**회원 관리 예제 - 백엔드 개발**

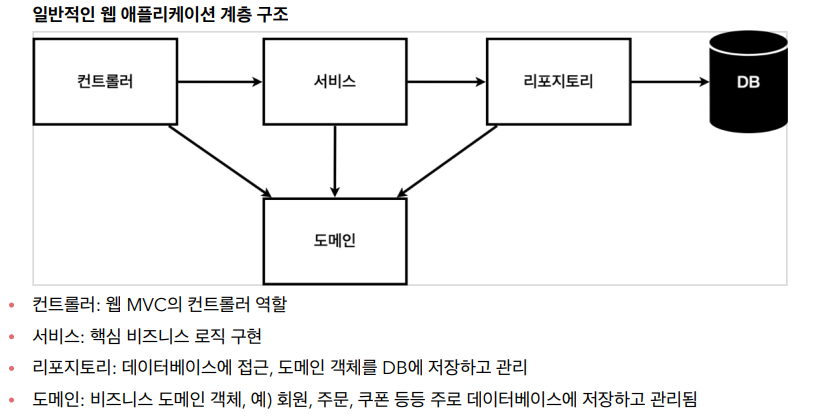
**📒 비즈니스 요구사항 정리**

**회원 관리 예제 프로젝트**를 만들기 위한 요구사항을 먼저 정리해야 된다.

회원 ID, 이름의 **데이터**, 그리고 회원을 등록하고 조회하는 **기능**이 요구된다.  
그리고 아직 데이터를 저장하는 DB가 선정되지 않았다는 가상의 시나리오를 가지고 프로젝트를 진행할 것이다.

**🖋️ 웹 어플리케이션 계층 구조**

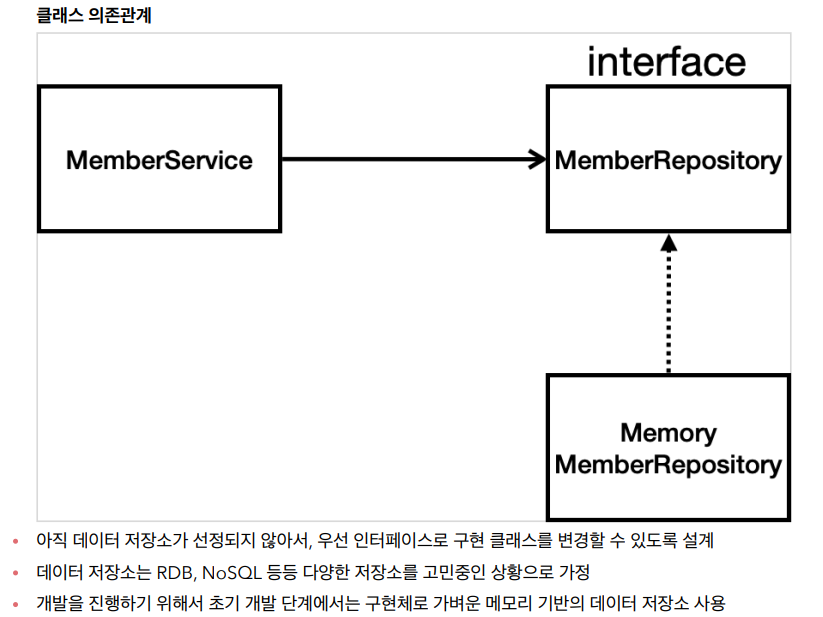
일반적인 웹 애플리케이션 계층 구조는 **컨트롤러, 서비스, 리포지토리, 도메인, DB**로 구성되어 있다.



* **컨트롤러**는 **웹 MVC**의 컨트롤러 역할과 **API**를 만드는 컨트롤러 역할
* **서비스**: **핵심 비지니스 로직**이 들어가 있어, 예를 들어 회원은 중복 가입이 안된다는 등의 로직들이 서비스 객체에 들어가 있음
* **도메인**: 회원, 주문, 쿠폰 등 주로 **데이터베이스에 저장하고 관리**되는 비즈니스 도메인 객체
* **리포지토리**: 비즈니스 도메인 객체를 가지고 **핵심 비즈니스가 동작**하도록 구현한 객체

**🖋️ 클래스 의존관계**

회원 관리 예제 프로젝트의 **클래스 의존관계**는 아래 사진과 같은 형태로 만들 것이다.

