섹션 1. 도서관리 애플리케이션 리팩토링 준비하기

이번 섹션의 목표
1강. 도서관리 애플리케이션 이해하기
    사용자 관련 기능
    책 관련 기능
2강. 테스트 코드란 무엇인가, 그리고 왜 필요한가?!
3강. 코틀린 코드 작성 준비하기
4강. 사칙연산 계산기에 대한 테스트 코드 작성하기
5강. 사칙연산 계산기의 나눗셈 테스트 작성
6강. Junit5 사용법과 테스트 코드 리팩토링
7강. Junit5으로 Spring Boot 테스트하기
8강. 유저 관련 기능 테스트 작성하기
9강. 책 관련 기능 테스트 작성하기
10강. 테스트 작성 끝! 다음으로!

이번 섹션의 목표

1. Java로 작성된 도서관리 애플리케이션을 이해한다.
2. 테스트 코드가 무엇인지, 왜 필요한지 이해하고
   Junit5를 사용해 Spring Boot의 테스트 코드를 작성한다.
3. 실제 만들어진 Java 프로젝트에 대해 Kotlin으로 테스트를 작성하며 Kotlin 코드 작성에 익숙해진다.

1강. 도서관리 애플리케이션 이해하기

> 영상과 PPT를 함께 보시면 더욱 좋습니다 😊

우리는 준비 시간 때, 도서관리 애플리케이션이 동작하는 것을 확인했다.

이제 우리의 미션은 Java로 작성된 애플리케이션을 이해하는 것이다.

기존 요구사항과 작성되어 있는 API, 도메인 구조, 코드의 흐름을 하나씩 살펴보자.

사용자 관련 기능

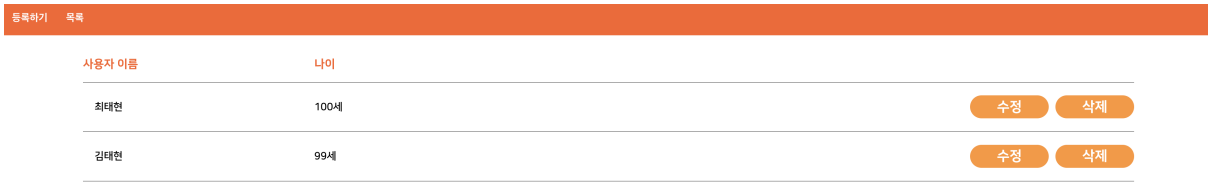
우선은 사용자와 관련된 기능이다.

- 도서관의 사용자를 등록할 수 있다. (이름 필수, 나이 선택)
- 도서관 사용자의 목록을 볼 수 있다.
- 도서관 사용자 이름을 업데이트 할 수 있다.
- 도서관 사용자를 삭제할 수 있다.

실제 개발되어 있는 화면에 들어가 확인해보니, 관련된 기능이 잘 동작하는 것을 알 수 있다.

```
💡 서버를 동작시킨 이후, http://localhost:8080/v1/index.html 으로
접속하시면 아래 화면을 보실 수 있습니다!
```

(사용자 등록 기능)



(사용자 조회 / 이름수정 / 삭제 기능)

이 기능과 관련한 API는 총 4가지가 존재한다.

`POST /user (유저 생성 API)`

요청

```
{
  "name": String
  "age": Int
}
```

응답 : 성공시 200 OK

`GET /user (유저 목록조회 API)`

요청 : 파라미터 없음

응답

```
[{
  "id": long
  "name": String
  "age": Int
}, ...]
```

`PUT /user (유저 이름변경 API)`

요청

```
{
  "id": Long
  "name": String // 변경되어야 하는 이름
}
```

응답 : 성공시 200 OK

`DELETE /user (유저 삭제 API)`

요청

```
{
  "name": String
}
```

응답 : 성공시 200 OK

책 관련 기능

다음은 책과 관련된 기능이다.

- 도서관에 책을 등록할 수 있다.
- 사용자가 책을 빌릴 수 있다.
    - 진작 대출되어 있는 책을
- 사용자가 책을 반납할 수 있다.

실제 개발되어 있는 화면을 보니, 관련된 기능이 잘 동작하는 것을 알 수 있다.

(책 등록 / 대출 기능 / 반납 기능)

이 기능과 관련한 API는 총 3가지가 존재한다.

`POST /book (도서 등록 API)`

요청

```
{
  "name": String // 책 이름
}
```

응답 : 성공시 200 OK

`POST /book/loan (도서 대출 API)`

요청

```
{
  "userName": String
  "bookName": String
}
```

응답 : 성공시 200 OK

`PUT /book/return (도서 반납 API)`

요청

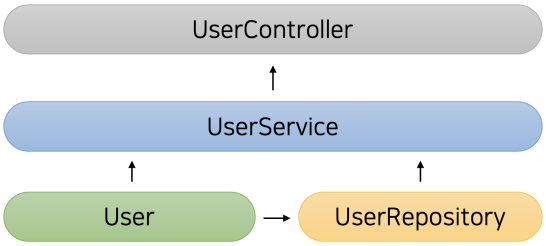
```
{
  "userName": String
  "bookName": String
}
```

응답 : 성공시 200 OK

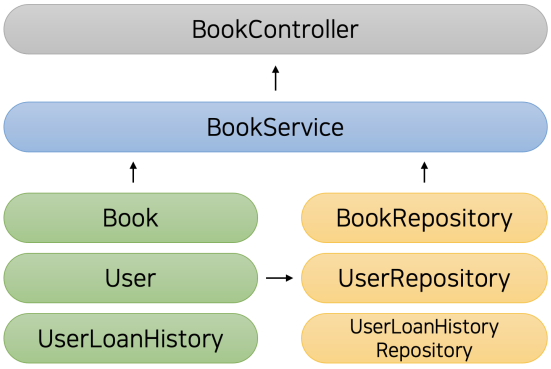
유저 및 책과 관련된 기능이 어떻게 구현되어 있는지 Controller → Service → Repository → Domain 들을 한 번 살펴보자

유저, 책 관련 기능에 관연하는 클래스들은 다음과 같다.

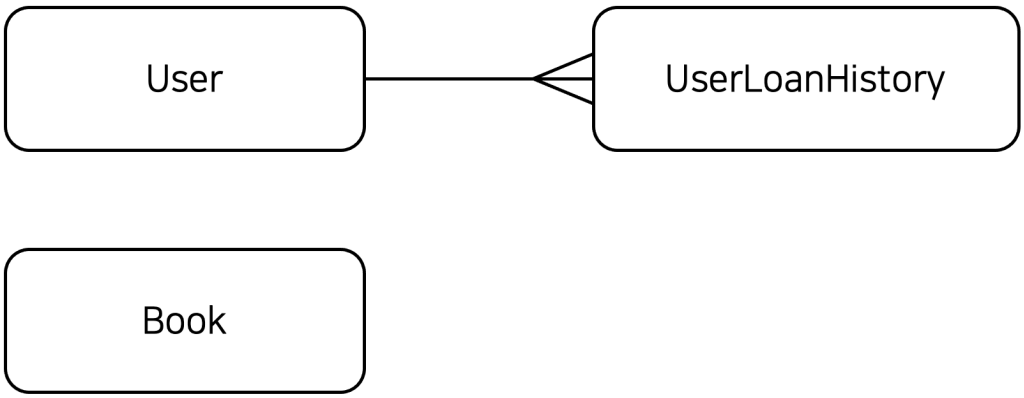
User 관련 기능 클래스 확인



Book 관련 기능 클래스 확인



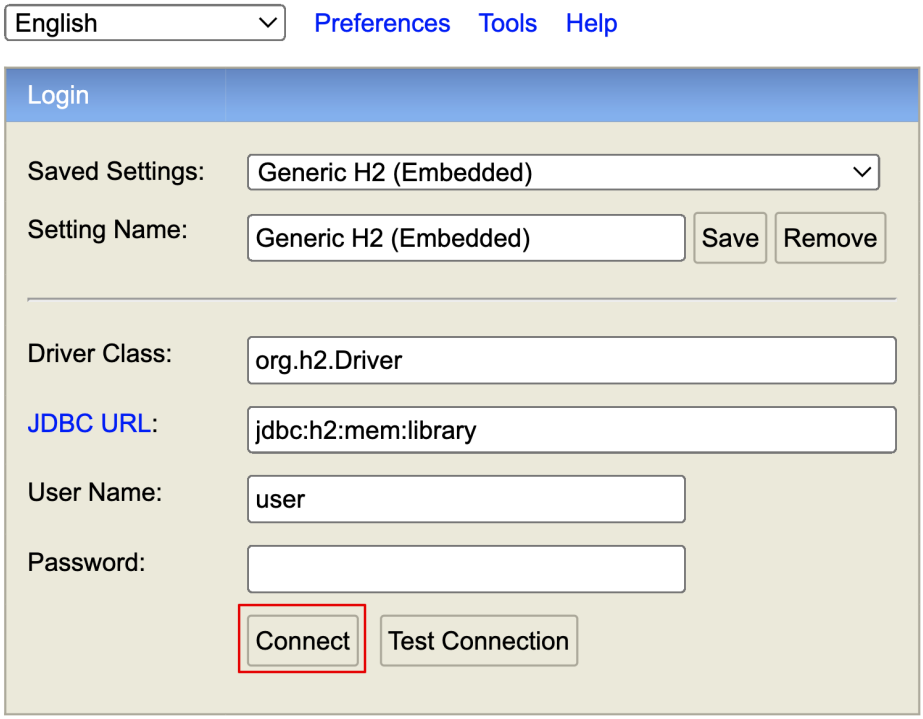
도메인을 정리하면 다음과 같다.



application.yml 의 설정은 다음과 같다.

```yaml
spring:
  datasource:
    url: 'jdbc:h2:mem:library'
    username: 'user'
    password: ''
    driver-class-name: org.h2.Driver
  jpa:
    hibernate:
      ddl-auto: create
    properties:
      hibernate:
        format_sql: true
        show_sql: true
  h2:
    console:
      enabled: true
      path: '/h2-console'
```

서버가 구동된 상태로 http://localhost:8080/h2-console 로 접속하면



H2에 접속해서 메모리 DB에 저장된 내용을 확인할 수 있다!!

