#6 네 번째 요구사항 추가하기 - Querydsl

1. JPQL과 Querydsl의 장단점을 이해할 수 있다.

2. Qureydsl을 Kotlin + Spring Boot와 함께 사용할 수 있다.

3. Querydsl을 활용해 기존에 존재하던 Repository를 리팩토링 할 수 있다.

37강. Querydsl 도입하기

기술적인 요구사항

- 현재 사용하는 JPQL은 몇 가지 단점이 있다.
- Querydsl을 적용해서 단점을 극복하자.

Query DSL

JPQL은 무슨 단점이 있을까?

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

문자열이기 때문에 '버그'를 찾기가 어렵다!

JPQL은 무슨 단점이 있을까?

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

JPQL 문법이 일반 SQL와 조금 달라
복잡한 쿼리를 작성할 때마다 찾아보아야 한다.

Spring Data JPA는 무슨 단점이 있을까?

```
fun findByName(userName: String): User?

fun findByNameAndAge(userName: String, age: Int?): User?

fun findByNameAndAgeAndId(userName: String, age: Int?, id: Long?): User?
```

조건이 복잡한 동적쿼리를 작성할 때 함수가 계속해서 늘어난다.

Spring Data JPA는 무슨 단점이 있을까?

```
fun findByBookNameAndStatus(bookName: String, status: UserLoanStatus): UserLoanHistory?
```

프로덕션 코드 변경에 취약하다.

JPQL과 Spring Data JPA의 단점 정리!

1. 문자열로 쿼리를 작성하기에 버그를 찾기 어렵다.

2. 문법이 조금 달라 그때마다 검색해 찾아보야아 한다.

3. 동적 쿼리 작성이 어렵다.

4. 도메인 코드 변경에 취약하다.

이런 단점을 보완하기 위해 Querydsl이 등장!

Spring Data JPA와 Querydsl을 함께 사용하며 서로를 보완해야 한다!

Querydsl : 코드로 쿼리를 작성하게 해주는 도구!

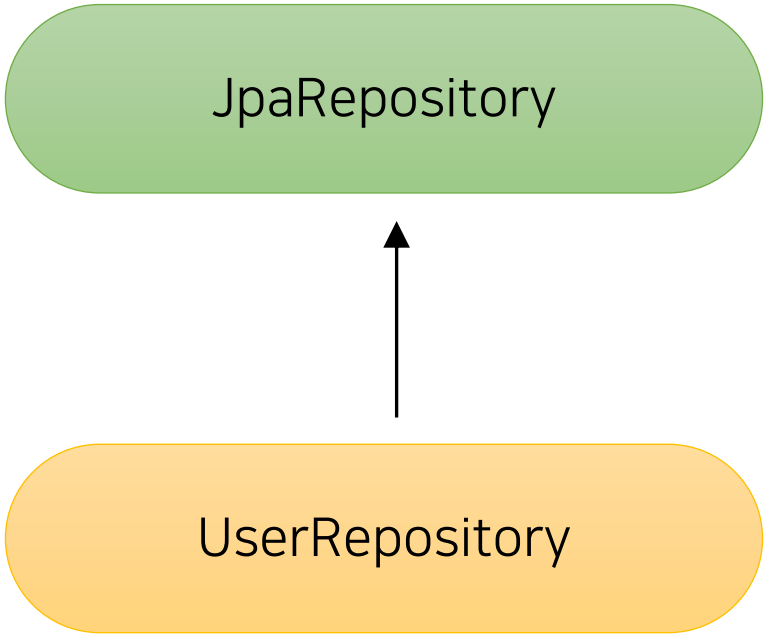
이런 단점을 보완하기 위해 Querydsl이 등장!

```
fun findAll(name: String): List<User> {
  return queryFactory.select(user)
    .from(user)
    .where(
      user.name.eq(name)
    )
    .fetch()
}
```

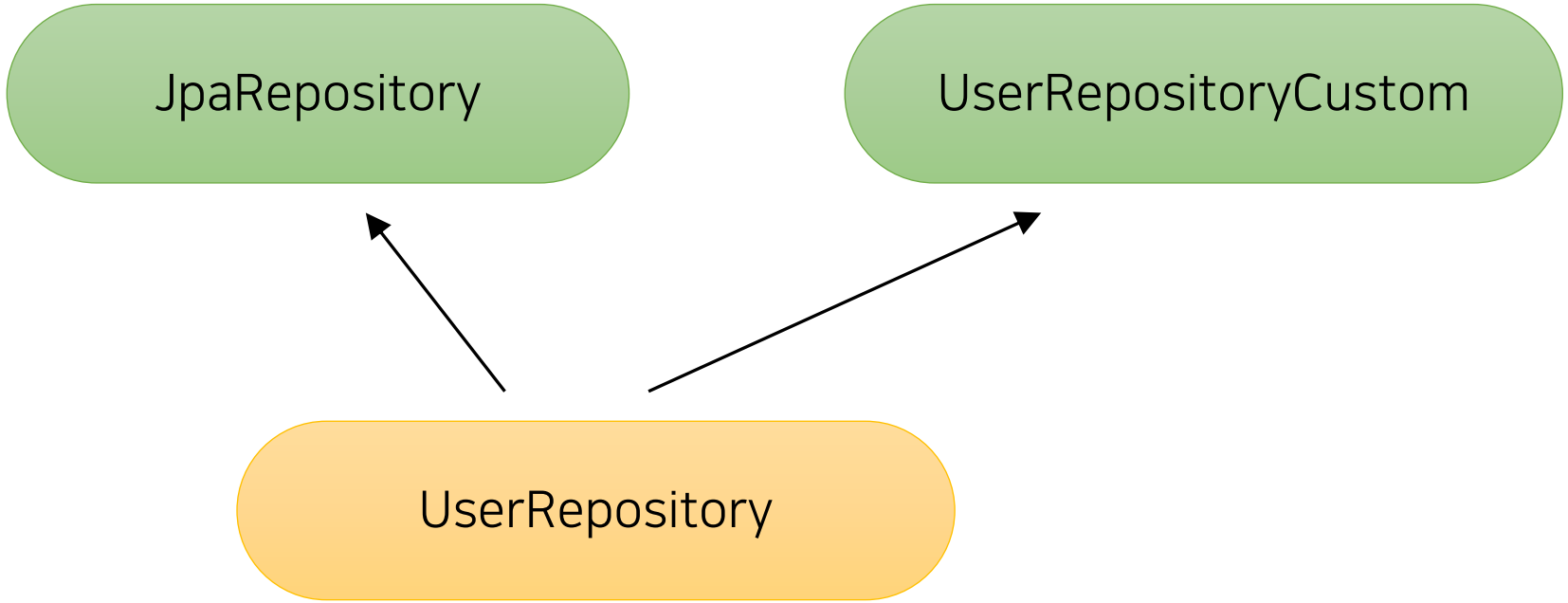
이제 프로젝트에 Querydsl을 적용 해보자!

