



섹션 6. 네 번째 요구사항 추가하기 - Querydsl

이번 섹션의 목표

37강. Querydsl 도입하기

38강. Querydsl 사용하기 - 첫 번째 방법

39강. Querydsl 사용하기 - 두 번째 방법

40강. UserLoanHistoryRepository를 Querydsl으로 리팩토링 하기

41강. 마지막 요구사항 클리어!

이번 섹션의 목표

1. JPQL과 Querydsl의 장단점을 이해할 수 있다.
2. Querydsl을 Kotlin + Spring Boot와 함께 사용할 수 있다.
3. Querydsl을 활용해 기존에 존재하던 Repository를 리팩토링할 수 있다.

37강. Querydsl 도입하기

지난 시간까지 우리는 추가적인 Business 요구사항 세 가지를 적용했다. 이번 시간에는 Spring Data JPA를 그대로 사용하는 것의 불편함을 이해하고, Querydsl 설정을 프로젝트에 진행할 것이다.

주어진 기술적인 요구사항은 다음과 같다.

기술적인 요구사항

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



영상과 PPT를 함께 보시면 더욱 좋습니다 😊

본격적으로 프로젝트에 Querydsl을 적용하기 전에 `@Query` 를 사용한 JPQL의 단점과 Spring Data JPA의 단점을 알아보자! 이들은 다음 몇 가지 단점을 가지고 있다.

1. 문자열로 쿼리를 작성하기에 버그를 찾기 어렵다.
2. 문법이 조금 달라 그때마다 검색해 찾아보아야 한다.
3. 동적 쿼리 작성이 어렵다.
4. 도메인 코드 변경에 취약하다.
5. 함수 이름 구성에 제약이 있다. (의미있는 이름을 붙이기 어렵다)

때문에 이런 단점을 보완하기 위해서 Querydsl을 함께 사용해야 한다.

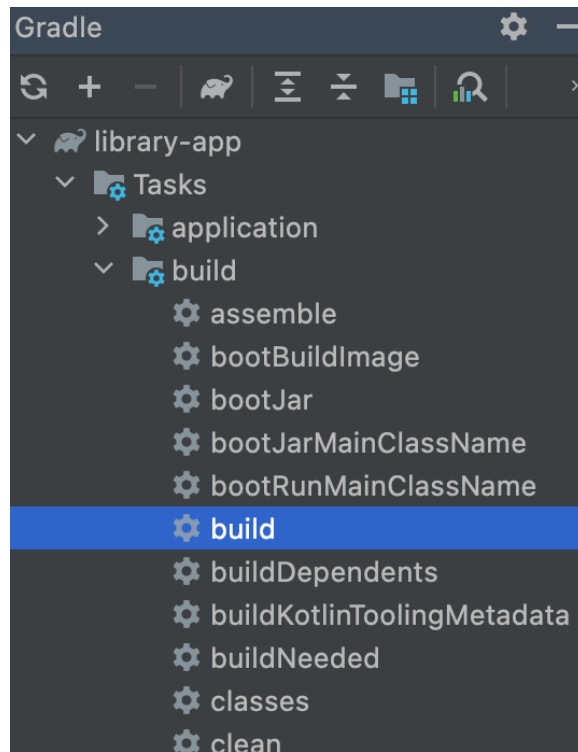
Querydsl은 “코드로 쿼리를 작성하게 해주는 도구”로, 예시를 보면 다음과 같다!

