Spring Boot Part3

스프링의 핵심(IOC, DI, AOP)
lombok(롬북) 라이브러리
롬북과 생성자 주입

Spring Framework의 사용 목적

스프링의 핵심 개념을 말하기 앞서, 우선 왜 스프링 프레임워크를 사용하는지 생각해 볼 필요가 있다. 사용하는 이유와 목적이 결국 핵심을 관통하기 때문이다. 스프링 프레임워크를 왜 사용하는가? 스프링 부트를 소개할 때 말했지만 스프링 프레임워크는 자바 기반의 웹 애플리케이션 개발에 사용하는 프레임워크이다. 결국, 스프링은 자바 기반으로 웹 개발을 할 수 있는 여러 기능을 제공하며, 웹 개발의 틀을 제공하기 때문에 대부분의 개발자가 이 틀만 잘 지킨다면 평균 이상의 코딩을 할 수 있도록 해주는 도구인 것이다. 그래서 사용하는 것이다. 여기서 눈여겨 볼 것은 스프링이 그럼 웹 개발을 위해 어떤 기능을 제공하는지도 있겠지만, 현재 더 관심을 가져야 하는 부분은 코딩의 틀이 어떻게 이루어져 있냐는 것이다. 우리는 이 틀을 잘 이해해야 스프링이 요구하는 형식대로 코드를 작성할 수 있으며, 이 틀만 잘 지켜면 최소한 나쁜 코딩은 하지 않게 되는 것이다.

Spring Framework가 지향하는 코드

스프링이 추구하는 코드는 결합도는 낮고, 응집도는 높은 코드이다. 그리고 이러한 코드는 spring 뿐만 아니라, 사실 모든 코드에서 추구되는 방향성이다. 하지만 항상 업무 시간이 부족한 현실에서 매번 이러한 조건을 지키며 코드를 작성하는 것은 이상적일 생각일 뿐, 현실 가능성이 매우 낮다. 그리고 이러한 문제점을 해결하기 위해 나온 것이 스프링이다. 스프링은 코드의 작성 방향을 개발자들이 따라올 수 있도록 틀을 만들었다. 그리고 이 틀을 잘 지키면 개발자가 신경쓰지 않아도 결합도는 낮고 응집도는 높은 코드를 구현할 수 있게 한 것이다. 결론적으로, 스프링이 구상한 Frame(틀)에 맞게 코드를 작성하면 낮은 결합도와 높은 응집도를 갖춘 코드를 개발자 모두가 작성할 수 있으며, 이러한 방향성이 spring의 탄생 배경이 된 것이다. 그리고 스프링은 결합도를 낮추기 위해 IOC, DI라는 개념을 도입하였고, 응집도를 높이기 위해서는 AOP라는 개념을 적용하였다. 결국, 스프링을 잘 이해하기 위해서는 IOC, DI, AOP 등의 스프링의 핵심 내용에 대해 반드시 알아야 한다. (AOP는 시간이되면 학습할 예정)

IOC(Inversion Of Control) : 제어의 역전

수업시간에 간단히 결합도가 높은 코드란 어떤 것이며, 결합도가 높을 경우 어떠한 문제점이 발생하는지 알아보았다. 간단하게 정리하면 결합도란 두 클래스 간의 연결 강도라고 알아두면 된다. 두 클래스의 연결이 강할수록 한 클래스의 코드가 수정되면, 연결된 클래스의 내용도 어떨 수 없이 수정해야 했다. 이러한 문제점은 상속, 인터페이스 등을 통해 어느정도 해소가 되었지만 결국 객체 생성 코드는 수정하거나 제거할 수 없었기에 결합도를 완전히 없애지 못했다. 다시 말하면 객체 생성 코드만 우리가 작성하지 않는다면 두 클래스 간의 결합도를 더욱 낮출 수 있다는 말이다. 그리고 스프링은 개발자가 직접 객체를 생성하는 코드를 작성하지 않아도 객체를 생성할 수 있게 설계되었다. 즉, 객체 생성 코드를 작성하지 않기 때문에 결합도를 없애버린 것이다.

