23.11.21 ~ 22 디자인 패턴

상태	Task
작성일시	@2023년 11월 20일 오후 4:58

Tasks

▼ Java 디자인 패턴

1. 디자인 패턴이란 ?

   개발하면서 발생하는 반복 적인 문제 들을 어떻게 해결할 것인지에 대한 해결 방안

   객체 지향 프로그래밍 설계를 할 때 자주 발생하는 문제들을 피하기 위해 사용되는 패턴

   객체 지향 4대 특성 (캡슐화 , 상속 , 추상화 , 다형성) 과 설계 원칙을 기반으로 구현되어 있다.

▼ 디자인 패턴의 장점

- **재 사용성** : 반복적인 문제에 대한 일반적인 해결책을 제공하므로, 이를 재 사용 하여 유사한 상황에서 코드를 더 쉽게 작성할 수 있다.

- **가독성** : 일정한 구조로 정리하고 명확하게 작성하여 개발자가 코드를 이해하고 유지보수하기 쉽게 만든다.
- **유지보수성** : 코드를 쉽게 모듈화 할 수 있으며, 변경이 필요한 경우 해당 모듈만 수정하여 유지보수가 쉬워진다.
- **확장성** : 새로운 기능을 추가하거나 변경할 때 디자인 패턴을 활용하여 기존 코드를 변경하지 않고도 새로운 기능을 통합할 수 있다.
- **안정성과 신뢰성** : 수많은 사람들이 인정한 모범 사례로 검증된 솔루션을 제공한다.

▼ 디자인 패턴의 종류

▼ 싱글톤 패턴 (Singleton 패턴)

- 프로그램 전반에서 하나의 인스턴스 만을 사용하게 하는 디자인 패턴
- 사용 목적 : 정보를 보관하고 공유하고자 하는 클래스가 한번의 메모리만 할당하고 그 할당한 메모리에 대해 객체로 관리하기 위함

▼ 장점

- 한번의 객체 생성으로 재사용이 가능 하기 때문에 메모리 낭비를 방지 할 수 있다.
- 싱글톤으로 생성된 객체는 전역성을 띄기 때문에 다른 객체 간에 공유가 쉽다.

▼ 단점

1. 의존성이 높아진다.

싱글톤 패턴을 사용하는 경우 클래스의 객체를 미리 생성한 뒤에 필요한 경우 정적 메서드를 이용하기 대문에 클래스 사이에 의존성이 높아지게 된다는 문제점이 있다. (= 높은 결합)

싱글톤의 인스턴스가 변경 되면 해당 인스턴스를 참조하는 모든 클래스들을 수정해야 하는 문제가 발생한다.

2. private 생성자 때문에 상속이 어렵다.

싱글톤 패턴은 기본 생성자를 private로 만들었기 때문에 상속을 통한 자식 클래스를 만들 수 없다는 문제점이 있다. 즉, 자바의 객체지향 언어의 장점 중 하나인 다형성을 적용하지 못한다는 문제로 이어진다.

3. 테스트하기가 힘들다.

싱글톤 패턴의 인스턴스는 자원을 공유하고 있다는 특징이 있다. 이는 서로 독립적이어야 하는 단위 테스트를 하는데 문제가 된다.

독립적인 테스트가 진행이 되려면 전역에서 상태를 공유하고 있는 인스턴스의 상태를 매번 초기화해야 한다. 초기화해주지 않으면 전역에서 상태를 공유 중이기 때문에 테스트가 정상적으로 수행되지 못할 가능성이 존재한다.

- 싱글톤의 공통 특징
- private 생성자만을 정의해 외부 클래스로부터 인스턴스 생성을 차단한다.
- 싱글톤을 구현하고자 하는 클래스 내부에 멤버 변수로써 private static 객체 변수를 만든다.
- public static 메서드를 통해 외부에서 싱글톤 인스턴스에 접근할 수 있도록 접점을 제공

```
public class Singleton {

    private Singleton(){}

    private static class SingletonHelper{
        private static final Singleton INSTANCE = new Singleton();
    }

    public static Singleton getInstance(){
        return SingletonHelper.INSTANCE;
    }
}

/*
SingletonHelper 클래스는 Singleton 클래스가 Load 될 때에도 Load 되지 않다가
getInstance()가 호출됐을 때 비로소 JVM 메모리에 로드되고, 인스턴스를 생성하게 됩니다.
*/
```

현재 가장 널리 쓰이고 있는 싱글톤 구현 방법 이다.

```
// 테스트 코드

@Test
void singletonServiceTest(){
    SingletonService singletonService1 = SingletonService.getInstance();
    SingletonService singletonService2 = SingletonService.getInstance();

    System.out.println("singletonService1 = " + singletonService1);
    System.out.println("singletonService2 = " + singletonService2);

    Assertions.assertThat(singletonService1).isSameAs(singletonService2);
}

// 동일한 객체가 반환 되는것을 확인 할 수 있다.
```

▼ 이로 인해 나타나는 문제점

순수 Java 코드로 싱글톤 패턴을 적용하는 방법이었는데 실제로 사용하기에는 불편하고 실질적인 문제점들이 존재한다.

