스터디 2주차(2021.08.22)

섹션 3. 회원 관리 예제 - 백엔드 개발

섹션 4. 스프링 빈과 의존관계

섹션 5. 회원 관리 예제 - 웹 MVC 개발

섹션 3. 회원 관리 예제 - 백엔드 개발

일반적인 웹 애플리케이션 계층 구조

- **컨트롤러(Controller) : 클라이언트의 요청을 받아 Requestmapping을 수행하고, 응답을 전달**
  👉 요청 url에 따라 뷰와 매핑
  👉 API 서비스를 사용하는 경우 @ResponseBody를 통해 반환 값을 HTTP response에 바로 씀
- **서비스 : 비지니스 로직을 처리 (ex : 회원가입은 중복으로 할 수 없다.)**
- **DAO(리포지토리) : 실제 DB에 접근하는 객체로, 서비스와 DB 사이의 연결 고리 역할**
  👉 도메인 객체를 DB에 저장하고 관리함
- **도메인(Entity) : 비지니스 도메인 객체 (ex: 회원, 주문, 쿠폰 등 DB에 저장되고 관리되는 객체)**
- **DTO : 계층간 데이터 교환을 위한 객체(Java Beans)**
  👉 로직을 갖고 있지 않은 순수한 데이터 객체이며 getter/setter 메서드만 가짐
  👉 DTO는 도메인 모델을 복사한 형태라고 볼 수 있음

❓ 도메인(Entity)와 DTO를 분리하는 이유?
1. View layer와 DB layer 역할을 철저히 분리하기 위해
2. Entity가 변경되면 DB와 연관된 여러 클래스에 영향을 끼치지만,
   View와 통신하는 DTO는 자주 변경되므로 분리시켜도 영향이 적음

TDD(Test Driven Development)

👉 테스트 주도 개발이란❓ 반복 테스트를 이용한 소프트웨어 방법론으로, 작은 단위의 테스트 케이스를 작성하고 이를 통과하는 코드를 추가하는 단계를 반복하여 구현한다.

즉, 테스트를 염두해둔 프로그램 개발 방법

- 테스트 코드는 빌드 코드에 포함되지 않는다
- 테스트 코드 메소드에 given, when, then 3요소 넣는다면 직관적스럽다.

❓ **TDD의 장점**
1. 테스트 코드를 먼저 작성한다면 명확한 기능과 구조를 설계할 수 있다
✅ 한 함수에 복잡한 기능을 몰아 넣는다면 테스트는 어려워지기에 재사용성을 고려
2. 설계 수정시간을 단축할 수 있다
✅ 테스트 코드를 작성함으로써 기능 구현 시 설계안을 토대로 구조 문제 발견 가능

Optional

메서드가 반환할 결과 값이 없다는 걸 표현하고 싶을 때, 혹은 null을 반환하면 에러가 발생할 가능성이 높은 상황에서 사용하기 위해 만든 반환 타입

```
// 같은 이름이 있는 중복 회원X
Optional<Member> result = memberRepository.findByName(member.getName());
result.ifPresent(m -> {
    throw new IllegalStateException("이미 존재하는 회원입니다.");
});
```

```
memberRepository.findByName(member.getName())
        .ifPresent(m -> {
    throw new IllegalStateException("이미 존재하는 회원입니다.");
});
```

Assertion

개발자는 해당 문이 그 문의 장소에서 언제나 참이라고 간주함. 런타임 중에 거짓으로 판단되면 실패를 초래하며 이 상황에서는 일반적으로 실행이 중단된다(Assert error)