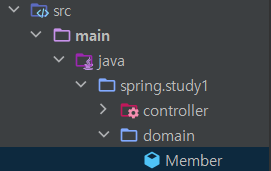


회원 비즈니스 로직에는 **회원 서비스(MemberService)**가 있고, **회원 저장소(MemberRepository)**는 **interface**로 설계할 것이다.  
왜냐하면 아직 DB가 선정되지 않았기 때문에 우선 단순하게 **메모리**로 데이터로 저장을 하며, 나중에 DB가 선정되고 메모리에 저장된 데이터를 DB로 바꿔야 하기 때문에 **interface**가 필요하다.

**📒 회원 도메인과 리포지토리 만들기**

**🖋️ 회원 도메인**

**회원 도메인**을 만들기 위해 src/main/java/spring.study1 폴더에 **domain** 이라는 새로운 **패키지**를 추가하고, 그 안에 **Member** 라는 **클래스**를 만들어 준다.



데이터로 **회원 ID**와 **이름**이 있어야 된다는 요구사항 때문에, **회원 ID**와 **이름**에 대한 getter와 setter를 만들어 준다.

package spring.study1.domain;

public class Member {

private Long id;

private String name;

// id의 getter, setter

public Long getId() {

return id;

}

public void setId(Long id) {

this.id = id;

}

// name의 getter, setter

public String getName() {

return name;

}

public void setName(String name) {

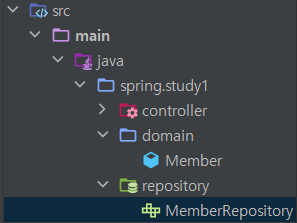
this.name = name;

}

}

**🖋️ 회원 리포지토리 interface**

위에 만든 회원 객체를 저장할 **리포지토리**를 만들어 주기 위해 src/main/java/spring.study1 폴더에 **repository** 라는 새로운 패키지를 만들고, DB 연결을 위한 **interface** 를 만들어 줘야 하기 때문에 **MemberRepository** 라는 **interface** 파일을 **repository** 패키지 안에 생성한다.



회원 저장 기능 4가지(save, findById, findByName, findAll)를 아래 코드와 같이 추가한다.

package spring.study1.repository;

import spring.study1.domain.Member;

import java.util.List;

import java.util.Optional;

public interface MemberRepository {

Member save(Member member); // save: 회원이 저장소에 저장

Optional<Member> findById(Long id); // findById: 저장소에서 id를 찾아 member 객체 반환

Optional<Member> findByName(String name); // findByName: 저장소에서 name을 찿아 member 객체 반환

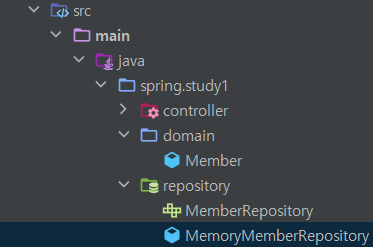
List<Member> findAll(); // findAll: 저장된 모든 회원의 member List 반환

}

여기서 **Optional**은 간단하게 설명을 하자면, 값을 반환할 때 **null** 일 수 있어 이를 처리해주는 java 방식이다.

**🖋️ 회원 리포지토리 메서드**

interface를 만들었으면, 그 다음으로 구현체를 만들기 위해 **repository** 패키지 안에 **MemoryMemberRepository** 라는 새로운 **파일**을 만든다.