1. **싱글톤 패턴을 구현하기 위한 코드가 늘어남**
2. **인스턴스를 반환해주는 구현 클래스를 직접 참조해야 하므로 DIP를 위반한다. (OCP 원칙)**
3. **내부 속성을 변경, 초기화하기가 어렵다.**

4. **private 생성자로 자식 클래스를 생성하기 어렵다.**
5. **유연성이 떨어진다.**

▼ 싱글톤 컨테이너 (Singleton Container)

스프링에서는 사용자가 이렇게 일일이 구현하지 않고 싱글톤 패턴의 문제점들도 보완해 주면서 싱글톤 패턴으로 클래스의 인스턴스를 사용하게 해 주는데 이것을 **싱글톤 컨테이너**라고 합니다.

```
@Test
void springContainer() {

    ApplicationContext ac = new AnnotationConfigApplicationContext(AppConfig.class);

    MemberService memberService1 = ac.getBean("memberService", MemberService.class);
    MemberService memberService2 = ac.getBean("memberService", MemberService.class);

    System.out.println("memberService1 = " + memberService1);
    System.out.println("memberService2 = " + memberService2);

    Assertions.assertThat(memberService1).isSameAs(memberService2);

}

// 결과

memberService1 = com.example.springdemostudy.member.MemberServiceImpl@78e16155
memberService2 = com.example.springdemostudy.member.MemberServiceImpl@78e16155
```

```
@Configuration
public class AppConfig {

    @Bean
    public MemberRepository memberRepository() {
        System.out.println("call AppConfig.memberRepository");
        return new MemoryMemberRepository();
    }

    @Bean
    public DiscountPolicy discountPolicy() {
        return new RateDiscountPolicy();
    }

    @Bean
    public MemberService memberService() {
        System.out.println("call AppConfig.memberService");
        return new MemberServiceImpl(memberRepository());
    }

    @Bean
    public OrderService orderService() {
        System.out.println("call AppConfig.orderService");
        return new OrderServiceImpl(memberRepository(), discountPolicy());
```

```
    }
}
```

하지만 이렇게 수동으로 직접 빈을 등록하는 작업은 빈으로 등록하는 클래스가 많아질수록 상당히 많은 시간을 차지할 것이고, 생산력 저하를 야기할 것이다. 그래서 스프링에서는 특정 어노테이션이 있는 클래스를 찾아서 빈으로 등록해주는 컴포넌트 스캔 기능을 제공한다.

[@Bean, @Configuration]

수동으로 스프링 컨테이너에 빈을 등록하는 방법
개발자가 직접 제어가 불가능한 라이브러리를 빈으로 등록할 때 불가피하게 사용
유지보수성을 높이기 위해 애플리케이션 전범위적으로 사용되는 클래스나 다형성을 활용하여 여러 구현체를 빈으로 등록 할 때 사용
1개 이상의 @Bean을 제공하는 클래스의 경우 반드시 @Configuration을 명시해 주어야 싱글톤이 보장됨

[@Component]

자동으로 스프링 컨테이너에 빈을 등록하는 방법
스프링의 컴포넌트 스캔 기능이 @Component 어노테이션이 있는 클래스를 자동으로 찾아서 빈으로 등록함
대부분의 경우 @Component를 이용한 자동 등록 방식을 사용하는 것이 좋음
@Component 하위 어노테이션으로 @Configuration, @Controller, @Service, @Repository 등이 있음

Java 힙 메모리는 애플리케이션 내에서 실행 중인 모든 스레드가 액세스할 수 있는 전역 공유 메모리입니다.

Spring 컨테이너가 싱글톤 스코프의 빈을 생성할 때 빈은 힙에 올라갑니다.

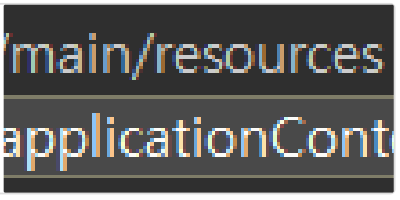
스프링의 방식대로라면, 모든 스레드가 동일한 Bean 인스턴스를 가리킬 수 있습니다.

https://github.com/seongminHong1/starlight/blob/main/Secret_joojoo/src/main/java/com/spring/view/controller/MainController.java

Annotation (@, 어노테이션)

new (= 객체화)를 스프링 컨테이너에게 객체 생성 및 관리하도록 부탁한다. 방법 1) .xml <bean&g...