👉 디버깅을 용이하게 함

Reference

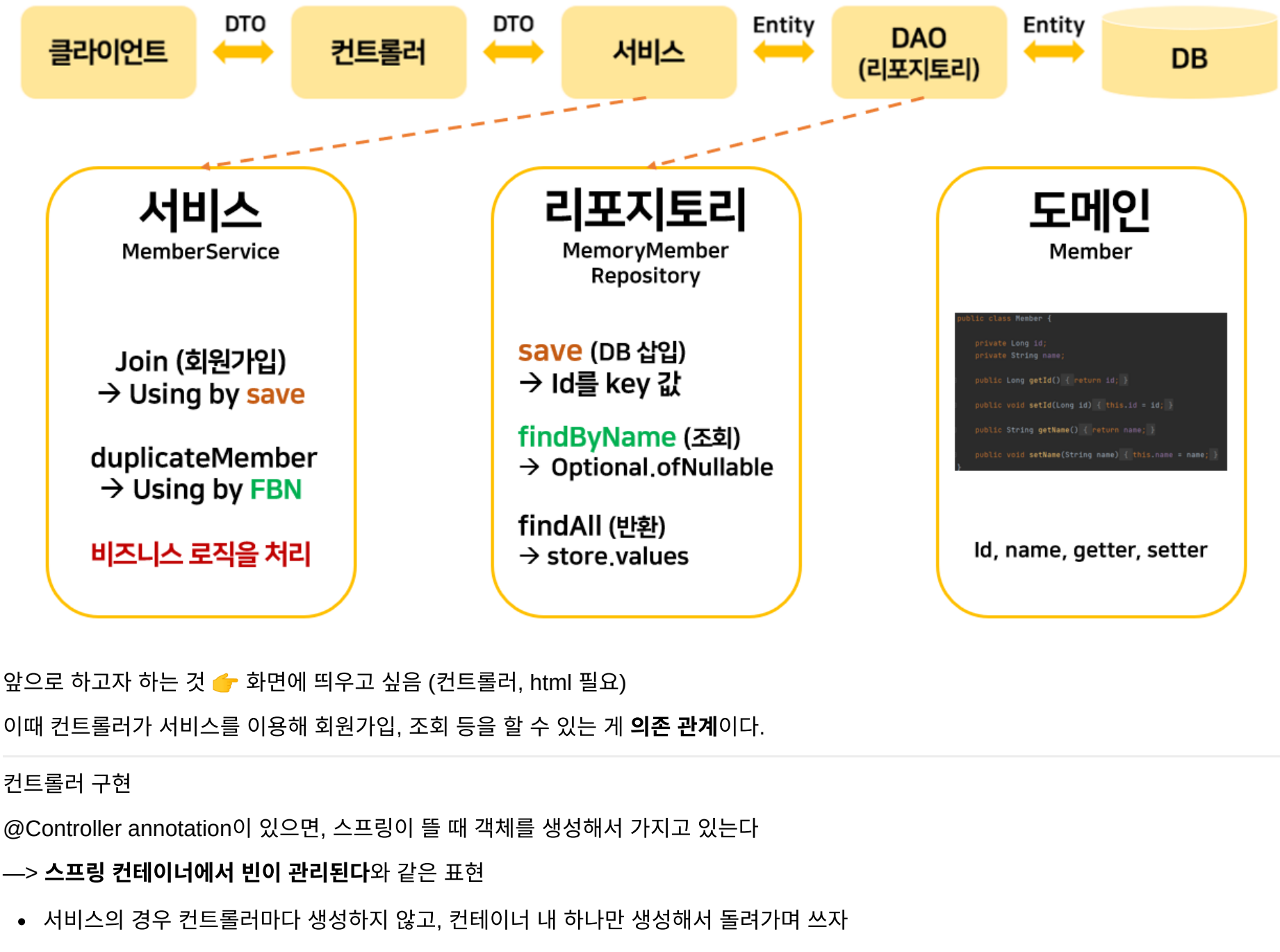
Spring 서비스 구조

https://dahye-jeong.gitbook.io/spring/spring/2020-04-12-layer

섹션 4. 스프링 빈과 의존 관계

✅ 스프링 빈을 등록하고 의존 관계를 어떻게 하는 지 알아보자

지난 시간까지 우리가 한 것 👉 멤버 객체, 서비스, 리포지토리, 테스트를 진행



앞으로 하고자 하는 것 👉 화면에 띄우고 싶음 (컨트롤러, html 필요)

이때 컨트롤러가 서비스를 이용해 회원가입, 조회 등을 할 수 있는 게 **의존 관계**이다.

컨트롤러 구현

@Controller annotation이 있으면, 스프링이 뜰 때 객체를 생성해서 가지고 있는다

—> **스프링 컨테이너에서 빈이 관리된다**와 같은 표현

- 서비스의 경우 컨트롤러마다 생성하지 않고, 컨테이너 내 하나만 생성해서 돌려가며 쓰자
- 이때 생성자를 만들면서 스프링으로부터 서비스를 바로 받아옴 (@Autowired)
- Autowired를 사용하기 위해선 자기 자신도 컨테이너에 올라가 있어야 함
- 받아오는 서비스에도 컨트롤러와 마찬가지로 @Service annotation 명시 필요

@Controller @Service @Repository annotation을 이용해 스프링 컨테이너에 bin으로 올려주자
👉 컴포넌트 스캔(Component Scan) 방식
컨트롤러를 통해 요청을 받고, 서비스를 이용해 비지니스 로직을 정의, 리포지토리로 데이터 저장

```
@Autowired
public MemberController(MemberService memberService) {
    this.memberService = memberService;
}
```

컨트롤러 생성 시 컨테이너에 있는 스프링을 **주입**시켜준다 (Dependency Injection)
—> 왜냐? 서비스를 통해 컨트롤러가 요청을 수행하니까 의존성 주입

스프링 빈을 등록하는 2가지 방법

1. 컴포넌트 스캔(@Service, Controller, Repository))과 자동 의존관계 생성(@Autowired)
2. 자바 코드로 직접 스프링 빈 등록하기

💡 컨테이너에 스프링 빈을 등록할 때는, 기본적으로 싱글톤으로 등록한다
💡 정형화된 방식은 컴포넌트 스캔 방식이나, 상황에 따라 구현 클래스를 변경할 경우 2번 사용

자바 코드로 직접 스프링 빈 등록하기

```java
@Configuration
public class SpringConfig {

    @Bean
    public MemberService memberService() {
        return new MemberService(memberRepository());
    }

    @Bean
    public MemberRepository memberRepository() {
        return new MemoryMemberRepository();
    }
}
```

1. Spring package 내에 Configuration class 만들기
2. annotation으로 Configuration 지정하기
3. @Bean annotation을 이용해 생성자 만들어주기 (의존성까지 고려)

💡 의존성 주입(Dependency injection) 방식에는 필드 주입, setter 주입, 생성자 주입이 있다.
setter의 경우 보안 문제가 발생할 수 있으며 생성자 주입이 가장 좋다. (처음 선언 후 변경 불가)