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

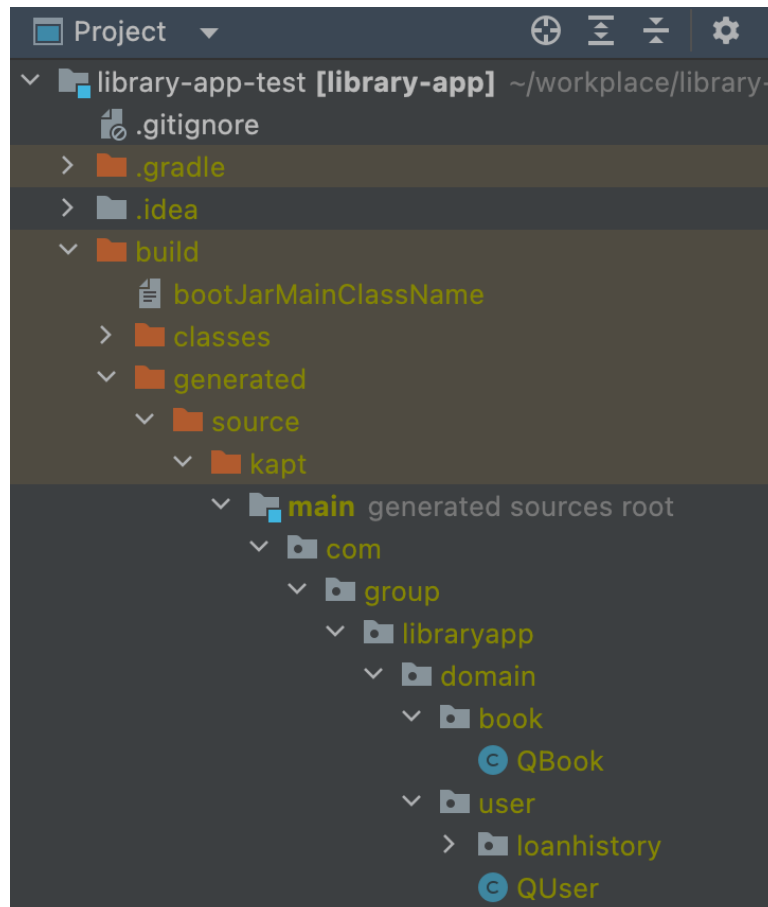
Querydsl을 직접 사용해보면 더 감이 올 것이다! 이제 Querydsl을 프로젝트에 설정해보자.
build.gradle 에 다음 부분을 추가해주자!

```
plugins {  
    // plugins 안에 아래 내용 추가  
    id "org.jetbrains.kotlin.kapt" version "1.6.21"  
}  
  
dependencies {  
    // dependencies 안에 아래 내용 추가  
    implementation("com.querydsl:querydsl-jpa:5.0.0")  
    kapt("com.querydsl:querydsl-apt:5.0.0:jpa")  
    kapt("org.springframework.boot:spring-boot-configuration-processor")  
}
```

모든 스크립트를 추가하였다면 gradle refresh를 해주고, build를 눌러보자!



그렇게 되면 프로젝트 build 결과물에서 **QClass** 를 확인할 수 있다!!! Querydsl은 이 QClass를 활용해, 코드로 쿼리를 작성한다.



이제 Querydsl을 사용할 준비는 끝이 났다!! 🎉 다음 시간에는 Querydsl을 사용하는 첫 번째 방법을 알아보고 UserRepository에 Querydsl을 적용해보자!

38강. Querydsl 사용하기 - 첫 번째 방법

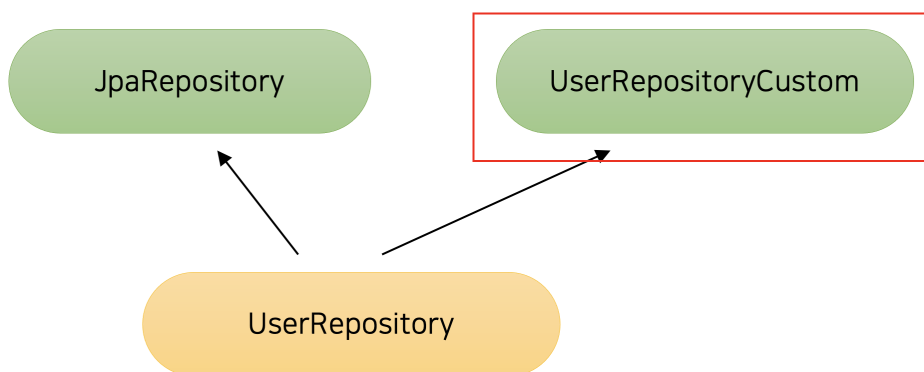
이번 시간에는 Querydsl을 사용하는 첫 번째 방법을 알아보고 UserRepository에 Querydsl을 적용해 볼 것이다.



