**회원 도메인과 리포지토리 만들기**

**회원객체**

package hello.hellospring.domain;

public class Member {

... // id, name 생성

}

**회원 리포지토리 인터페이스**

package hello.hellospring.repository;

import hello.hellospring.domain.Member;

import java.util.List;

import java.util.Optional;

public interface MemberRepository {

Member save(Member member);

Optional<Member> findById(Long id);

Optional<Member> findById(String name);

List<Member> findAll();

}

**회원 리포지토리 메모리 구현체**  
메모리 DB를 사용해서 데이터를 담기때문에 서버 종료시 소멸

package hello.hellospring.repository;

import hello.hellospring.domain.Member;

import java.util.\*;

/\*\*

\* 동시성 문제가 고려되어 있지 않음, 실무에서는 ConcurrentHashMap, AtomicLong 사용 고려

\*/

public class MemoryMemberRepository implements MemberRepository {

private static Map<Long, Member> store = new HashMap<>();

private static long sequence = 0L;

@Override

public Member save(Member member) {

member.setId(++sequence);

store.put(member.getId(), member);

return member;

}

@Override

public Optional<Member> findById(Long id) {

return Optional.ofNullable(store.get(id));

}

@Override

public List<Member> findAll() {

return new ArrayList<>(store.values());

}

@Override

public Optional<Member> findByName(String name) {

return store.values().stream()

.filter(member -> member.getName().equals(name))

.findAny();

}

public void clearStore() {

store.clear();

} }

* 구현체란

: 인터페이스를 구현하는 클래스

* 동시성 문제란

: 두 개 이상의 세션이 공통된 자원에 대해 모두 읽고 쓰는 작업(Read->Write)을 하려고 할 때 발생할 수 있는 문제를 말한다.

* 회원 리포지토리 테스트 케이스 작성  
  : 개발한 기능을 실행해서 테스트 할 때 자바의 main 메소드를 통해서 실행하거나, 웹 애플리케이션의 컨트롤러를 통해서 해당 기능을 실행한다. 이러한 방법은 준비하고 실행하는데 오래 걸리고, 반복 실행하기 어렵고 여러 테스트를 한번에 실행하기 어렵다는 단점이 있다. \_자바는 JUnit이라는 프레임워크로 테스트를 실행해서 이러한 문제를 해결한다.
* JUnit이란

: Java의 **단위 테스팅(Unit Testing)도구**이다. 단 하나의 jar 파일로 되어 있다.  
Testing 결과를 단순히 문서로 남기는 것이 아니라 Test Class를 그대로 남김으로써 추 후 개발자에게 테스트 방법 및 클래스의 히스토리를 넘겨줄 수 있다.  
즉, Java Testing Framework

**회원 리포지토리 메모리 구현체 테스트**

package hello.hellospring.repository;

import hello.hellospring.domain.Member;

import org.junit.jupiter.api.AfterEach;

import org.junit.jupiter.api.Test;

import java.util.List;

import java.util.Optional;

import static org.assertj.core.api.Assertions.\*;

class MemoryMemberRepositoryTest {

MemoryMemberRepository repository = new MemoryMemberRepository();

@AfterEach

public void afterEach() {

repository.clearStore();

}

@Test

public void save() {

//given

Member member = new Member();

member.setName("spring");

//when

repository.save(member);

//then

Member result = repository.findById(member.getId()).get();

assertThat(result).isEqualTo(member);

}

@Test

public void findByName() {

//given

Member member1 = new Member();

member1.setName("spring1");

repository.save(member1);

Member member2 = new Member();

member2.setName("spring2");

repository.save(member2);

//when

Member result = repository.findByName("spring1").get();

//then

assertThat(result).isEqualTo(member1);

}

@Test

public void findAll() {

//given

Member member1 = new Member();

member1.setName("spring1");

repository.save(member1);

Member member2 = new Member();

member2.setName("spring2");

repository.save(member2);

//when

List<Member> result = repository.findAll();

//then

assertThat(result.size()).isEqualTo(2);

}

}

* Given : 선언되고,
* When : 실행할 때,
* Then : 검증한다.
* @AfterEach: 한번에 여러 테스트를 실행하면 메모리 DB에 직전 테스트의 결과가 남을 수 있다. 이렇게되면 다음 이전 테스트 때문에 다음 테스트가 실패할 가능성이 있다. @AfterEach를 사용하면 각 테스트가 종료될 때 마다 이 기능을 실행한다. 여기서는 메모리 DB에 저장된 데이터를 삭제한다.
* 테스트는 각각 **독립적으로 실행**되어야 한다. 테스트 순서에 의존관계가 있는 것은 좋은 테스트가 아니다.

