2020-10-14

\*스프링 부트

스프링을 편리하게 사용할 수 있도록 지원한다.

단독으로 실행할 수 있는 스프링 애플리케이션을 쉽게 생성

Tomcat 같은 웹서버를 내장해서 별도의 웹서버를 설치하지 않아도 된다.

손쉬운 빌드 구성을 위한 starter 종속성 제공

스프링과 3rd parth 라이브러리 자동구성

모니터링 제공

\*스프링

스프링 DI컨테이너 기술

스프링 프레임워크

스프링 부트, 스프링 프레임워크 등을 모두 포함한 스프링 생태계

\*스프링을 만든이유?

핵심 개념, 컨셉?

스프링은 **좋은 객체 지향** 애플리케이션을 개발할 수 있게 도와주는 프레임 워크다.

\*객체 지향 프로그래밍

객체들의 모임으로 파악, 각각의 객체는 메시지를 주고받고, 데이터를 처리할 수 있다.

유연하고 변경이 용이하다. -> 내부의 내용을 알지못하고 다른 것으로 대체 가능 ..?!

-다형성

클라이언트는 인터페이스만 알면 된다.

내부 구조를 몰라도 된다.

내부 구조가 변경되어도 영향을 받지 않는다.

대상 자체를 변경해도 영향을 받지 않는다.

오버라이딩(다형성으로 인터페이스를 구현한 객체를 실행 시점에 유연하게 변경할 수 있다.)

**클라이언트를 변경하지 않고, 서버의 구현 기능을 유연하게 변경할 수 있다.**

개발시 인터페이스를 안정적으로 잘 설계하는 것이 중요하다.!!!!

\*좋은 객체 지향 설계의 5가지 원칙 (SOLID)

SRP:단일 책임 원칙(single responsibility principle)

한 클래스는 하나의 책임만 가져야한다.(중요한 기준은 변경이다.)

OCP:개방-폐쇄 원칙(Open/closed principle)

확장에는 열려 있으나 변경에는 닫혀 있어야한다.(다형성)

새로운 클래스를 하나 만들어서 새로운 기능을 구현하는 것 .

구현 객체를 변경하려면 클라이언트 코드를 변경해야한다.

다형성을 사용했지만 OCP 원칙을 지킬 수 없다.

스프링 컨테이너가 객체를 생성하고, 연관관계를 맺어주는 별도의 조립, 설정자가 되어주는 것이다.

LSP:리스코프 치환 원칙(Liskov substitution principle)

프로그램의 객체는 프로그램의 정확성을 깨뜨리지 않으면서 하위 타입의 인스턴스로 바꿀 수 있어야 한다.

ISP:인터페이스 분리 원칙(Interface segregation principle)

특정 클라이언트를 위한 인터페이스 여러 개가 범용 인터페이스 하나보다 낫다

DIP:의존관계 역전 원칙(Dependency inversion principle)

프로그래머는 추상화에 의존해야지 , 구체화에 의존하면 안된다.

정리

다형성만으로는 구현 객체를 변경할 때 클라이언트 코드도 함께 변경된다.

OCP, DIP를 지킬 수 없다. 뭔가 더필요하다!!!

\*스프링 ? 객체지향 ?

DI : 의존관계, 의존성 주입, DI 컨테이너제공

2020-10-15

\*비즈니스 요구사항과 설계

요구사항이 미확정 되어 있어도 인터페이스를 사용해서 미리 구현이 가능하다 .(역할과 구현을 참고)

\*도메인 설계

클라이언트 -> 서비스 -> 저장소( 메모리 저장소, DB 저장소 , 외부 시스템 연동 회원 저장소)

\*어떤 것을 할지 모를 때 메모리 저장소로 구현한다 .

클래스 다이어그램만으로는 확실 하게 알 수 없기 때문에 객체 다이어그램도 그려야 한다.

\*테스트

Junit – 테스트를 하기위해서

//@Test 어노테이션을 사용한후

//given 어떤 것이 주어졌을 때.

//when 무엇을 했을 때

//then 어떻게 되는 가

여기서는 Assertions.assertThat(member).isEqualTo(findMember); 과 같이 사용한다 .

테스트 코드는 필수다.

@BeforeEach 를 사용할 수 있다. Test 를 실행 하기전

\*TIP

인터페이스 구현체가 한 개 일 때 impl를 붙힌다 .

