덕션 코드는 개발이 모두 완료되었다! 다음 시간에는 이어서 테스트 코드를 작성해 동작을 확인하도록 하자!

## 26강. 유저 대출 현황 보여주기 - 테스트 코드 개발

이번 시간에는 지난 시간에 작성하였던 Service 코드에 대한 테스트 코드를 작성할 것이다. 이 Service에 대해서는 여러 case를 검증해야 한다. 검증해야 하는 몇 가지 경우를 살펴보자.

1. 사용자가 지금까지 한 번도 책을 빌리지 않은 경우
  - API 응답에 잘 포함되어 있어야 한다.
2. 사용자가 책을 빌리고 아직 반납하지 않은 경우
  - `isReturn` 값이 `false`로 잘 들어있어야 한다.

### 3. 사용자가 책을 빌리고 반납한 경우

- `isReturn` 값이 true로 잘 들어 있어야 한다.

### 4. 사용자가 책 여러권을 빌렸는데, 반납을 한 책도 있고 하지 않은 책도 있는 경우

- 중첩된 리스트에 여러권이 정상적으로 들어가 있어야 한다.

이중 2번, 3번은 4번을 검증할 때 자연스럽게 검증될 것이므로 1번 경우와 4번 경우에 대한 테스트를 작성해보자. 먼저 1번 경우이다.

```
@Test
@DisplayName("대출 기록이 없는 유저도 응답에 포함된다")
fun getUserLoanHistoriesTest1() {
    // given
    userRepository.save(User("A", null))

    // when
    val results = userService.getUserLoanHistories()

    // then
    assertThat(results).hasSize(1)
    assertThat(results[0].name).isEqualTo("A")
    assertThat(results[0].books).isEmpty()
}
```

테스트를 작성하였으면 가볍게 돌려보자~

다음은 4번 경우이다.

```
@Test
@DisplayName("대출 기록이 많은 유저의 응답이 정상 동작한다")
fun getUserLoanHistoriesTest2() {
    // given
    val savedUser = userRepository.save(User("A", null))
    userLoanHistoryRepository.saveAll(listOf(
        UserLoanHistory(savedUser, "책1", UserLoanStatus.LOANED),
        UserLoanHistory(savedUser, "책2", UserLoanStatus.LOANED),
        UserLoanHistory(savedUser, "책3", UserLoanStatus.RETURNED),
    ))

    // when
    val results = userService.getUserLoanHistories()

    // then
    assertThat(results).hasSize(1)
    assertThat(results[0].name).isEqualTo("A")
    assertThat(results[0].books).hasSize(3)
    assertThat(results[0].books).extracting("name").containsExactlyInAnyOrder("책1", "책2", "책3")
}
```

```
        assertThat(results[0].books).extracting("isReturn").containsExactlyInAnyOrder(false,
    false, true)
}
```

LOANED를 2개, RETURNED를 1개 해준 이유는 `containsExactlyInAnyOrder` 에서 LOANED이 false로 변환됨을 보장하기 위해서이다.

만약 각각 1개씩만 했다면, REURNED가 `isReturn = false` 로 변환되더라도 알아차리지 못했을 것이다.

테스트의 핵심 목적 중 하나는 '버그 방지'이므로, 테스트를 작성할 때에 프로덕션 코드가 잘못 변경된 것을 알아차릴 수 있는 테스트인지 꼭 고민해야 한다.

이렇게 2가지 테스트 코드를 작성해보았는데, 사실 아래와 같이 테스트 코드를 하나로 합치는 것도 한가지 방법이다

```
@Test
@DisplayName("방금 두 경우가 합쳐진 테스트")
fun getUserLoanHistoriesTest3() {
    // given
    val savedUsers = userRepository.saveAll(listOf(
        User("A", null),
        User("B", null),
    ))
    userLoanHistoryRepository.saveAll(listOf(
        UserLoanHistory(savedUsers[0], "책1", UserLoanStatus.LOANED),
        UserLoanHistory(savedUsers[0], "책2", UserLoanStatus.LOANED),
        UserLoanHistory(savedUsers[0], "책3", UserLoanStatus.RETURNED),
    ))

    // when
    val results = userService.getUserLoanHistories()

    // then
    assertThat(results).hasSize(2)
    val userAResult = results.first { it.name == "A" }
    assertThat(userAResult.books).hasSize(3)
    assertThat(userAResult.books).extracting("name").containsExactlyInAnyOrder("책1", "책
    2", "책3")
    assertThat(userAResult.books).extracting("isReturn").containsExactlyInAnyOrder(fals
    e, false, true)

    val userBResult = results.first { it.name == "B" }
    assertThat(userBResult.books).isEmpty()
}
```