38강. Querydsl 사용하기 - 첫 번째 방법

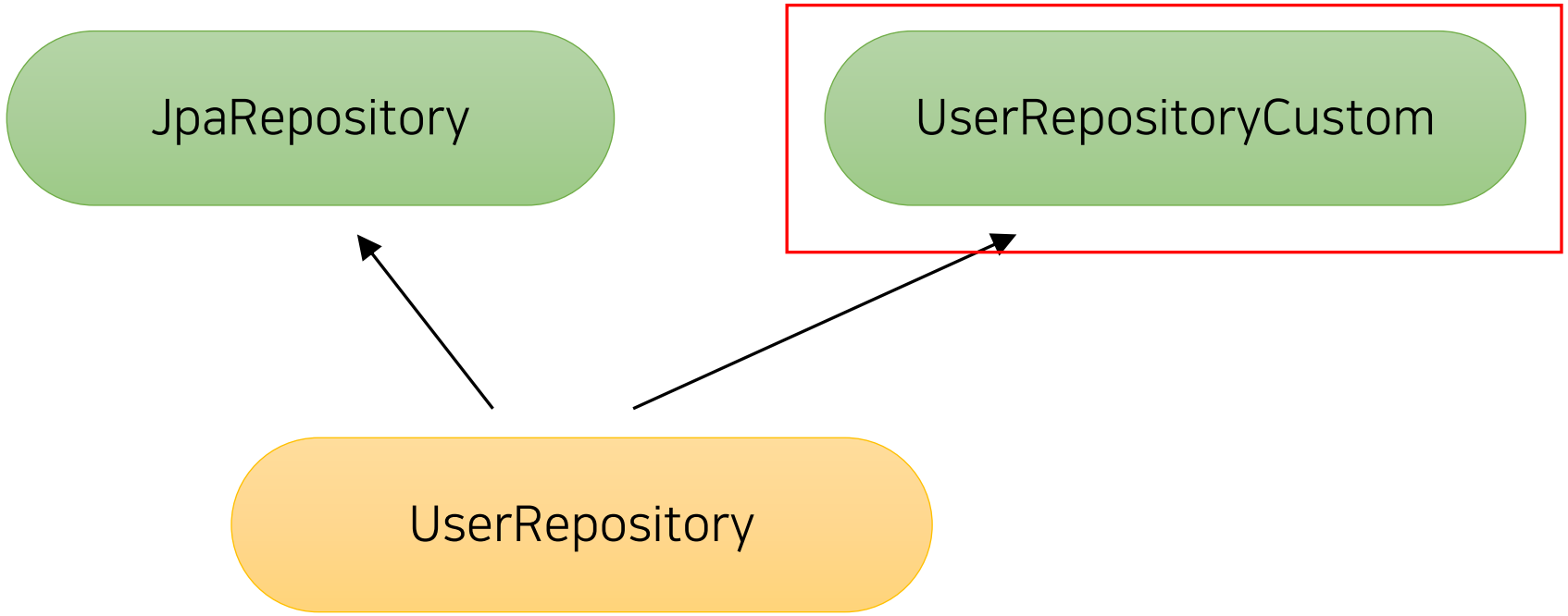
기본 Repository 구조



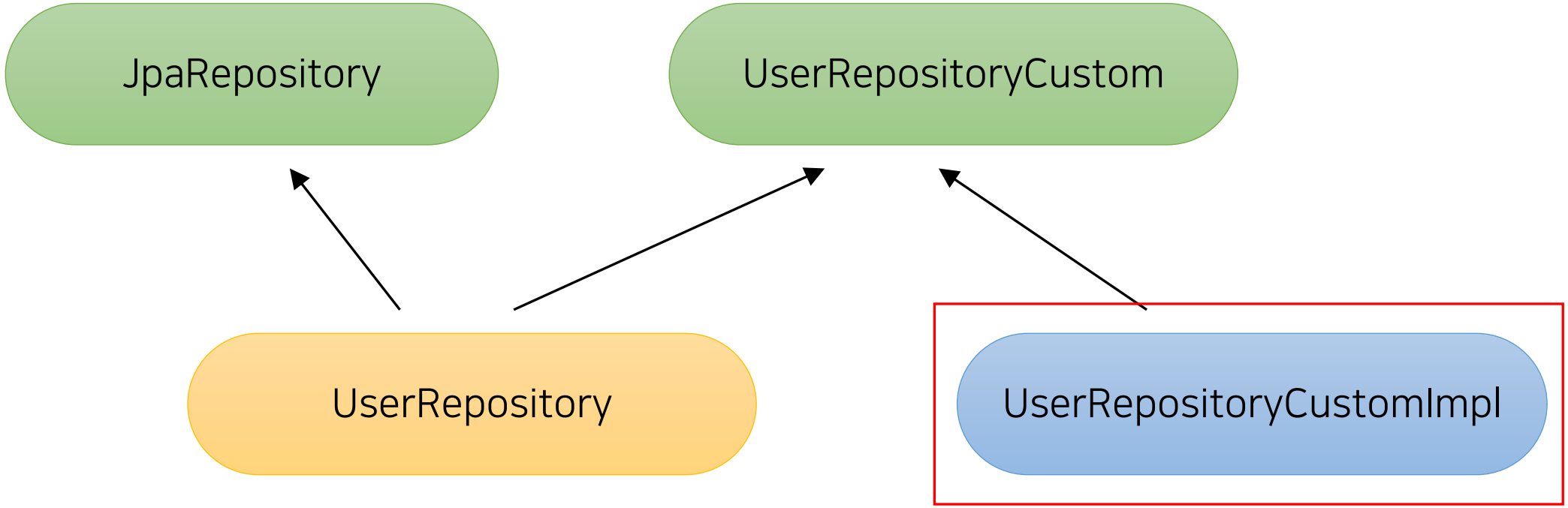
Querydsl을 적용한 새로운 Repository 구조



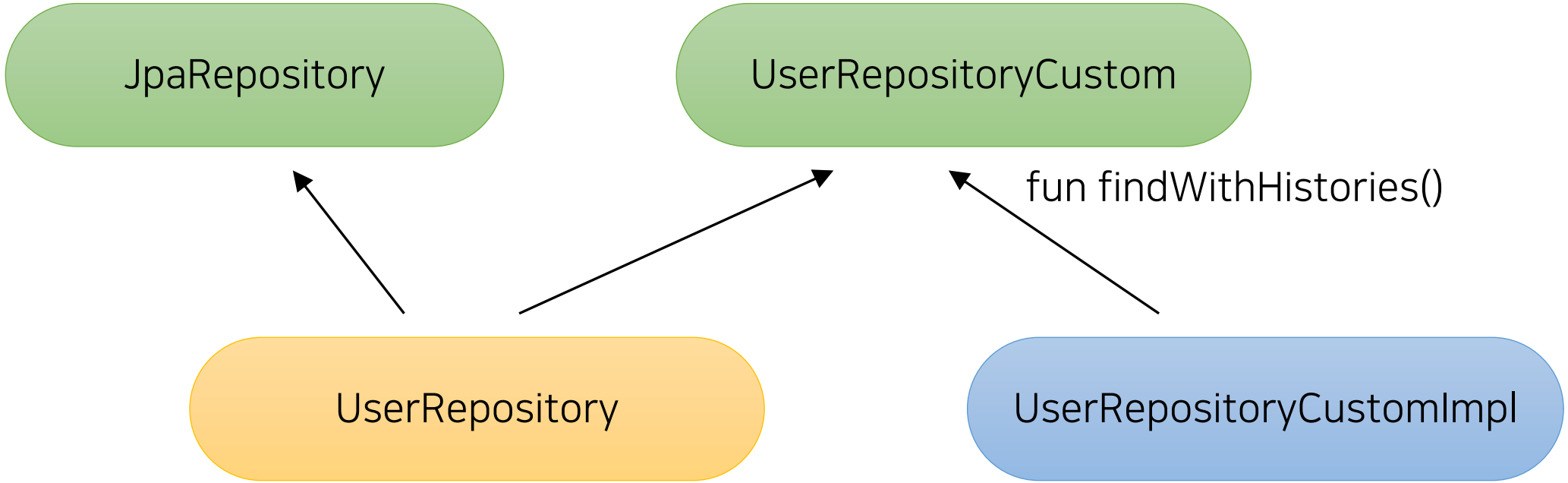
Querydsl을 적용한 새로운 Repository 구조



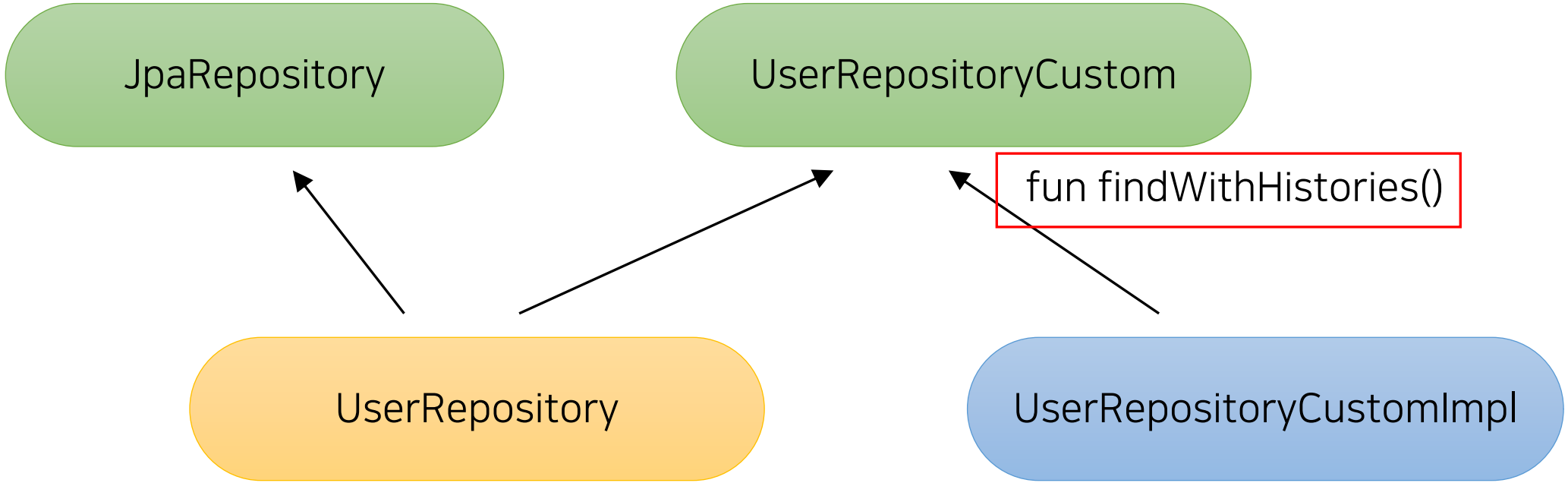
Querydsl을 적용한 새로운 Repository 구조



Querydsl을 적용한 새로운 Repository 구조



Querydsl을 적용한 새로운 Repository 구조



작성한 Query

```
override fun findAllWithHistories(): List<User> {
    return queryFactory.select(user).distinct()
        .from(user)
        .leftJoin(userLoanHistory).on(userLoanHistory.user.id.eq(user.id)).fetchJoin()
        .fetch()
}
```

작성한 Query

```
override fun findAllWithHistories(): List<User> {
    return queryFactory.select(user).distinct()
        .from(user)