Enum Type은 == 을 쓴다. Member.getGrade() == Grade.VIP

성공테스트 뿐만 아니라 실패 테스트도 만들어 보아야 한다.

**DIP를 지키기 위해서 인터페이스에만 의존하도록 설계와 코드를 변경하면된다. 하지만 NPE가 발생하게된다 .**

**따라서 누군가 구현객체를 생성 시켜줘야한다 .**

Class 안에 static class를 만들었다는건 그안에서 scope를 설정 하겠다는 뜻.

**\***관심사의 분리

AppConfig 등장

구현 객체를 생성하고 , 연결하는 책임을 가지는 별도의 설정 클래스

생성자 주입을 한다.

의존관계에 대한 고민은 외부에 맡기고 실행에만 집중하면 된다. (DIP 완성)

**2020-10-16**

\*AppConfig

설정정보

애플리케이션이 크게 사용 영역과, 객체를 생성하고 구성(Configuration)하는 영역으로 분리되었다**.**

영향 범위가 작아진다.

\*IOC(Inversion of Control)

프로그램에 대한 제어 흐름을 넘기는 것 (예: Appconfig)

프로그램의 제어흐름을 직접 제어하는 것이 아니라 외부에서 관리하는 것을 제어의 역전이라 한다.

\*프레임워크 vs 라이브러리

프레임워크 : 내가 작성한 코드를 제어하고 , 대신 실행하면 그것은 프레임 워크가 맞다.(JUnit)

라이브러리 : 내가 작성한 코드가 직접 에어의 흐름을 담당한다면 라이브러리 이다.

\*의존관계 주입 DI

실제 어떤 구현 객체가 사용될지는 모른다.

정적인 클래스 의존관계와, 실행 시점에 결정되는 동적인 객체(인스턴스)의존관계 둘을 분리해서 생각 해야한다.

정적인 클래스 의존관계

클래스가 사용하는 import코드만 보고 의존관계 쉽게 판단 가능.

실행하지 않아도 분석 가능

동적인 객체 인스턴스 의존관계

애플리케이션 실행 시점에 실제 생성된 객체 인스턴스의 참조가 연결된 것이다.

**실행 시점(런 타임)에 외부에서 실제 구현 객체를 생성하고 클라이언트에 전달해서 클라이언트와 서버의 실제 의존관계가 연결되는 것을 의존 관계 주입이라고 한다.**

**정적인 클래스 의존관계를 변경하지 않고, 변경 가능하다.**

\*IoC컨테이너, DI컨테이너

appConfig처럼 객체를 생성 관리 하면서 의존관계 연결해주는 것

**2020-10-17**

\*ApplicationContext

스프링 컨테이너라 한다. (인터페이스다)

AppConfig를 사용해서 객체생성 , DI를 했지만, 스프링 컨테이너를 사용한다.

스프링 컨테이너는 @Configuration이 붙은 설정 정보를 사용한다 .

XML기반으로 만들 수 있고, 애노테이션 기반의 자바 설정 클래스로 만들 수 있다.

설정 버로를 참고해서 의존관계를 주입한다. (DI)

new AnnotationConfigApplicationContext(XXX.class) 예) Appconfig.class

\*스프링 빈

스프링 컨테이너에 등록된 객체를 스프링 빈이라 한다.

빈 이름은 항상 다른 이름을 부여해야 한다.

직접 등록한 애플리케이션 빈(Role ROLE\_APPLICATION)과 스프링이 내부에서 사용하는 빈(ROLE\_INFRASTRUCTURE)이 있다.

\*스프링 빈 조회

빈 이름으로 조회

이름 없이 타입으로만 조회

구체 타입으로 조회(많이 사용 할 일 없음)

빈 이름으로 조회 되지않는 경우 (NoSuchBeanDefinitionException 발생)

조회 시 같은 타입의 스프링 빈이 둘 이상이면 오류 발생 : 이때는 빈 이름을 지정

상속관계

**부모 타입으로 조회하면, 자식 타입도 함께 조회한다.**

\*BeanFactory 와 ApplicationContext

BeanFactory

스프링 컨테이너의 최상위 인터페이스이다.

스프링 빈을 관리하고 조회하는 역할을 담당한다.

