#4 두 번째 요구사항 추가하기 - 도서 대출 현황

1. join 쿼리의 종류와 차이점을 이해한다.

2. JPA N + 1 문제가 무엇이고 발생하는 원인을 이해한다.

#4 두 번째 요구사항 추가하기 - 도서 대출 현황

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

요구사항 2 확인

유저 대출 현황 화면

- 유저 대출 현황을 보여준다.

등록하기 목록 히스토리 통계

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.

현재 Client는 진작 개발되어 있다

때문에 API 스펙도 정해져 있고,
우리는 이 스펙에 맞추어 서버 개발을 하면 된다!

GET /user/loan

요청 : 파라미터 없음

응답

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

새로운 API를 만들 때 코드의 위치를 어떻게 해야 할까?!

Controller를 예로 들어 보자

새로운 API를 만들 때 코드의 위치를 어떻게 해야 할까?!

1. 새로운 Controller를 만들어야 할까?

2. 기존의 Controller에 추가해야 할까?

3. 기존의 Controller에 추가한다면
   어떤 Controller에 추가해야 할까?!

Controller를 구분하는 3가지 기준

1. 화면에서 사용되는 API끼리 모아 둔다

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.

Controller를 구분하는 3가지 기준

1. 화면에서 사용되는 API끼리 모아 둔다

2. 동일한 도메인끼리 API를 모아 둔다

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.

Controller를 구분하는 3가지 기준

1. 화면에서 사용되는 API끼리 모아 둔다

2. 동일한 도메인끼리 API를 모아 둔다

3. (간혹) 1 API 1 Controller를 사용한다

1 API 1 Controller를 사용한다

장점

- 화면 위치와 무관하게 서버 코드는 변경되지 않아도 된다.

단점

- 이 API가 어디서 사용되는지 서버 코드만 보고 알기는 어렵다.

새로운 API를 만들 때 코드의 위치를 어떻게 해야 할까?!

프로젝트가 낯선 사람 입장에서
어떤 기능에 대한 코드가 어디 있는지 찾을 수 있는 것

새로운 API를 만들 때 코드의 위치를 어떻게 해야 할까?!

Controller를 찾으면 사용되는
Service, Repository, Domain을 확인할 수 있다!

새로운 API를 만들 때 코드의 위치를 어떻게 해야 할까?!

프로젝트가 낯선 사람 입장에서
Controller가 어디 있는지 찾을 수 있는 것

Controller를 찾을 수 있는 몇 가지 방법

전제 : API를 알고 있다

Controller를 찾을 수 있는 몇 가지 방법

Find in Files 38 matches in 3 files

File mask: *.yml

/user

In Project Module Directory Scope

```
@PostMapping("/user")                    UserController.kt 21
@GetMapping("/user")                     UserController.kt 26
@PutMapping("/user")                     UserController.kt 31
@DeleteMapping("/user")                  UserController.kt 36
@GetMapping("/user/loan")                UserController.kt 41
```

UserController.kt src/main/kotlin/com/group/libraryapp/controller/user

```
18    private val userService: UserService,
19  ) {
20
21    @PostMapping("/user")
22    fun saveUser(@RequestBody request: UserCreateRequest) {
23      userService.saveUser(request)
24    }
25
26    @GetMapping("/user")
```

Open results in new ...

Open in Find Window

Controller를 찾을 수 있는 몇 가지 방법

URL들을 모아두기

Controller를 찾을 수 있는 몇 가지 방법

(유료) IntelliJ의 endpoints

Controller를 찾을 수 있는 몇 가지 방법

Endpoints

Module: All Type: All Framework: All

library-app.main

/user [GET] HTTP Server Spring MVC Controllers	UserController
/user [PUT] HTTP Server Spring MVC Controllers	UserController
/user [POST] HTTP Server Spring MVC Controllers	UserController
/user [DELETE] HTTP Server Spring MVC Controllers	UserController
/user/loan [GET] HTTP Server Spring MVC Controllers	UserController
/book [POST] HTTP Server Spring MVC Controllers	BookController
/book/loan [POST] HTTP Server Spring MVC Controllers	BookController
/book/return [PUT] HTTP Server Spring MVC Controllers	BookController

26강. 유저 대출 현황 보여주기 - 테스트 코드 개발

무엇을 검증해야 할까?!

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 false로 잘 들어 있어야 한다

무엇을 검증해야 할까?!

3. 사용자가 책을 빌리고 반납한 경우
   isReturn 값이 true로 잘 들어 있어야 한다

4. 사용자가 책을 여러권 빌렸는데,
   반납을 한 책도 있고 하지 않은 책도 있는 경우
   중첩된 리스트에 여러 권이 정상적으로 들어가 있어야 한다

무엇을 검증해야 할까?!

2번, 3번은 4번을 검증할 때 자연스럽게 검증될 것이므로
1번 case와 4변 case에 대한 테스트를 작성해보자.

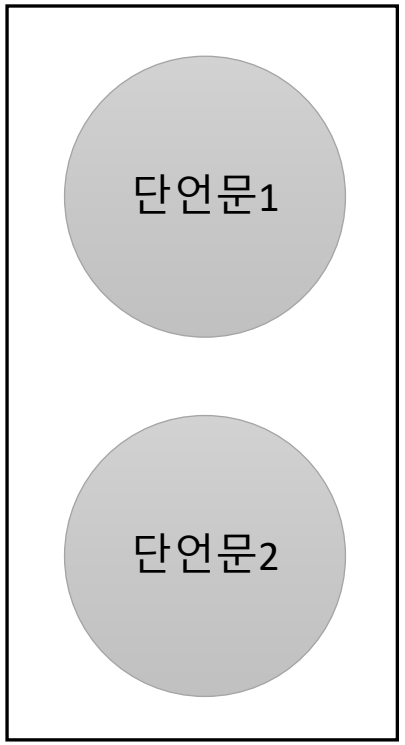
큰 테스트 코드 1개보다 작은 테스트 코드 2개가 나은 이유

복잡한 테스트 1개 보다, 간단한 테스트 2개가 유지보수하기 용이하다.