하지만 개인적으로는 전자를 선호한다. 그 이유는

- 복잡한 테스트 1개보다, 간단한 테스트 2개가 유지보수하기 용이하기 때문이다.
- 또한 테스트가 합쳐지게 되면, 앞 부분에서 실패가 나는 경우 뒷 부분은 아예 검증되지 않는다.
  - 만약 테스트가 나눠져 있었다면 정확히 어떤 부분들이 문제 되는지 알 수 있다.

만약, 유저가 여러 명인 경우, 리스트가 정상 반환되는지 궁금하다면, 그 경우를 따로 작성하는 편이 좋다.

`getUserLoanHistoriesTest3()` 을 지워주고 전체 테스트도 한 번 돌려보자~!! 테스트가 모두 통과하였다면 매우 좋다 😊 우리는 새로운 요구사항을 완벽하게 구현하였다!!!

하지만 하늘 아래 완벽은 없다고 했던가..!

사실 이 구현은 한 가지 문제를 가지고 있다 😞 어떤 문제가 있는지, 왜 그런 문제가 발생하는 다음 시간에 알아보도록 하자!!

## 27강. N + 1 문제와 N + 1 문제가 발생하는 이유



영상과 PPT를 함께 보시면 더욱 좋습니다 😊

이번 시간에는 문제가 무엇인지, 왜 발생하는지 살펴볼 것이다.

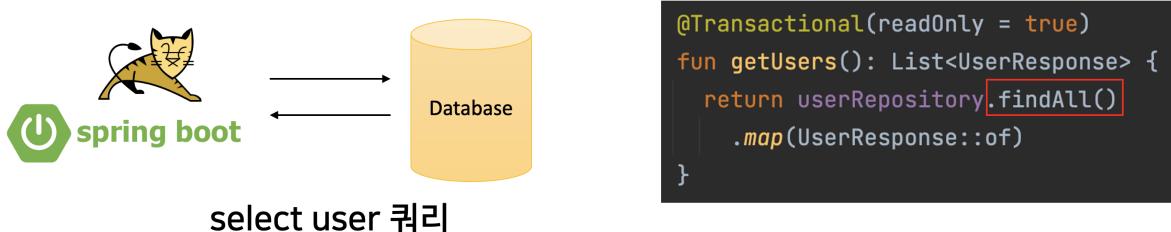
자 다시 한 번 Service 코드를 살펴보자.

```
@Transactional(readOnly = true)
fun getUserLoanHistories(): List<UserLoanHistoryResponse> {
    return userRepository.findAll().map { user ->
        UserLoanHistoryResponse(
            name = user.name,
            books = user.userLoanHistories.map { history ->
                BookHistoryResponse(
                    name = history.bookName,
                    isReturn = history.status == UserLoanStatus.RETURNED
                )
            }
        )
    }
}
```

