스프링핵심원리-객체지향 설계

2021년 8월 2일 월요일 오후 2:32

스프링의 핵심 가치 : 객체지향 프로그래밍

<역사이야기>
자바 진영의 정파 기술 EJB , 오픈소스는 사파 느낌
자바진영에서 표준으로 인정한 기술이여서 영업이 잘됨=> 보급이 많이됨
컨테이너, 트랜젝션, 분산기술 등의 장점이 있었음
Entity bean 이라는 orm 기술도 가지고 있었음
BUT 비쌌다...
이론상으로는 너무 좋음 그러나 어렵고 복잡한데 느려
POJO(plan old java object) : 옛날 순수한 자바로 돌아가자 라는 말이 나올 정도

개빈 킹이라는 사람이 답답해서 하이버네이트를 만듬
=> EJB의 Entity Bean 기술을 대체
=> 자바가 개빈 킹을 영입하여 하이버네이트가 JPA 새로운 표준이 됨

하이버네이트는 실무 개발자들이 답답해서 만든 것이라 개발자친화적인데 좀 날 것의 맛이 났는데 이것이 자바 표준으로 들어오면서 표현 등이 정제되어 깔끔해져서 더 좋아졌다

로드 존슨 이라는 사람이 답답해서 EJB의 문제점 지적하면서 새로운 방법에 대한 책을 씀 이 책의 예제 코드가 스프링의 핵심 개념, 기반 코드가 됨 (BeanFactory, ApplicationContext, POJO, 제어의 역전, 의존관계 주입 등)
=> 유겐휠러, 얀카로프가 오픈소스 프로젝트를 제안하여 이것이 스프링으로 발전

JPA - orm 기술(자바 객체를 db에 편하게 저장하고 꺼내는 기술)

스프링이란? - 여러 기술의 집합체

- (필수)스프링 프레임워크 - 핵심
- (필수)스프링 부트 - 스프링 기술들을 편하게 사용할 수 있게 도와줌
- 스프링 데이터 - CRUD(등록 수정 삭제 조회)를 편리하게 사용할 수 있게 도와줌, 스프링 데이터 Jpa 제일 많이 사용
- 세션 - 세션 기능 도와줌
- 시큐리티 - 보안 관련
- Rest Docs -문서화 편리하게 해줌

- 배치 - 천만명의 데이터를 실시간 업데이트 어렵 => 천 건 처리하고 배치하고... 반복함 이러한 배치 처리에 특화
- 클라우드 -클라우드 관련

스프링 프레임워크
- 핵심 기술 : 스프링 DI 컨테이너, AOP, 이벤트, 기타 등등
- 웹 기술 : 스프링 MVC, 스프링 WebFlux
- 데이터 접근 기술 : 트랜잭션, JDBC, ORM 지원, XML 지원
- 기술 통합 : 캐시, 이메일, 원격접근, 스케줄링
- 테스트 : 스프링 기반 테스트 지원
- 언어 : Kotlin, Groovy
- 최근에는 스프링 부트 통해서 스프링 프레임워크 기술들을 편리하게 사용가능

스프링 부트
- 스프링을 편리하게 사용할 수 있게 지원, 최근에는 기본으로 사용
- 단독으로 실행할 수 있는 스프링 어플리케이션을 쉽게 생성
- Tomcat 같은 웹 서버를 내장하여 별도의 웹 서버를 설치하지 않아도 됨
- 손쉬운 빌드 구성을 위한 starter 종속성 제공
- 스프링과 서드파티 라이브러리 자동 구성
- 메트릭, 상태 확인, 외부 구성 같은 프로덕션 준비 기능 제공
- 관례에 의한 간결한 설정

핵심 개념
- 이 기술을 왜 만들었나?
- 이 기술의 핵심 컨셉은?

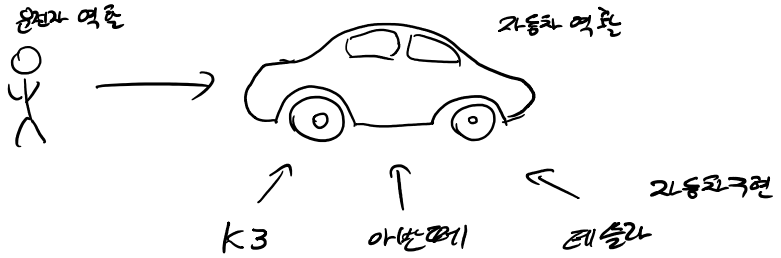
스프링은 자바 언어 기반 프레임워크
자바 언어의 가장 큰 특징인 객체 지향 언어의 강력함을 살려내는 프레임워크
=> 스프링은 좋은 객체 지향 어플리케이션을 개발할 수 있게 해주는 프레임워크
좋은 객체 지향 프로그밍이란?