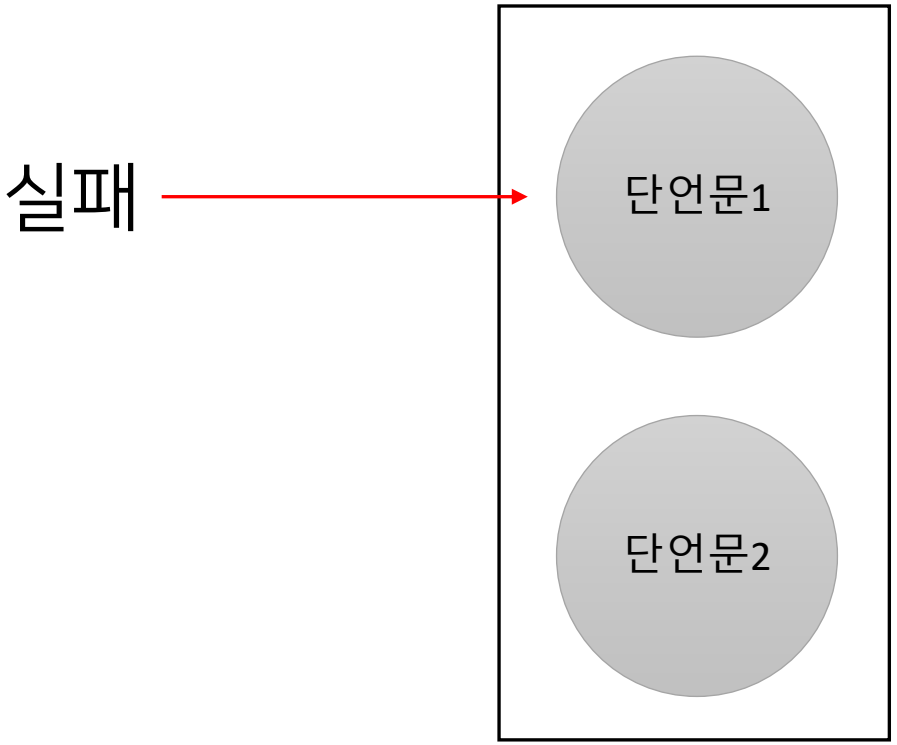
큰 테스트 코드 1개보다 작은 테스트 코드 2개가 나은 이유

테스트가 합쳐지게 되면,
앞 부분에서 실패가 나는 경우 뒷 부분은 아예 검증되지 않는다.