매우 좋다 👍 이제 우리는 현재 버전의 도서관리 애플리케이션을 모두 이해하였다.

이제 Java로 만들어진 애플리케이션을 Kotlin으로 바꿀 것이다.

그 전에!! 매우 중요한 작업이 있다. 바로 **현재 기능에 대한 테스트를 작성하는 것**이다.

다음 시간에는 테스트 코드란 무엇인지, 그리고 왜 필요한지에 대해 살펴볼 예정이다.

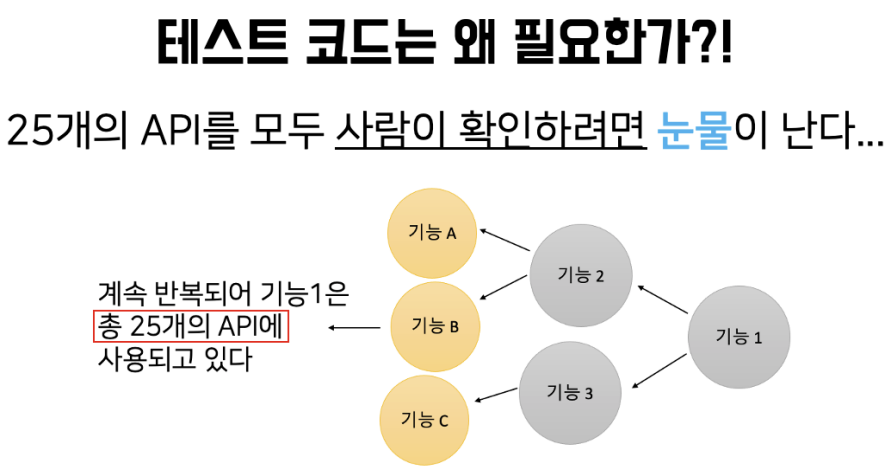
2강. 테스트 코드란 무엇인가, 그리고 왜 필요한가?!

> 💡 영상과 PPT를 함께 보시면 더욱 좋습니다 😊

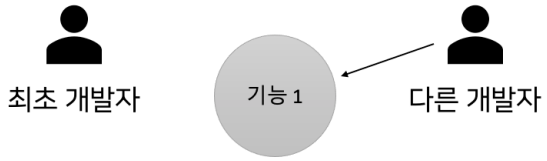
테스트 코드란 무엇인가

- 프로그래밍 코드를 사용해 무엇인가를 검증한다
- 즉, 자동으로 (사람의 손을 거치지 않고) 테스트를 할 수 있다!

테스트 코드는 왜 필요한가



테스트 코드는 왜 필요한가?!



최초 개발한 사람만큼 이해도가 높지 않을 수 있어,
테스트가 없다면 버그 확률이 높아진다 = 눈물

테스트 코드는 왜 필요한가?!

그런데, 리팩토링을 할 때마다
사람이 직접 수동으로 기능을 검증한다면?

1. 개발 과정에서 문제를 미리 발견할 수 있다.
2. 기능 추가와 리팩토링을 안심하고 할 수 있다.
3. 빠른 시간 내 코드의 동작 방식과 결과를 확인할 수 있다.
4. 좋은 테스트 코드를 작성하려 하다보면, 자연스럽게 좋은 코드가 만들어진다.
5. 잘 작성한 테스트는 문서 역할을 한다. (코드 리뷰를 돕는다.)

3강. 코틀린 코드 작성 준비하기

다음시간부터는 간단한 사칙연산 계산기를 만들고, 순수한 Kotlin으로 테스트를 작성해 볼 것이다.

본격적으로 Kotlin 코드를 작성하는 것이다!

우선 Kotlin 코드를 작성하기 위해 `build.gradle`에 다음과 같은 부분을 추가해주어야 한다.

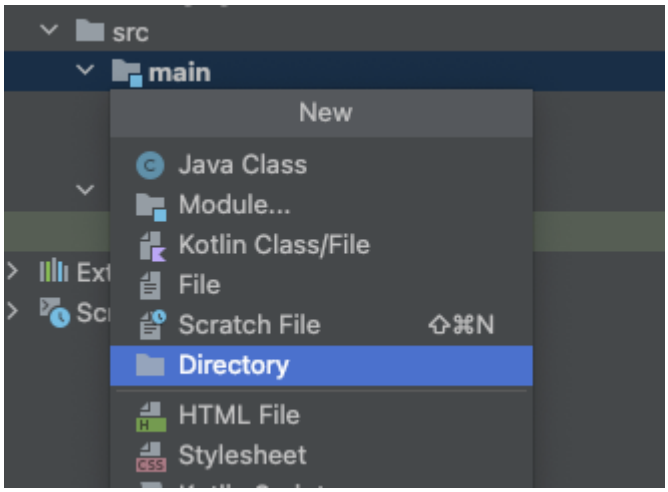
```
plugins {
    id 'org.jetbrains.kotlin.jvm' version '1.6.21'
}

dependencies {
    implementation "org.jetbrains.kotlin:kotlin-stdlib-jdk8"
}

compileKotlin {
    kotlinOptions {
        jvmTarget = "11"
    }
}
compileTestKotlin {
    kotlinOptions {
        jvmTarget = "11"
    }
}
```

모두 추가하고 나면, `build.gradle` 이 다음과 같이 변하게 된다.

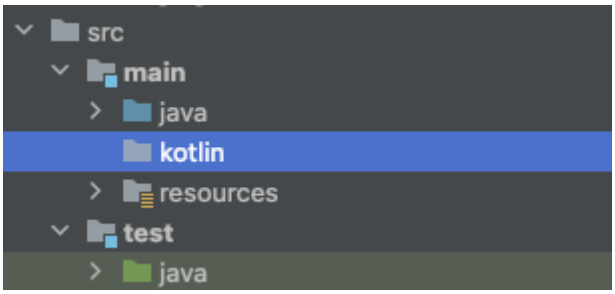
```
plugins {
    id 'org.springframework.boot' version '2.6.8'
    id 'io.spring.dependency-management' version '1.0.11.RELEASE'
    id 'java'
    id 'org.jetbrains.kotlin.jvm' version '1.6.21'
}

group = 'com.group'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = '11'

repositories {
    mavenCentral()
}

dependencies {
    implementation "org.jetbrains.kotlin:kotlin-stdlib-jdk8"
    implementation 'org.springframework.boot:spring-boot-starter-data-jpa'
    implementation 'org.springframework.boot:spring-boot-starter-web'
    runtimeOnly 'com.h2database:h2'
    testImplementation 'org.springframework.boot:spring-boot-starter-test'
}

tasks.named('test') {
    useJUnitPlatform()
}

compileKotlin {
    kotlinOptions {
        jvmTarget = "11"
    }
}
compileTestKotlin {
```

```
    kotlinOptions {
        jvmTarget = "11"
    }
}
```