        .leftJoin(userLoanHistory).on(userLoanHistory.user.id.eq(user.id)).fetchJoin()
        .fetch()
}
```

select(user) : select *

작성한 Query

```
override fun findAllWithHistories(): List<User> {
    return queryFactory.select(user).distinct()
        .from(user)
        .leftJoin(userLoanHistory).on(userLoanHistory.user.id.eq(user.id)).fetchJoin()
        .fetch()
}
```

distinct() : distinct

작성한 Query

```
override fun findAllWithHistories(): List<User> {
    return queryFactory.select(user).distinct()
        .from(user)
        .leftJoin(userLoanHistory).on(userLoanHistory.user.id.eq(user.id)).fetchJoin()
        .fetch()
}
```

from(user) : from user

작성한 Query

```
override fun findAllWithHistories(): List<User> {
    return queryFactory.select(user).distinct()
        .from(user)
        .leftJoin(userLoanHistory).on(userLoanHistory.user.id.eq(user.id)).fetchJoin()
        .fetch()
}
```

leftJoin(userLoanHistory) : left join user_loan_history

작성한 Query

```
override fun findAllWithHistories(): List<User> {
    return queryFactory.select(user).distinct()
        .from(user)
        .leftJoin(userLoanHistory).on(userLoanHistory.user.id.eq(user.id)).fetchJoin()
        .fetch()
}
```

on(userLoanHistory.user.id.eq(user.id) :
on user_loan_history.user_id = user.id

작성한 Query

```
override fun findAllWithHistories(): List<User> {
    return queryFactory.select(user).distinct()
        .from(user)
        .leftJoin(userLoanHistory).on(userLoanHistory.user.id.eq(user.id)).fetchJoin()
        .fetch()
}
```

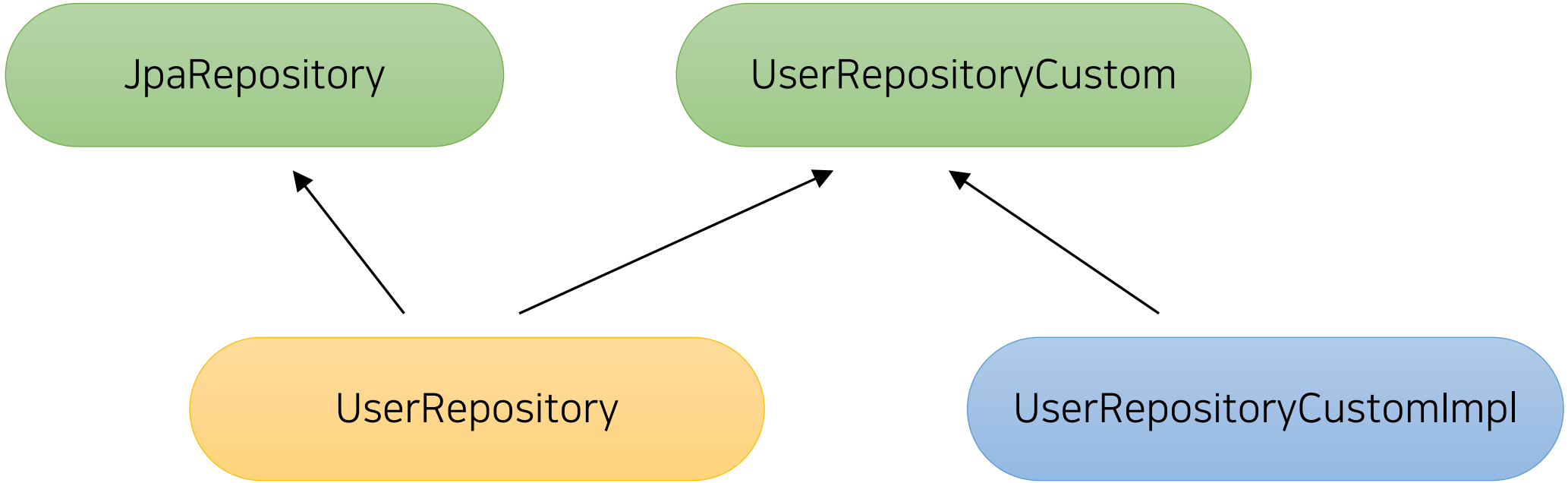
fetchJoin() : 앞의 join을 fetch join으로 간주한다.