# 서버 코드를 보고 Query를 생각할 수 있어야 한다!

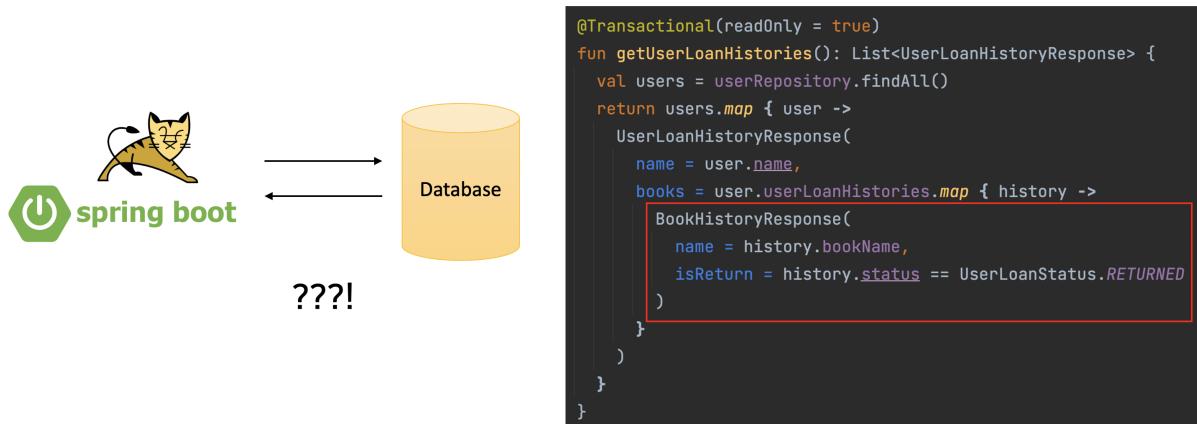


insert user 쿼리

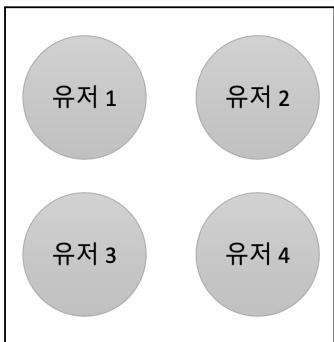


select user 쿼리





`select * from user;`



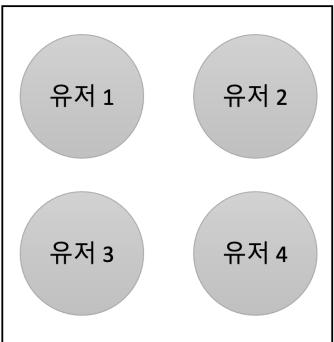
서버 메모리

```

@Transactional(readOnly = true)
fun getUserLoanHistories(): List<UserLoanHistoryResponse> {
    val users = userRepository.findAll()
    return users.map { user ->
        UserLoanHistoryResponse(
            name = user.name,
            books = user.userLoanHistories.map { history ->
                BookHistoryResponse(
                    name = history.bookName,
                    isReturn = history.status == UserLoanStatus.RETURNED
                )
            }
        )
    }
}

```

### 유저별로 반복



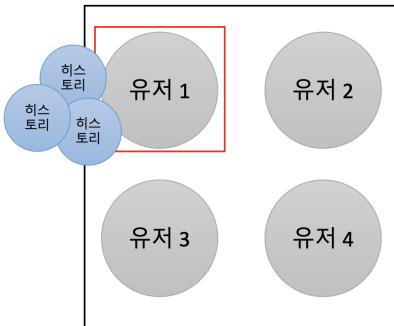
서버 메모리

```

@Transactional(readOnly = true)
fun getUserLoanHistories(): List<UserLoanHistoryResponse> {
    val users = userRepository.findAll()
    return users.map { user ->
        UserLoanHistoryResponse(
            name = user.name,
            books = user.userLoanHistories.map { history ->
                BookHistoryResponse(
                    name = history.bookName,
                    isReturn = history.status == UserLoanStatus.RETURNED
                )
            }
        )
    }
}

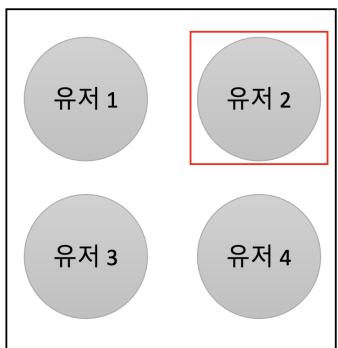
```

```
select * from  
user_loan_history  
where user_id = 1
```



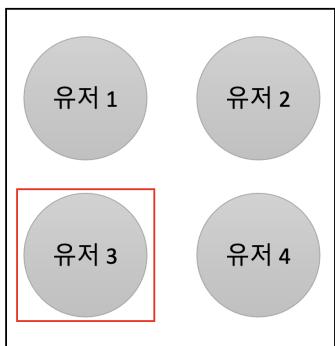
```
@Transactional(readOnly = true)  
fun getUserLoanHistories(): List<UserLoanHistoryResponse> {  
    val users = userRepository.findAll()  
    return users.map { user ->  
        UserLoanHistoryResponse(  
            name = user.name,  
            books = user.userLoanHistories.map { history ->  
                BookHistoryResponse(  
                    name = history.bookName,  
                    isReturn = history.status == UserLoanStatus.RETURNED  
                )  
            }  
        )  
    }  
}
```

```
select * from  
user_loan_history  
where user_id = 2
```