영상과 PPT를 함께 보시면 더욱 좋습니다 😊

먼저, 기존 Repository 구조에서 UserRepositoryCustom interface를 추가한다.

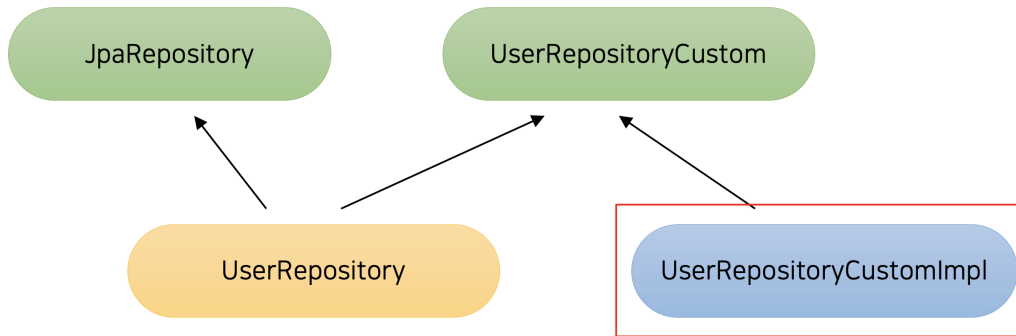
UserRepositoryCustom 은 UserRepository 와 같은 패키지에 넣어주었다.



```
interface UserRepositoryCustom {  
}
```

```
interface UserRepository : JpaRepository<User, Long>, UserRepositoryCustom {  
    fun findByName(userName: String): User?  
}
```

다음으로 UserRepositoryCustomImpl Class를 만들어 UserRepositoryCustom 을 구현하도록 한다. UserRepositoryCustomImpl 역시 UserRepository , UserRepositoryCustom 과 같은 패키지에 넣어주자.



```

class UserRepositoryCustomImpl : UserRepositoryCustom {

}

```

이제 다음과 같이 `JPAQueryFactory` 를 스프링 Bean으로 등록해준다. 이 `JPAQueryFactory` 를 활용해서 querydsl 코드를 작성할 예정이다. `QuerydslConfig` 는 `com.group.libraryapp.config` 패키지 안에 만들었다.

```

@Configuration
class QuerydslConfig(
    private val em: EntityManager,
) {

    @Bean
    fun querydsl(): JPAQueryFactory {
        return JPAQueryFactory(em)
    }

}

```

이제 `UserRepositoryCustom` 에 필요한 함수를 입력하고, `UserRepositoryCustomImpl`에 querydsl을 이용해 구현하면 된다!

```

interface UserRepositoryCustom {

    fun findAllWithHistories(): List<User>

}