테스트 메소드

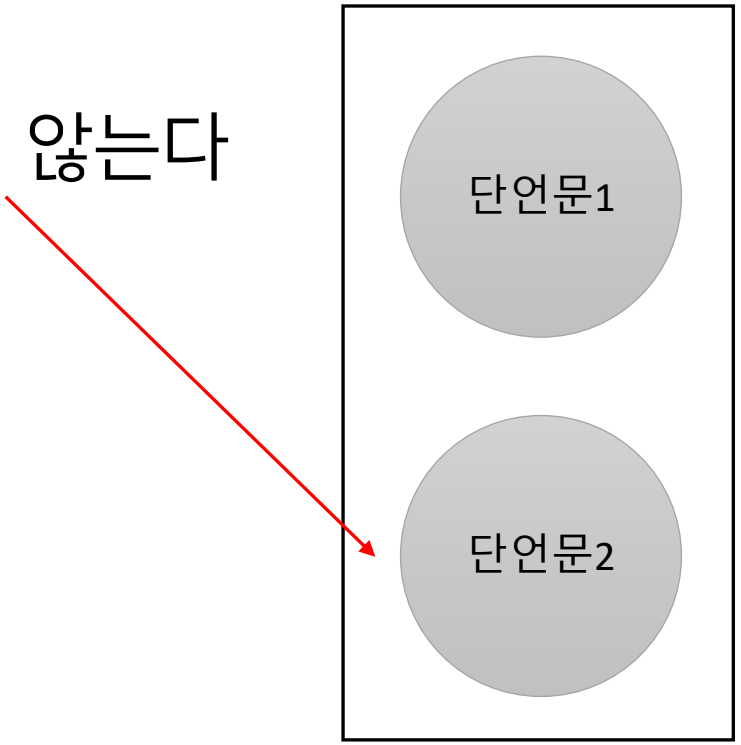


테스트 메소드

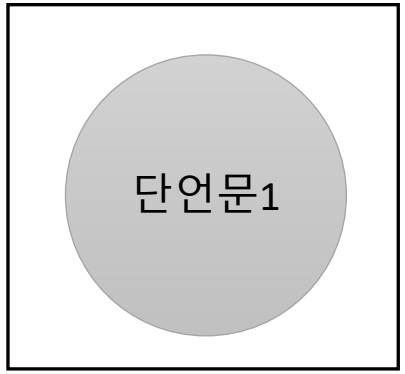


테스트 메소드

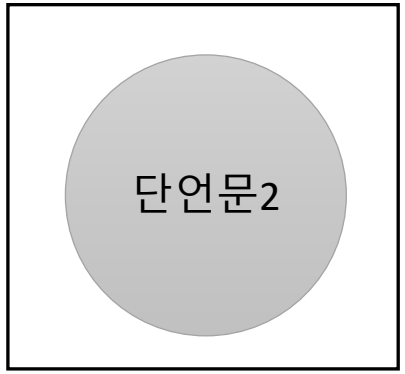
아예 수행이 되지 않는다



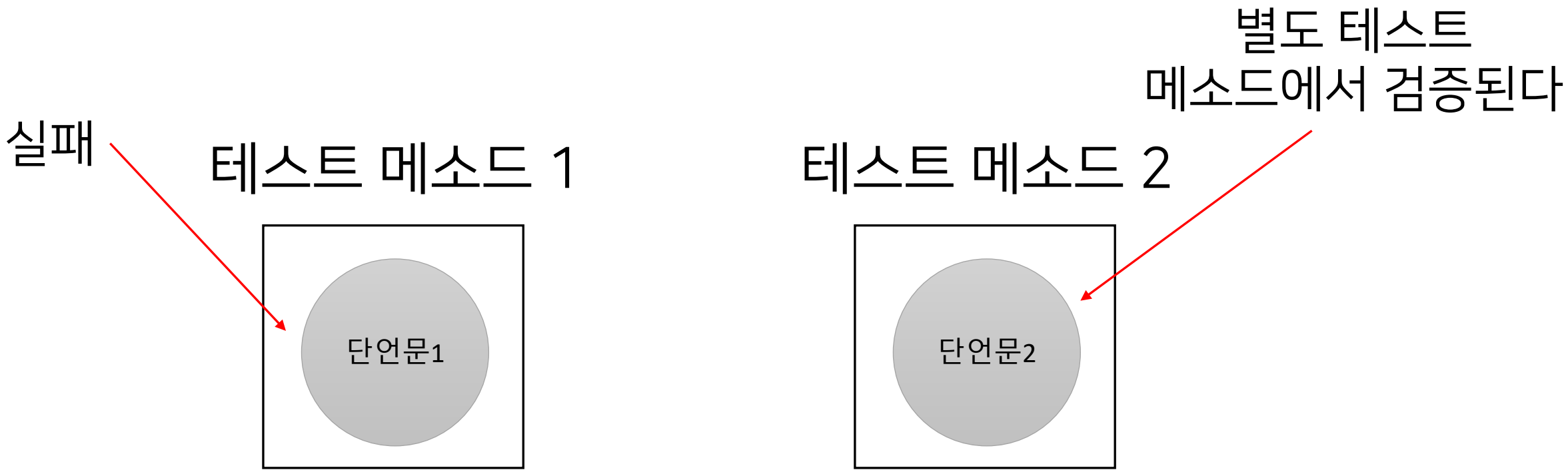
테스트 메소드 1



테스트 메소드 2



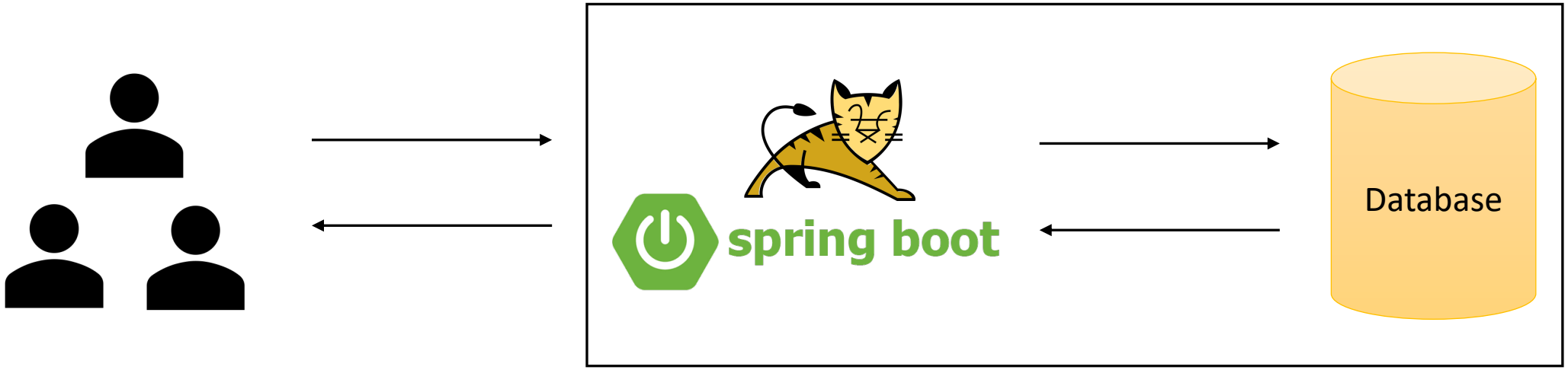
실패
테스트 메소드 1
단언문1
테스트 메소드 2
단언문2


별도 테스트
메소드에서 검증된다
실패
테스트 메소드 1
테스트 메소드 2
단언문1
단언문2

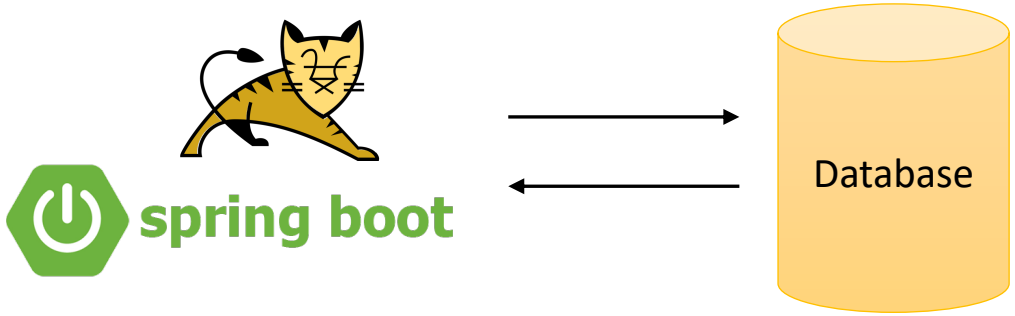
그래서 복잡한 테스트 1개보다 작은 테스트 N개를 선호합니다.