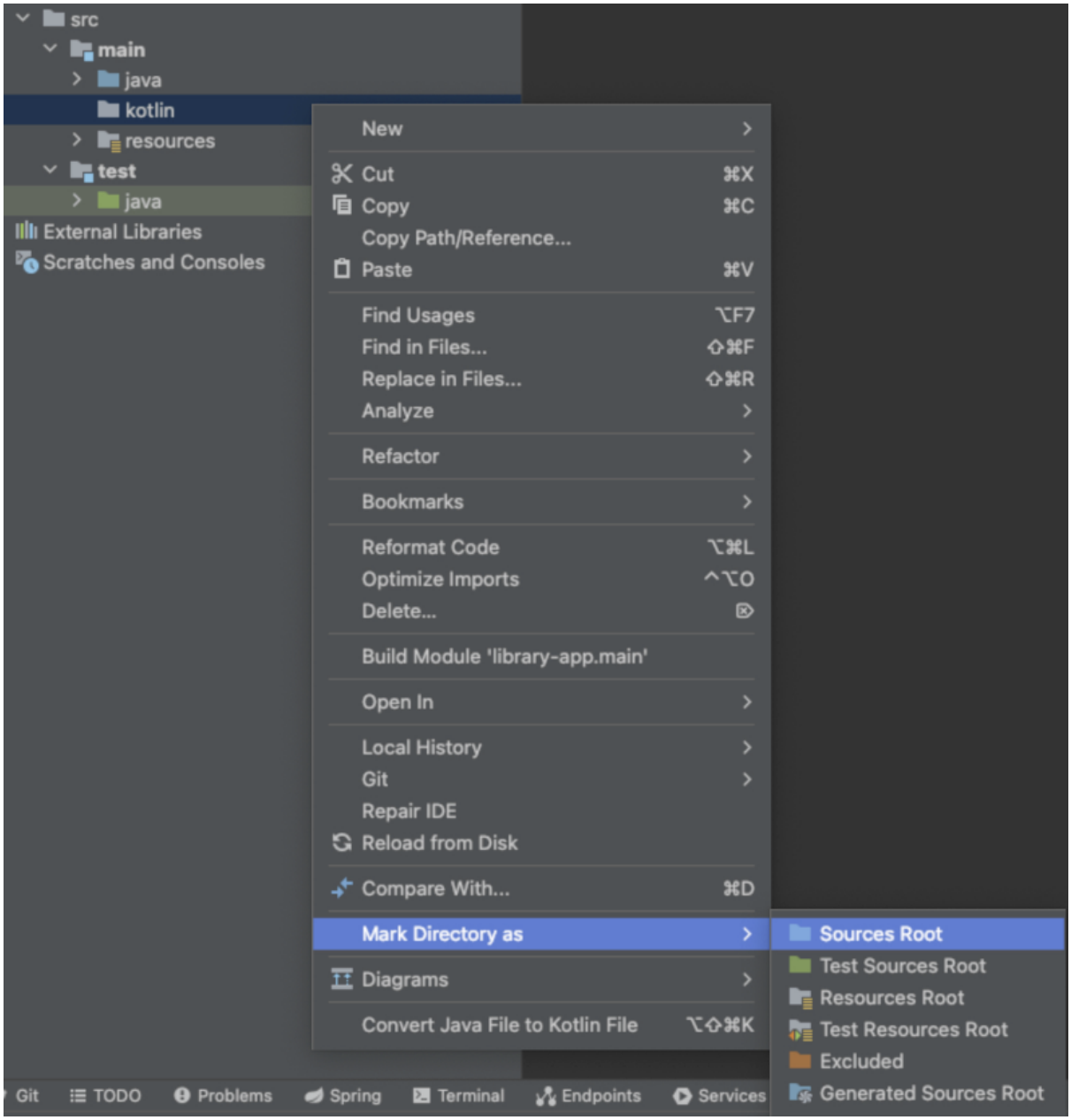
이제 kotlin 코드를 `src/main/` 과 `src/test/` 에 추가하기 위해 패키지 작업을 해주자.



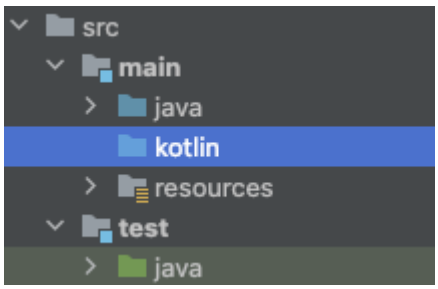
`src/main/` 아래에 kotlin 이라는 이름으로 Directory를 만들자.



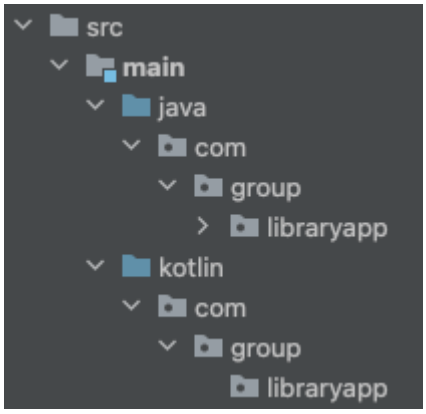
그리고 kotlin Directory에 마우스를 올리고 우클릭 > Mark Directory as > Source Root 를 클릭해주자.



그러면 kotlin 폴더의 색이 변한 것을 확인할 수 있다.

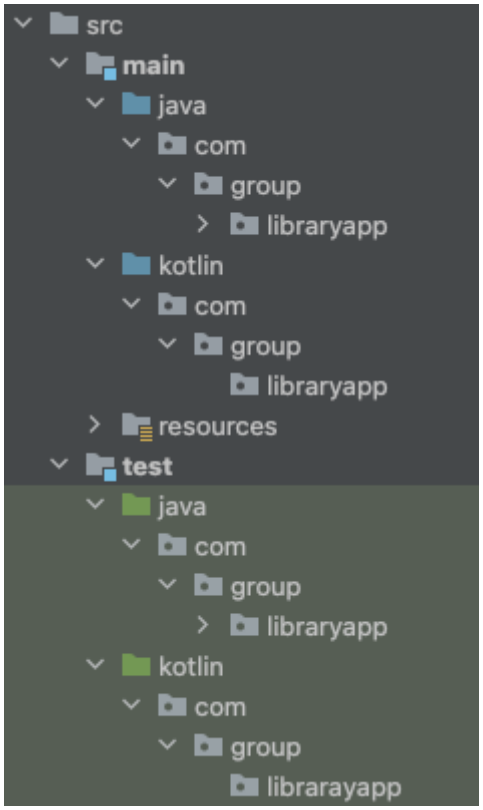


이제 `src/main/java/` 와 동일하게 `src/main/kotlin` 내부에 `com.goup.libraryapp` 패키지를 만들어주자



비슷하게 `src/test/` 에도 `kotlin` Directory를 만든 다음 우클릭 > Mark Directory as > Test Source Root 를 해주고, `com.group.libraryapp` 패키지를 만들어주자.

최종적으로 다음과 같은 패키지 구조를 가지게 된다.



4강. 사칙연산 계산기에 대한 테스트 코드 작성하기

Kotlin 코드를 프로젝트에 추가하기 위한 준비는 끝났다. 이번 강의에서는 사칙 연산 계산기를 만들고, Kotlin만을 이용해 계산기에 대한 테스트 코드를 만들어 볼 것이다.

사칙 연산 계산기에 대한 요구사항은 다음과 같다.

1. 계산기는 정수만을 취급한다.
2. 계산기가 생성될 때 숫자를 1개 받는다.
3. 최초 숫자가 기록된 이후에는 연산자 함수를 통해 숫자를 받아 지속적으로 계산한다.

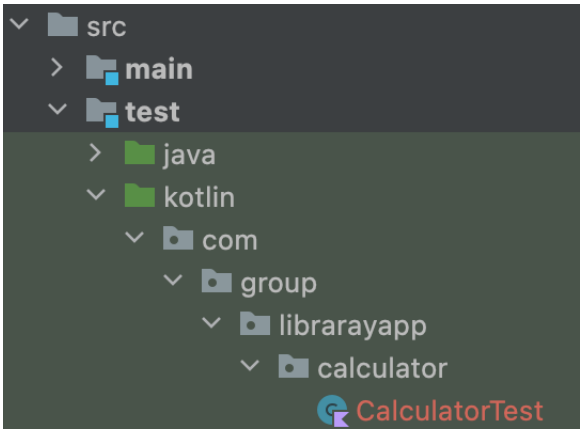
계산기는 `com.group.libraryapp.calculator` 패키지에 만들었다.

```
💡 알림 💡
본 강의에서는 글자 크기를 키워 보여드리고 있는데, 한 화면에서 조금이라도 많은 글자를 보여드리기 위해 들여쓰기로 스페이스바 2칸을 사용하고 있습니다.

코드를 복사 붙여넣기 하신 후
- MAC : option + command + L
- Windows or Linux : Ctrl + Alt + L
단축키를 사용하시면 여러분들 설정에 맞게 포맷팅 됩니다!😊
```

```
class Calculator(
  private var number: Int,
) {

  fun add(operand: Int) {
    this.number += operand
  }

  fun minus(operand: Int) {
    this.number -= operand
  }

  fun multiply(operand: Int) {
    this.number *= operand
  }

  fun divide(operand: Int) {
    if (operand == 0) {
      throw IllegalArgumentException("0으로 나눌 수 없습니다")
    }
    this.number /= operand
  }

}
```

이제 테스트 클래스를 만들자. 테스트 클래스는 `src > test > kotlin > com > group > libraryapp > calculator` 에 CalculatorTest로 만들어주었다.

- 테스트 클래스를 만들 때에는 test directory내에 같은 패키지를 사용하고, 테스트 대상 클래스 뒤에 Test라는 이름을 붙이는 것이 관례이다.