그럼 도대체 어떻게 객체를 생성하지 않아도 코드가 실행된단 말인가. 스프링에서는 우리가(개발자가) 객체를 생성하지 않고, 스프링 컨테이너가 우리를 대신해서 객체를 생성해준다. 코드에는 직접적으로 객체 생성 문법이 없기 때문에 결합도가 생기지 않는다. 하지만 스프링 컨테이너가 개발자 대신 객체를 생성해주기 때문에 코드 실행에는 문제가 없다. 이러한 결론이다. 그리고 이렇게, 객체를 생성해주는 주체가 개발자에서 스프링 컨테이너로 변경된 것을 제어의 역전(IOC)이라 부른다.

결론 : 결합도가 낮은 코드 구현을 위해 개발자가 객체 생성 코드를 작성하지 않는다. 그 대신 스프링 컨테이너가 객체를 만들어준다. 이렇게 객체를 만드는 주체가 개발자에서 스프링 컨테이너로 제어권이 넘어간 것을 제어의 역전(IOC)라 한다.

프레임워크는 코드를 구성하는 틀(방향성)을 제공한다. 이러한 방향성을 지키지 않으면 프레임워크를 사용할 필요가 없다. 그러니 우리는 스프링이 지향하는대로 객체 생성 코드를 스프링에서는 최소화 할 것이다. (객체 생성 문법을 사용한다고 오류가 나는 것은 아님)

IOC(Inversion Of Control:제어의 역전)의 개념으로 어떻게 스프링 컨테이너가 객체를 만들어줄까?

스프링 컨테이너가 개발자 대신 객체를 만들어줌으로써, 스프링 프레임워크는 느슨한 결합도를 제공할 수 있게 되었다.

그럼 어떻게 스프링 컨테이너가 객체를 만들어줄 수 있을까. 스프링 컨테이너가 어떤 클래스에 대한 객체를 만들지는 알 수 없기 때문에 개발자가 만들어질 객체의 클래스는 지정해줘야 한다. 객체가 필요한 클래스를 개발자가 지정하면 스프링 컨테이너가 알아서 객체를 생성해준다. 그렇기 때문에 우리는 객체가 필요한 클래스를 지정하는 방법만 알면 된다. 그 방법은 다음과 같이 두 단계로 이루어진다.

1. **우선, 객체가 필요한 클래스는 반드시 default 패키지 안에 생성되어야 한다.**
   = main > java > 패키지명 안에 생성

   default 패키지란 프로젝트 생성 시 자동으로 만들어진 패키지를 말한다. 이 패키지 안에서 작성된 클래스만 스프링 컨테이너가 객체를 생성할 수 있는 후보 클래스가 된다. 즉, deault 패키지 밖에서 만들어진 클래스는 스프링 컨테이너가 자동으로 객체를 생성해주지 못한다.

2. **객체가 필요한 클래스 위에 객체 생성을 명령하는 어노테이션(annotation)을 추가한다.**

   객체 생성 기능을 가진 어노테이션을 클래스 위에 선언하면 해당 클래스의 객체가 자동으로 만들어진다.

   이때, 자동으로 생성되는 객체명을 지정하지 않으면 클래스명에서 첫글자만 소문자로 바꾼 이름으로 지정된다.

객체 생성 어노테이션(annotation)	설 명
@Component(“객체명“)	객체를 생성한다.
@Controller(“객체명“)	객체를 생성하고, controller 역할을 부여한다.
@RestController(“객체명“)	객체를 생성하고, Rest API 기능의 컨트롤러 역할을 부여한다.
@Service(“객체명“)	객체를 생성하고, 비지니스 로직이 포함된 클래스임을 알려준다.
@Configuration(“객체명“)	객체를 생성하고, 설정내용이 작성된 크래스임을 알려준다.
@Bean	객체를 생성한다. 해당 어노테이션은 메서드 선언문 위에 작성한다.