27강. N + 1 문제와 N + 1 문제가 발생하는 이유

서버 코드를 보고 Query를 생각할 수 있어야 한다!

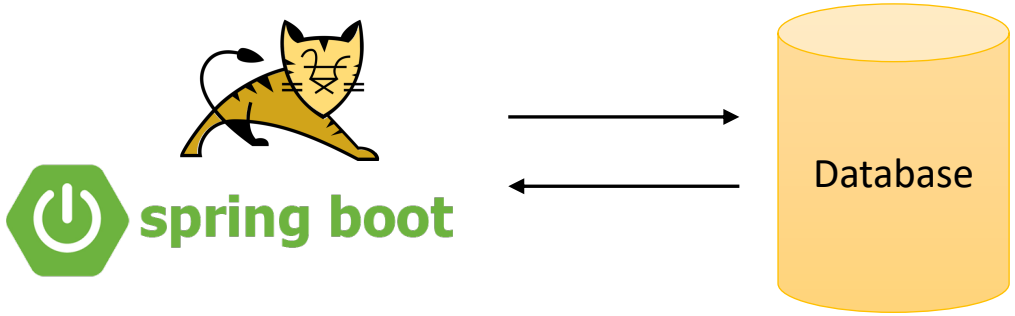


서버 코드를 보고 Query를 생각할 수 있어야 한다!




```
@Transactional
fun saveUser(request: UserCreateRequest) {
  userRepository.save(User(request.name, request.age))
}
```

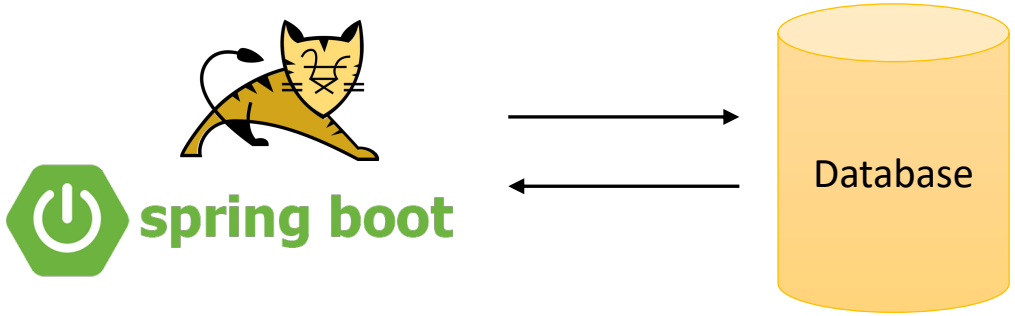
서버 코드를 보고 Query를 생각할 수 있어야 한다!



insert user 쿼리

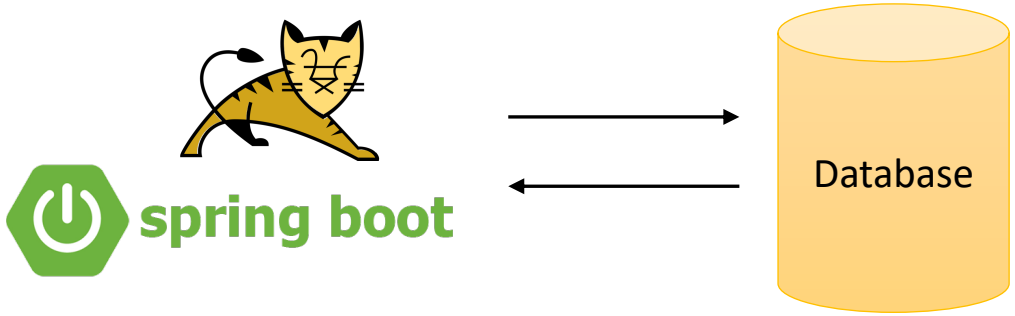
```
@Transactional
fun saveUser(request: UserCreateRequest) {
  userRepository.save(User(request.name, request.age))
}
```

서버 코드를 보고 Query를 생각할 수 있어야 한다!




```
@Transactional(readOnly = true)
fun getUsers(): List<UserResponse> {
  return userRepository.findAll()
    .map(UserResponse::of)
}
```

서버 코드를 보고 Query를 생각할 수 있어야 한다!



select user 쿼리

```
@Transactional(readOnly = true)
fun getUsers(): List<UserResponse> {
  return userRepository.findAll()
    .map(UserResponse::of)
}
```