```

```
class UserRepositoryCustomImpl(
    private val queryFactory: JPAQueryFactory,
) : UserRepositoryCustom {

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

여기서 사용된 `user`가 우리가 설정으로 만든 QClass (QUser) 의 `user`이다.

이때 사용된 Querydsl 문법의 의미는 다음과 같다.

- `select(user)` : `select user`
- `distinct()` : select 결과에 DISTINCT를 추가한다.
- `from(user)` : `from user`
- `leftJoin(userLoanHistory)` : `left join user_loan_history`
- `on(userLoanHistory.user.id.eq(user.id))` : `on user_loan_history.user_id = user.id`
- `fetchJoin` : 앞의 join을 fetch join으로 처리한다.
- `fetch()` : 쿼리를 실행하여 결과를 `List` 로 가져온다.

SQL과 거의 유사하고, 쿼리를 작성할 때 코드를 사용함으로써 오타가 발생하면 컴파일 단계에서 알 수 있다.

이제 `findAllWithHistories` 를 사용하는 서비스 테스트를 돌려 정상적으로 동작함을 확인해 보자~! 🍷

Test Results	885 ms
com.group.libraryapp.service.user.UserService	885 ms
✓ 유저 저장이 정상 동작한다	573 ms
✓ 유저 업데이트가 정상 동작한다	21 ms
✓ 유저 조회가 정상 동작한다	19 ms
✓ 대출 기록이 없는 유저도 응답에 포함된다	142 ms
✓ 대출 기록이 많은 유저의 응답이 정상 동작한다	34 ms
✓ 유저 삭제가 정상 동작한다	96 ms

이 방식의 장점은 다음과 같다.

- 서비스단에서 `UserRepository` 하나만 사용하면 된다.

단점은 다음과 같다.

- 인터페이스와 클래스를 항상 같이 만들어주어야 하는 것이 부담이고, 여러모로 번거롭다.

이제 다음 시간에는 Querydsl을 사용하는 두 번째 방법을 살펴보고 `BookRepository`의 `getStatus()`를 Querydsl로 변경해보자!

39강. Querydsl 사용하기 - 두 번째 방법

이번 시간에는 지난 시간에 했던 방법과 다른 방법을 활용하여 Querydsl을 사용할 것이다!

먼저 `BookQuerydslRepository`를 만들자. `Book.kt`가 위치한 `domain` 패키지에 만들어도 되지만, 이번에는 `com.group.libraryapp.repository.book` 패키지를 만들어 그 안에 넣어주겠다.

```
@Component
class BookQuerydslRepository(
    private val queryFactory: JPAQueryFactory,
) {

}
```

`JPAQueryFactory`를 주입 받을 수 있도록 `@Component` 어노테이션을 붙여주자! `@Repository`를 붙이더라도 상관 없다.

좋다~ 이제 바로 코드를 작성할 수 있다.

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

```
.fetch()
}
```

- 이번엔 `Projections.constructor()` 를 사용하였다.
 - 주어진 DTO의 생성자를 호출한다는 뜻이다.
 - `Projections.constructor()` 안에는 세 가지 파라미터가 들어갔다.
`BookStatResponse::class.java`, `book.type`, `book.id.count()`
- `select from`을 포함해 SQL으로 바꾸면 다음과 같다.
 - `select book.type, count(book.id) from book`
- `groupBy(book.type)` 은 SQL로 다음과 같다.
 - `group by type`

이전보다 살짝 복잡한 Querydsl 예시이지만, 직관적으로 사용할 수 있다.

이 Repository를 사용하려면 `BookService` 에서 `BookQuerydslRepository` 에 대한 의존성을 추가로 불러와야 한다.