https://blog.naver.com/itchild_/223172382364



@ 어노테이션의 종류

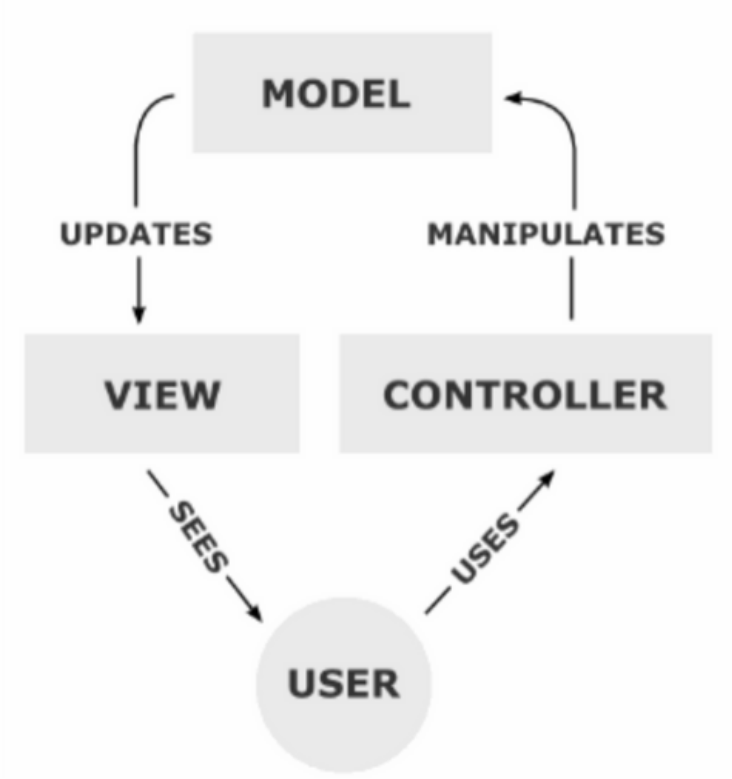
지금 까지 배운 내용들 중심으로 @ 어노테이션의 종류에 대해 정리 해보았다. @Component (스프링 컨테이...

https://blog.naver.com/itchild_/223174239418

우리가 프로젝트에서 적용한 싱글톤 방식은 Ioc 제어의 역행을 해서 객체화를 직접 안해주고 했던거 , 의존주입 한것들 이런것들이 싱글톤 패턴에 해당

▼ MVC 패턴 (Model , View , Controller 패턴)

- 관심사를 분리하여 업무의 분할 관리를 제공 (모듈화를 강제해서 개발의 편의성을 도모)
- 사용자 인터페이스로부터 비즈니스 로직을 분리하여, 서로 영향없이 독립적으로 유지보수 가능
- 개발 효율성 및 유지보수성이 높아짐



▶️ **M** Model 모델 (DB와 관련되어있다.)

데이터(DB)와 관련된 로직을 담당하는 파트

1)DB에서 데이터를 다루는 방법에 대한 코딩

== SQL

2)DB의 데이터 <-> JAVA환경으로 가져올수 있도록

== JDBC

: CRUD 기능을 구현하는 파트 (사용자에게 제공할 서비스를 코딩 하는 부분)

: CRUD 란? 생성(Create) , 읽어오기(Read), 업데이트 (Update), 삭제(Delete) 의 앞 이니셜을 따서 CRUD 라고 부르며 이 로직을 큰 틀로 잡고 코딩을 진행한다.

: 비즈니스 메서드, 핵심 로직

▶️ **V** View 뷰 (사용자와 관련된거)

사용자가 보는 화면에 대한 모든것을 담당하는 파트

사용자 편의성 (UI /UX) 고려

유효성 검사도 필요한 부분

사용자와의 입출력을 담당하는 파트

▶️ **C** Controller 컨트롤러 (연결을 관리 하는거)

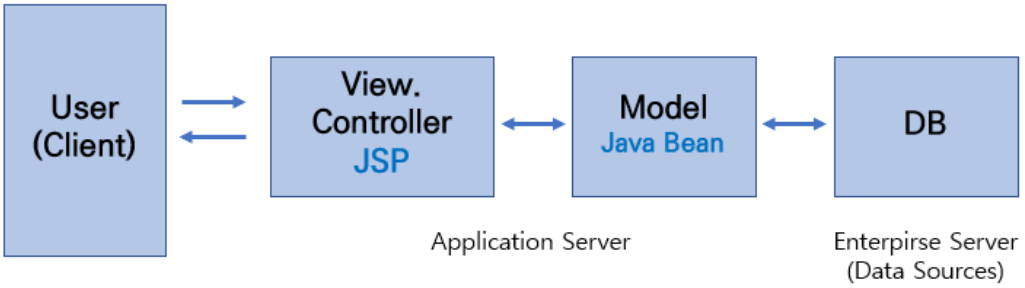
View <---> Model을 연결하는 파트

★ **여기서 MVC 패턴의 핵심은 ,**

Model 과 View가 절대 서로 딱 ! 붙어있지 않게 하는것 !