가장 간단한 덧셈부터 테스트를 만들어 보자.

```
class CalculatorTest {

  fun addTest() {
    val calculator = Calculator(5)
    calculator.add(3)
  }

}
```

5라는 숫자를 가지고 있는 계산기 인스턴스를 만들어 3을 더해주었다. 이제 우리는 계산기 안에 8이라는 값이 있는 것을 기대하므로 8이 들어있지 않다면 예외를 낼 것이다. **이때 계산기 안에 8이 있는지 확인하는 방법은 2가지가 있다.**

첫 번째는 data class의 `equals`를 사용하는 방법이다. Calculator를 data class로 변경하고 아래 코드를 실행시켜보자

```
data class Calculator(
  private var number: Int,
) {
  // 생략...
}
```

```
fun addTest() {
  val calculator = Calculator(5)
  calculator.add(3)
  val expectedResult = Calculator(8)
  if (calculator != expectedResult) {
    throw IllegalStateException()
  }
}
```

data class의 경우, 자동으로 `equals` 를 만들어 주기 때문에 8을 가지고 있는 계산기 인스턴스를 만든 다음 `!=` 연산자로 두 계산기가 같은지 다른지 확인할 수 있다.

두 번째 방법을 소개하기 전에, 첫 테스트 메소드를 만들었으니 main 함수를 만들어 실행시켜보자.

```
fun main() {
  val calculatorTest = CalculatorTest()
  calculatorTest.addTest()
}
```

실행을 해보면, 아무런 에러가 나지 않는다! 즉 테스트가 성공한 것이다. 만약 Calculator의 add메소드가 누군가에 의해 오동작하게 바뀐다면 main 함수에서는 에러가 날 것이고 우리는 코드를 통해 이를 확인할 수 있다.

```
// Calculator.kt의 add 메소드
// 본문을 주석처리하고 main 함수를 실행시켜 보자

fun add(operand: Int) {
  // this.number += operand
}
```

두 번째 방법은, Calculator로부터 number를 가져오는 방법이다. 현재는 Calculator가 private number를 가지고 있기 때문에 `calculator.number` 로 number 프로퍼티를 가져올 수 없다.

number를 가져오기 위해서는 간단히 private 대신 public을 사용할 수도 있고

```
class Calculator(
  var number: Int,
) {
  // 생략...
}
```

원래 존재하던 number 라는 프로퍼티를 `_number` 라는 이름으로 변경하고 불변 `number` 를 추가하는 방법도 있다.

```
class Calculator(
  private var _number: Int,
) {

  val number: Int
    get() = this._number
```

```
  fun add(operand: Int) {
    this._number += operand // this._number로 바꿔야 한다.
  }

  // 나머지 메소드 생략...
}
```

이를 Kotlin에서는 `backing property` 라고 하며, backing property에는 `_` 를 쓰는 것이 컨벤션이다.

public 프로퍼티를 사용하는 방법은 setter가 노출된다는 단점이 있고, backing property를 사용하는 방법은 코드가 장황해진다는 단점이 있다.

개인적으로는 setter가 노출되더라도 setter를 바로 사용하지 않는 것이 익숙해지면, 간결한 코드가 주는 효과가 더 크다고 생각해 본 강의에서는 첫 번째 방법을 주로 사용할 예정이다.

테스트 코드는 다음과 같이 변경된다.

```
fun addTest() {
  val calculator = Calculator(5)
  calculator.add(3)
  if (calculator.number != 8) {
    throw IllegalStateException()
  }
}
```

data class를 활용하는 것보다 조금 간결해짐을 확인할 수 있다.

테스트 메소드를 들여다보면, 크게 3가지 과정으로 구성되어 있는 것을 확인할 수 있다.

1. 테스트 대상을 만들어 준비하는 과정
2. 실제 우리가 테스트 하고 싶은 기능을 호출하는 과정
3. 호출 이후 의도한대로 결과가 나왔는지 확인하는 과정

이 과정을 보다 명확하게 구분하기 위해서 다음과 같이 각 과정에 앞에 주석을 적어주고 한 칸씩 띄여쓰기를 할 수 있다.

```
fun addTest() {
  // given
  val calculator = Calculator(5)

  // when
  calculator.add(3)

  // then
  if (calculator.number != 8) {
```

```
    throw IllegalStateException()
  }
}
```

이것을 given - when - then 패턴이라고 한다.

여기까지 중간 정리를 해보자. 우리는 계산기를 만들고, 계산기에 대한 덧셈 기능 테스트 코드를 만들었다.

테스트 코드를 만들 때 결과를 검증하는 부분에서는

1. data class를 활용해 객체 자체로 비교(`equals()`)를 할 수도 있었고
2. 필요한 프로퍼티를 getter를 통해 가져와 비교할 수도 있었다.

이제 이어서 뺄셈과 곱셈에 대한 테스트도 만들어보자.

```
fun minusTest() {
  // given
  val calculator = Calculator(5)

  // when
  calculator.minus(3)

  // then
  if (calculator.number != 2) {
    throw IllegalStateException()
  }
}

fun multiplyTest() {
  // given
  val calculator = Calculator(5)

  // when
  calculator.multiply(3)

  // then
  if (calculator.number != 15) {
    throw IllegalStateException()
  }
}
```

테스트 코드를 작성하고 메인 메소드에도 추가해 실행을 시켜보자.

완전 좋다! 우리는 자동화된 테스트 코드를 만들었다!! 다음 시간에는 나눗셈 기능에 대한 테스트를 작성할 예정이다.

5강. 사칙연산 계산기의 나눗셈 테스트 작성

이제 나눗셈을 테스트 차례이다. 나눗셈은 다른 기능들과 다르게 예외가 있기 때문에 2가지로 테스트를 해야 한다.

1. 0이 아닌 다른 숫자를 넣었을때 나눗셈이 정상적으로 동작하는지
2. 0을 넣었을때 우리가 의도한 예외가 발생하는지

첫 번째 부터 빠르게 작성해보자.

```
fun divideTest() {
  // given
  val calculator = Calculator(5)

  // when
  calculator.divide(2)

  // then
  if (calculator.number != 2) {
    throw IllegalStateException()
  }
}
```

테스트가 잘 통과한다!

이제 0을 넣었을 때 우리가 의도한 예외가 발생하는지 테스트를 작성해보자.

```
fun divideExceptionTest() {
  // given
  val calculator = Calculator(5)

  // when & then
  try {
    calculator.divide(0)
  } catch(e: java.lang.IllegalArgumentException) {
    // 테스트 통과
    return
  } catch(e: Exception) {
    throw IllegalStateException()
  }
  throw IllegalStateException("기대하는 예외가 발생하지 않았습니다.")
}
```

`try catch` 구문과 `throw` 를 이용해 IllegalArgumentException이 나왔을 때만 테스트가 통과하게 해두었다.

추가적으로 메시지를 확인해줄 수도 있다. 첫 번째 catch에 한 번 if문을 중첩해 처리하는 방식이다.

```
try {
  calculator.divide(0)
} catch(e: IllegalArgumentException) {
  if (e.message != "0으로 나눌 수 없습니다") {
    throw IllegalStateException("메시지가 다릅니다.")
  }
  return
}
```

프로덕션 코드(`Calculator`)의 메시지를 수정하고 테스트를 돌려보면 실패하는 것을 확인할 수 있다.

매우 좋다. 우리는 모든 경우에 대해 철저하게 테스트 코드를 작성하였다. 이제 누군가 Calculator의 코드를 잘못 건들면 테스트를 통해 쉽게 확인할 수 있다.

하지만 이렇게 수동으로 만든 테스트 코드에는 다음과 같은 단점이 있다.

1. 테스트 클래스가 많아지고 메소드가 생길 때마다 메인 메소드에 수동으로 코드를 작성해 주어야 하고, 메인 메소드가 아주 커질 것이다.
    - 테스트 메소드를 개별적으로 실행하기도 어렵다.
2. 테스트가 실패한 경우 무엇을 기대하였고, 어떤 잘못된 값이 들어와 실패했는지 알려주지 않는다. 또한 예외를 던지거나, try catch를 사용해야 하는 등 직접 구현해야 할 부분이 많다.
3. 테스트 메소드별로 공통적으로 처리해야 하는 기능이 있다면, 메소드마다 중복이 생기게 된다.
    - 예를 들어, 현재 모든 테스트 메소드에서는 5를 초기값으로 가지고 있는 계산기를 만들어 준다.

이러한 단점을 극복하기 위해 테스트 프레임워크가 등장하였는데, Java-Kotlin 진영에서 가장 많이 사용되는 Juni5를 알아보자.

6강. Junit5 사용법과 테스트 코드 리팩토링

이번 시간에는 Junit5의 사용법을 알아보고, 테스트 코드를 리팩토링 해보자.

이 프로젝트에는 Junit5 설정이 미리 되어 있기 때문에 추가적으로 설정해줄 부분은 없다.