```
@Service
class BookService(
    private val bookRepository: BookRepository,
    private val bookQuerydslRepository: BookQuerydslRepository, // 추가
    private val userRepository: UserRepository,
    private val userLoanHistoryRepository: UserLoanHistoryRepository,
) {
    // 생략...
}
```

이 방법의 장단점은 첫 번째 방법과 반대이다.

장점은

- 클래스만 바로 만들면 되어 간결하다는 점이고

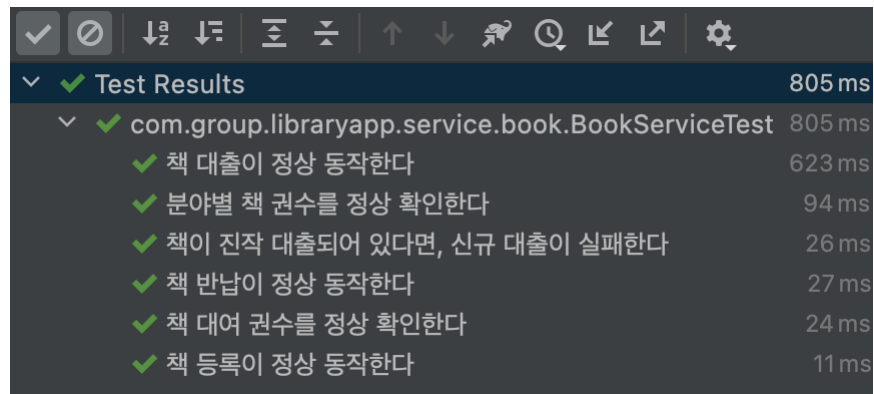
단점은

- 서비스단에서 필요에 따라 두 Repository를 모두 사용해주어야 한다는 점이다.

그렇다면 둘 중 어떤 방법을 사용하면 좋을까?!

개인적으로는 방금 사용한 두 번째 방법을 선호한다. 그 이유는 멀티 모듈을 사용할 경우, 모듈별로 각기 다른 Repository를 사용하는 경우가 많아, 단점이 상쇄되고 장점이 극대화되기 때문이다.

이제 테스트 코드까지 한 번 돌려주고~ 다음 시간에 이어서 UserLoanHistoryRepository에 있는 기능들도 Querydsl로 변경해주자!



✓	Test Results	805 ms
✓	com.group.libraryapp.service.book.BookServiceTest	805 ms
✓	책 대출이 정상 동작한다	623 ms
✓	분야별 책 권수를 정상 확인한다	94 ms
✓	책이 진작 대출되어 있다면, 신규 대출이 실패한다	26 ms
✓	책 반납이 정상 동작한다	27 ms
✓	책 대여 권수를 정상 확인한다	24 ms
✓	책 등록이 정상 동작한다	11 ms

40강. UserLoanHistoryRepository를 Querydsl으로 리팩토링 하기

이번 시간에는 UserLoanHistoryRepository에 있는 기능들을 Querydsl으로 옮겨볼 것이다. 그 전에 잠시 현재 UserLoanHistoryRepository를 살펴보자.

```
interface UserLoanHistoryRepository : JpaRepository<UserLoanHistory, Long> {  
  
    fun findByBookNameAndStatus(bookName: String, status: UserLoanStatus): UserLoanHistory?  
  
    fun countByStatus(status: UserLoanStatus): Long  
  
}
```

지금 존재하는 쿼리들은 모두 @Query를 사용하지 않고 Spring Data JPA가 자동으로 만들어 준 쿼리들이다. 이런 쿼리들도 Querydsl로 옮겨야 할까?!

개인적으로는 이 쿼리들 역시 Querydsl로 옮기는 것을 선호한다. 그 이유는 Querydsl을 사용함으로써 얻을 수 있는 장점인 '동적 쿼리의 간편함' 때문이다.

예를 들어 UserLoanHistoryRepository에 다음 함수도 추가로 존재했다고 하자.

```
fun findByBookName(bookName: String): UserLoanHistory?
```

이 기능은 책 이름을 기준으로 `UserLoanHistory` 를 찾는 기능으로 `findByBookNameAndStatus` 와 유사하지만 status가 존재하지 않는다.

이렇게 2가지 쿼리, `findByBookName` 과 `findByBookNameAndStatus` 를 보면서 느껴지는 기운이 있다!! 🤔

바로 Repository의 함수가 정말 많이 늘어날 수 있다는 사실이다!!!

예를 들어, 다음과 같은 조건들을 모두 만족하는 Repository 함수가 필요하다고 해보자. 단순한 and 조건의 쿼리이지만, 그 필드의 종류가 계속해서 달라질 수 있다.

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

이런 요구사항을 구현하려면 다음과 같이 많은 수의 메소드가 필요하다.

- findByA
- findByAAndB
- findByAAndC
- ...
- findByAAndBAndCAndDAndE

수학시간은 아니지만, 총 16개의 함수가 생길 것이다. 만약 선택적인 필드가 더 늘어난다면..?! 더욱더 많은 수의 함수가 생기게 된다.

이렇게 함수가 늘어나는 불편함은 Querydsl을 이용하면 간단히 해결된다.

`UserLoanHistoryRepository` 를 Querydsl로 변경해보자! 두 번째 방법을 사용해보겠다.

