섹션 5. 세 번째 요구사항 추가하기 - 책 통계

이번 섹션의 목표

1. SQL의 다양한 기능들(sum, avg, count, group by, order by) 을 이해한다
2. 간결한 함수형 프로그래밍 기법을 사용해보고 익숙해진다
3. 동일한 기능을 애플리케이션과 DB로 구현해보고, 차이점을 이해한다

32강. 책 통계 보여주기 - 프로덕션 코드 개발

드디어 마지막 Business 요구사항을 구현할 시간이다. 이번 시간에는 세 번째 추가 요구사항인 “책 통계 보여주기"에 대해 알아보고, 가장 간단한 방법을 활용해 API를 구현해보자.

책 통계 화면

- 현재 대여 중인 책이 몇 권이 보여준다.
- 분야별로 도서관에 등록되어 있는 책이 각각 몇 권인지 보여준다.

이 요구사항을 달성하기 위한 클라이언트 개발 역시 미리 끝나 있는 상황이다. 이제 우리가 해야할 것은 원하는 스펙에 맞춰 API를 만드는 것이다. 클라이언트는 다음과 같은 스펙을 요구하였다.

GET /book/loan (현재 대여 중인 책의 권수 보여주기)

요청 : 파라미터 없음

응답 (바로 숫자가 반환)

```
number
```

GET /book/stat (분야별로 등록되어 있는 책의 권수 보여주기)

요청 : 파라미터 없음

응답

```
[{
  "type": "COMPUTER",
  "count": 10
}, ...]
```

- count가 0이면, 반환 리스트에 존재하지 않아도 된다.

이 스펙에 맞춰 API를 구현해보자! 먼저 간단한, `GET /book/loan` 부터 개발하자!
BookController - BookService에 API를 구현할 것이다.

가장 간단한 방법은 다음과 같다.

```
// Controller
@GetMapping("/book/loan")
fun countLoanedBook(): Int {
  return bookService.countLoanedBook()
}
```

```
// Service
@Transactional(readOnly = true)
fun countLoanedBook(): Int {
  return userLoanHistoryRepository.findAllByStatus(UserLoanStatus.LOANED).size
}
```

```
// Repository
fun findAllByStatus(status: UserLoanStatus): List<UserLoanHistory>
```

매우 간단한 API이다! 좋다~ 👍 이어서 `GET /book/stat` 역시 개발하도록 하자.

DTO 구현부터 빠르게 작업해보자.

```
// DTO
data class BookStatResponse(
  val type: BookType,
  var count: Int,
) {
  fun plusOne() {
    this.count += 1
  }
}
```

```
// Controller
@GetMapping("/book/stat")
fun getBookStatistics(): List<BookStatResponse> {
  return bookService.getBookStatistics()
}
```

```
// Service
@Transactional(readOnly = true)
fun getBookStatistics(): List<BookStatResponse> {
  val result = mutableListOf<BookStatResponse>()
  val books = bookRepository.findAll()
  for (book in books) {
    result.firstOrNull { dto -> dto.type == book.type }?.plusOne()
      ?: result.add(BookStatResponse(book.type, 1))
  }
  return result
}
```

두 API의 개발이 끝났으니, 서버를 가동하고 데이터를 넣어 화면에 들어가보자!

모두 정상적으로 동작하는 것을 확인할 수 있다~!

등록된 책 분류	총 권 수
과학	0권
사회	0권
경제	0권
컴퓨터	2권
언어	0권

대출 중인 권 수 : **1권**

이제 다음 시간에는 테스트 코드를 만들고 조금 더 좋은 코드로 리팩토링 할 예정이다.

33강. 책 통계 보여주기 - 테스트 코드 개발과 리팩토링

이번 시간에는 테스트 코드를 통해 코드의 안정성을 확보하고, 통계 기능에 대한 리팩토링을 진행할 예정이다.

우선 현재 대여중인 책 권수 API부터 테스트 코드를 작성해보자.

```
@Test
@DisplayName("책 대여 권수를 정상 확인한다")
fun countLoanedBookTest() {
  // given
  val savedUser = userRepository.save(User("최태현", null))
  userLoanHistoryRepository.saveAll(
    listOf(
      UserLoanHistory.fixture(savedUser, "A"),
      UserLoanHistory.fixture(savedUser, "B", UserLoanStatus.RETURNED),
      UserLoanHistory.fixture(savedUser, "C", UserLoanStatus.RETURNED),
    )
  )

  // when
  val result = bookService.countLoanedBook()

  // then
  assertThat(result).isEqualTo(1)
}
```

이어서 책 통계 기능도 테스트 코드를 작성해보자!