객체 지향 특징
- 추상화
- 캡슐화
- 상속
- 다형성

객체 지향 프로그래밍 : 컴퓨터 프로그램을 명령어의 목록이 아닌 여러개의 독립된 단위 객체들의 모임으로 파악하려는 것. => 프로그램이 유연하고 변경 용이해짐
(레고 블록 조립, 컴퓨터 부품 갈아 끼우는 느낌으로다가) 컴포넌트를 쉽고 유연하게 변경하면서 개

발할 수 있는 방법

다형성 : Ex) 역활과 구현으로 세상을 구분



자동차를 바꾼다고 해서 운전자한테 영향X
자동차들은 자동차 역할(인터페이스)에 맞게 만들어졌고
운전자는 자동차 역할(인터페이스)에 대해서 알기만 하면(의존) 사용가능
운전자가 자동차의 내부 구현을 알 필요X, 기존 자동자 역활을 지키면 문제X

=> 운전자(클라이언트) 에게 영향 주지 않고 새로운 기능을 제공할 수 있다 +
운전자(클라이언트)는 바뀔 필요가 없다
why? 역활과 구현으로 구분하였기 때문에

다시 정리

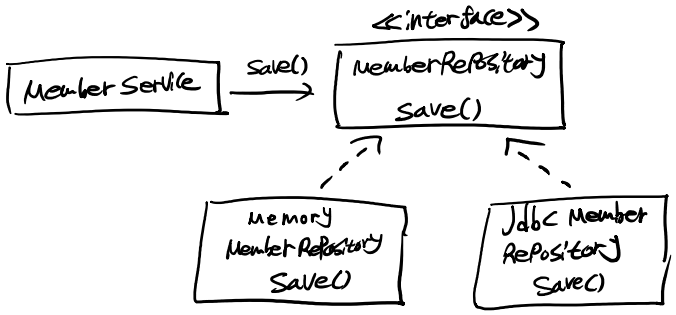
- 클라이언트는 대상의 역할(인터페이스)만 알면 됨
- 클라이언트는 구현 대상의 내부 구조 몰라도 됨
- 클라이언트는 구현 대상의 내부 구조가 변경되어도 영향 받지X
- 클라이언트는 구현 대상 자체를 변경해도 영향받지X

자바에서
역할 = 인터페이스
구현 = 인터페이스를 구현한 클래스, 구현 객체
객체를 설계할 때 역활과 구현을 명확하게 분리
객체 설계시 역할(인터페이스)를 먼저 부여하고, 역할 수행하는 구현체 만들어야...

객체의 협력이라는 관계부터 생각해야 한다
혼자있는 객체는 없다
클라이언트는 : 요청하는 사람 서버: 응답하는 사람
수많은 객체 클라이언트와 객체 서버는 서로 협력 관계를 가짐

자바의 다형성

오버로딩(overloading) - 같은이름의 메서드 여러개를 가지고
매개변수의 유형과 개수가 다르도록 하는 기술
오버라이딩(overriding) - 상위 클래스가 가지고 있는 메소드를
하위 클래스가 재정의해서 사용
다형성으로 인터페이스를 구현한 객체를 실행 시점에 유연하게 변경가능




```
public class MemberService {

   private MemberRepository memberRepository = new MemoryMemberRepository();

}
```

```
public class MemberService {

//  private MemberRepository memberRepository = new MemoryMemberRepository();
    private MemberRepository memberRepository = new JdbcMemberRepository();


}
```

다형성의 본질

- 인터페이스를 구현한 객체 인스턴스를 **실행 시점**에 유연하게 변경가능
- 다형성의 본질을 이해하려면 협력이라는 객체사이의 관계에서 시작해야함
- 클라이언트를 변경하지 않고, 서버의 구현기능을 유연하게 변경 가능