ApplicationContext

BeanFactory 기능을 모두 상속받아서 제공

애플리케이션을 개발할 때는 관리 , 조회 기능 말고 부가기능을 제공

MessageSource–메시지소스를 활용한 국제화기능(한국에서 들어오면 한국어, 영어권이면 영어로출력)

EnvironmentCapable- 환경 변수에 관련된 처리를 한다. 로컬,개발,운영 등 을 구분해서 처리

ApplicationEventPublisher – 이벤트를 발행하고 구독하는 모델을 편리하게 지원

ResourceLoader – 파일 , 클래스패스 , 외부 등에서 리소스를 편리하게 조회

\*XML 설정

ApplicationContext를 사용해서

AnnotationConfig ApplicationContext AppConfig.class 사용

GenericXmlApplicationContext appConfig.xml 사용

Xxx ApplicationContext appConfig.xxx 사용

예) <bean id="memberService" class="hello.core.member.MemberServiceImpl">

<constructor-arg name="memberRepository" ref="memberRepository" />

</bean>

**2020-10-19**

\*스프링 빈 설정 메타 정보 -BeanDefinition

BeanDefinition을 빈 설정 메타정보라 한다.

@Bean , <bean> 당 각각 하나씩 메타 정보가 생성된다.

스프링 컨테이너는 이 메타정보를 기반으로 스프링 빈을 생성한다.

스프링 컨테이너는 BeanDefinition에 의존한다.

\*annotated BeanDefinitionReader

ApplicationContext -> AnnotationConfigApplicationContext -> Annotated BeanDefinitionReader ->AppConfig.class

AnnotatedBeanDefinitionReader 은 설정 정보를 AppConfig.class에서 읽고 빈 메타정보를 생성한다.

\*XmlBeanDefinitionReader

ApplicationContext -> GenericXmlApplicationContext -> XmlBeanDefinitionReader ->appConfig.xml

XmlBeanDefinitionReader은 설정 정보를 appConfig.xml에서 읽고 BeanDefinition 을 생성한다.

\*BeanDefinition

정보

BeanClassName: 생성할 빈의 클래스 명

factoryBeanName: 팩토리 역할의 빈을 사용할 경우

factoyrMethodName: 빈을 생성할 팩토리 메소드 지정

Scope: 싱글톤 (Default 값)

lazyInit: 스프링 컨테이너를 생성할 때 생성하는 것이 아니라, 실제 빈을 사용할 때 까지 지연처리 하는지 여부

InintMethodName: 빈을 생성하고 , 의존관계를 적용한 뒤에 호출되는 초기화 메서드명

DestroyMethodName: 빈의 생명주기가 끝나서 제거하기 직전에 호출되는 메서드명

Constructor arguments, Properties: 의존관계 주업에서 사용한다.

\*메타정보를 사용해서 빈을 생성한다 .

BeanDefinition을 직접 생성해서 스프링 컨테이너에 등록할 수도 있다.

스프링은 다양한 형태의 설정 정보를 BeanDefinition으로 추상화 해서 사용한다.

자주 사용하는 빈을 등록하는 법

직접 스프링빈을 등록

팩토리 메서드를 사용해서 등록(java AppConfig를 사용해서 하는 것 )

\*웹 애플리케이션과 싱글톤

웹어플리케이션은 여러 고객이 요청이 매우 많다.

스프링 없는 순수한 DI 컨테이너를 사용하게 되는 경우에 객체를 새로 생성한다 .

메모리 낭비가 매우 심하다 -> 싱글톤으로 해결

\*싱글톤 패턴

인스턴스가 딱 1개만 생성되는 것을 보장하는 디자인 패턴이다.

Static 영역에 객체 instance를 미리 하나 생성해서 올려둔다.

생성자를 private으로 막는다 .

스프링컨테이너를 쓰면 알아서 싱글톤 패턴으로 관리해준다 .

문제점.

싱글톤 패턴을 구현하는 코드 자체가 많이 들어간다.

의존관계상 클라이언트가 구체 클래스에 의존한다. DIP를 위반한다.

클라이언트가 구체 클래스에 의존해서 OCP 원칙을 위반할 가능성이 높다.

테스트하기 어렵다.

내부 속성을 변경하거나 초기화 하기 어렵다.

private 생성자로 자식 클래스를 만들기 어렵다.

유연성이 떨어진다.

안티패턴으로 불리기도 한다.