작성한 Query

```
override fun findAllWithHistories(): List<User> {
    return queryFactory.select(user).distinct()
        .from(user)
        .leftJoin(userLoanHistory).on(userLoanHistory.user.id.eq(user.id)).fetchJoin()
        .fetch()
}
```

fetch() : 쿼리를 실행한다.

Querydsl을 적용한 새로운 Repository 구조



장점

서비스단에서 UserRepository 하나만 사용하면 된다.

단점

인터페이스와 클래스를 항상 같이 만들어 주어야 하는 것이
부담이고 여러모로 번거롭다.

다음 시간에는

새로운 방식을 활용해 BookRepository의 getStatus()를 바꿔보자!

39강. Querydsl 사용하기 - 두 번째 방법

추가적인 Querydsl 기능

```
fun getStats(): List<BookStatResponse> {
  return queryFactory
    .select(
      Projections.constructor(
        BookStatResponse::class.java,
        book.type,
        book.id.count(),
      )
    )
    .from(book)
    .groupBy(book.type)
    .fetch()
}
```

추가적인 Querydsl 기능

```
fun getStats(): List<BookStatResponse> {
  return queryFactory
    .select(
      Projections.constructor(
        BookStatResponse::class.java,
        book.type,
        book.id.count(),
      )
    )
    .from(book)
    .groupBy(book.type)
    .fetch()
}
```

주어진 DTO의 생성자를
호출한다는 의미!

추가적인 Querydsl 기능

```
fun getStats(): List<BookStatResponse> {
  return queryFactory
    .select(
      Projections.constructor(
        BookStatResponse::class.java,
        book.type,
        book.id.count(),
      )
    )
    .from(book)
    .groupBy(book.type)
    .fetch()
}
```

주어진 DTO의 생성자를
호출한다는 의미!

추가적인 Querydsl 기능

```
fun getStats(): List<BookStatResponse> {
  return queryFactory
    .select(
      Projections.constructor(
        BookStatResponse::class.java,
        book.type,
        book.id.count(),
      )
    )
    .from(book)
    .groupBy(book.type)
    .fetch()
}
```

이때 뒤에 나오는 파라미터들이
생성자로 들어간다.

추가적인 Querydsl 기능

```
fun getStats(): List<BookStatResponse> {
  return queryFactory
    .select(
      Projections.constructor(
        BookStatResponse::class.java,
        book.type,
        book.id.count(),
      )
    )
    .from(book)
    .groupBy(book.type)
    .fetch()
}
```

이때 뒤에 나오는 파라미터들이
생성자로 들어간다.

추가적인 Querydsl 기능

```
fun getStats(): List<BookStatResponse> {
  return queryFactory
    .select(
      Projections.constructor(
        BookStatResponse::class.java,
        book.type,
        book.id.count(),
      )
    )
    .from(book)
    .groupBy(book.type)
    .fetch()
}
```

여기까지를 SQL로 바꾸면
다음과 같다.

추가적인 Querydsl 기능

```
fun getStats(): List<BookStatResponse> {
  return queryFactory
    .select(
      Projections.constructor(
        BookStatResponse::class.java,
        book.type,
        book.id.count(),
      )
    )
    .from(book)
    .groupBy(book.type)
    .fetch()
}
```

여기까지를 SQL로 바꾸면
다음과 같다.

```
select type, count(book.id)
from book;
```

추가적인 Querydsl 기능

```
fun getStats(): List<BookStatResponse> {
  return queryFactory
    .select(
      Projections.constructor(
        BookStatResponse::class.java,
        book.type,
        book.id.count(),
      )
    )
    .from(book)
    .groupBy(book.type)
    .fetch()
}
```

group by type

추가적인 Querydsl 기능

```
fun getStats(): List<BookStatResponse> {
  return queryFactory
    .select(
      Projections.constructor(
        BookStatResponse::class.java,
        book.type,
        book.id.count(),
      )
    )
    .from(book)
    .groupBy(book.type)
    .fetch()
}
```

```
select type, count(book.id)
from book
group by type;
```

2번째 방법의 장점

클래스만 바로 만들면 되어 간결하다.