```
@Component
class UserLoanHistoryQuerydslRepository(
    private val queryFactory: JPAQueryFactory,
) {

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

```
}
}
```

우선 가장 첫 번째 함수인 `findByBookName` 을 Querydsl로 옮겨 보았다.

새로 나온 Querydsl 문법의 의미를 살펴보면 다음과 같다.

- `limit(1)` : SQL의 `limit 1` 이라는 의미로 모든 검색 결과에서 1개만을 가져온다는 의미이다.
- `fetchOne()` : `List<Entity>` 로 조회 결과를 반환하는 대신 `Entity` 하나만으로 조회 결과를 반환한다.

좋다~ 이제 `findByBookNameAndStatus` 역시 추가로 구현을 해볼 것이다. 이때 우리는 Kotlin의 특성을 활용해 다음과 같이 구현할 수 있다.

```
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()
}
```

- `find` 함수가 `status`라는 파라미터를 추가로 받게 했다. 이때 default parameter를 null로 넣었고 덕분에 외부에서는 `bookName`만 사용할 수도 `bookName`과 `status`를 같이 사용할 수도 있다.
- `?.let` 을 활용해 `status` 파라미터가 null인 경우에는 where 조건에 `user_loan_history.status = status` 가 들어가지 않도록 하였다.
 - where에 들어오는 조건이 null이면 Querydsl은 이를 무시하게 된다.
 - 또한 where에 여러 조건이 들어오면 각 조건이 AND로 결합한다.
 - 즉, `status`가 null이 아닌 경우에만 `book_name = ? and status = ?` 가 실행된다는 의미이다.

이어서 `countByStatus` 도 변경해보자.

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

추가적인 Querydsl 문법은 다음과 같다.

- `userLoanHistory.count()` : `count(id)` 로 변경된다
- `fetchOne() ?: 0L` : `count`의 결과는 숫자 1개이므로 `fetchOne()` 을 사용해준다! 혹시나 결과가 비어 있다면 0L을 반환하도록 `elvis` 연산자를 사용한다.

매우 좋다~ 🎉 이제 이전 Repository 기능들을 날려버리고, QuerydslRepository로 갈아끼워보자!

그런 다음 BookServiceTest 돌리면 Querydsl 적용 요구사항 완료이다!!! 🏃

✓ Test Results	978 ms
✓ com.group.libraryapp.service.book.BookServiceTest	978 ms
✓ 책 대출이 정상 동작한다	849 ms
✓ 분야별 책 권수를 정상 확인한다	31 ms
✓ 책이 진작 대출되어 있다면, 신규 대출이 실패한다	38 ms
✓ 책 반납이 정상 동작한다	30 ms
✓ 책 대여 권수를 정상 확인한다	20 ms
✓ 책 등록이 정상 동작한다	10 ms

41강. 마지막 요구사항 클리어!

우리는 아래와 같은 기술적인 요구사항을 완벽하게 적용하였다~!!! 🎉🎉🎉🎉

기술적인 요구사항

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

덕분에 이번 **Section 6. 네 번째 요구사항 추가하기 - Querydsl** 을 통해 다음과 같은 내용을 배울 수 있었다.

1. JPQL과 Querydsl의 장단점을 이해한다.
2. Querydsl을 Kotlin + Spring Boot와 함께 사용하고, 2가지 방식의 장단점을 이해한다.
3. Querydsl의 기본적인 사용법을 익힌다.
4. Querydsl을 활용해 기존 Repository를 리팩토링한다.

추가적으로 아직 Spring Data JPA를 활용한 기능들이 남아 있는데, Querydsl에 아직 익숙하지 않다면 이 기능들에도 Querydsl을 적용해 볼 수 있을 것이다! 👍



정말 고생 많으셨습니다!!! 🙏