위 표는 대표적인 객체 생성 어노테이션을 나타낸 표이다. 표의 설명 부분에서 중요한 내용은 모두 객체를 생성하는 기능을 가진 어노테이션이라는 것이다. 해당 어노테이션이 클래스 선언 위에 작성되면(@Bean은 메서드 선언 위에) 클래스의 객체를 만들어준다. 만들어지는 객체명은 어노테이션의 () 안에 작성한 이름으로 지정되며, 만약 ()를 사용하지 않으면 클래스명에서 첫 앞글자만 소문자로 바꾼 이름으로 객체를 생성한다.

```
@Component
class Test{...}
```
-> 내부적으로 Test test = new Test(); 코드를 실행하여 객체 생성

```
@Component(“aaa”)
class Test{...}
```
-> 내부적으로 Test aaa = new Test(); 코드를 실행하여 객체 생성

지금까지 스프링 컨테이너가 우리 대신 객체를 생성하는 방법에 대해 알아보았다. main 메서드에 아래의 코드를 작성하여 정말 객체를 만들어주는지 확인해보자. (이 코드는 웹 개발에 사용되지 않습니다. 객체 생성 확인용 코드이니, 확인 후 그냥 넘어가면 됩니다.)

```
public static void main(String[] args) {
    ApplicationContext context = SpringApplication.run(Test1Application.class, args);

    //스프링 컨테이너가 생성한 모든 객체의 이름을 배열로 가져온다.
    //스프링에서는 객체를 bean이라 표현합니다.
    String[] beanNames = context.getBeanDefinitionNames();

    //만들어진 객체의 이름과 자료형을 차례대로 출력한다.
    for(String beanName : beanNames){
        Object beanType = context.getBean(beanName);
        System.out.println("객체명 : " + beanName + ", 자료형 : " + beanType);
    }
}
```

DI(Dependency Injection) : 의존성 주입

지금까지 IOC를 통해 스프링 컨테이너가 객체를 생성해주는 방법에 대해 알아보았다. 그럼 이렇게 생성된 객체를 우리가 어떻게 사용하면 될까?

IOC 개념에 의해 스프링 컨테이너가 객체를 생성해주면 우리는 만들어준 객체를 사용하면 된다. 이때, 생성된 객체를 우리가 사용하기 위해 가져오는 것을 의존성 주입(DI)이라 한다. 의존성을 주입하는 방법은 생성자 주입, 수정자(Setter) 주입, 필드 주입, 일반 메서드 주입 등 4가지 방법을 제공한다. 4가지 방법 모두 알면 좋으나, 우리는 생성자 주입 방식만을 사용할 예정이다. 스프링이 4가지의 방법 중 생성자 주입 사용을 권장하고 있기 때문이다.

생성자를 통한 의존성 주입(생성자 주입)

생성자 주입은 말 그대로 생성자를 통해 의존성을 주입하는 방법을 말한다. 생성자에 @Autowired 라는 어노테이션을 사용하면 스프링 컨테이너가 생성해놓은 객체 중 생성자에 필요한 빈(객체)을 주입해준다. 만약 생성자가 1개만 존재하면 @Autowired 어노테이션을 생략 할 수 있다. 다음 예제를 보자.

```
@Service
public class MemberService{...}

@Service("service")
public class ItemService{...}
```

위 두 코드로 인해 memberService, service
두 객체를 스프링 컨테이너가 생성함

```
public class Test{
  private ItemService itemService;

  //생성자 주입
  @Autowired
  public Test(ItemService itemService){
    this.itemService = itemService
  }
}
```

Lombok(롬북) 라이브러리

코드를 작성하다보면 자주 사용하는 코드들이 있다. 대표적으로 생성자, setter, getter, toString 등 이다.