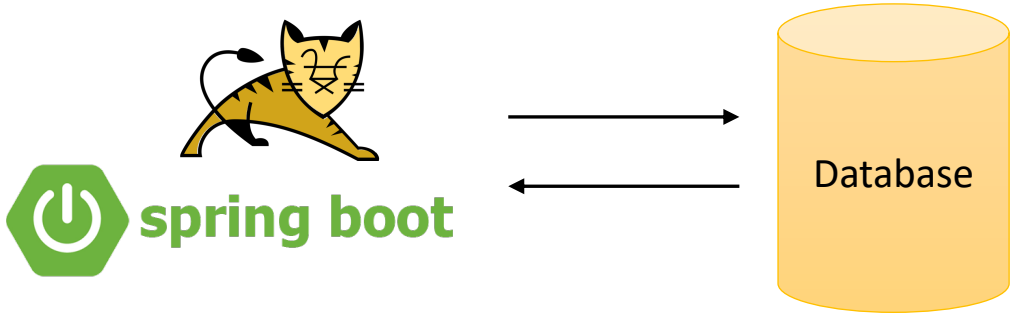
서버 코드를 보고 Query를 생각할 수 있어야 한다!

```
@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
        )
      }
    )
  }
}
```

서버 코드를 보고 Query를 생각할 수 있어야 한다!

```
@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
        )
      }
    )
  }
}
```

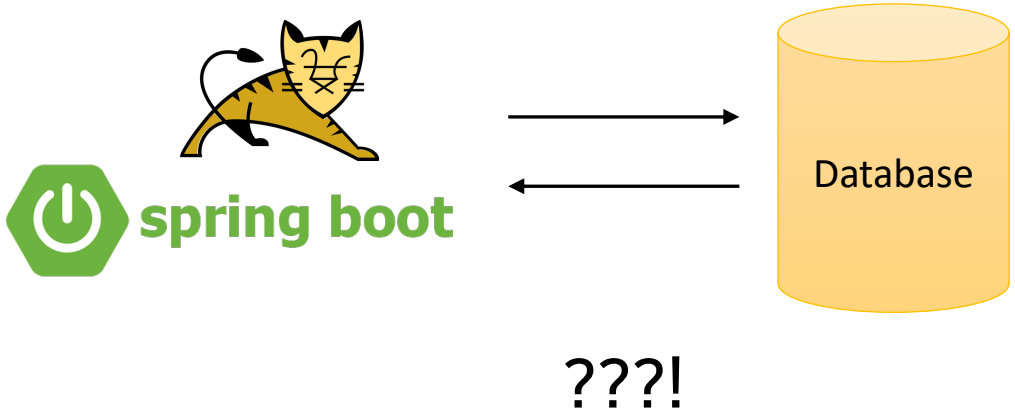
서버 코드를 보고 Query를 생각할 수 있어야 한다!



select user 쿼리

```
@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
        )
      }
    )
  }
}
```

서버 코드를 보고 Query를 생각할 수 있어야 한다!




```
@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
        )
      }
    )
  }
}
```

서버 코드를 보고 Query를 생각할 수 있어야 한다!

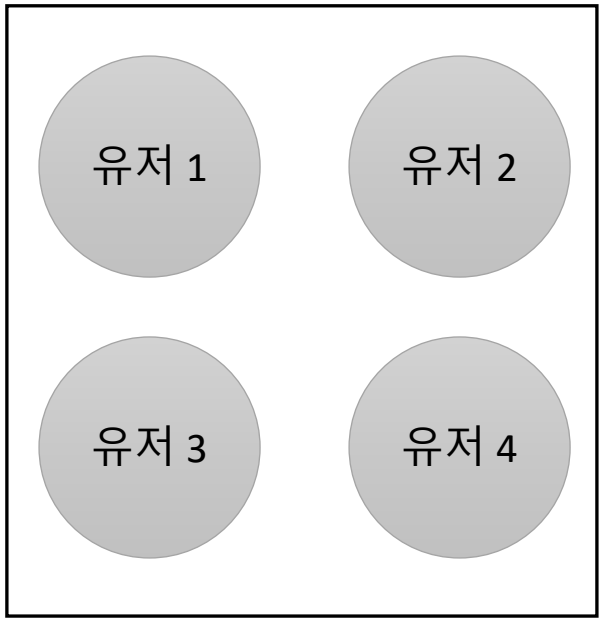
엇..?! UserLoanHistory를 가져온적이 없는데?!

서버 코드를 보고 Query를 생각할 수 있어야 한다!

사실은 이렇다!

서버 코드를 보고 Query를 생각할 수 있어야 한다!

select * from user;

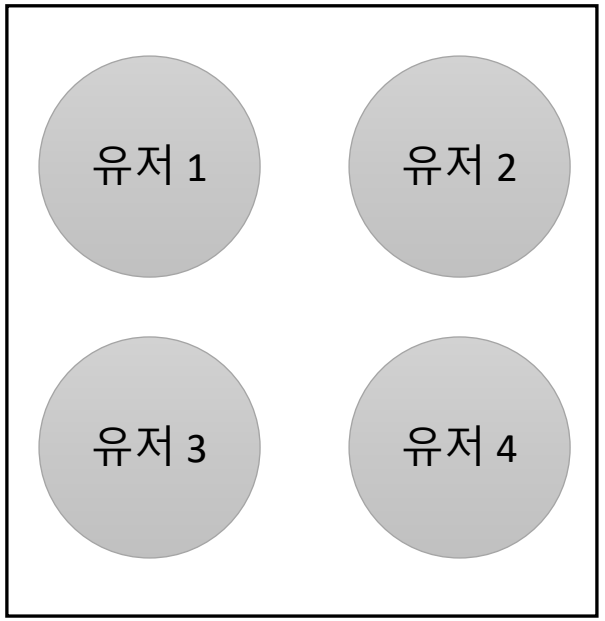


서버 메모리

```
@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
        )
      }
    )
  }
}
```

서버 코드를 보고 Query를 생각할 수 있어야 한다!