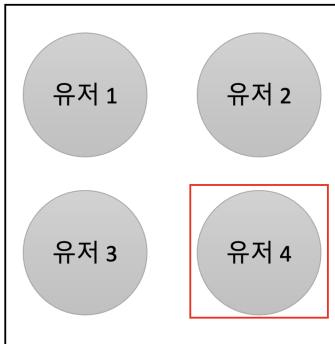
```
@Transactional(readOnly = true)  
fun getUserLoanHistories(): List<UserLoanHistoryResponse> {  
    val users = userRepository.findAll()  
    return users.map { user ->  
        UserLoanHistoryResponse(  
            name = user.name,  
            books = user.userLoanHistories.map { history ->  
                BookHistoryResponse(  
                    name = history.bookName,  
                    isReturn = history.status == UserLoanStatus.RETURNED  
                )  
            }  
        )  
    }  
}
```

```
select * from  
user_loan_history  
where user_id = 3
```



```
@Transactional(readOnly = true)  
fun getUserLoanHistories(): List<UserLoanHistoryResponse> {  
    val users = userRepository.findAll()  
    return users.map { user ->  
        UserLoanHistoryResponse(  
            name = user.name,  
            books = user.userLoanHistories.map { history ->  
                BookHistoryResponse(  
                    name = history.bookName,  
                    isReturn = history.status == UserLoanStatus.RETURNED  
                )  
            }  
        )  
    }  
}
```

```
select * from
user_loan_history
where user_id = 4
```



```
@Transactional(readOnly = true)
fun getUserLoanHistories(): List<UserLoanHistoryResponse> {
    val users = userRepository.findAll()
    return users.map { user ->
        UserLoanHistoryResponse(
            name = user.name,
            books = user.userLoanHistories.map { history ->
                BookHistoryResponse(
                    name = history.bookName,
                    isReturn = history.status == UserLoanStatus.RETURNED
                )
            }
        )
    }
}
```

때문에 유저가 100명이라면 총 101번의 쿼리가 발생하고, 유저가 1000명이라면 총 1001번의 쿼리가 발생한다.

조회 한 번에 쿼리가 많이 발생하게 될 수록, DB에 부하가 심해져 성능적인 문제가 생길 수밖에 없다. 이런 문제를 N + 1 문제라고 부른다.

디버거를 사용해 조금 더 확인해보자. 디버거 사용법의 경우는 아래 링크를 참조하면 좋다.

(TODO - 유튜브 링크)

User를 기준으로 해당 사용자의 List<UserLoanHistory>를 계속해서 가져오는 이유는, User 객체까지는 온전히 로딩이 잘 되지만 UserLoanHistory는 가짜 객체가 들어있기 때문이다. 이 가짜객체에 실제 접근을 하는 타이밍에 Hibernate가 DB에 SQL을 보내 온전히 로딩하는 방식이다.

이러한 방식을 Lazy Fetching이라 한다.

그렇다면 이러한 문제는 어떻게 해결할 수 있을까?! 이 성능적인 문제를 해결하기 위해서는 먼저 SQL join에 대한 이해가 필요하다. 다음 시간에는 SQL join의 종류와 특징을 알아보자.

## 28강. SQL join에 대해 알아보자



영상과 PPT를 함께 보시면 더욱 좋습니다 😊

이번 시간에는 SQL 쿼리의 종류 중 inner join과 left join에 대해서 알아볼 것이다.

User 테이블과 UserLoanHistory 테이블을 예시로 설명할 것이고 이해를 돋기 위해, 표 형식으로 데이터를 준비하였다.

유저 테이블		유저 도서 대출 테이블			
id	이름	id	유저 id	책 이름	반납 여부
1	A	1	1	책 1	반납 완료
2	B	2	1	책 2	대출중
3	C	3	2	책 3	대출중

데이터를 해석하면 다음과 같다.

- A 유저는 책 1를 빌렸다 반납하였고 책 2는 대출 중이다.
- B 유저는 책 3을 대출 중이다.
- C 유저는 아직까지 책을 빌린 적이 없다.

이 상황에서 단순한 select 쿼리 말고, 두 테이블의 결과를 한 번에 보고 싶다고 해보자.

즉 유저 테이블의 id 와 유저 도서 대출 테이블의 id를 연결해서 보는 것이다. 이럴 때 사용할 수 있는 것이 join이다.

```
select * from user
join user_loan_history on user.id = user_loan_history.user_id
```

- user : 기준 테이블
- user\_loan\_history : 합쳐지는 테이블
- user.id = user\_loan\_history.user\_id : 합쳐지는 조건

좋다. join의 결과는 다음과 같다.