2020-10-20

\*스프링 컨테이너

싱글톤 컨테이너, 싱글톤 패턴의 문제점을 해결하면서 , 인스턴스를 1개로 관리한다.

싱글톤 패턴을 적용하지 않아도, 객체 인스턴스를 싱글톤으로 관리한다.

이미 만들어진 객체를 고융해서 효율적으로 재사용 할 수 있다.

요청할 때마다 새로운 객체를 생성해서 반환하는 기능도 제공한다.

(특별한 기능을 할 때 , 요청할 때마다 새로 꺼낸 다거나 HTTP LifeCycle 을 맞출 때 등)

\*싱글톤 레지스트리

싱글톤 객체를 생성하고 관리하는 기능

\*싱글톤 방식의 주의점

여러 클라이언트가 하나의 같은 객체 인스턴스를 공유하기 때문에 싱글톤 객체는 상태를 유지하게 설계하면 안된다.

**무상태로 설계해야 한다**.

특정 클라이언트에 의존적인 필드가 있으면 안된다.

특정 클라이언트가 값을 변경할 수 있는 필드가 있으면 안된다!

가급적 읽기만 가능해야한다.

필드 대신에 자바에서 공유되지 않는, 지역변수, 파라미터, ThreadLocal등을 사용해야 한다.

ThreadLocal을 이용하면 쓰레드 영역에 변수를 설정할 수 있기 때문에,

특정 쓰레드가 실행하는 모든 코드에서 그 쓰레드에 설정된 변수 값을 사용할 수 있게 된다.

\*Configuration

bean = class hello.core.AppConfig$$EnhancerBySpringCGLIB$$dc3bb617

순수한 클래스라면 ‘class hello.core.AppConfig 로 출력되어야한다.

xxxCGLIB가 붙으면서 상당히 복잡해진 것을 볼 수 있다. 이것은 내가

만든 클래스가 아니라 스프링이 CGLIB라는 바이트코드 조작 라이브러리를 사용해서 AppConfig 클래스

를 상속받은 임의의 다른 클래스를 만들고, 그 다른 클래스를 스프링 빈으로 등록한 것이다!

If( 이미 스프링 컨테이너에 등록 되어 있으면 ) 찾아서 반환

Else 새로 만들어서 등록한후 반환

\*@Bean이 붙은 메서드마다 이미 스프링 빈이 존재하면 존재하는 빈을 반환, 빈이 없으면 생성해서

빈으로 등록하고 반환하는 코드가 **동적으로 만들어진다.** (조작라이브러리를 이용해서)

2020-10-22

\*컴포넌트 스캔

스프링빈이 수십,수백개가 되면 일일이 등록하기도 귀찮고, 설정 정보도 커지고, 누락하는 문제도 발생한다.

스프링 설정 정보가 없어도 자동으로 스프링 빈을 등록하는 하는 것이 컴포넌트 스캔이다 .

의존 관계도 자동으로 주입하는 ‘@Autowired 라는 기능도 제공한다. (생성자에 붙혀준다.)

기본 조희 전략은 타입이 같은 빈을 찾아서 주입한다.

@ComponentScan - @Component를 전부 찾아서 스프링빈으로 등록을 해준다.

이때 스프링 븐의 기본 이름은 클래스명을 사용하되 맨 앞글자만 소문자를 사용한다.

\*탐색 위치와 기본 스캔 대상

@ComponentScan(

basePackages = “hello.core”, … // 여기서부터 컴포넌트 스캔이 시작된다.

)

만약 basepackages 를 지정 하지않으면 @ComponentScan 이 붙은 설정 정보 클래스의 패키지가 시작 위치가 된다.

권장하는 방법

패키지 위치를 지정하지 않고, 설정 정보를 클래스의 위치를 프로젝트 최상단에 두는 것이다.

기본대상

@Component : 컴포넌트 스캔에서 사용

@Controller : 스프링 MVC 컨트롤러에서 사용 (부가기능: 스프링 MVC 컨트롤러 인식)

@Service : 스프링 비즈니스 로직에서 사용 (부가기능:특별한 처리를 하지않는다. 대신 개발자들이 핵심

비즈니스 로직이 여기에 있겠구나 라고 비즈니스 계층을 인식하는데 도움이 된다.)