무조건! 분리 되어 있어야함 !!!

그리고 사용자가 DB에 직접 접근 하는 일은 절대 없어야 한다



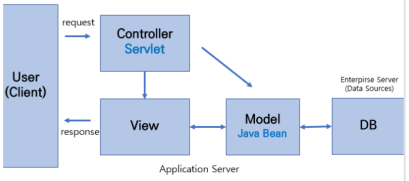
https://github.com/jihyean/Web/tree/main/day42/webapp

MVC 2(FrontController) 패턴 , Spring MVC

기존 JSP -> Spring 이관작업 흐름 정리

JSP MVC2 패턴 (FrontController) 기존 JSP MVC2 패턴 (FrontController) 을 이용하여 작...

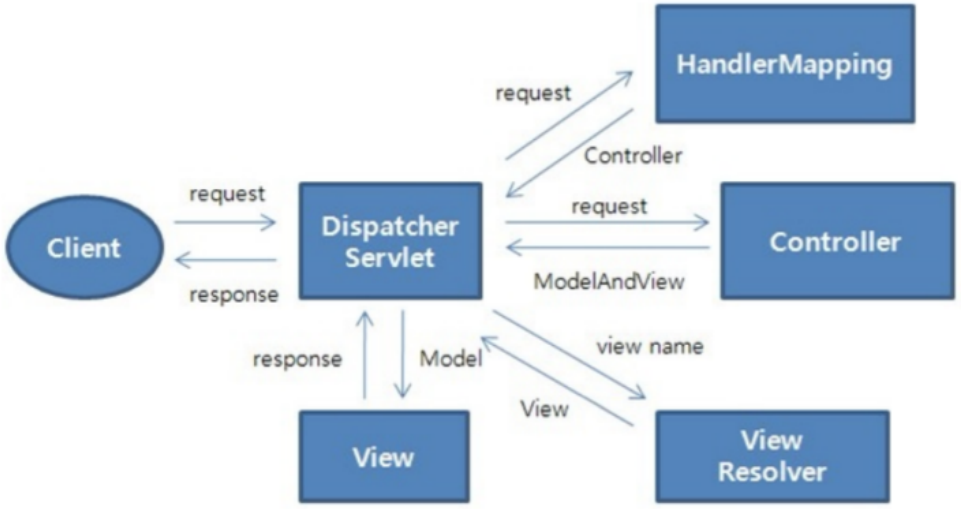
https://blog.naver.com/itchild_/223173182847



https://github.com/jihyean/Web/tree/main/day48

▼ Factory 패턴

- 객체명으로 객체를 반환받을수있도록 해주는 패턴
- 이미 메모리에 생성되어있던(적재,load) 객체를 반환받음
- 결합도를 낮춤



1. 클라이언트가 요청
2. DS 이 반응 한다. == FrontController 역할을 합니다.
3. 요청에 맞는 Controller를 반환하는 HM (실제로 만드는 사람)

요청에 맞는 컨트롤러 뭔가 OUTPUT 이 나올수 있다.

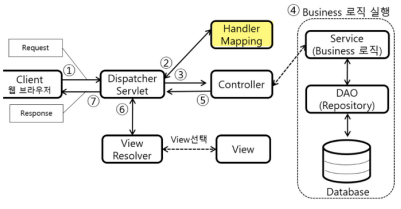
어떠한 컨트롤러가 와야 하는지 HM을 통해서 받습니다.

4. 서버에서 클라이언트에게 응답하기 위한 페이지를 반환하는 VR 페이지가 반환

Handler Mapping과 Request Mapping이란?

HANDLER MAPPING, REQUEST MAPPING 모두 웹 애플리케이션에서 요청을 처리하는데 사용되는 개념이다. 이 두 개념은 Spring MVC에서 주로 사용되는 개념으로 웹 애플리케이션에서 클라이언트

https://sharonprogress.tistory.com/196



https://github.com/seongminHong1/starlight/blob/main/Secret_joojoo/src/main/java/com/spring/view/controller/MainController.java

팩토리 패턴은 "객체줘"했을 때 해당 객체 주는 느낌이라
HandlerMapping이 Controller객체 뱉어주는게 좀더 정확한 패턴 이라고 이해

▼ Template 패턴

- JSP 기반의 웹 프로젝트

https://github.com/seongminHong1/starlight/tree/main/별빛역_TeamProject

- Spring 에서의 Template 패턴

https://blog.naver.com/itchild_/223185404577