유저별로 반복

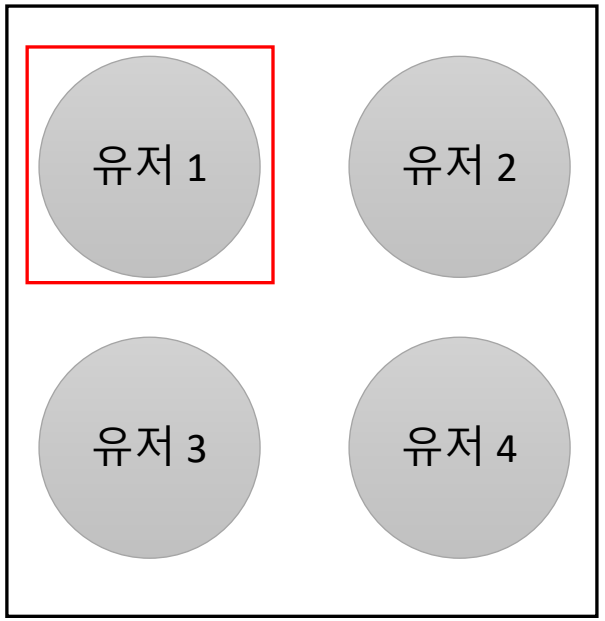


서버 메모리

```
@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
        )
      }
    )
  }
}
```

서버 코드를 보고 Query를 생각할 수 있어야 한다!

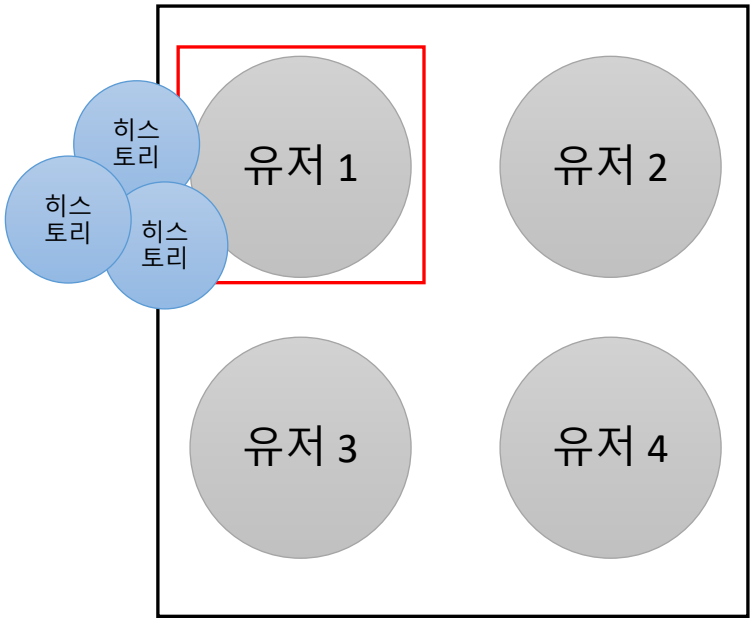
select * from
user_loan_history
where user_id = 1




```
@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
        )
      }
    )
  }
}
```

서버 코드를 보고 Query를 생각할 수 있어야 한다!

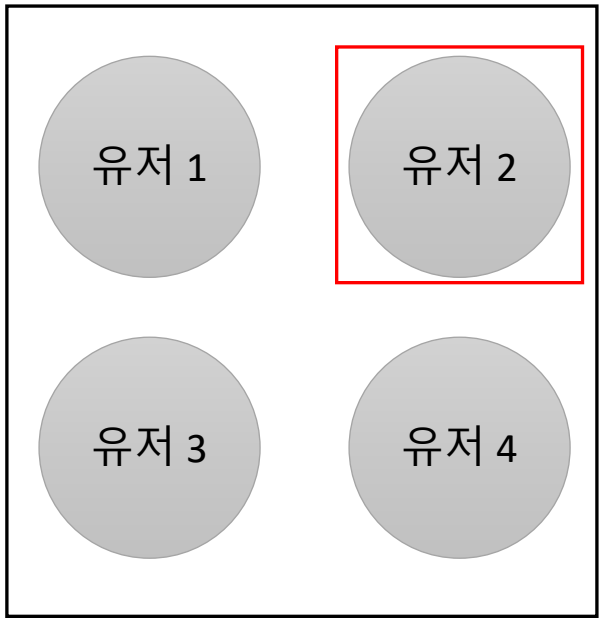
select * from
user_loan_history
where user_id = 1




```
@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
        )
      }
    )
  }
}
```

서버 코드를 보고 Query를 생각할 수 있어야 한다!

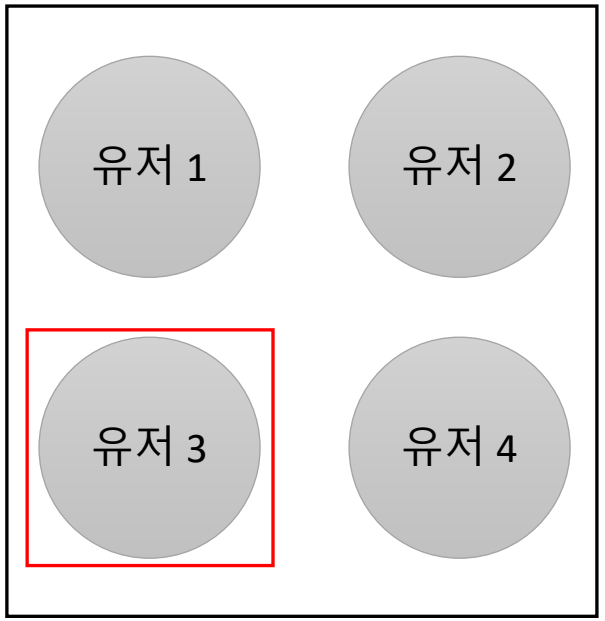
select * from
user_loan_history
where user_id = 2




```
@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
        )
      }
    )
  }
}
```

서버 코드를 보고 Query를 생각할 수 있어야 한다!