이러한 코드들은 경험을 통해 익숙해지면 점점 작성하는 것 조차 귀찮아지게 된다. lombok 라이브러리는 이렇게 우리가 반복적으로 작성하던 코드를 간단한 어노테이션으로 작성할 수 있도록 도와주는 라이브러리이다. 또한, 룸복 라이브러리를 활용하면 생성자 주입을 위해 매번 작성하던 생성자도 훨씬 간편하게 사용할 수 있게 된다. 생성자 주입을 간단하게 사용하는 방법은 다음에 알아보고, 지금은 lombok 라이브러리가 어떠한 기능이 있는지 간단히 자주 사용하는 것들만 아래 표로 정리하였다. 앞으로는 lombok 라이브러리로 대체할 수 있는 코드는 lombok을 사용하겠다.

Lombok 어노테이션(annotation)	설 명
@Setter	setter를 자동으로 생성해준다.
@Getter	getter를 자동으로 생성해준다.
@ToString	toString 메서드를 자동으로 생성해준다.
@Data	setter, getter, toString, equals(), 생성자 등을 자동으로 생성해준다. 사용 권장 X
@NoArgsConstructor	매개변수가 없는 default 생성자를 생성해준다.
@AllArgsConstructor	모든 맴버변수를 매개변수로 갖는 생성자를 생성해준다.
@RequiredArgsConstructor *	final 키워드가 붙은 맴버변수만을 매개변수로 갖는 생성자를 생성해준다. 생성자 주입에 사용
@slf4j	로그를 사용할 수 있게 해 준다.

스프링은 IOC(제어의 역전)를 통해 스프링 컨테이너가 객체를 생성해주고, 개발자는 DI(의존성 주입)를 통해 객체를 사용하였다. 그리고 의존성 주입 방식 중 우리는 생성자 주입 방식을 사용하고 있다. 그렇기에 의존성 주입이 필요한 클래스에 매번 생성자를 작성하였고, 반복되는 생성자 구현은 그닥 반기운 일은 아니였을 것이다. 그래서 생성자 주입 방식을 보다 간결하게 사용할 수 있는 방법을 소개하려 한다. 이를 위해 먼저 생성자 주입에 대한 이해력을 좀 더 올려야 하며, lombok 라이브러리가 제공하는 @RequiredArgsConstructor 어노테이션에 대한 지식도 필요하다.

스프링이 생성자 주입을 권장하는 이유는 뭘까?

의존성 주입은 생성자 주입, 수정자(Setter) 주입, 필드 주입, 일반 메서드 주입 등 4가지 방법을 사용하며, 이 중 생성자 주입을 스프링이 권장한다고 하였다. 왜 일까? 여러 이유가 있지만 '객체의 불변성 확보'에 가장 적합한 방법이기 때문이다. 객체의 불변성이란, 한 번 객체가 할당되면 그 객체는 변하지 않는 것이 좋다는 것이다.(이제는 알겠지만 객체가 변하지 않는다는 것은 객체가 가진 맴버변수의 값이 변하지 않는다는 것이 아니다. 주소값이 달라지는 것을 의미한다). 실제로 개발을 하다보면 객체가 변경되는 일은 거의 없다. 그렇기에 의존성 주입을 통해 사용하는 객체는 그 값을 변경할 수 없도록 애초에 막아주는 것이 좋다. 하지만 수정자 주입, 필드 주입, 메서드 주입 등은 의존성 주입을 통해 가져온 객체가 변경될 수 있는 가능성이 존재한다. 반대로, 생성자 주입은 객체의 변경 가능성을 배제시키도록 코드를 작성할 수 있기 때문에 스프링에서 권장하고 있는 것이다.

그렇다면 생성자 주입 사용 시 코드를 어떻게 작성하면 객체의 변경 가능성을 배제시킬 수 있을까?

맴버변수 선언 시 final 키워드의 사용이 해답이다. 변수 선언문에 final 키워드를 붙이면 상수(변하지 않는 값)가 된다.