유저 테이블 - 유저 도서 대출 테이블 join 결과					
유저테이블.id	유저테이블.이름	도서대출테이블.id	도서대출테이블.유저id	도서대출테이블.책이름	도서대출테이블.반납여부
1	A	1	1	책1	반납 완료
1	A	2	1	책2	대출중
2	B	3	2	책3	대출중

쿼리를 작성할 때 다음과 같이 테이블별로 별칭을 줄 수 있다.

```
select * from user u
join user_loan_history ulh on u.id = ulh.user_id
```

자 그런데 결과가 이상하다. 3번 유저는 왜 결과에서 빠졌는가?!!

그 이유는 우리가 사용한 join 쿼리가 사실은 'inner join' 이기 때문이다.

inner join은 join을 하고 싶은 테이블 양쪽에 데이터가 모두 존재하는 경우에만 하나로 합쳐 준다.

```
select * from user u
inner join user_loan_history ulh on u.id = ulh.user_id
```

그렇다면 3번 유저의 데이터도 보고 싶다면 어떻게 해야 할까?

inner join 대신 left join을 사용해주면 된다. left join은 기준 테이블에만 데이터가 있고, join 대상 테이블에 데이터가 없더라도 합쳐진 결과를 보여준다.

```
select * from user u
left join user_loan_history ulh on u.id = ulh.user_id
```

유저 테이블 - 유저 도서 대출 테이블 left join 결과					
유저테이블.id	유저테이블.이름	도서대출테이블.id	도서대출테이블.유저id	도서대출테이블.책이름	도서대출테이블.반납여부
1	A	1	1	책1	반납 완료
1	A	2	1	책2	대출중
2	B	3	2	책3	대출중
3	C	null	null	null	null

left join에서 주의할 점으로는 '기준 테이블에 데이터가 있고 대상 테이블에 없는 경우'를 조회 결과에 포함한다는 점이다.

즉, 기준 테이블에 데이터가 없다면 조회 결과에 포함되지 않는다.

```
select * from user user_loan_history ulh
  left join user u on u.id = ulh.user_id
```

`user_loan_history` 테이블을 기준으로 데이터를 가져오는 경우에는 3번 유저를 볼 수 없다는 의미이다.

자 이상으로 SQL의 inner join, left join이 무엇인지, 그리고 각각의 특징은 어떻게 되는지를 살펴보았다. 다음 시간에는 join을 활용해 N + 1 문제를 해결해보자!

## 29강. N + 1 문제를 해결하는 방법! fetch join

이번 시간에는 지난 시간에 살펴보았던 join을 활용해 N + 1 문제를 해결해보자!

N + 1 문제가 발생하는 이유를 한 줄로 요약하면 다음과 같았다.

- User 데이터만 최초 1회 가져온 다음,  
각 User 별로 UserLoanHistory 정보들을 추가로 가져온다.

그러니 이런 발상을 해볼 수 있다! 최초 유저 정보만 쿼리를 통해 가져오지 않고, join을 활용해 유저 정보와, 유저 대출 기록 정보를 같이 가져온다면?!

바로 한 번 작성해보자! Repository에서 직접 쿼리를 작성하려면 `@Query` 어노테이션을 사용해야 한다.

```
@Query
fun findAllWithHistories(): List<User>
```

이제 `@Query` 안에 쿼리를 작성해주자! 순수한 SQL은 아니고 JPA에서 만든 JPQL이 여기 들어가야 한다.

단계별로 구현해보면서 필요한 부분을 살펴보자. 우선 User만 모두 가져와보자. SQL로는 아래와 같았다.

```
select * from user;
```

JPQL은 아래와 같다. \* 대신 User의 별칭인 u를 사용해 주었다.

```
@Query("SELECT u FROM User u")
fun findAllWithHistories(): List<User>
```

이제 join을 해보자! 대출 기록이 없는 유저도 포함되어야 하므로 left join이어야 한다.

```
@Query("SELECT u FROM User u LEFT JOIN u.userLoanHistories")
fun findAllWithHistories(): List<User>
```

LEFT JOIN 뒤에 유저에 필드로 존재하는 `u.userLoanHistories` 라고 적어 주었다.

