
spring

테스트 코드

- 작성한 코드가 의도대로 잘 동작하고 예상치 못한 문제가 없는지 확인할 목적으로 작성하는 코드가 바로 테스트 코드이다.

- 테스트 코드는 test 디렉토리에서 작업하게 된다.

- 테스트 코드에는 다양한 패턴이 있는데, 그 중 학습할 패턴은 given-when-then 패턴이다.
- given-when-then 패턴은 세 단계로 작성하는 방식이다.
  - 테스트의 실행을 준비하는 given 단계
  - 테스트를 진행하는 when 단계
  - 테스트 결과를 검증하는 then 단계

JUnit

- Junit은 자바 언어를 위한 단위 테스트 프레임워크이다.
- 단위 테스트란, 작성한 코드가 의도대로 작동하는지 작은 단위로 검증하는 것을 의미한다. 여기서 작은 단위는 보통 메소드가 된다.

- Junit이 갖는 특징은 아래와 같다.
  - 테스트 방식을 구분할 수 있는 애너테이션을 제공한다.
  - @Test 애너테이션으로 메소드를 호출할 때마다 새 인스턴스를 생성하여 독립적인 테스트가 가능하다.
  - 예상 결과를 검증하는 assertion 메소드를 제공한다.
  - 사용 방법이 단순하여, 테스트 코드 작성 시간이 적다.
  - 자동 실행, 자체 결과를 확인하고 즉각적인 피드백을 얻을 수 있다.

JUnit

```java
public class JUnitTest {
    @DisplayName("1 + 2는 3이다.") // 테스트 이름
    @Test // 테스트 메소드
    public void junitTest() {
        int a = 1;
        int b = 2;
        int sum = 3;
        Assertions.assertEquals(sum, a + b); // assertEquals(기대값, 검증할값)
    }

    @DisplayName("1 + 3는 3이다.") // 테스트 이름
    @Test // 테스트 메소드
    public void junitTest() {
        int a = 1;
        int b = 3;
        int sum = 3;
        Assertions.assertEquals(sum, a + b); // assertEquals(기대값, 검증할값)
    }
}
```

JUnit

- Junit에서 사용할 수 있는 애너테이션 몇 가지를 알아보자.

  - @BeforeAll
    - 전체 테스트를 시작하기 전에 처음으로 한 번만 실행 (정적 메소드로 선언)
    - DB 연결 또는 테스트 환경 초기화할 때 사용

  - @BeforeEach
    - 각 테스트 케이스를 시작하기 전에 매번 실행
    - 테스트 메소드에서 사용하는 객체 초기화 또는 테스트에 필요한 값을 미리 넣을 때 사용

  - @AfterAll
    - 전체 테스트를 마치고 종료하기 전에 한 번만 실행 (정적 메소드로 선언)
    - DB 연결 종료 또는 공통 사용 자원 해제 시 사용

  - @AfterEach
    - 각 테스트 케이스를 종료하기 전에 매번 실행
    - 테스트 후 특정 데이터를 삭제하는 경우 사용

JUnit

```java
public class JUnitCycleTest {

    @BeforeAll
    public static void beforeAll() { System.out.println("@BeforeAll \t 전체 테스트 시작 전에 1회 실행"); }

    @BeforeEach
    public void beforeEach() { System.out.println("@BeforeEach \t 테스트 케이스를 시작하기 전마다 실행"); }

    @Test
    public void test1() { System.out.println("테스트 1"); }
    @Test
    public void test2() { System.out.println("테스트 2"); }
    @Test
    public void test3() { System.out.println("테스트 3"); }

    @AfterAll
    public static void afterAll() { System.out.println("@AfterAll \t 전체 테스트 종료 전에 1회 실행"); }

    @AfterEach
    public void afterEach() { System.out.println("@AfterEach \t 테스트 케이스를 종료하기 전마다 실행"); }
}
```

AssertJ

- AssertJ는 Junit과 함께 사용해 검증문의 가독성을 높여주는 라이브러리이다.

- 앞서 작성했던 아래 코드는 기대값과 검증할 값을 명시하지 않으므로 비교 대상이 헷갈릴 수 있다.

```
Assertions.assertEquals(sum, a + b);
```

- 이를 AssertJ로 변경하면 아래와 같다.

```
assertThat(a + b).isEqualTo(sum);
```

AssertJ

- AssertJ에는 값을 비교하기 위한 다양한 메소드들을 제공한다.

- 자주 사용되는 메소드는 아래 표와 같다.