2번째 방법의 단점

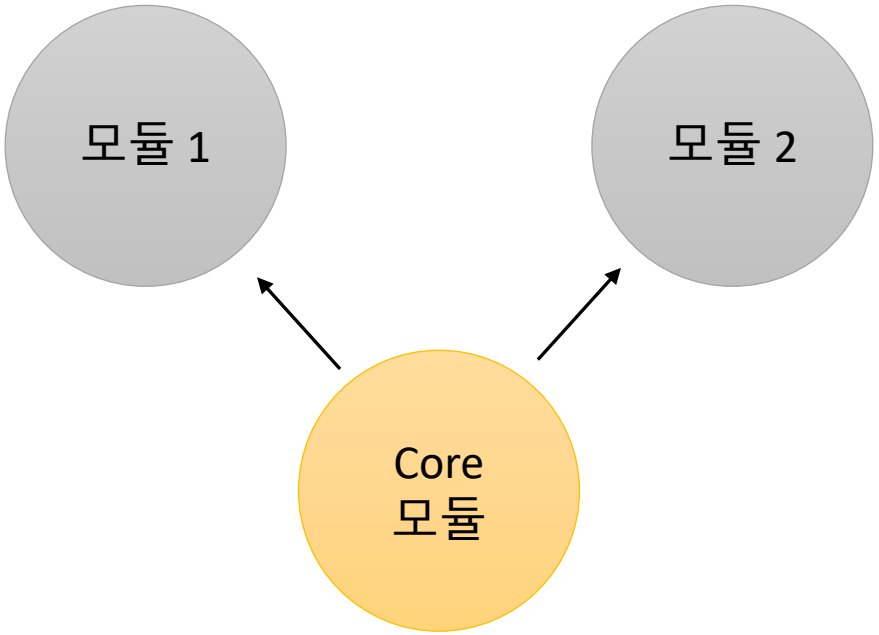
필요에 따라 두 Repository를 모두 불러와야 한다.

어떤 방식이 더 좋을까?!

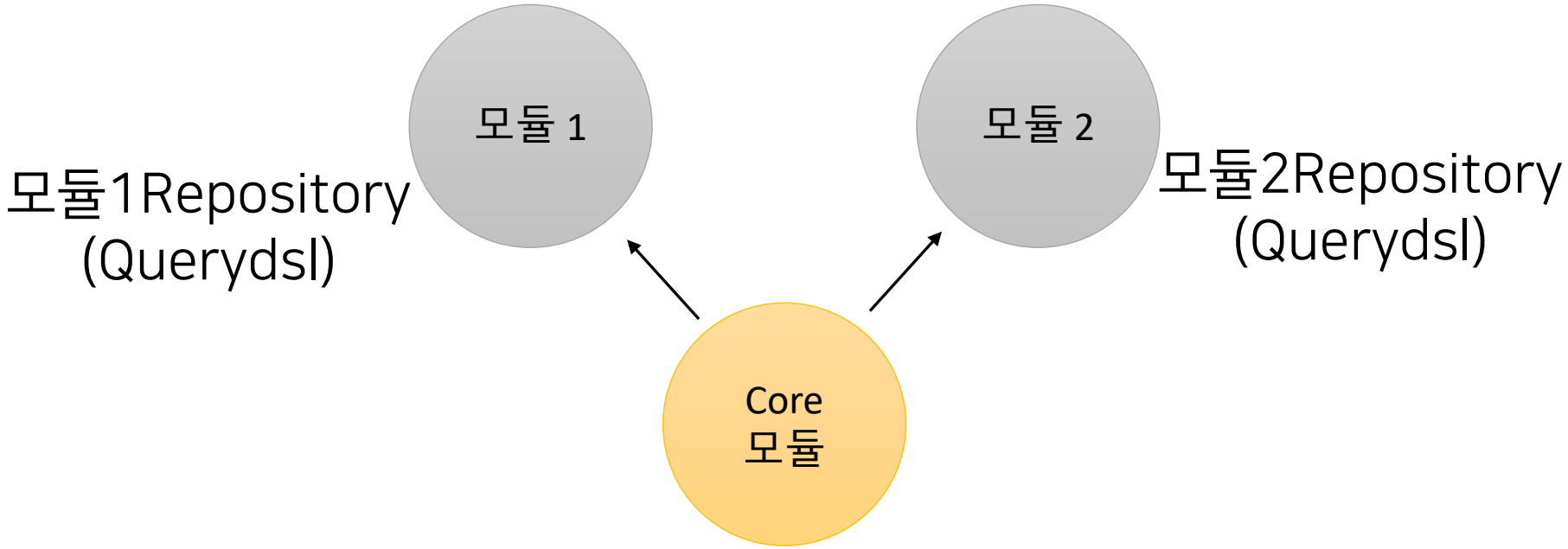
개인적으로는 지금 방법을 선호한다.