스프링과 객체 지향에서 다형성이 가장 중요
스프링은 다형성을 극대화해서 이용할 수 있게 해줌
스프링에서 이야기하는 제어의 역전(IOC), 의존관계 주입(DI)은 다형성을 활용해서 역활과 구현을 편리하게 다룰 수 있도록 지원
스프링을 사용하면 마치 레고 블록 조립하듯이 구현을 편리하게 변경할 수 있음

좋은 객체 지향 설계의 5가지 원칙 (SOLID) (면접에서 많이 물어봄;;)

- SRP : 단일 책임 원칙 (single responsibility principle)
- OCP : 개방-폐쇄 원칙 (Open/closed principle)

- LSP : 리스코프 치환 원칙 (Liskov subsitution principle)
- ISP : 인터페이스 분리 원칙 (Interface segregation principle)
- DIP : 의존관계 역전 원칙 (Dependency inversion principle)

SRP : 단일 책임 원칙 (single responsibility principle)

- 한 클래스는 하나의 책임만 가져야 한다 (but 하나의 책임이라는 것이 모호)
- 중요한 기준은 변경, 변경이 있을 때 파급효과가 적으면 단일 책임 원칙을 지킨 것

OCP : 개방-폐쇄 원칙 (Open/closed principle)

- 소프트웨어 요소는 확장에는 열려있으나 변경에는 닫혀 있어야 한다
- 다형성을 활용하여 가능하게 함
- 문제점
    - 구현 객체를 변경하려면 클라이언트 코드를 변경해야함
    - 분명 다형성을 사용했지만 OCP 원칙을 지킬 수 없다
    - => 객체를 생성하고 연관관계를 맺어주는 별도의 조립, 설정자가 필요

ex)

```
public class MemberService {

   private MemberRepository memberRepository = new MemoryMemberRepository();

}
```

```
public class MemberService {

//  private MemberRepository memberRepository = new MemoryMemberRepository();
    private MemberRepository memberRepository = new JdbcMemberRepository();


}
```

LSP : 리스코프 치환 원칙 (Liskov subsitution principle)

- 프로그램의 객체는 프로그램의 정확성을 깨뜨리지 않으면서 하위 타입의 인스턴스로 바꿀 수 있어야 한다
- 다형성에서 하위 클래스는 인터페이스 규약을 다 지켜야 한다는 것, 다형성을 지원하기 위한 원칙, 인터페이스를 구현한 구현체는 믿고 사용하려면 이 원칙이 필요함
- EX) 자동차 인터페이스의 엑셀은 앞으로 가는 기능, 뒤로 가게 구현하면 위반, 느리더라도 앞으로 가면 LSP 지킨 것

ISP : 인터페이스 분리 원칙 (Interface segregation principle)

- 특정 클라이언트를 위한 인터페이스 여러 개가 범용 인터페이스 하나보다 낫다
- EX) 자동차 인터페이스 -> 운전 인터페이스, 정비 인터페이스
  사용자 클라이언트 -> 운전자 클라이언트, 정비사 클라이언트
- 분리하면 정비 인터페이스가 자체가 변해도 운전자 클라이언트에 영향X
- 인터페이스 명확해지고, 대체 가능성 높아진다

DIP : 의존관계 역전 원칙 (Dependency inversion principle)

- 의존성 주입은 '추상화에 의존해야지, 구체화에 의존하면 안된다' 라는 법칙을 따르는 방법 중 하나 == 클라이언트 코드가 구현 클래스를 바라보지 말고 인터페이스를 바라봐야 된다는 것
- 즉 역활에 의존하게 해야 한다.
- 그런데 OCP에서 설명한 MemberService 는 인터페이스에 의존하지만, 구현 클래스도 동시에 의존함
- MemberService 클라이언트가 구현 클래스를 직접 선택
    - MemberRepository m = new MemoryMemberRepository();
- => DIP 위반

마무리
객체 지향의 핵심은 다형성
BUT 다형성 만으로는 구현 객체를 변경할 때 클라이언트 코드도 함께 변경되어
OCP, DIP를 지킬 수 없음
=> 스프링의 DI(Dependency Injection, 의존관계 주입), DI 컨테이너 를 통해 위를 해결함

But 이상적으로는 인터페이스를 도입하면 추상화라는 비용이 발생함
기능을 확장할 가능성이 없다면, 구체 클래스를 직접 사용하고, 향후 꼭 필요할 때 리팩터링해서 인터페이스를 도입하는 것도 방법이다.