자바스크립트와 비교하자면 final 키워드가 붙은 변수는 const 로 선언된 변수가 되는 것이라 생각하면 된다.

```
public static void main(String[] args){
    int a = 10;
    a = 20;                 //변수는 값 변경 가능

    final int b = 20;
    b = 30;                 //오류! final 키워드가 붙은 변수는 값 변경 불가

    final Member m = new Member();
    m.setName("kim");           //가능! 이 코드는 객체의 값을 변경시키는 코드가 아님!
    m = new Member();           //불가능! 이 코드는 객체의 값을 변경시키는 코드라 오류 발생!
}
```

다음은 생성자 주입 시 final 키워드를 사용하지 않은, 지금까지의 코드이다.

```
@Service
public class MemberService{...}

@Service("service")
public class ItemService{...}
```

위 두 코드로 인해 memberService, service
두 객체를 스프링 컨테이너가 생성함

```
public class Test{
  private ItemService itemService;

  //생성자 주입
  @Autowired
  public Test(ItemService itemService){
    this.itemService = itemService
  }
}
```

다음은 생성자 주입 시 final 키워드를 사용한 변경된 코드이다.

```
@Service
public class MemberService{...}

@Service("service")
public class ItemService{...}
```

위 두 코드로 인해 memberService, service
두 객체를 스프링 컨테이너가 생성함

```
public class Test{
  private final ItemService itemService;

  //생성자가 1개만 존재하면 @Autowired 생략해도 의존성 주입 됨
  public Test(ItemService itemService){
    this.itemService = itemService
  }
}
```

생성자 주입에 대한 좀 더 알아본 결과, 우리는 의존성 주입이 필요한 객체 선언 시 이제부터는 final 키워드를 붙이는 것이 좋다는 것을 알게 되었다.

하지만, 여기까지만 놓고 보면 기존의 우리가 사용하던 생성자 주입보다 간결해지기는 커녕 오히려 코드가 늘어난 느낌이다.

마지막으로 lombok 라이브러리에서 제공하는 @RequiredArgsConstructor 어노테이션만 이해하면 생성자 주입을 짧게 사용할 수 있다.

@RequiredArgsConstructor

required는 '요구되는', args는 arguments의 줄임말로 논쟁, 주제 등을 의미하며 프로그래밍에서 args는 매개변수를 뜻한다.

마지막으로 constructor는 생성자이기에 직역하면 '요구되는 매개변수를 가진 생성자' 정도가 되겠다. 그럼 요구되는 매개변수란 무엇일까?

'요구되는 매개변수'란 반드시 값이 필요한 변수에 값을 할당할 목적으로 전달되는 매개변수를 의미한다. 그리고 '반드시 값이 필요한 변수'는 final 키워드가 붙은 변수를 말한다. final 키워드가 붙은 변수는 한 번 값이 결정되면 그 값을 변경할 수 없기에, 최초 한 번은 반드시 값이 할당되어야 하기 때문이다.

위의 내용을 종합해보면 @RequiredArgsConstructor 어노테이션은 'final 키워드가 붙은 맴버변수의 값을 매개변수로 갖는 생성자'를 생성해준다는 의미인 것이다.

그럼 final 키워드가 붙은 맴버변수와 @RequiredArgsConstructor 어노테이션을 사용한 생성자 주입을 코드로 보겠다.

먼저 기존의 생성자 주입 코드이다.

```
@Service
public class MemberService{...}

@Service("service")
public class ItemService{...}
```

위 두 코드로 인해 memberService, service 두 객체를 스프링 컨테이너가 생성함

```
public class Test{
  private ItemService itemService;

  @Autowired
  public Test(ItemService itemService){
    this.itemService = itemService
  }
}
```

우측의 코드는 변경된 생성자 주입 코드이다.

@RequiredArgsConstructor 어노테이션으로 final 키워드가 붙은 ItemService 객체를 매개변수로 갖는 생성자로 자동으로 생성한다. 생성자가 1개만 존재할 때는, @Autowired 어노테이션도 생략 가능하기에 결국, 클래스 안에는 코드 한 줄 적지않아도 자동으로 생성자 주입을 받는다.

```
@RequiredArgsConstructor
public class Test{
  private final ItemService itemService;
}
```