멀티 모듈을 사용하는 경우
모듈 별로만 Repository를 쓰는 경우가 많기 때문

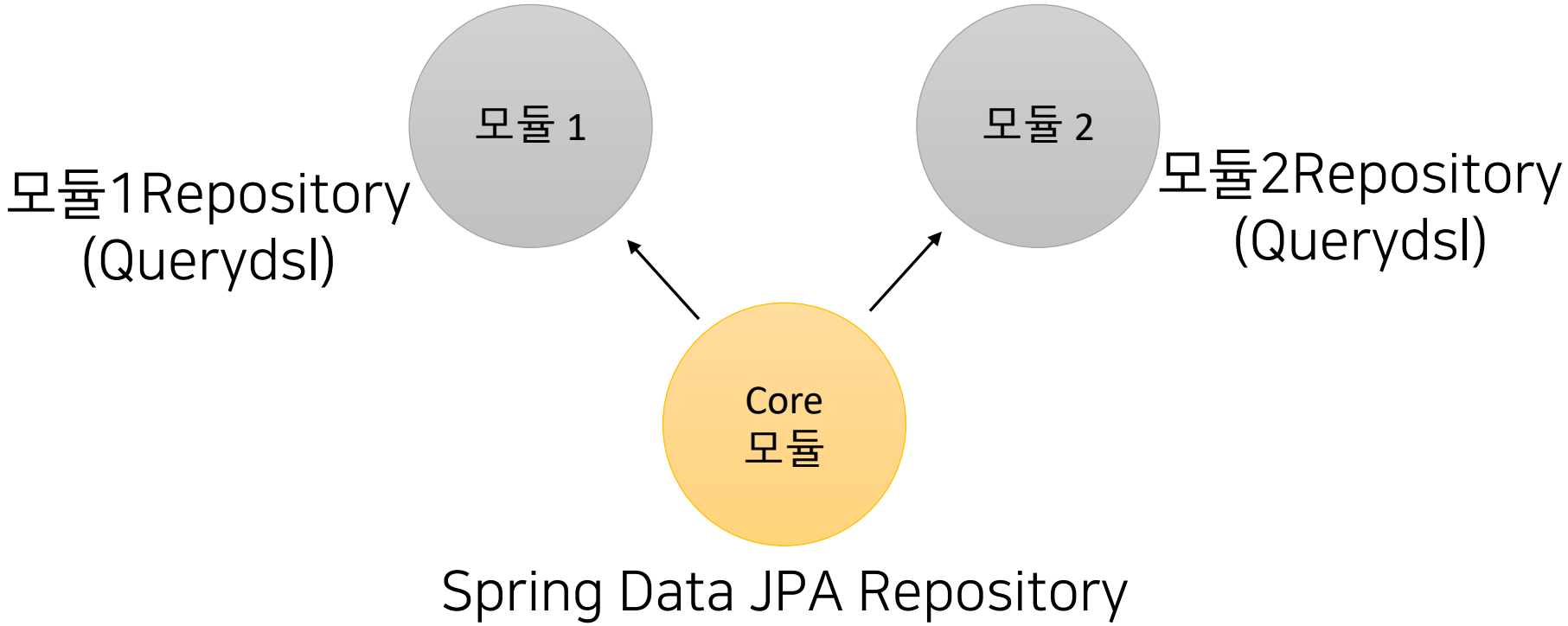
어떤 방식이 더 좋을까?!



어떤 방식이 더 좋을까?!


모듈 1
모듈 2
모듈1Repository
(Querydsl)
모듈2Repository
(Querydsl)
Core
모듈

어떤 방식이 더 좋을까?!


모듈 1
모듈 2
모듈1Repository
(Querydsl)
모듈2Repository
(Querydsl)
Core
모듈
Spring Data JPA Repository

40강. UserLoanHistoryRepository를 Querydsl으로 리팩토링 하기

현재의 UserLoanHistoryRepository

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

    fun findByBookNameAndStatus(bookName: String, status: UserLoanStatus): UserLoanHistory?

    fun countByStatus(status: UserLoanStatus): Long

}
```

@Query를 사용하지 않은 Repository 기능도 Querydsl로 옮겨야 할까?!

현재의 UserLoanHistoryRepository

개인적으로는 Querydsl로 옮기는 것을 선호합니다!

동적 쿼리의 간편함 때문!!

가상의 UserLoanHistoryRepository

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

    fun findByBookName(bookName: String): UserLoanHistory?

    fun findByBookNameAndStatus(bookName: String, status: UserLoanStatus): UserLoanHistory?

    fun countByStatus(status: UserLoanStatus): Long

}
```

가상의 UserLoanHistoryRepository

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

    fun findByBookName(bookName: String): UserLoanHistory?

    fun findByBookNameAndStatus(bookName: String, status: UserLoanStatus): UserLoanHistory?

    fun countByStatus(status: UserLoanStatus): Long

}
```

가상의 UserLoanHistoryRepository

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

    fun findByBookName(bookName: String): UserLoanHistory?

    fun findByBookNameAndStatus(bookName: String, status: UserLoanStatus): UserLoanHistory?

    fun countByStatus(status: UserLoanStatus): Long

}
```

함수가 정말 많이 늘어날 수 있다!!

함수가 늘어나는 예시

다음의 조건을 만족하는 Repository 함수를 만들어주세요!

- A 필드는 필수적으로 들어온다.
- B, C, D, E 필드는 선택적으로 들어온다.

findByA

함수가 늘어나는 예시

다음의 조건을 만족하는 Repository 함수를 만들어주세요!

- A 필드는 필수적으로 들어온다.
- B, C, D, E 필드는 선택적으로 들어온다.

findByA, findAAndB

함수가 늘어나는 예시

다음의 조건을 만족하는 Repository 함수를 만들어주세요!

- A 필드는 필수적으로 들어온다.
- B, C, D, E 필드는 선택적으로 들어온다.

findByA, findAAndB, findAAndC

함수가 늘어나는 예시

다음의 조건을 만족하는 Repository 함수를 만들어주세요!

- A 필드는 필수적으로 들어온다.
- B, C, D, E 필드는 선택적으로 들어온다.

findByA, findAAndB, findAAndC, ...

함수가 늘어나는 예시

다음의 조건을 만족하는 Repository 함수를 만들어주세요!

- A 필드는 필수적으로 들어온다.
- B, C, D, E 필드는 선택적으로 들어온다.

findByA, findAAndB, findAAndC, ..., findAAndBAndCAndDAndE

함수가 늘어나는 예시

총 16개의 Repository 함수가 생긴다! 필드가 더 늘어난다면..?!