여기까지 되었다면 Service 로직에서 `UserRepository.findAllWithHistories()` 를 호출하도록 변경하고, 테스트를 돌려보자! 테스트를 통해 실제 나가는 Query를 확인할 수 있다.  
`@AfterEach` 에서 날려주는 쿼리와 구분하기 위해 간단한 로그를 찍어 주었다.

```
@AfterEach
fun clean() {
    println("UserServiceTest CLEAN")
    userRepository.deleteAll()
}
```

테스트를 돌려보니 두 가지가 이상하다.

- User 결과는 1개여야 하는데, 3개가 들어 있다 → 테스트가 실패했다!
- 여전히 UserLoanHistory를 가져온다

우선 첫 번째 이슈를 해결해보자! 1개가 아닌 3개가 들어 있는 이유는 left join의 결과를 생각해보면 당연하다.

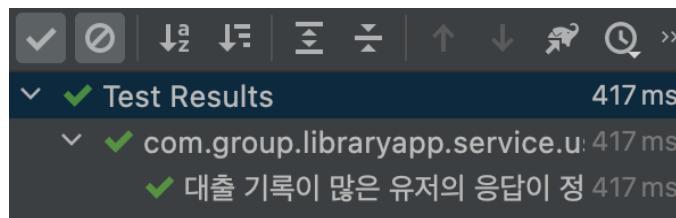
유저 테이블 - 유저 도서 대출 테이블 left join 결과					
유저테이블.id	유저테이블.이름	도서대출테이블.id	도서대출테이블.유저id	도서대출테이블.책이름	도서대출테이블.반납여부
1	A	1	1	책1	반납 완료
1	A	2	1	책2	대출중
2	B	3	2	책3	대출중
3	C	null	null	null	null

지난 시간에 살펴보았던 left join 테이블에서 유저가 한 명인 경우라도, 여러 줄의 결과가 나온다. 이 결과가 모두 `List<User>`에 담겨 오는 것이다.

이를 해결하기 위해서는 SELECT 뒤에 DISTINCT를 붙여 주어야 한다. DISTINCT는 결과 `List<User>`에 같은 유저라면 데이터를 1개만 넣게 해주는 효과가 있다.

```
@Query("SELECT DISTINCT u FROM User u LEFT JOIN u.userLoanHistories")
fun findAllWithHistories(): List<User>
```

이렇게 DISTINCT를 추가하고 테스트를 다시 돌려보자! 통과하는 것을 확인할 수 있다~!



다음 두 번째 이슈를 해결해보자. 분명 우리는 LEFT JOIN을 사용하였는데 N + 1 문제가 여전히 발생하고 있다. 그 이유는 join을 통해 UserLoanHistory 정보를 가져와, 실제 UserLoanHistory 객체를 만들어 주어야 하기 때문이다.

UserLoanHistory 객체까지 만들어주려면 JOIN 뒤에 FETCH를 붙이도록 하자!

```
@Query("SELECT DISTINCT u FROM User u LEFT JOIN FETCH u.userLoanHistories")
fun findAllWithHistories(): List<User>
```

이제 다시 테스트를 돌려 쿼리를 확인해보면 우리가 원하는대로 한 번만 쿼리가 나오는 것을 확인할 수 있다!! 🎉

매우 좋다~ 우리는 Lazy Fetching으로 인한 N + 1 문제를 fetch join 으로 해결하였다~ 😊

다음 시간에는 간단하게 코드를 조금 더 깔끔하게 변경해보자.

## 30강. 조금 더 깔끔한 코드로 변경하기

이번 시간에는 코드를 조금 더 깔끔하게 변경해볼 것이다. 지금의 Service 로직도 함수형 프로그래밍을 사용해서 깔끔한 편이긴 하지만, 관심사를 분리함으로써 조금 더 Clean하게 변경해보자.

우리가 활용할 방법은 바로 정적 팩토리 메소드이다. 두 가지 DTO에 정적 팩토리를 만들자.