메소드	설명
isEqualTo(A)	A 값과 같은지 검증
isNotEqualTo(A)	A 값과 다른지 검증
contains(A)	A 값을 포함하는지 검증
doesNotContain(A)	A 값을 포함하지 않는지 검증
startsWith(A)	A로 시작하는지 검증
endsWith(A)	A로 끝나는지 검증
isEmpty()	비어있는 값인지 검증
isNotEmpty()	비어있지 않은 값인지 검증
isPositive()	양수인지 검증
isNegative()	음수인지 검증
isGreaterThan(A)	A 보다 큰 값인지 검증
isLessThan(A)	A 보다 작은 값인지 검증

테스트 코드 실습

- 문제 1.
  - String으로 선언한 변수 3개(name1, name2, name3)가 있다.
  - 변수 모두 NULL이 아니다.
  - name1과 name2는 같은 값이며, name3은 다른 값을 갖는다.

```
public class JUnitPractice {
    @DisplayName("문제1")
    @Test
    public void practice1() {
        String name1 = "이제훈";
        String name2 = "이제훈";
        String name3 = "이재훈";
        // 1. 모든 변수가 NULL이 아닌지 확인
        // 2. name1과 name2가 같은지 확인
        // 3. name1과 name3이 다른지 확인
    }
}
```

테스트 코드 실습

- 문제 2.
  - int로 선언한 변수 3개(num1 = 15, num = 0, num3 = -5)가 있다.

```
@DisplayName("문제2")
@Test
public void practice2() {
    int num1 = 15;
    int num2 = 0;
    int num3 = -5;
    // 1. num1이 양수인지 확인
    // 2. num2가 0인지 확인
    // 3. num3가 음수인지 확인
    // 4. num1은 num2보다 큰 값인지 확인
    // 5. num3은 num2보다 작은 값인지 확인
}
```

테스트 코드 실습

- 문제 2.
  - int로 선언한 변수 3개(num1 = 15, num = 0, num3 = -5)가 있다.

```
@DisplayName("문제2")
@Test
public void practice2() {
    int num1 = 15;
    int num2 = 0;
    int num3 = -5;
    // 1. num1이 양수인지 확인
    // 2. num2가 0인지 확인
    // 3. num3가 음수인지 확인
    // 4. num1은 num2보다 큰 값인지 확인
    // 5. num3은 num2보다 작은 값인지 확인
}
```

테스트 코드 실습

- 문제 3.
  - 새로운 클래스 [JUnitCyclePractice] 를 생성
  - 각각의 테스트를 시작하기 전에 "Hellllo!" 를 출력.
  - 모든 테스트를 다 마친 후에는 "Bye!" 를 출력

```
public class JUnitCyclePratice {
    @Test
    public void test1() {
        System.out.println("첫번째 테스트");
    }

    @Test
    public void test2() {
        System.out.println("두번째 테스트");
    }
}
```

테스트 코드 실습

- 이제 본격적인 테스트 코드를 작성해보자.

```
@SpringBootTest // 스프링 부트 애플리케이션 컨텍스트를 로드해 통합 테스트를 수행
@AutoConfigureMockMvc // MockMvc를 자동 구성하여 컨트롤러를 테스트할 때 사용
class BoardApplicationTests {
    @Autowired
    protected MockMvc mockMvc; // MockMvc 인스턴스를 주입 받아 HTTP 요청을 모방해 테스트

    @Autowired
    private WebApplicationContext context; // 웹 애플리케이션의 설정과 빈을 관리하는 컨텍스트

    @BeforeEach
    public void mockMvcSetUp() {
        // 각 테스트 실행 전에 MockMvc 인스턴스를 웹 애플리케이션 컨텍스트로 초기화
        this.mockMvc = MockMvcBuilders.webAppContextSetup(context).build();
    }
}
```

테스트 코드 실습

- 이제 본격적인 테스트 코드를 작성해보자.

```
@DisplayName("Board List 조회 테스트")
@Test
public void testBoardList() throws Exception {
    // 테스트할 URL 경로
    final String url = "/board/list";
    // MockMvc를 사용해 해당 URL로 GET 요청을 수행
    final ResultActions result = mockMvc.perform(get(url).accept(MediaType.TEXT_HTML));

    result
        .andExpect(status().isOk()) // 응답 상태가 HTTP 200 OK인지 확인
        .andExpect(view().name("board/list")) // 반환된 뷰의 이름이 "board/list"인지 확인
        .andExpect(model().attributeExists("list")) // 모델에 "list"라는 속성이 존재하는지 확인
        .andExpect(content().contentType("text/html;charset=UTF-8")); // 응답 콘텐츠 타입 확인
}
```

JPA

- 백엔드 개발에서 데이터베이스 관련 로직을 처리하는데 오랜 시간을 할애하게 된다.
- 해당 로직에는 반복되는 코드가 많아지면서 데이터베이스와 관련된 개발의 생산성이 떨어지자 이 문제를 해결하기 위해 프레임워크가 등장한다.

- JDBC를 직접 사용하는 방식에서부터 ibatis, MyBatis까지 SQL Mapper 프레임워크를 사용하는 방법까지 다양한 대안이 제시되었다.
- 최근에는 JPA를 이용하는데, JPA(Java Persistence API)는 자바 객체와 데이터베이스 테이블 간의 매핑을 처리하는 ORM 기술의 표준이다.
- ORM(Object Relational Mapping) 기술은 객체와 관계를 설정하는 것을 의미한다.