위에서 만든 **interface**를 구현하기 위해, 이에 대한 **implemnets 메서드**를 생성해준다.

package spring.study1.repository;

import spring.study1.domain.Member;

import java.util.\*;

public class MemoryMemberRepository implements MemberRepository{

// Map: java의 데이터 저장 방식 중 하나

// key: Long, value: Member

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

// sequence: key값을 생성해줌

private static long sequence = 0L;

@Override

public Member save(Member member) { // save

member.setId(++sequence); // sequence값 증가(key값)

store.put(member.getId(), member); // store에 저장

return member;

}

@Override

public Optional<Member> findById(Long id) { // findById

// null일 수 있어 Optional.ofNullable() 로 감쌈

return Optional.ofNullable(store.get(id)); // id값으로 해당 객체 store에서 꺼내서 return

}

@Override

public Optional<Member> findByName(String name) { // findByName

return store.values().stream() // value값들을 루프를 돌아서

.filter(member -> member.getName().equals(name)) // param로 넘어온 name과 같은 것을 filtering

.findAny(); // 처음 filtering 된 요소 1개를 return

}

@Override

public List<Member> findAll() { // findAll

return new ArrayList<>(store.values()); // 저장소에 있는 value들을 List로 반환

}

}

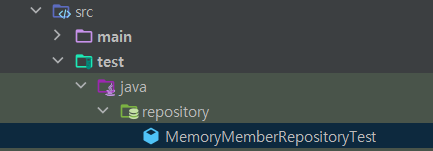
각각 **save**, **findById**, **findByName**, **findAll**에 대한 메서드를 작성하였다.

**📒 회원 리포지토리 테스트 케이스 작성**

**🖋️ 테스트 케이스 작성 준비**

위에서 만든 **MemoryMemberRepository**가 정삭적으로 동작하는 지 검증을 하는 **테스트 케이스**를 작성하기 위해, 자바의 **JUnit** 이라는 프레임워크로 테스트를 실행할 것이다.

테스트 파일은 src/test/java에 **repository** 라는 폴더를 만들고 그 안에 **MemoryMemberRepositoryTest** 라는 클래스 파일을 만든다.



테스트를 검증하기 위한 새로운 저장소 **repository** 를 만들고, **Test 코드**를 작성할 것이다.

package repository;

class MemoryMemberRepositoryTest { // 굳이 public 으로 할 필요없어서 지움

// Test를 하기 위한 저장소

MemoryMemberRepository repository = new MemoryMemberRepository();

}

**🖋️ 테스트 케이스 작성 - Test**

**save** 메서드가 정상적으로 작동이 되는지 확인하기 위해, **save Test 코드**를 클래스 안에 아래와 같이 작성한다.

@Test // Test

public void save() {

Member member = new Member(); // member 객체 만듦

member.setName("spring");

repository.save(member); // 저장소에 객체 member 저장

// member의 id 값으로, 저장소에 저장된 객체 result 가져옴

// get(): 값을 꺼내옴

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

// 저장한 member와 저장소에서 가져온 result가 같은 지 검증

System.out.println("result = " + (result == member));

Assertions.assertEquals(member, result);

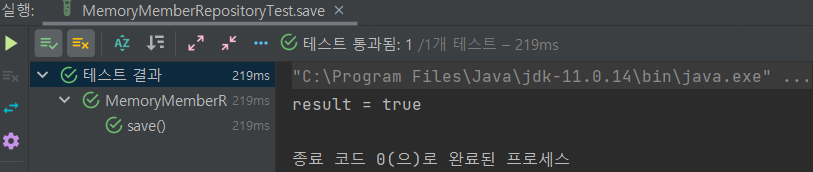
assertThat(result).isEqualTo(member);

}

**save Test 코드**를 실행하는 방법은 아래 사진처럼 실행 버튼을 눌러 실행을 한다.



**Test**를 실행하면 다음과 같이 테스트가 정상적으로 통과됐다는 것을 쉽게 확인 할 수 있다.



다음으로 **findByName** 메서드가 정상적으로 작동이 되는지 확인하기 위해, **findByName Test 코드**를 클래스 안에 아래와 같이 작성한다.

@Test // Test

public void findByName() {

Member member1 = new Member(); // member1 객체 만들어 저장소에 저장

member1.setName("spring1");

repository.save(member1);

Member member2 = new Member(); // member2 객체 만들어 저장소에 저장

member2.setName("spring2");

repository.save(member2);

// member1의 이름으로, 저장소에 저장된 객체 가져옴

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

// member1과 저장소에서 가져온 result가 같은 지 검증

assertThat(result).isEqualTo(member1);

}

마지막으로 **findAll** 메서드가 정상적으로 작동이 되는지 확인하기 위해, **findAll Test 코드**를 클래스 안에 아래와 같이 작성한다.