동적 쿼리

이렇게 where 조건이 동적으로 바뀌는 쿼리는
Querydsl을 이용하면 쉽게 구현 가능하다!

findByBookName 변경

```
fun find(bookName: String): UserLoanHistory? {
    return queryFactory.select(userLoanHistory)
        .from(userLoanHistory)
        .where(
            userLoanHistory.bookName.eq(bookName),
        )
        .limit(1)
        .fetchOne()
}
```

findByBookName 변경

```
fun find(bookName: String): UserLoanHistory? {
    return queryFactory.select(userLoanHistory)
        .from(userLoanHistory)
        .where(
            userLoanHistory.bookName.eq(bookName),
        )
        .limit(1)
        .fetchOne()
}
```

함수 이름이 간결해졌다!

findByBookName 변경

```kotlin
fun find(bookName: String): UserLoanHistory? {
    return queryFactory.select(userLoanHistory)
        .from(userLoanHistory)
        .where(
            userLoanHistory.bookName.eq(bookName),
        )
        .limit(1)
        .fetchOne()
}
```

limit(1)
SQL의 limit 1

모든 검색 결과에서 1개만
가져온다는 의미이다.

findByBookName 변경

```
fun find(bookName: String): UserLoanHistory? {
    return queryFactory.select(userLoanHistory)
        .from(userLoanHistory)
        .where(
            userLoanHistory.bookName.eq(bookName),
        )
        .limit(1)
        .fetchOne()
}
```

fetchOne()

List<Entity>로 조회 결과를
가져오는 대신
Entity 하나만 가져온다.

findByBookNameAndStatus 변경

```
fun find(bookName: String, status: UserLoanStatus? = null): UserLoanHistory? {
    return queryFactory.select(userLoanHistory)
        .from(userLoanHistory)
        .where(
            userLoanHistory.bookName.eq(bookName),
            status?.let { userLoanHistory.status.eq(status) }
        )
        .limit(1)
        .fetchOne()
}
```

findByBookNameAndStatus 변경

```
fun find(bookName: String, status: UserLoanStatus? = null): UserLoanHistory? {
    return queryFactory.select(userLoanHistory)
        .from(userLoanHistory)
        .where(
            userLoanHistory.bookName.eq(bookName),
            status?.let { userLoanHistory.status.eq(status) }
        )
        .limit(1)
        .fetchOne()
}
```

default parameter에 null을 넣어 외부에서는 bookName만 쓸 수도, status까지 같이 쓸 수도 있다.

findByBookNameAndStatus 변경

```
fun find(bookName: String, status: UserLoanStatus? = null): UserLoanHistory? {
    return queryFactory.select(userLoanHistory)
        .from(userLoanHistory)
        .where(
            userLoanHistory.bookName.eq(bookName),
            status?.let { userLoanHistory.status.eq(status) }
        )
        .limit(1)
        .fetchOne()
}
```

status가 null이 아닌 경우에만
User_loan_history.status = ? 가 들어간다.

findByBookNameAndStatus 변경

```kotlin
fun find(bookName: String, status: UserLoanStatus? = null): UserLoanHistory? {
    return queryFactory.select(userLoanHistory)
        .from(userLoanHistory)
        .where(
            userLoanHistory.bookName.eq(bookName),
            status?.let { userLoanHistory.status.eq(status) }
        )
        .limit(1)
        .fetchOne()
}
```

Where에 여러 조건에 들어오면 각 조건은 AND로 연결된다.

findByBookNameAndStatus 변경

```kotlin
fun find(bookName: String, status: UserLoanStatus? = null): UserLoanHistory? {
    return queryFactory.select(userLoanHistory)
        .from(userLoanHistory)
        .where(
            userLoanHistory.bookName.eq(bookName),
            status?.let { userLoanHistory.status.eq(status) }
        )
        .limit(1)
        .fetchOne()
}
```

Where 조건에 null이 들어오면 무시하게 된다.

countByStatus 변경

```
fun count(status: UserLoanStatus): Long {
  return queryFactory.select(userLoanHistory.count())
    .from(userLoanHistory)
    .where(
      userLoanHistory.status.eq(status)
    )
    .fetchOne() ?: 0L
}
```

countByStatus 변경

```kotlin
fun count(status: UserLoanStatus): Long {
  return queryFactory.select(userLoanHistory.count())
    .from(userLoanHistory)
    .where(
      userLoanHistory.status.eq(status)
    )
    .fetchOne() ?: 0L
}
```

SQL의
count(id)로 변경된다!

countByStatus 변경

```
fun count(status: UserLoanStatus): Long {
    return queryFactory.select(userLoanHistory.count())
        .from(userLoanHistory)
        .where(
            userLoanHistory.status.eq(status)
        )
        .fetchOne() ?: 0L
}
```

Count의 결과는 숫자 1개 이므로 fetchOne()을 사용했다.

countByStatus 변경

```
fun count(status: UserLoanStatus): Long {
    return queryFactory.select(userLoanHistory.count())
        .from(userLoanHistory)
        .where(
            userLoanHistory.status.eq(status)
        )
        .fetchOne() ?: 0L
}
```

Count의 결과는 숫자 1개 이므로 fetchOne()을 사용했다.

혹시 비어 있는 경우는 0L을 반환하도록 했다.

41강. 마지막 요구사항 클리어!

(기술적인) 요구사항 4 추가하기

기술적인 요구사항

- 현재 사용하는 JPQL은 몇 가지 단점이 있다.
- Querydsl을 적용해서 단점을 극복하자.

Query DSL

#6 네 번째 요구사항 추가하기 - Querydsl

1. JPQL과 Querydsl의 장단점을 이해한다.

2. Querydsl을 Kotlin + Spring Boot와 함께 사용하고,
   2가지 방식의 장단점을 이해한다.

#6 네 번째 요구사항 추가하기 - Querydsl

3. Querydsl의 기본적인 사용법을 익힌다.

4. Querydsl을 활용해 기존 Repository를 리팩토링 한다.

감사합니다