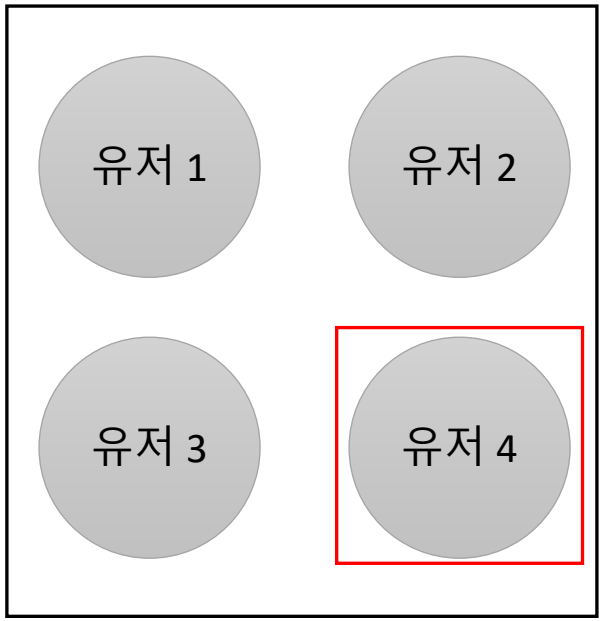
select * from
user_loan_history
where user_id = 3




```
@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
        )
      }
    )
  }
}
```

서버 코드를 보고 Query를 생각할 수 있어야 한다!