@Test // Test

public void findAll() { // findAll

Member member1 = new Member(); // member1 객체 만들어 저장소에 저장

member1.setName("spring1");

repository.save(member1);

Member member2 = new Member(); // member2 객체 만들어 저장소에 저장

member2.setName("spring2");

repository.save(member2);

// 저장소에 저장된 객체를 List로 가져옴

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

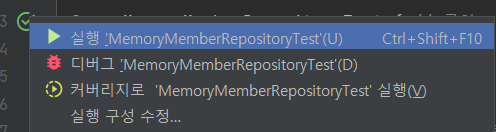
// 저장소에서 가져온 List의 size가 2와 같은 지 검증

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

}

**🖋️ 테스트 케이스 작성 - clearStore**

**Test 코드**를 각각 개별적으로 하나하나씩 실행할 수도 있지만, 전체 클래스도 실행할 수 있어 보다 빠르고 간편하게 **Test**를 할 수 있다.



그러나 위에 사진과 같이 바로 실행을 하면 **Test**가 실패를 한다.

왜냐하면 findAll과 findByName에서 모두 **똑같은** member1과 member2 **객체**가 둘 다 생성 되었기 때문이다.

그래서 이를 해결하기 위해 메서드의 실행이 끝날 때마다 실행되는 **@AfterEach** 메서드에 만들어, 저장소의 내용들을 모두 삭제하는 코드를 작성할 것이다.

@AfterEach // 메서드 실행이 끝날 때마다 실행됨

public void afterEach() {

repository.clearStore(); // 저장소 내용 다 지움

}

위에 코드에서 사용한 저장소 내용을 다 지워주는 **clearStore** 메서드를 만들어야 하기 때문에, src/main/java/spring.study1/repository/**MemoryMemberRepository** 에서 저장소에 있는 내용을 다 지워주는 **clearStore** 메서드를 새롭게 추가해 준다.

public void clearStore() { // clearStore

store.clear(); // 저장소 내용 지움

}

그러면 **Test 클래스**를 전체 실행 했을 때 아무런 오류없이 정상적으로 실행이 된다.

**🖋️ 테스트 케이스 전체 코드**

package repository;

import org.junit.jupiter.api.AfterEach;

import org.junit.jupiter.api.Assertions;

import org.junit.jupiter.api.Test;

import spring.study1.domain.Member;

import spring.study1.repository.MemoryMemberRepository;

import java.util.List;

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

class MemoryMemberRepositoryTest { // 굳이 public 으로 할 필요없어서 지움

MemoryMemberRepository repository = new MemoryMemberRepository();

@AfterEach // 메서드 실행이 끝날 때마다 실행됨

public void afterEach() {

repository.clearStore(); // 저장소 내용 다 지움

}

@Test // Test

public void save() {

Member member = new Member(); // member 객체 만듦

member.setName("spring");

repository.save(member); // 저장소에 객체 member 저장

// member의 id 값으로, 저장소에 저장된 객체 가져옴

// get(): 값을 꺼내옴

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

// member와 저장소에서 가져온 result가 같은 지 검증

System.out.println("result = " + (result == member));

Assertions.assertEquals(member, result);

assertThat(result).isEqualTo(member);

}

@Test // Test

public void findByName() {

Member member1 = new Member(); // member1 객체 만들어 저장소에 저장

member1.setName("spring1");

repository.save(member1);

Member member2 = new Member(); // member2 객체 만들어 저장소에 저장

member2.setName("spring2");

repository.save(member2);

// member의 이름으로, 저장소에 저장된 객체 가져옴

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

// member1과 저장소에서 가져온 result가 같은 지 검증

assertThat(result).isEqualTo(member1);

}

@Test // Test

public void findAll() { // findAll

Member member1 = new Member(); // member1 객체 만들어 저장소에 저장

member1.setName("spring1");

repository.save(member1);

Member member2 = new Member(); // member2 객체 만들어 저장소에 저장

member2.setName("spring2");

repository.save(member2);