우선 Junit5에서 필수적으로 사용되는 5가지 어노테이션을 알아보자.

```
class JunitTest {

  companion object {
    @JvmStatic
    @BeforeAll
    fun beforeAll() {
      println("모든 테스트 시작 전")
    }

    @JvmStatic
    @AfterAll
    fun afterAll() {
      println("모든 테스트 실행 후")
    }
  }

  @BeforeEach
  fun beforeEach() {
    println("각 테스트 시작 전")
  }

  @AfterEach
  fun afterEach() {
    println("각 테스트 실행 후")
  }

  @Test
  fun test1() {
    println("테스트 1")
  }

  @Test
  fun test2() {
    println("테스트 2")
  }

}
```

각 어노테이션은 다음과 같은 의미를 가진다.

- `@Test` : 테스트 메소드를 지정한다. 테스트 메소드를 실행하는 과정에서 오류가 없으면 성공이다.
- `@BeforeEach` : 각 테스트 메소드가 수행되기 전에 실행되는 메소드를 지정한다.
- `@AfterEach` : 각 테스트가 수행된 후에 실행되는 메소드를 지정한다.
- `@BeforeAll` : 모든 테스트를 수행하기 전에 최초 1회 수행되는 메소드를 지정한다.
    - 코틀린에서는 `@JvmStatic` 을 붙여 주어야 한다.

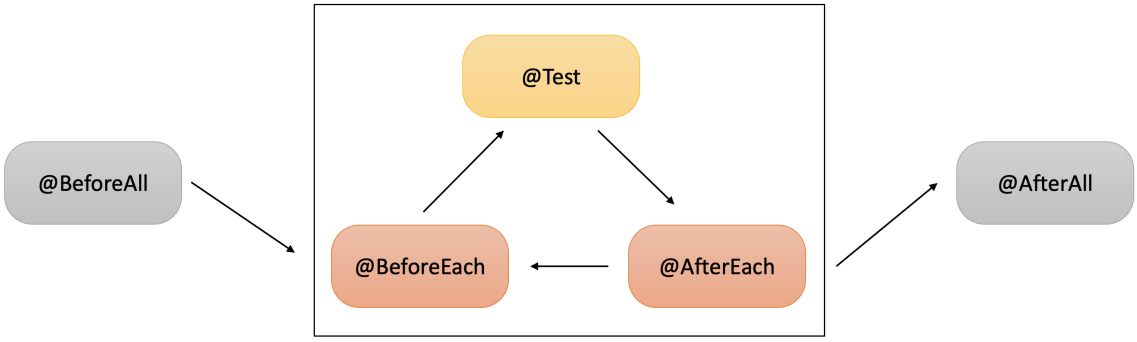
- `@AfterAll` : 모든 테스트를 수행한 후 최후 1회 수행되는 메소드를 지정한다.
    - 코틀린에서는 `@JvmStatic` 을 붙여 주어야 한다.

Junit5를 사용하면, 메소드 단위로 테스트를 실행시킬 수 있고 클래스 단위로 테스트를 실행시킬 수도 있다.

위의 테스트 클래스 전체를 실행시키면, 다음과 같은 결과가 console에 출력된다.

```
모든 테스트 시작 전
각 테스트 시작 전
테스트 1
각 테스트 실행 후
각 테스트 시작 전
테스트 2
각 테스트 실행 후
모든 테스트 실행 후
```

아래 그림과 같은 순서로 어노테이션이 실행되는 것이다.



이제 계산기 테스트 코드를 Junit5 기반으로 변경해보자. 먼저 덧셈 테스트이다.

```
class CalculatorTestV2 {

  @Test
  fun addTest() {
    // given
    val calculator = Calculator(5)

    // when
    calculator.add(3)

    // then
    assertThat(calculator.number).isEqualTo(8)
  }
```

```
}
```

또한 `assertThat(calculator.number).isEqualTo(8)` 라는 코드가 등장하였는데 `calculator.number` 는 8이길 기대한다로 해석할 수 있다.

이렇게 단언문(assert문)을 작성해 놓으면 에러가 났을 때 기대하는 것이 8이었는데 다른 값이 들어와 오류가 났다고 알려준다.

```
org.opentest4j.AssertionFailedError:
expected: 8
 but was: 5
```

단언문은 `assertThat(확인하고싶은값)` 으로 시작하고 뒤에 `.isEqualTo( )` 가 붙게 된다. 이때 `isEqualTo` 는 정확히 동일한 것을 기대한다는 의미이다.

자주 사용되는 단언문을 몇 가지 소개해보겠다.

- 주어진 값이 true인지 / false인지 검증한다.

```
val isNew = true

assertThat(isNew).isTrue
assertThat(isNew).isFalse
```

- 주어진 컬렉션이 size가 원하는 값인지 검증한다.

```
val people = listOf(Person("A"), Person("B"))
assertThat(people).hasSize(2)
```

- 주어진 컬렉션 안의 item 들에서 name 이라는 프로퍼티를 추출한 후 (extracting), 그 값을 검증한다.
    - 이때 순서는 중요하지 않다.

```
val people = listOf(Person("A"), Person("B"))
assertThat(people).extracting("name").containsExactlyInAnyOrder("A", "B")
```

- 주어진 컬렉션 안의 item 들에서 name 이라는 프로퍼티를 추출한 후 (extracting), 그 값을 검증한다.

- 이때 **순서도 중요**하다.

```
val people = listOf(Person("A"), Person("B"))
assertThat(people).extracting("name").containsExactly("A", "B")
```

- 함수(`function1()`)를 실행했을 때 원하는 예외가 나오는지 검증한다.

```
assertThrows<IllegalArgumentException> {
  function1()
}
```

- 예외 메시지까지 검증할 수 있다.

```
val message = assertThrows<IllegalArgumentException> {
  function1()
}.message
assertThat(message).isEqualTo("잘못된 값이 들어왔습니다")
```

이어서 계산기의 나머지 테스트 코드도 Junit5로 리팩토링 해보자.

```
@Test
fun minusTest() {
  // given
  val calculator = Calculator(5)

  // when
  calculator.minus(3)

  // then
  assertThat(calculator.number).isEqualTo(2)
}

@Test
fun multiplyTest() {
  // given
  val calculator = Calculator(5)

  // when
  calculator.multiply(3)

  // then
  assertThat(calculator.number).isEqualTo(15)
}

@Test
fun divideTest() {
```

```
  // given
  val calculator = Calculator(5)

  // when
  calculator.minus(3)

  // then
  assertThat(calculator.number).isEqualTo(2)
}

@Test
fun divideExceptionTest() {
  // given
  val calculator = Calculator(5)

  // when & then
  val message = assertThrows<IllegalArgumentException> {
    calculator.divide(0)
  }.message
  assertThat(message).isEqualTo("0으로 나눌 수 없습니다")
}
```

<자바 개발자를 위한 코틀린 입문> 강의에서 다루었던 scope function을 활용하면 예외에 대한 메시지 검증을 아래와 같이 리팩토링 할 수도 있다.

```
// when & then
assertThrows<IllegalArgumentException> {
  calculator.divide(0)
}.apply {
  assertThat(message).isEqualTo("0으로 나눌 수 없습니다")
}
```

매우 좋다! 이제 다음 시간에는 Juni5를 활용해 Spring Boot 애플리케이션을 테스트 하는 방법에 대해 다룰 예정이다.

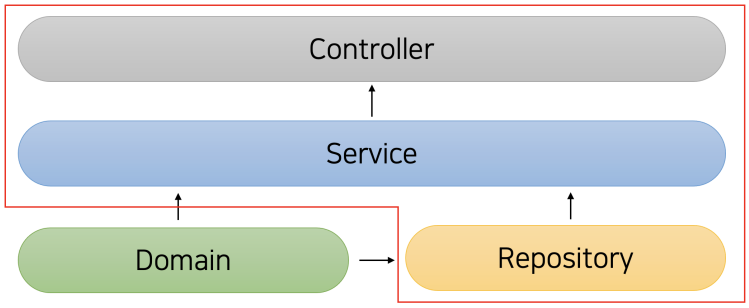
7강. Junit5으로 Spring Boot 테스트하기

> 💡 영상과 PPT를 함께 보시면 더욱 좋습니다 😊

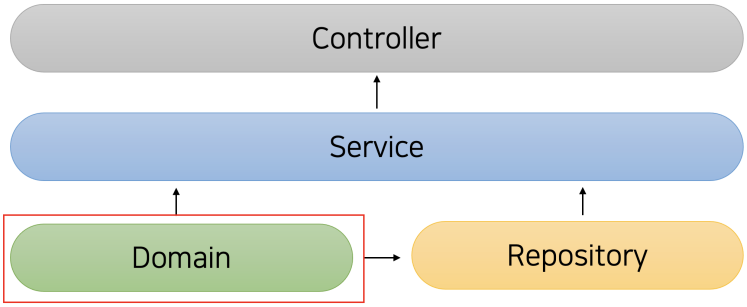
이번 시간에는 Junit5을 활용해 Spring Boot 애플리케이션을 어떻게 테스트하지 다룰 예정이다.

그 전에, Controller - Service - Repository - Domain 계층 중 어디를 테스트 하는 것이 좋을지 잠깐 생각해보자.