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
        )
      }
    )
  }
}
```

서버 코드를 보고 Query를 생각할 수 있어야 한다!

즉, 최초에 모든 유저를 가져오고 (쿼리 1회)
Loop를 통해 유저별로 히스토리를 가져온다 (쿼리 N회)

서버 코드를 보고 Query를 생각할 수 있어야 한다!

만약 유저가 100명이면? 1,000명이면? 100,000명이면?

서버 코드를 보고 Query를 생각할 수 있어야 한다!

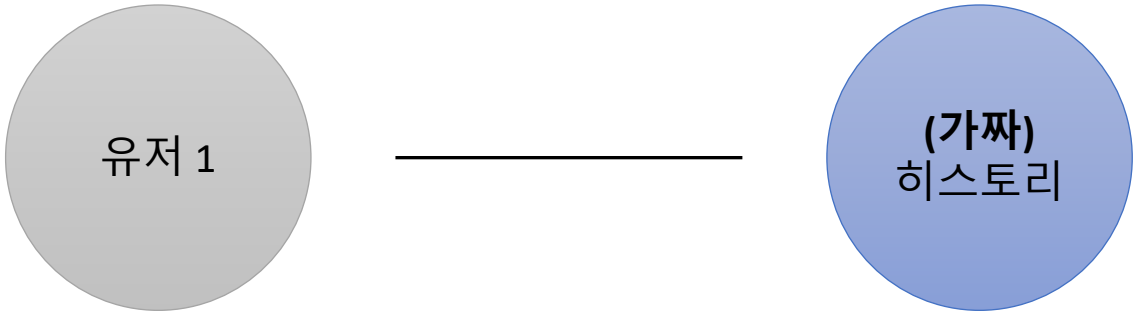
즉, 최초에 모든 유저를 가져오고 (쿼리 1회)
Loop를 통해 유저별로 히스토리를 가져온다 (쿼리 N회)

서버 코드를 보고 Query를 생각할 수 있어야 한다!

N + 1 문제!

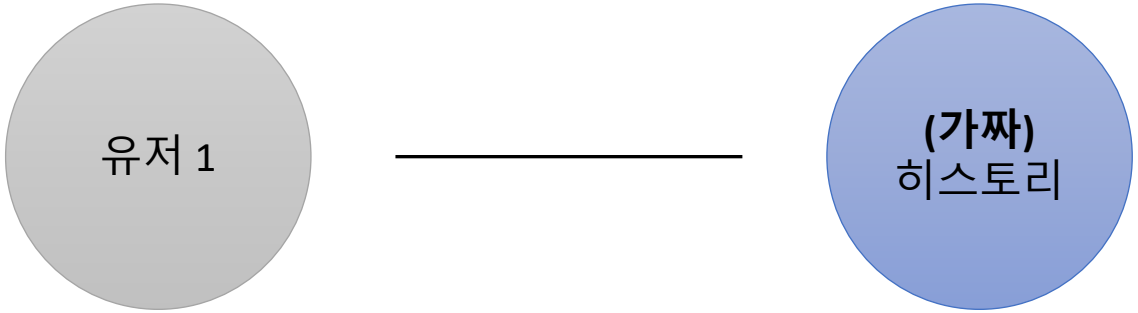
디버거로도 한 번 확인해보자!

JPA 1 : N 연관관계의 동작 원리



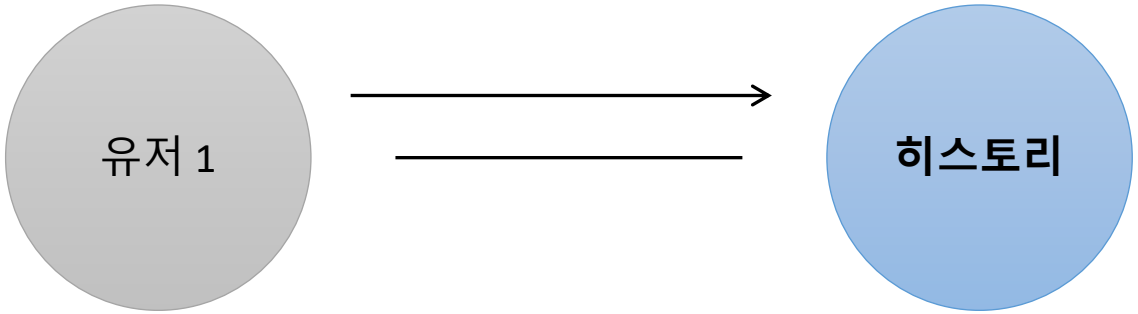
최초 유저 로딩시 가짜 List<UserLoanHistory>가 들어간다

JPA 1 : N 연관관계의 동작 원리



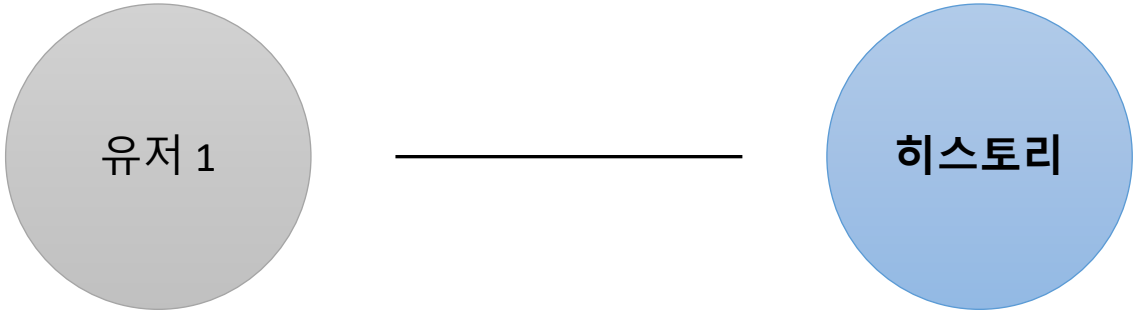
시작부터 모든 진짜 히스토리를 들고오는 것은
비효율적일 수 있기도 때문

JPA 1 : N 연관관계의 동작 원리



실제 히스토리에 접근할때에 진짜 UserLoanHistory를 불러온다

JPA 1 : N 연관관계의 동작 원리



이러한 전략을 Lazy Fetching이라고 한다.

N + 1 문제를 해결하는 방법

SQL의 join query를 알아야 한다!

28강. SQL join에 대해 알아보자

데이터를 예시로 join query를 이해해보자!

유저 테이블	
id	이름
1	A
2	B
3	C

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

유저 테이블	
id	이름
1	A
2	B
3	C

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

A 유저는 책 1을 빌렸다 반납하였고, 책 2는 대출 중이다

유저 테이블	
id	이름
1	A
2	B
3	C

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

B 유저는 책 3을 대출 중이다.

유저 테이블	
id	이름
1	A
2	B
3	C

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

C 유저는 아직까지 책을 빌린 적이 없다.

쿼리 한 번으로 두 테이블의 결과를 한 번에 보는 법!

join 을 사용하자!

쿼리 한 번으로 두 테이블의 결과를 한 번에 보는 법!

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

쿼리 한 번으로 두 테이블의 결과를 한 번에 보는 법!

기준 테이블

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

쿼리 한 번으로 두 테이블의 결과를 한 번에 보는 법!

합쳐지는 테이블

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

쿼리 한 번으로 두 테이블의 결과를 한 번에 보는 법!

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

합쳐지는 조건

유저 테이블	
id	이름
1	A
2	B
3	C

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

유저 테이블	
id	이름
1	A
2	B
3	C

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

유저 테이블	
id	이름
1	A
2	B
3	C

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

유저 테이블	
id	이름
1	A
2	B
3	C

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

유저 테이블	
id	이름
1	A
2	B
3	C

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

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

join을 사용할 때 '별칭'을 줄 수도 있다

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

쿼리 결과에 대한 한 가지 사실

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

3번 유저에 대한 결과는 없다!

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

3번 유저에 대한 결과는 없다!

우리가 사용했던 쿼리가 inner join 이었기 때문이다.

inner join

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

inner join

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

inner join

inner join은 테이블 양쪽에 데이터가
모두 존재하는 경우에만 하나로 합쳐준다.

inner join

합쳐지는 테이블에는 데이터가 없더라도 보여주고 싶다면?

left join

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 join

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

left join

User를 기준으로 데이터를 조회하였기 때문에
3번 유저에 대해 null로써 정보가 존재한다.

N + 1 문제를 해결하자!

이제 SQL join 쿼리가 무엇인지 알았으니
N + 1 문제를 해결하러 가자!

31강. 두 번째 요구사항 클리어!

요구사항 2 추가하기

유저 대출 현황 화면

- 유저 대출 현황을 보여준다.

등록하기 목록 히스토리 통계

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.

#4 두 번째 요구사항 추가하기 - 책의 분야

1. 새로운 기능을 추가할 때, 위치에 관한 고민과 선택에 따른 장단점

2. 복잡한 기능을 추가할 때, 테스트 코드를 작성하는 방법

#4 두 번째 요구사항 추가하기 - 책의 분야

3. SQL의 inner join, left join 특징과 차이

4. N + 1 문제를 해결하기 위해 fetch join을 사용하는 방법

감사합니다