- `BookHistoryResponse` 에 정적 팩토리 메소드 `of`를 만들고 `UserLoanHistory` 를 파라미터로 받게 하자.
- `UserLoanHistoryResponse` 에 정적 팩토리 메소드 `of`를 만들고 `User` 를 파라미터로 받게 하자.

```
data class BookHistoryResponse(  
    val name: String, // 책의 이름  
  
    @get:JsonProperty("isReturn")  
    val isReturn: Boolean  
) {  
    companion object {  
        fun of(userLoanHistory: UserLoanHistory): BookHistoryResponse {  
            return BookHistoryResponse(  
                name = userLoanHistory.bookName,  
                isReturn = userLoanHistory.status == UserLoanStatus.RETURNED  
            )  
        }  
    }  
}
```

`isReturn` 의 경우 `UserLoanHistory`에 custom getter 프로퍼티를 만들어 주었다. 이렇게 도메인 객체에 로직을 넣어두면 나중에 필요한 경우 재활용할 수 있게 된다.

```
@Entity  
class UserLoanHistory constructor(  
    // 생략 ...  
) {  
    val isReturn: Boolean  
        get() = this.status == UserLoanStatus.RETURNED  
}
```

```
data class UserLoanHistoryResponse(  
    val name: String, // 유저 이름  
    val books: List<BookHistoryResponse>  
) {  
    companion object {
```

```

        fun of(user: User): UserLoanHistoryResponse {
            return UserLoanHistoryResponse(
                name = user.name,
                books = user.userLoanHistories.map { history ->
                    BookHistoryResponse.of(history)
                }
            )
        }
    }
}

```

이제 Service 로직에서는 정적 팩토리만 호출하면 되므로 코드가 깔끔하게 변경된다.

```

@Transactional(readOnly = true)
fun getUserLoanHistories(): List<UserLoanHistoryResponse> {
    val users = userRepository.findAllWithHistories()
    return users.map { user -> UserLoanHistoryResponse.of(user) }
}

```

Service 계층은 본래의 목적인 1) 트랜잭션 관리 2) Repository를 통한 도메인 조회 및 제어에 집중하게 된 것이고 데이터를 DTO에 매핑하는 역할은 DTO 쪽에 넘기게 된 것이다.

또한 depth가 깊고, 세로로 길어 한 눈에 보기 힘들던 코드가 간단한 코드 조각들로 변경되며 유지보수가 용이해졌다.

리팩토링을 다 마무리 하였다면, 테스트 코드를 돌려보자. 역시 통과한다~! 이제 다음 시간에는 이번 Section에서 다루었던 내용들을 간단히 정리해 보도록 하자.

## 31강. 두 번째 요구사항 클리어!

우리는 아래와 같은 두 번째 요구사항을 완벽하게 구현하였다.

### 유저 대출 현황 화면

- 유저 대출 현황을 보여준다.

사용자 이름	책 이름	대여 상태
번개맨	엘리스를 찾아서	반납 완료
아이언맨	-	-

- 과거에 대출했던 기록과 현재 대출 중인 기록을 보여준다.
- 아무런 기록이 없는 유저도 화면에 보여져야 한다.

덕분에 이번 [Section 4. 두 번째 요구사항 추가하기 - 도서 대출 현황](#) 을 통해 다음과 같은 내용을 배울 수 있었다.

1. 새로운 기능을 추가할 때, 위치에 관한 고민과 선택에 따른 장단점
2. 복잡한 기능을 추가할 때, 테스트 코드를 작성하는 방법
3. SQL의 inner join, left join
4. N + 1 문제를 해결하기 위해 fetch join을 사용하는 방법

이제 다음 추가 기능을 구현하러 가보자!! 🚀🔥

## 부록

19강에서 Kotlin Jackson Module을 사용하지 않은 경우 발생하는 문제와 원인, 그리고 해결 방법을 알아보자.

<http://localhost:8080/v2/index.html> 에 접속해서 확인을 위한 데이터를 넣어 주고, 히스토리를 들어가보자.

데이터는 다음과 같이 넣어주었다.

- 사용자 2명 - A, B
- 책 2권 - 책1, 책2
- A는 책 1을 빌려 반납하였고, 책 2는 현재 대출 중이다.

사용자 이름	책 이름	대여 상태
A	책1	-
A	책2	-
B	-	-

이상하다. 대여 상태가 나오지 않는다! 분명 `isReturn` 을 잘 만들어 주었는데 왜 그럴까?  
개발자 도구를 활용해 응답이 반환되는 Response를 살펴보자.

```
▼ 0: {name: "A", books: [{name: "책1", return: true}, {name: "책2", return: false}]}
  ▼ books: [{name: "책1", return: true}, {name: "책2", return: false}]
    ▼ 0: {name: "책1", return: true}
      name: "책1"
      return: true
    ▼ 1: {name: "책2", return: false}
      name: "책2"
      return: false
    name: "A"
  ▼ 1: {name: "B", books: []}
    books: []
    name: "B"
```

응답에는 `isReturn` 대신 `return` 이라는 이름이 있는 것을 확인할 수 있다.