Spring Boot 각 계층별 특징

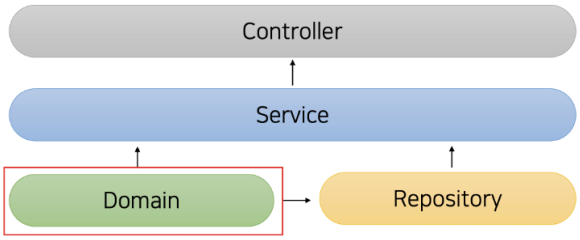


스프링 컨텍스트에 의해 관리되는 Bean



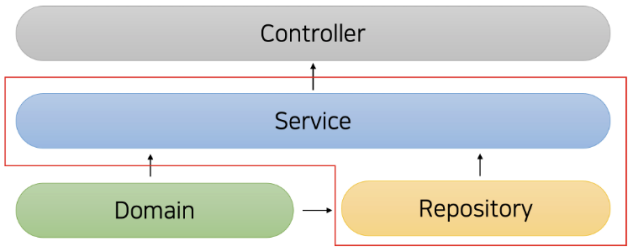
순수한 Java 객체 (POJO)

Spring Boot의 각 계층을 테스트하는 방법



Domain 계층

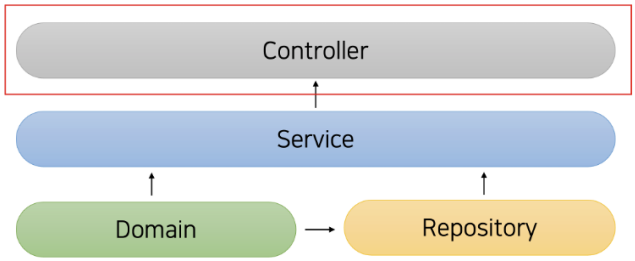
클래스를 테스트하는 것과 동일



Service, Repository 계층

스프링 빈을 사용하는
테스트 방법 사용
(@SpringBootTest)

데이터 위주의 검증



Controller계층

스프링 빈을 사용하는
테스트 방법 사용
(@SpringBootTest)

응답 받은 JSON을 비롯한
HTTP 위주의 검증

어떤 계층을 테스트 해야 할까?!

- 당연히 최선은 모든 계층에 대해 많은 경우를 검증하는 것!
- 하지만 현실적으로 코딩 시간을 고려해 딱 1개 계층만 테스트 한다면
  일반적인 상황에서는, 개인적으로 Service 계층 테스트를 선호한다.
- Service 계층을 테스트 함으로써 A를 보냈을 때 B가 잘 나오는지, 또는 원하는 로직을 잘 수행하는지 검증할 수 있기 때문이다.

그럼 이제 `UserService.java` 의 유저 저장 로직을 테스트 해보자.

```
@SpringBootTest
class UserServiceTest @Autowired constructor(
  private val userService: UserService,
  private val userRepository: UserRepository,
) {

  @Test
  fun saveUserTest() {
    // given
    val request = UserCreateRequest("최태현", null)

    // when
    userService.saveUser(request)

    // then
    val users = userRepository.findAll()
    assertThat(users).hasSize(1)
  }

}
```

- `@SpringBootTest` : 스프링 컨텍스트를 띄우는 테스트 임을 표시한다. 이 테스트가 실행될 때는 컨텍스트가 자동으로 뜨게 된다.

    - UserServiceTest 클래스가 `com.group.libraryapp.service.user` 에 있어야 어노테이션 만으로 동작한다.
- `@Autowired` : 생성자를 통해 Bean을 주입받기 위한 어노테이션

위의 테스트 코드는 `user` 테이블에 데이터를 저장하도록 명령을 내리고, 실제 DB를 조회해 1명의 유저가 들어 있는지 확인한다. 첫 Spring Boot Service 테스트이다! 아주 좋다~ 👍

추가로 저장된 유저의 이름과 나이가 맞는지도 확인해보자.

```
assertThat(users[0].name).isEqualTo("최태현")
assertThat(users[0].age).isNull()
```

`assertThat().isNull()` 이라는 새로운 단언문을 사용하였다.

위의 테스트 코드를 실행해보면 `users[0].age` 에서 에러가 나게 된다!!

```
users[0].age must not be null
```

그 이유는 다음과 같다.

- User getAge()가 Java 타입으로 Integer라고 되어 있는데 (즉, 플랫폼 타입이다.)
- Kotlin 입장에서는 이 Int가 nullable인지 non-nullable인지 몰라 `users[0].age` 라고만 타이핑한 경우 null이 아닌 변수에 age를 담으려고 한다.
- 하지만 실제 값은 null이었기 때문에 null을 non-null 변수에 넣으려가 에러가 난 것이다!

이를 해결하기 위해서는 User 테이블의 getter 2개에 Annotation을 붙여주면된다.

name은 null이 불가능하니 `@NotNull` 을 붙여주고, age는 null이 가능하니 `@Nullable` 을 붙여주자.

`org.jetbrains.annotations.NotNull` , `org.jetbrains.annotations.Nullable` 을 활용하면 된다.

```
@NotNull
public String getName() {
  return name;
}
```

```
@Nullable
public Integer getAge() {
  return age;
}
```

그후 테스트를 다시 돌리면, Kotlin이 어노테이션을 참고해 변수를 할당하기 때문에 age를 nullable한 변수로 인식하고, 테스트가 정상적으로 동작하게 된다!! 🎉

매우 좋다~!! 다음 시간에는 아직 작성하지 않은 유저 조회, 수정, 삭제 기능에 대한 테스트 코드를 작성해보자.

8강. 유저 관련 기능 테스트 작성하기

이번 시간에는 이전 시간에 작성했던 '유저 생성 테스트'를 제외한 나머지 테스트를 작성할 계획이다.

먼저, 조회 테스트부터 작성해보자. 조회의 경우 유저가 DB에 진작 저장되어 있어야 할 것이다.

```
@Test
fun getUsersTest() {
  // given
  userRepository.saveAll(listOf(
    User("A", 20),
    User("B", null),
  ))

  // when
  val results = userService.getUsers()

  // then
  assertThat(results).hasSize(2)
  assertThat(results).extracting("name").containsExactlyInAnyOrder("A", "B")
  assertThat(results).extracting("age").containsExactlyInAnyOrder(20, null)
}
```

테스트를 작성하고, `getUsersTest()` 를 실행시켜 보면 정상적으로 동작한다!!!

`userService.getUsers()` 부분에서 getter method는 코틀린의 프로퍼티로 바꿀 수 있다는 Weak Warning이 나오는데 실제 `getUsers()` 는 단순한 getter가 아니니 다음과 같이 바꾸지 않도록 주의하자.

```
// Kotlin에서는 Java의 getUsers() 함수를 다음과 같이 바꿀 수 있지만,
// 이는 단순한 getter가 아니므로 그렇지 않도록 주의해야 한다.
```

```
val results = userService.users
```

이전 6강에서 Junit5를 사용하면 전체 테스트를 한 번에 실행할 수 있다고 언급한 적이 있다. 이제 테스트가 2개 되었으니 전체 테스트를 실행해보자

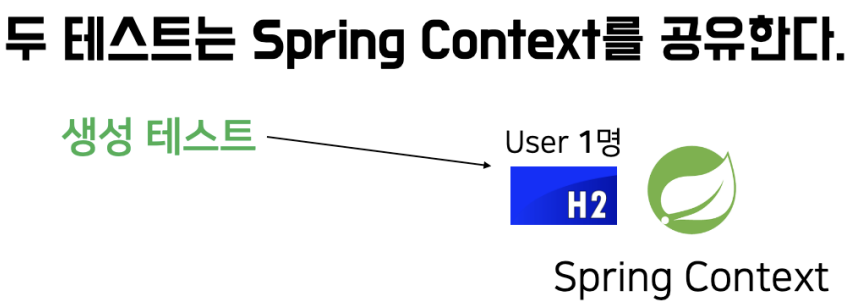
전체 테스트를 실행해보니 에러가 난다!!! 분명 1개만 실행시켰을 때는 잘 동작하였는데 왜 에러가 난 것일까?!

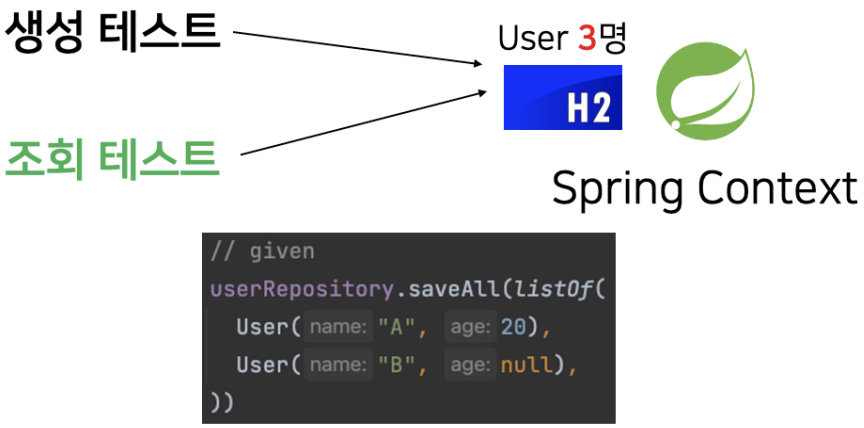
```
에러메시지 :
Expected size: 2 but was: 3 in:
[com.group.libraryapp.dto.user.response.UserResponse@43fdef43,
    com.group.libraryapp.dto.user.response.UserResponse@3f36c191,
    com.group.libraryapp.dto.user.response.UserResponse@73973e77]
```

(만약 다른 에러가 발생하더라도 원인은 동일하다.)