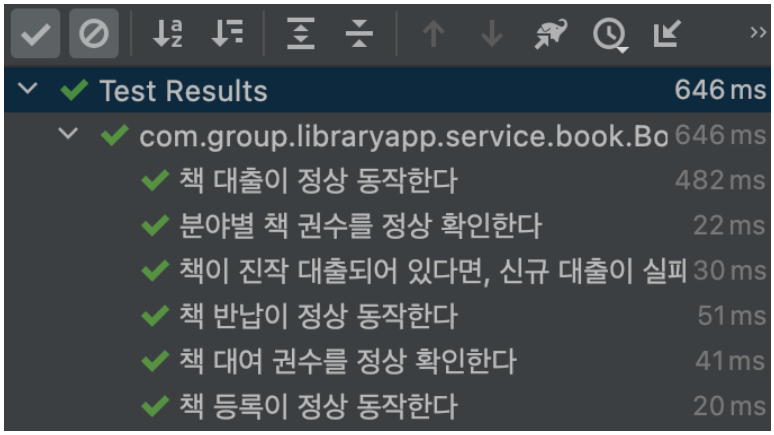
```
@Test
@DisplayName("분야별 책 권수를 정상 확인한다")
fun getBookStatisticsTest() {
  // given
  bookRepository.saveAll(listOf(
    Book.fixture("A", BookType.COMPUTER),
    Book.fixture("B", BookType.COMPUTER),
    Book.fixture("C", BookType.SCIENCE),
  ))

  // when
  val result = bookService.getBookStatistics()

  // then
  assertThat(result).hasSize(2)
  assertCount(result, BookType.COMPUTER, 2)
  assertCount(result, BookType.SCIENCE, 1)
}

private fun assertCount(result: List<BookStatResponse>, type: BookType, expectedCount:
Int) {
  assertThat(result.first { it.type == type }.count).isEqualTo(expectedCount)
}
```

BookServiceTest 전체를 돌리면 모두 잘 통과하는 것을 확인할 수 있다~!



이제 다시 한 번, 분야별 책 통계를 확인하는 서비스 코드를 봐보자. 현재 이 코드는

1. 로직이 복잡하고 call chain이 길며,
2. 가변 변수들이 많아, 다른 개발자가 실수할 여지가 많다.

실제로 서비스 코드 안의 result는 `MutableList`로 가변 리스트이고, DTO의 필드도 `var` 키워드를 사용해 다른 곳에서도 변경될 수 있다.

이럴 때 함수형 프로그래밍(Functional Programming)을 사용할 수 있다. <자바 개발자를 위한 코틀린 입문> 에서도 나왔던 `groupBy` 를 사용해보자.

```
// Service
@Transactional(readOnly = true)
fun getBookStatistics(): List<BookStatResponse> {
  return bookRepository.findAll()
    .groupBy { book -> book.type }
    .map { (type, books) -> BookStatResponse(type, books.size) }
}
```

```
// DTO
data class BookStatResponse(
  val type: BookType,
  val count: Int,
)
```

방금 만들었던 테스트 코드로 확인해보자. 매우 좋다~! 리팩토링을 통해 구현이 깔끔해졌다!

이제 정말 최선의 방법일까?! 아니면 추가적으로 개선할 수 있는 방법이 있을까?

다음 시간에는 추가적인 개선을 위해 필요한 SQL을 알아볼 예정이다.

34강. 다양한 SQL을 알아보자!

> 💡 영상과 PPT를 함께 보시면 더욱 좋습니다 😊

sum : 주어진 column의 합계를 계산한다

```
select sum(age) from user;
```

avg : 주어진 column의 평균을 계산한다

```
select avg(age) from user;
```

count : 개수를 센다

```
select count(*) from user;
```

group by : 주어진 column을 기준으로 그룹핑을 한다. 우리가 사용했던 groupBy 를 생각하면 된다.

```
select type, count(1) from book group by type;
```

order by : 주어진 column을 기준으로 정렬을 한다. 내림차순과 오름차순을 지정할 수 있다.

```
select * from book order by type desc; # 내림차순
select * from book order by type asc; # 오름차순, asc는 생략할 수 있다
```

이제 다음 시간에는 count, group by 등을 활용해 기존의 애플리케이션 로직의 단점을 살펴보고, 개선을 해보도록 하자.