**회원 서비스 개발**

package hello.hellospring.service;

import hello.hellospring.domain.Member;

import hello.hellospring.repository.MemberRepository;

import java.util.List;

import java.util.Optional;

public class MemberService {

private final MemberRepository memberRepository = new MemoryMemberRepository();

/\*\*

\* 회원가입

\*/

public Long join(Member member) {

validateDuplicateMember(member); //중복 회원 검증

memberRepository.save(member);

return member.getId();

}

private void validateDuplicateMember(Member member) {

memberRepository.findByName(member.getName())

}

.ifPresent(m -> {

throw new IllegalStateException("이미 존재하는 회원입니다.");

});

/\*\*

\*전체 회원 조회

\*/

public List<Member> findMembers() {

return memberRepository.findAll();

}

public Optional<Member> findOne(Long memberId) {

return memberRepository.findById(memberId);

} }

**회원 서비스 테스트**

* 기존에는 회원 서비스가 메모리 회원 리포지토리를 직접 생성하게 했다.

public class MemberService {

private final MemberRepository memberRepository = new MemoryMemberRepository();

}

회원 리포지토리의 코드가 **회원 서비스 코드를 DI가능하게 변경한다.**

public class MemberService {

private final MemberRepository memberRepository;

public MemberService(MemberRepository memberRepository) {

this.memberRepository = memberRepository;

}

... }

* DI(의존성 주입)이란

: Spring 프레임워크는 3가지 핵심 프로그래밍 모델을 지원하고 있는데, 그 중 하나가 의존성 주입(DI)이다. DI란 외부에서 두 객체 간의 관계를 결정해주는 디자인 패턴으로, 인터페이스를 사이에 둬서 클래스 레벨에서는 의존관계가 고정되지 않도록하고 런타임 시에 관계를 다이나믹하게 주입하여 유연성을 확보하고 결합도를 낮출 수 있게해준다. 의존성이란 한 객체가 다른객체를 사용할 때 의존성 있다고 한다.

**회원 서비스 테스트**

package hello.hellospring.service;

import hello.hellospring.domain.Member;

import hello.hellospring.repository.MemoryMemberRepository;

import org.junit.jupiter.api.BeforeEach;

import org.junit.jupiter.api.Test;

import static org.assertj.core.api.Assertions.\*;

import static org.junit.jupiter.api.Assertions.\*;

class MemberServiceTest {

MemberService memberService;

MemoryMemberRepository memberRepository;

@BeforeEach

public void beforeEach() {

memberRepository = new MemoryMemberRepository();

memberService = new MemberService(memberRepository);

}

@AfterEach

public void afterEach() {

memberRepository.clearStore();

}

@Test

public void 회원가입() throws Exception {

//Given

Member member = new Member();

member.setName("hello");

//When

Long saveId = memberService.join(member);

//Then

Member findMember = memberRepository.findById(saveId).get();

assertEquals(member.getName(), findMember.getName());

}

@Test

public void 중복\_회원\_예외() throws Exception {

//Given

Member member1 = new Member();

member1.setName("spring");

Member member2 = new Member();

member2.setName("spring");

//When

memberService.join(member1);

IllegalStateException e = assertThrows(IllegalStateException.class,

() -> memberService.join(member2));//예외가 발생해야 한다. assertThat(e.getMessage()).isEqualTo("이미 존재하는 회원입니다.");

} }

* @BeforeEach: 각 테스트 실행 전에 호출된다. 테스트가 서로 영향이 없도록 항상 새로운 객체를 생성하고, 의존관계도 새로 맺어준다.