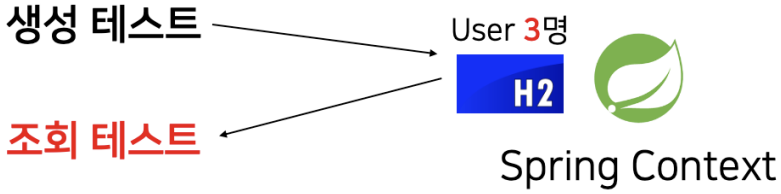
그 이유는 다음과 같다.

> 💡 영상과 PPT를 함께 보시면 더욱 좋습니다 😊






```
// given
userRepository.saveAll(listOf(
  User( name: "A",  age: 20),
  User( name: "B",  age: null),
))
```




```
// then
assertThat(results).hasSize(2)
```

위의 그림에서 알 수 있듯이, 두 테스트를 함께 실행시킬 경우, 테스트가 실패하는 이유는 같은 Spring Context 안의 H2 DB를 공유하기 때문이다.

이러한 문제를 해결하기 위해서는, 테스트가 끝날 때마다 DB를 깨끗이 비워 주어야 한다. then 절 가장 마지막에 `userRepository.deleteAll()` 을 추가해주자!

```
// then (생성 테스트)
val users = userRepository.findAll()
assertThat(users).hasSize(1)
assertThat(users[0].name).isEqualTo("최태현")
assertThat(users[0].age).isEqualTo(null)
userRepository.deleteAll() // 추가된 deleteAll

// then (조회 테스트)
assertThat(results).hasSize(2)
assertThat(results).extracting("name").containsExactlyInAnyOrder("A", "B")
assertThat(results).extracting("age").containsExactlyInAnyOrder(20, null)
userRepository.deleteAll() // 추가된 deleteAll
```

하지만 이렇게 되면, 테스트가 추가 될 때마다 `userRepository.deleteAll()` 이 중복될 것이다! 😢

이럴때 Junit5의 `@AfterEach` 를 활용할 수 있다. `@AfterEach` 는 각 테스트가 수행된 후 바로 수행되는 함수를 지정할 수 있으므로 다음과 같이 설정해줄 수 있다.

```
@AfterEach
fun clean() {
  userRepository.deleteAll()
}
```

매우 좋다~! 이제 유저 관련 테스트를 추가하더라도 `userRepository.deleteAll()` 이 중복되지 않는다.

이제 User 이름 수정 기능에 대한 테스트를 작성하자.

```
@Test
fun updateUserNameTest() {
  // given
  val savedUser = userRepository.save(User("A", null))
  val request = UserUpdateRequest(savedUser.id!!, "B")

  // when
  userService.updateUserName(request)

  val result = userRepository.findAll()[0]
  assertThat(result.name).isEqualTo("B")
}
```

테스트를 작성하는 과정에서 `User.java` 에 다음과 같은 코드를 추가로 작성해주었다.

```
@Nullable
public Long getId() {
  return id;
}
```

User가 최초 DB에 저장되기 이전에는 id가 null로 들어갈 수 있기 때문에 `@Nullable` 로 처리해주었고, 테스트 코드에서 널 아님 단언을 사용해 주었다.

테스트가 잘 통과하는 것을 확인할 수 있다.

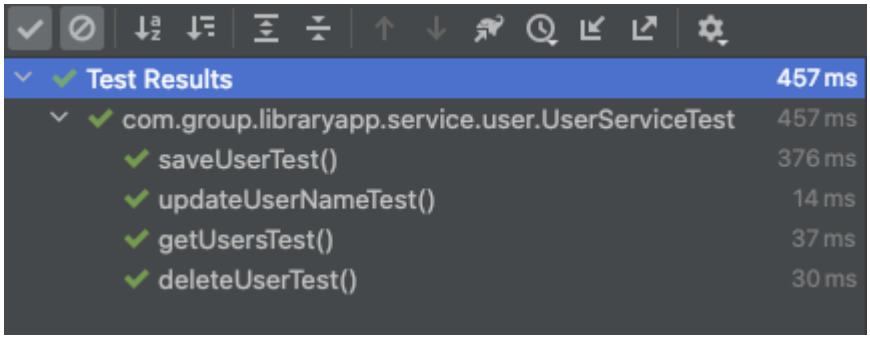
마지막으로 User 삭제 기능에 대한 테스트를 작성하자.

```
@Test
fun deleteUserTest() {
```

```
  // given
  userRepository.save(User("A", null))

  // when
  userService.deleteUser("A")

  // then
  assertThat(userRepository.findAll()).isEmpty()
}
```

`assertThat().isEmpty()` 라는 추가적인 단언을 사용하였다.

이제 전체 테스트를 실행시켜보면

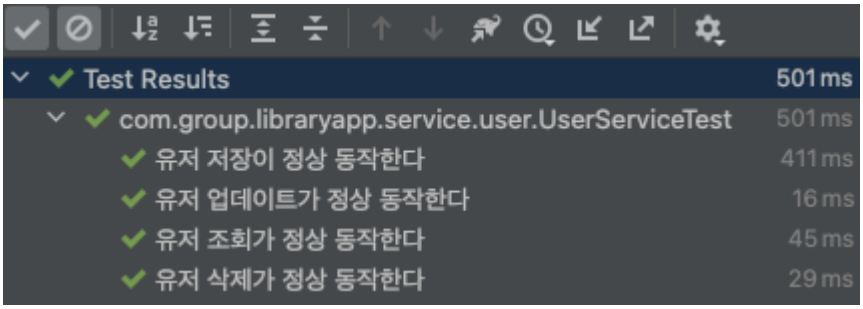


위의 사진과 같이 모두 통과한 테스트 코드를 얻을 수 있다!!!

테스트 실행을 할 때에 조금 더 직관적인 한글 이름을 부여하고 싶다면 Junit5의 `@DisplayName` 어노테이션을 테스트 메소드에 붙여줄 수도 있다!! 예를 들어 삭제 테스트의 경우는 다음과 같이 어노테이션을 붙일 수 있다.

```
@Test
@DisplayName("유저 삭제가 정상 동작한다")
fun deleteUserTest() {
  // given
  userRepository.save(User("A", null))

  // when
  userService.deleteUser("A")

  // then
  assertThat(userRepository.findAll()).isEmpty()
}
```

나머지 테스트들에도 `@DisplayName` 을 붙여주면, 테스트 실행화면은 조금 더 친숙한 언어로 바뀌게 된다. 😊



아주 좋다~! 🎉🎉 우리는 이제 Java를 Kotlin으로 마음껏 리팩토링 할 때에 코드의 동작을 보장할 수 있다!!! 👍

또한 Kotlin으로 테스트 코드를 작성하며 Kotlin 문법에 익숙해지고, 플랫폼 타입과 같은 Java와 Kotlin을 사용할 때 주의할 점 몇 가지를 익혔다.

다음 시간에는 지금까지 연습했던 노하우를 총 적용해 `BookService.java` 에 대한 테스트 코드들을 작성해보자.

9강. 책 관련 기능 테스트 작성하기

이제 책과 관련한 기능에 대해 테스트를 작성해보자! 먼저 책 등록에 관한 테스트이다.

```
@SpringBootTest
class BookServiceTest @Autowired constructor(
  private val bookService: BookService,
  private val bookRepository: BookRepository,
) {

  @AfterEach
  fun clean() {
    bookRepository.deleteAll()
  }

  @Test
  @DisplayName("책 등록이 정상 동작한다")
  fun saveBookTest() {
    // given
    val request = BookRequest("이상한 나라의 엘리스")

    // when
    bookService.saveBook(request)

    // then
    val books = bookRepository.findAll()
    assertThat(books).hasSize(1)
    assertThat(books[0].name).isEqualTo("이상한 나라의 엘리스")
  }
```

```
}
```

지금까지 배웠던 기능들이 대거 등장했다.

- `@SpringBootTest` : 스프링 애플리케이션을 테스트 하기 위한 Context 조성한다.
- `@Autowired constructor` : 생성자 앞에 `@Autowired` 를 사용해 생성자 Bean 주입을 받게 한다.
- `@AfterEach` : 각 테스트가 끝나면 실행되는 함수를 지정한다.
- `@Test` : 테스트 메소드를 지정한다.
- `@DisplayName` : 테스트가 실행될 때 어떤 이름을 보여줄지 지정한다.
- `@NotNull` : 책의 이름은 null이 될 수 없으므로 `Book.java` 의 `getName()` 메소드에 어노테이션을 붙여주었다.