35강. 애플리케이션 대신 DB로 기능 구현하기

이번 시간에는 지난 시간에 다루었던 SQL을 활용해 현재 존재하는 통계 기능들을 개선해보고, 어떻게 판단을 해야 하는지 살펴볼 예정이다.

```
// Repository
fun countByStatus(status: UserLoanStatus): Long
```

Repository에서는 Spring Data JPA에서 제공하는 `countByXXX` 를 활용할 수 있다.

```
@Transactional(readOnly = true)
fun countLoanedBook(): Int {
  return userLoanHistoryRepository.countByStatus(UserLoanStatus.LOANED).toInt()
}
```

toInt() 라는 타입 변환 메소드를 사용해주었다.

이제 테스트 코드를 돌려보면, 다음과 같은 SQL이 나오는 것을 확인할 수 있다!

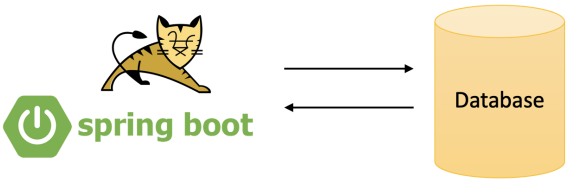
```
select
  count(userloanhi0_.id) as col_0_0_
from
  user_loan_history userloanhi0_
where
  userloanhi0_.status=?
```

그렇다면 기존의 구현과 어떤 차이가 있을까?!

> 💡 영상과 PPT를 함께 보시면 더욱 좋습니다 😊

이전 기능

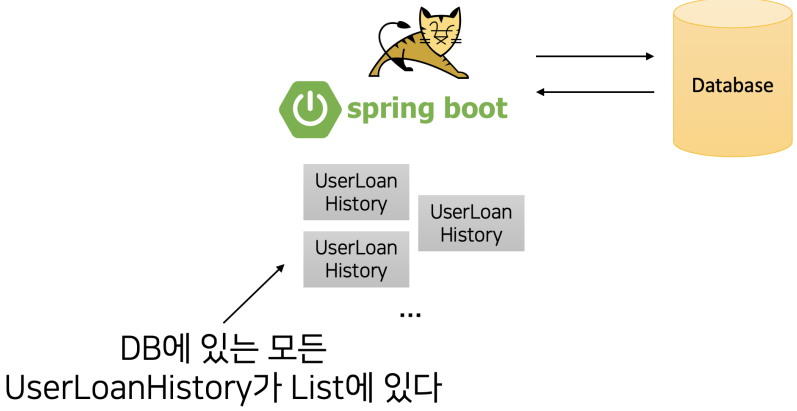
서버 코드를 보고 Query를 생각할 수 있어야 한다!

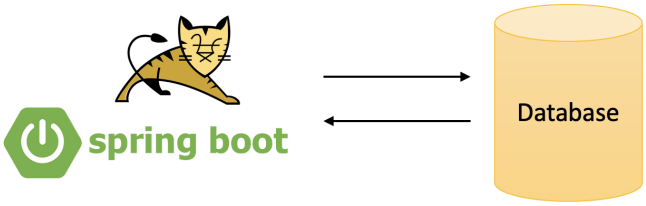


select * from user_loan_history where status = ?;

```
@Transactional(readOnly = true)
fun countLoanedBook(): Int {
  return userLoanHistoryRepository.findAllByStatus(UserLoanStatus.LOANED).size
}
```

select * from user_loan_history where status = ?;

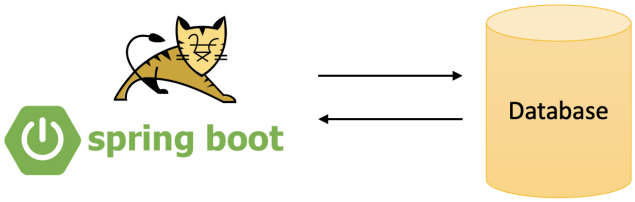




메모리에 존재하는 List의 size를 계산한다.

```
@Transactional(readOnly = true)
fun countLoanedBook(): Int {
  return userLoanHistoryRepository.findAllByStatus(UserLoanStatus.LOANED).size
}
```