- JPA는 각 기능의 동작이 어떻게 되어야 한다는 것을 정의한 기술 명세로, 기술 명세에 따라 실제 기능을 구현한 구현체가 필요하다.
- JPA의 구현체를 JPA 프로바이더라고 하며, 하이버네이트, 이클립스링크 등이 있다.
- 가장 많이 사용되는 JAP 프로바이더는 하이버네이트(Hibernate)이다.

JPA

- JPA의 장점
  - 개발이 편리하다.
    - (웹 애플리케이션에서 반복적으로 작성하는 기본적인 CRUD를 SQL로 직접 작성하지 않아도 된다.
  - 데이터베이스에 독립적인 개발이 가능하다.
    - (JPA는 특정 데이터베이스에 종속되지 않기 때문에 데이터베이스가 변경되어도 문제 없다.)
  - 유지보수가 쉽다.
    - (MyBatis와 같은 프레임워크는 테이블이 변경될 경우 관련 코드를 모두 수정해야 하는데, JPA를 이용하면 객체만 수정하면 된다.)

- JPA의 단점
  - 학습곡선(Learning Curve)이 크다.
  - 특정 데이터베이스의 기능을 사용할 수 없다.
  - 객체지향 설계가 필요하다.

스프링 데이터 JPA

- 스프링 데이터 JPA는 스프링 프레임워크 하위 프로젝트 중 하나로, JPA를 스프링에서 쉽게 사용할 수 있도록 해주는 라이브러리이다.

- 하이버네이트와 같은 JPA 프로바이더를 직접 사용할 경우에는 엔티티 매니저(Entity Manager)를 설정하고 이용하는 등 여러 진입장벽이 있다.

- 스프링 데이터 JPA는 리포지터리(Repository)라는 인터페이스를 제공하여, 해당 인터페이스의 규격에 맞게 메소드만 작성하면 된다.
- 그렇게 하면, 내부적으로 하이버네이트를 사용하여 동작하게 된다.
  즉, 하이버네이트를 모르더라도 프레임워크가 하이버네이트를 이용해 적절한 코드를 생성하기 때문에 JPA를 보다 쉽게 사용할 수 있게 된다.

스프링 데이터 JPA

```
dependencies {
    runtimeOnly 'com.mysql:mysql-connector-j'
    implementation 'org.springframework.boot:spring-boot-starter-data-jpa'
    implementation 'org.springframework.boot:spring-boot-starter-thymeleaf'
    implementation 'org.springframework.boot:spring-boot-starter-web'
    compileOnly 'org.projectlombok:lombok'
    developmentOnly 'org.springframework.boot:spring-boot-devtools'
    annotationProcessor 'org.projectlombok:lombok'
    testImplementation 'org.springframework.boot:spring-boot-starter-test'
    testRuntimeOnly 'org.junit.platform:junit-platform-launcher'
}
```

```
CREATE DATABASE `jpa_test_db`;
```

스프링 데이터 JPA

- src/main/resources 에 data.sql 파일 생성

```sql
INSERT INTO member_tbl (id, name) VALUES (1, '둘리');
INSERT INTO member_tbl (id, name) VALUES (2, '도우너');
INSERT INTO member_tbl (id, name) VALUES (3, '희동이');
```

스프링 데이터 JPA

```yaml
spring:
    application:
        name: jpa_project
    datasource:
        driver-class-name: com.mysql.cj.jdbc.Driver
        url: jdbc:mysql://localhost:3306/jpa_test_db
        username: root
        password: 1313
    jpa:
        database: mysql
        # 자동으로 테이블 생성과 같은 스크립트 실행 (실제는 false로 변경)
        generate-ddl: true
        show-sql: true
        open-in-view: false
     sql:
         init:
             mode: never # data.sql 파일 절대 실행 안함
```

스프링 데이터 JPA

```java
package com.jpa.entity;

@Entity(name = "member_tbl") // JPA 엔티티임을 명시 (테이블 연결)
@NoArgsConstructor
@Data
public class Member {
    // 기본키
    @Id
    // 데이터베이스의 기본키 자동 증가 전략 사용
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    @Column(nullable = false)
    private String name;
}
```

```
Hibernate: create table member_tbl (id integer not null auto_increment, name
varchar(255) not null, primary key (id)) engine=InnoDB
```

스프링 데이터 JPA

```
package com.jpa.repository;

public interface MemberRepository extends JpaRepository<Member, Integer>{
    Optional<Member> findByName(String name);
}
```

스프링 데이터 JPA

```java
@DataJpaTest
@AutoConfigureTestDatabase(replace = AutoConfigureTestDatabase.Replace.NONE) // 실제 데이터베이스로 테스트
public class MemberRepositoryTest {

    @Autowired
    MemberRepository memberRepository;

    @DisplayName("전체 회원 조회")
    @Sql("/data.sql") // 테스트 실행 전 SQL문 실행
    @Test
    void getAllMembers() {
        List<Member> all = memberRepository.findAll();
        assertThat(all.size()).isEqualTo(3);
    }

}
```