@Repository : 스프링 데이터 접근 계층에서 사용(부가기능: 스프링 데이터 접근 계층으로 인식하고

, 데이터 계층의 예외를 스프링 예외로 변환해준다.)

@Configuration : 스프링 설정 정보에서 사용 (부가기능:스프링 설정 정보로 인식하고,

스프링 빈이 싱글톤을 유지하도록 추가 처리를 한다.)

해당 클래스의 소스를 보면 @component 를 포함하고있다.

\*애노테이션에는 상속관계라는 것이 없다. 그래서 이렇게 애노테이션이 특정 애노테이션을 들고 있는 것을 인식

할 수 있는 것은 자바 언어가 지원하는 기능이 아니고, 스프링이 지원하는 기능이다.

\*필터

FilterType 옵션

ANNOTATION: 기본값 , 애노테이션을 인식해서 동작한다.

ASSIGNABLE\_TYPE: 지정한 타입과 자식타입을 인식해서 동작한다.

ASPECTJ : AspectJ 패턴사용

REGEX:정규표현식

CUSTOM : TypeFilter 이라는 인터페이스를 구현해서 처리

\*중복 등록과 충돌

자동 빈 등록 vs 자동 빈 등록

ConflictingBeanDefinitionException 예외 발생

수동 빈 등록 vs 자동 빈 등록

수동 빈 등록이 우선권을 가진다. (수동 빈이 자동 빈을 오버라이딩 해버린다.)

스프링 부트로 실행 해보면 오류가 난다.

수동 빈 등록과 자동 빈 등록이 충돌나면 오류가 발생하도록 기본 값을 바꾸었다.== false

Resources -> application.properties 에 spring.main.allow-bean-definition-overriding= true 로하면 가능

2020-10-23

다양한 의존관계 주입 방법

생성자 주입

생성자 호출시점에 딱 1번만 호출되는 것이 보장된다.

\*불변,필수 의존관계에 사용

생성자가 딱 1개만 있으면 @Autowired 생략 가능

수정자 주입 (setter 주입)

setter라 불리는 필드의 값을 변경하는 수정자 메서드를 통해서 의존관계를

주입하는 방법

선택, 변경 가능성이 있는 의존관계에 사용

자바빈 프로퍼티 규약의 수정자 메서드 방식을 사용하는 방법이다.

@Autowired(required = false) : @Autowired 의 기본 동작은 주입할 대상이 없으면 오류가 발생하지만 동작하기위해서 사용

필드 주입

이름 그대로 필드에 바로 주입

코드가 간결 하지만 외부에서 변경이 불가능 해서 테스트 할 수 있는 방법 X

(순수한 자바 테스트를 할 때 )

DI프레임워크가 없으면 아무것도 할 수 가 없다.

일반 메서드 주입

한번에 여러 필드를 주입 받을 수 있다.

일반적으로 자주 사용 하지않는다 .

\*의존관계 자동 주입은 스프링 컨테이너가 관맇나느 스프링 빈이어야 동작한다.

옵션처리

주입할 스프링 빈이 없어도 동작해야 할 때가 있다.

@Autowired만 사용 하면 required 옵션의 기본값은 ture이기 때문에 오류 발생

@Autowired(required =false) : 자동 주입 대상이 없으면 메서드 자체가 호출 X

@Nullable Member member : 자동 주입할 대상이 없으면 null이 입력

Optional<Member> member : 자동 주입 대상이 없으면 Optional.empty가 입력

생성자 주입을 선택해라

불변

대부분의 의존관계 주입은 한번 일어나면 종료시점 까지 의존관계를

변경할 일이 없다. 그리고 대부분은 변하면 안된다.

수정자 주입을 사용하면, setXxx메서드를 public으로 열어두어야한다.

누군가 실수로 변경 할 수도 있다.

생성자 주입은 객체를 생성할 때 딱 1번만 호출되므로 이후에 호출되는

일이 없다. 따라서 불변하게 설계 할 수있다.

누락

프레임 워크 없이 순수한 자바 코드를 단위 테스트 하는 경우에 수정자 의존

관계를 사용 하게 되면 NPE 가 발생한다. 더미를 만들어야한다.

생성자 주입을 하게되면 어떤 매개변수가 필요한지 정확하게 알 수있고,

테스트가 용이하다.

final 키워드를 사용 시 final키워드 사용한 필드는 생성자로도 넣을 수있다.