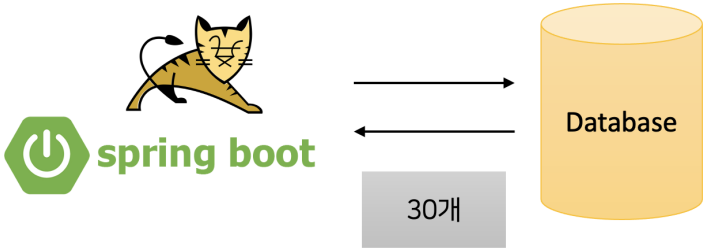
변경된 기능

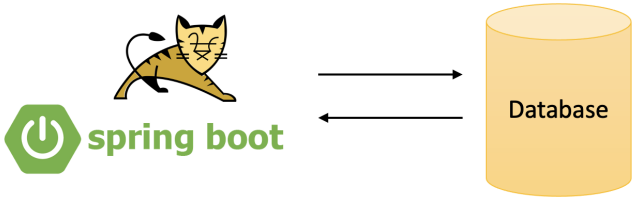


select count(*) from user_loan_history where status = ?;

```
@Transactional(readOnly = true)
fun countLoanedBook(): Int {
  return userLoanHistoryRepository.countByStatus(UserLoanStatus.LOANED).toInt()
}
```

select count(*) from user_loan_history where status = ?;





30L을 30으로 변환한다.

```
@Transactional(readOnly = true)
fun countLoanedBook(): Int {
  return userLoanHistoryRepository.countByStatus(UserLoanStatus.LOANED).toInt()
}
```

이어서 책의 분야별 통계 API도 변경해보자.

```
// Repository
@Query(
  "SELECT NEW com.group.libraryapp.dto.book.response.BookStatResponse(b.type, COUNT(b.
```

```
id)) " +
  "FROM Book b GROUP BY b.type"
)
fun getStats(): List<BookStatResponse>
```

```
// Service
@Transactional(readOnly = true)
fun getBookStatistics(): List<BookStatResponse> {
  return bookRepository.getStats()
}
```

group by 에서 사용된 count 역시 기본적으로 Long 타입의 결과가 나오기 때문에 DTO와 테스트 코드 역시 수정해주어야 한다.

```
// DTO
data class BookStatResponse(
  val type: BookType,
  val count: Long, // Int에서 Long으로 변경되었다
)
```

```
// BookServiceTest 클래스 내
private fun assertCount(result: List<BookStatResponse>, type: BookType, expectedCount:
Long) {
  assertThat(result.first { it.type == type }.count).isEqualTo(expectedCount)
}
```

현재 Spring Data JPA만을 사용해 불편한 점이 몇 가지 있는데, 다음 Section에서 Querydsl을 적용하면 해결될 것이다.

이렇게 코드를 바꾸고 나면, 아까와 비슷한 차이가 존재한다.

어떤 방법이 더 좋을까?!

전체 데이터 쿼리
메모리 로딩 + grouping

group by 쿼리

(상황에 따라 다르지만)
DB 및 Network 부하, 애플리케이션 부하가 덜 든다.
인덱스를 이용해 튜닝할 여지가 있다.

즉 상황에 따라서 코드 역시 변경되어야 하는 것이다. 추가로, 데이터가 일정 수준을 넘어가면 다양한 컴포넌트를 활용하는 시스템 아키텍처를 고민해야 한다. 예를 들어 다음과 같은 구성을 고민할 수 있다.

- 대용량 통계 처리 배치를 이용한 구조
- 이벤트 발행과 메시징 큐를 이용한 구조

매우 좋다~! 😊 😊 이번 시간에는 동일한 기능을 애플리케이션에서 처리하는 경우와, DB에서 처리하는 경우에 해당하는 구현을 각각 비교해보고, 어떤 차이가 있는 살펴보았다. 다음 시간에는 이번 Section을 간략히 정리해보도록 하자!

36강. 세 번째 요구사항 클리어!

우리는 아래와 같은 세 번째 요구사항을 완벽하게 구현하였다. 🎉🎉🎉

책 통계 화면

- 현재 대여 중인 책이 몇 권이 보여준다.
- 분야별로 도서관에 등록되어 있는 책이 각각 몇 권인지 보여준다.

덕분에 이번 `Section 5. 세 번째 요구사항 추가하기 - 책 통계` 를 통해 다음과 같은 내용을 배울 수 있었다.

1. SQL의 다양한 기능들(sum, avg, count, group by, order by) 을 이해한다
2. 간결한 함수형 프로그래밍 기법을 사용해보고 익숙해진다
3. 동일한 기능을 애플리케이션과 DB로 구현해보고, 특징과 장단점을 이해한다

이제 마지막 기술적인 추가 요구사항을 구현하러 가보자!! 🏃‍♀️🔥