// 저장소에 저장된 객체를 List로 가져옴

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

// 저장소에서 가져온 List의 size가 2와 같은 지 검증

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

}

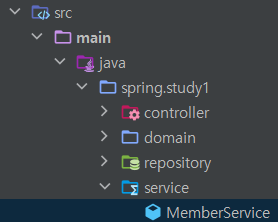
}

**📒 회원 서비스 개발**

**🖋️ 회원 서비스 개발**

**회원 서비스**는 회원 repository와 domain을 활용하여 실제 비즈니스 로직을 작성해야 한다.

이를 작성하기 위해 src/main/java/spring.study1에 **service**라는 패키지를 만들고 그 안에 **MemberService**라는 클래스를 생성한다.



먼저 회원 가입 서비스를 위한 **join** 메서드를 작성한다.

저장소에 저장을 할 때, 중복된 이름이 있으면 안된다는 조건이 있어 **중복회원을 검증**하는 **validateDuplicateMember** 메서드를 새롭게 추가해 준다.  
만약 저장소 안에 이미 저장할 이름이 있다면 "이미 존재하는 회원입니다." 라는 메세지 예외를 주고, 중복 회원이 없다면 **member**를 저장소에 **save** 해준다.

package spring.study1.service;

import spring.study1.domain.Member;

import spring.study1.repository.MemberRepository;

import spring.study1.repository.MemoryMemberRepository;

import java.util.Optional;

public class MemberService {

// 저장소

private final MemberRepository memberRepository = new MemoryMemberRepository();

// 회원가입

public Long join(Member member) {

// 중복회원 검증 메서드 호출

validateDuplicateMember(member);

// member를 저장소에 저장

memberRepository.save(member);

// member id 반환

return member.getId();

}

// 중복회원 검증 메서드

private void validateDuplicateMember(Member member) {

memberRepository.findByName(member.getName())

.ifPresent(m -> { // 이름이 같은 회원이 존재하면

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

});

}

}

다음으로 **전체 회원 조회** 서비스를 위한 메서드를 **MemberService** 클래스 안에 작성한다.

// 전체 회원 조회

public List<Member> findMembers() {

return memberRepository.findAll();

}

마지막으로 **id로 회원 조회** 서비스를 위한 메서드를 작성한다.

// id로 회원 조회

public Optional<Member> findOne(Long memberId) {

return memberRepository.findById(memberId);

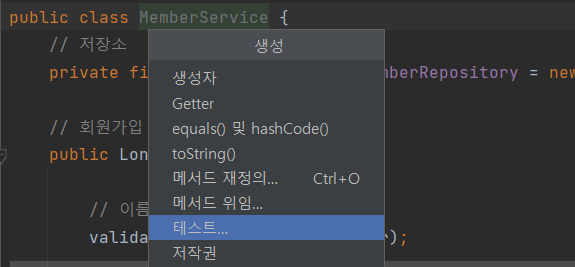
}

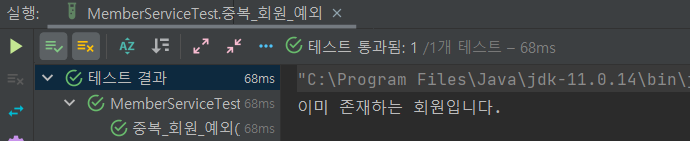
**📒 회원 서비스 테스트**

**🖋️ 회원 서비스 테스트 환경**

위에서 작성한 회원가입, 전체 회원 조회 등 **서비스**를 검증하기 위해 **회원 서비스 테스트** 코드를 작성할 것이다.