이러한 이유는 바로 Jackson의 동작 원리에 있다. Jackson은 객체를 JSON 형태로 바꾸어 클라이언트로 전달해주는 역할을 한다. 이때 Jackson은 객체의 getter를 보고 이름을 정하게 되는데 원래는 다음과 같다.

- `getApple(): String` → apple
- `isReturn(): Boolean` → return

반환 타입이 Boolean인 getter는 관례상 `is필드이름` 이 사용되고 때문에 `필드이름` 만 JSON에 사용하게 되는 것이다.

이는 Java와 Lombok을 함께 사용하더라도 동일하게 적용된다.

해결하는 방법은 다음과 같다.

```
data class BookHistoryResponse(  
    val name: String, // 책의 이름  
  
    @get:JsonProperty("isReturn")  
    val isReturn: Boolean  
)
```

Jackson에서 제공하는 어노테이션인 `@JsonProperty` 를 사용해, 필드 이름을 직접 지정해주어야 한다.

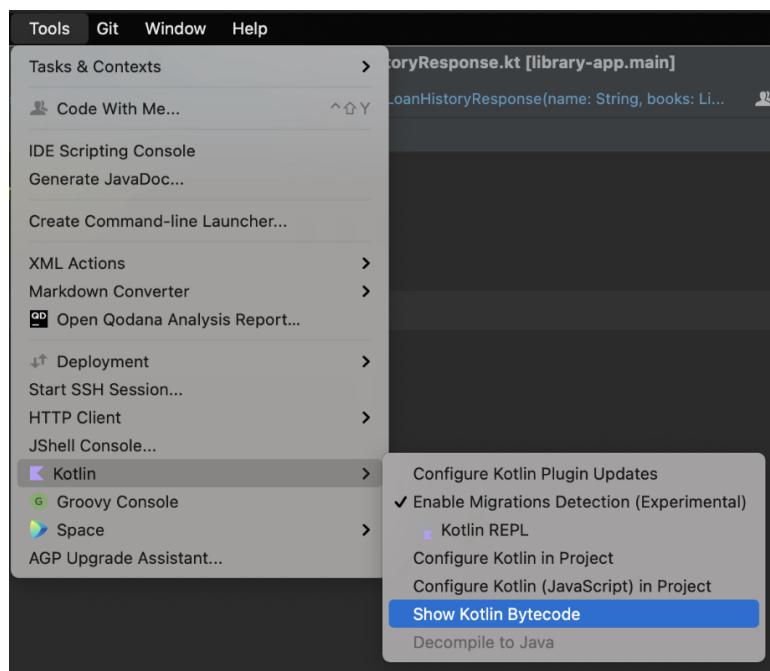
코틀린에서는

```
val isReturn: Boolean
```

만 보았을 때 여러 의미가 있을 수 있다.

1. isReturn이라는 생성자 파라미터일 수도 있고
2. isReturn이라는 필드일 수도 있고
3. isReturn이라는 getter일 수도 있다.

때문에 `@get:` 을 앞에 붙여, 어디에 어노테이션을 붙이려는지 명확히 보여주어야 한다.  
Decompile을 활용해보면 쉽게 감이 올 것이다.



Kotlin Bytecode

Decompile  Inline  Optimization  Assertions  IR Target: 1.8

```

1 // =====com/group/libraryapp/dto/use
2 // class version 52.0 (52)
3 // access flags 0x31
4 public final class com/group/libraryapp/dto/use
5
6
7     // access flags 0x12
8     private final Ljava/lang/String; name
9     @Lorg/jetbrains/annotations/NotNull;() // inv
10
11    // access flags 0x11

```

응답 DTO에 어노테이션을 추가해주었다면 이제 화면에 들어가 데이터를 확인해보자. 정상적으로 노출되는 것을 확인할 수 있다~! 🎉

등록하기	목록	히스토리	통계
사용자 이름	책 이름	대여 상태	
A	책1	반납 완료	
A	책2	대출 중	
B	-	-	

프로퍼티의 이름을 지정해줄 수 있는 `@JsonProperty` 외에 `@JsonIgnore` 도 알아두면 좋다!  
이 필드는 객체에 존재하는 특정 필드를 JSON으로 옮기지 않을 때 사용한다.