이제 추가로 책 대여 기능을 테스트해보자. 대여 기능은 지금까지 테스트 했던 기능들 중에 제일 복잡하다.

책 대여 기능은 다음과 같은 흐름을 가지고 있다.

- 책을 가져온다
  - 해당 책이 대출이 되어 있는지 확인한다
  - 대출 되어있다면 에러를 발생시키고, 대출되어 있지 않다면, 다음 단계로 넘어간다
- 유저를 가져와 책을 대여시킨다
  - 이때 UserLoanHistory가 생기게 된다

대여 기능은 2가지를 테스트해야 한다.

책이 대출이 정상적으로 잘 되는 경우 (happy case) 대출이 진작 되어 있는 책을 빌려 대출이 실패하는 경우

먼저 대출이 정상적으로 잘 되는 경우부터 테스트 해보자.

```
@Test
@DisplayName("책 대출이 정상 동작한다")
fun loanBookTest() {
  // given
  bookRepository.save(Book("이상한 나라의 엘리스"))
  val savedUser = userRepository.save(User("최태현", null))
  val request = BookLoanRequest("최태현", "이상한 나라의 엘리스")

  // when
```

```
  bookService.loanBook(request)

  // then
  val results = userLoanHistoryRepository.findAll()
  assertThat(results).hasSize(1)
  assertThat(results[0].bookName).isEqualTo("이상한 나라의 엘리스")
  assertThat(results[0].user.id).isEqualTo(savedUser.id)
  assertThat(results[0].isReturn).isFalse
}
```

대출이 되었다는 의미는 해당 유저와 연결된 UserLoanHistory 데이터 1개가 생겼다는 의미이므로, 이를 활용해 검증해줄 수 있다.

이 테스트를 실행하게 되면 User 데이터와 UserLoanHistory 데이터가 생기게 되니 `clean()` 에 `userRepository.deleteAll()` 을 추가해주자!

```
@AfterEach
fun clean() {
  bookRepository.deleteAll()
  userRepository.deleteAll()
}
```

`userRepository.deleteAll()` 메소드만 호출하면 되는 이유는, 연관관계에 있는 자식 테이블까지 찾아서 데이터를 모두 지워주기 때문이다.

테스트를 실행시켜보면, 성공하는 것을 확인할 수 있다!

좋다~! 🎉 이제 예외 case인 대출이 진작 되어 있는 책을 빌리려해 대출이 실패하는 경우를 테스트 해보자.

```
@Test
@DisplayName("책이 진작 대출되어 있다면, 신규 대출이 실패한다")
fun loanBookFailTest() {
  // given
  bookRepository.save(Book("이상한 나라의 엘리스"))
  val savedUser = userRepository.save(User("최태현", null))
  userLoanHistoryRepository.save(UserLoanHistory(savedUser, "이상한 나라의 엘리스", fals
e))
  val request = BookLoanRequest("최태현", "이상한 나라의 엘리스")

  // when & then
  assertThrows<java.lang.IllegalArgumentException> {
    bookService.loanBook(request)
  }.apply {
    assertThat(message).isEqualTo("진작 대출되어 있는 책입니다")
```

```
  }
}
```

- given 절에서 아직 대출중인 (isReturn = false) 책 대여 기록 (UserLoanHistory) 을 만들어 주었고
- when & then 절에서는 `assertThrows` 를 활용해 이 메소드를 실행할 때 `IllegalArgumentException` 가 나는지 확인해주었다.

마지막으로, 책을 반납하는 경우를 테스트 해보자.

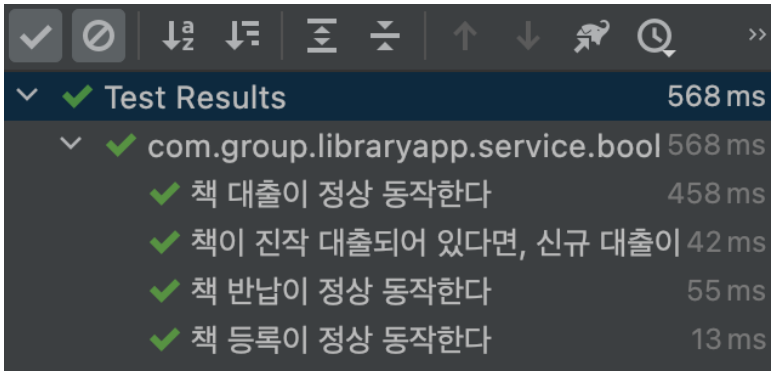
```
@Test
@DisplayName("책 반납이 정상 동작한다")
fun returnBookTest() {
  // given
  bookRepository.save(Book("이상한 나라의 엘리스"))
  val savedUser = userRepository.save(User("최태현", null))
  userLoanHistoryRepository.save(UserLoanHistory(savedUser, "이상한 나라의 엘리스", fals
e))
  val request = BookReturnRequest("최태현", "이상한 나라의 엘리스")

  // when
  bookService.returnBook(request)

  // then
  val results = userLoanHistoryRepository.findAll()
  assertThat(results).hasSize(1)
  assertThat(results[0].isReturn).isTrue
}
```

드디어 BookService에 대한 모든 테스트를 작성했다!!

BookServiceTest 옆의 테스트 실행 버튼을 눌러, BookServiceTest에 존재하는 모든 테스트를 실행시켜보자!

```
@SpringBootTest
class BookServiceTest @Autowired constructor(
  private val bookService: BookService,
  private val bookRepository: BookRepository,
  private val userRepository: UserRepository,
  private val userLoanHistoryRepository: UserLoanHistoryRepository,
) {
```

모든 테스트가 성공한 것을 확인할 수 있다~!!!



10강. 테스트 작성 끝! 다음으로!

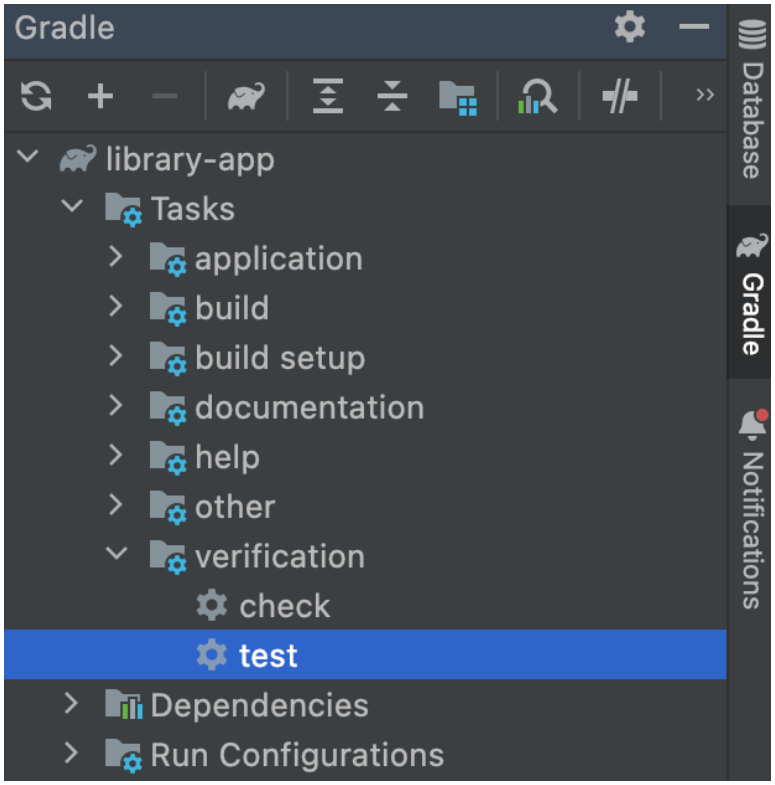
우리는 현재 도서관리 애플리케이션에 존재하는 기능들에 대한 테스트를 작성했다.

만약 모든 테스트를 돌리고 싶다면, 터미널에서

```
./gradlew test
```

를 타이핑 하거나

IntelliJ에서 `gradle > Tasks > verification > test` 를 더블클릭 하면 된다.



우리는 `Section 1. 도서관리 애플리케이션 리팩토링 준비하기` 를 통해 다음을 배울 수 있었다.

1. Kotlin을 사용하기 위해 필요한 설정 방법
2. 테스트란 무엇이고, 왜 중요한가
3. Junit5의 기초 사용법
4. Junit5와 Spring Boot를 함께 사용해 테스트를 작성하는 방법
5. 여러 API에 대한 Service 계층 테스트 실습

또한 주요 기능에 대한 테스트를 모두 작성해두었기 때문에 다음 Section에서 **Java로 작성된 프로덕션 코드를 Kotlin으로 변경할 때 보다 안심하고 작성**할 수 있게 되었다. 다음 시간부터는 어떤 방식으로 리팩토링을 진행할 것인지 살펴보고, 본격적인 리팩토링에 착수할 예정이다.