다음과 같이 **테스트**를 생성해주는 기능을 사용하면 보다 빠르게 **Test 코드**를 작성할 수 있는 환경을 만들어 준다.





package spring.study1.service;

import org.junit.jupiter.api.Test;

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

class MemberServiceTest {

@Test

void join() {

}

@Test

void findMembers() {

}

@Test

void findOne() {

}

}

**🖋️ 회원 서비스 테스트 코드 - Test**

**회원가입 서비스**의 **Test 코드**는 아래 코드처럼 작성을 하며, **Test 코드**는 실제로 빌드할 때 포함되지 않아 **한글**로 작성해도 무방하다.

MemberService memberService = new MemberService(); // member service

@Test

void 회원가입() {

// given

Member member = new Member(); // member 객체 생성

member.setName("hello"); // member name에 hello을 넣음

// when

// member 객체를 회원가입하고, 반환된 id를 saveId

Long saveId = memberService.join(member);

// then

// 회원가입한 member의 id가 저장소에 있으면, 해당 member 객체를 findMember로

Member findMember = memberService.findOne(saveId).get();

// 회원가입한 member와, 저장소에서 가져온 member의 이름이 같은 지 검증

Assertions.assertThat(member.getName()).isEqualTo(findMember.getName());

}

작성한 **Test 코드**를 실행하면 정상적으로 동작하는 것을 확인할 수 있다.

뿐만 아니라 회원가입을 할 때 **중복회원**일 경우도 있어 예외적인 상황에서도 정상적으로 동작하는 지 확인하기 위해 중복*회원*예외 **Test 코드**도 추가적으로 작성해야 한다.

@Test

public void 중복\_회원\_예외() {

// given

// 이름이 같은 중복 회원 member 객체 생성

Member member1 = new Member();

member1.setName("spring");

Member member2 = new Member();

member2.setName("spring");

// when

memberService.join(member1);

// memberService.join(member2)에 IllegalStateException 예외 검증

IllegalStateException e = assertThrows(IllegalStateException.class,

() -> memberService.join(member2));

System.out.println(e.getMessage()); // 예외 메시지 출력

}

저장소에 중복된 회원이 있을 경우 정상적으로 예외 처리되는 것을 **Test 코드**로 확인 할 수 있다.

**🖋️ 회원 서비스 테스트 코드 - clearStore**

위에서 작성한 **회원 리포지토리 테스트 케이스**에서 각 메서드의 실행이 끝날 때마다 저장소에 있는 내용을 다 지운 것 처럼, 이번에도 저장소에 있는 내용을 다 지우는 메서드를 추가해야 한다.

MemberService memberService = new MemberService();

MemoryMemberRepository memberRepository = new MemoryMemberRepository();

@AfterEach // 메서드 실행이 끝날 때마다 실행됨

public void afterEach() {

memberRepository.clearStore(); // 저장소 내용 다 지움

}

🚨그러나 **회원 리포지토리 테스트 케이스** 처럼 **memberRepository**의 내용을 지우면 문제가 발생할 수 있다.

왜냐하면 **memberService 객체**가 아닌 새로운 **memberRepository 저장소 객체**를 만들었기 때문이다. 그래서 다른 내용을 지울 수 있는 위험이 있기 때문에 위에 방법이 아닌 다른 방법으로 **clearStore()** 를 해야한다.

지금은 **MemoryMemberRepository** 저장소가 **static**이라서 문제는 없지만, 혹시라도 **static**이 아닌 경우 서비스의 저장소가 아닌 다른 저장소를 처리하게 되는 문제가 발생하게 된다.

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

**🖋️ 회원 서비스 테스트 코드 - clearStore, DI**

이러한 문제를 해결하기 위해 **MemberService**에 **memberRepository** 를 외부에서 넣어주도록 변경해야 한다.

src/main/java/spring.study1/service/MemberService

private final MemberRepository memberRepository;

public MemberService(MemberRepository memberRepository) {

this.memberRepository = memberRepository;

}

src/test/java/spring.study/service/MemberServiceTest

MemberService memberService;

MemoryMemberRepository memberRepository;

@BeforeEach // 메서드 시작되기 전

public void beforeEach() {

memberRepository = new MemoryMemberRepository();

memberService = new MemberService(memberRepository);

}

@AfterEach // 메서드 시작된 후

public void afterEach() {

memberRepository.clearStore();

}

그러면 메서드가 시작되기 전에 각각 memberRepository와 memberService를 생성해주면 같은 memberRepository가 생성이 되고 메서드가 끝나면 다시 memberRepository를 clear 하게 되며, 이를 **DI(Dependency Injection)** 라고 한다.