## 토비의 스프링 3.1 스터디 후 정리한 내용으로 토비의 스프링 책이 바탕으로 되어있다.

제어의 역전(IoC)과 싱글톤 레지스트리

제어의 역전 IoC Inversion of Control (p.92~)

제어의 역전이란?

프로그램의 제어 프름구조가 뒤바뀌는 것

제어의 흐름의 개념을 거꾸로 뒤집는 것

오브젝트가 자신이 사용할 오브젝트를 스스로 선택하지 않는다. 당연히 생성하지도 않는다.

또 자신도 어떻게 만들어지고 어디서 사용되는지를 알 수 없다. 모든 제어 권한을 자신이 아닌 다른 대상에게 위임하기 때문이다.

라이브러리 VS 프레임워크

\* 라이브러리 : 라이브러리를 사용하는 애플리케이션 코드는 애플리케이션 흐름을 직접 제어한다.

\* 프레임워크 : 거꾸로 어플리케이션 코드가 프레임워크에 의해 사용된다. 보통 프레임워크 위에 개발한 클래스를 등록해 두고, 프레임워크가 흐름을 주도하는 중에 개발자가 만든 애플리케이션 코드를 사용하도록 만드는 방식이다.

프레임워크에는 분명한 제어의 역전 개념이 적용되어 있어야 한다.

스프링의 IoC

빈 : 스프링이 제어권을 가지고 직접 만들고 관계를 부여하는 오브젝트

스프링 빈 : 스프링 컨테이너가 생성, 관계설정, 사용등을 제어해주는 제어의 역전이 적용된 오브젝트

빈팩토리 / 애플리케이션 컨텍스트 : 빈의 생성과 관계 설정 같은 제어를 담당하는 IoC 오브젝트

애플리케이션 컨텍스트 동작방식

+) DaoFactory(p.88)를 오브젝트로 직접 사용했을 때 vs 애플리케이션 컨텍스트를 사용했을 때

- 클라이언트는 구체적인 팩토리 클래스를 알 필요가 없다.

- 애플리케이션 컨텍스트는 종합 IoC 서비스를 제공한다.

싱글톤 레지스트리와 오브젝트 스코프 (p.102)

오브젝트 Factory vs 애플리케이션 컨텍스트 (가장 큰 차이점)

오브젝트 Factory : new DaoFactory / 2개의 객체 생성시 2개는 다른 객체

애플리케이션 컨텍스트 : getBean("userDao", UserDao.class); / 2개의 객체 생성시 2개는 같은 객체 => 스프링은 여러번에 걸쳐 빈을 요청하더라도 매번 동일한 오브젝트를 돌려준다.

싱글톤 레지스트리로서의 애플리케이션 컨텍스트

애플리케이션 컨텍스트는 IoC 컨테이너이면서 싱글톤을 저장하고 관리하는 싱글톤 레지스트리이다.

(별 다른 설정을 하지 않으면 내부에서 생성하는 빈 오브젝트를 모두 싱글톤으로 만든다. but 디자인 패턴의 싱글톤과 다르다!)

?? why? 스프링은 왜 싱글톤으로 빈을 만들까?

=> 스프링이 주로 적용되는 대상이 자바 엔터프라이즈 기술을 사용하는 서버환경이기 때문이다.

+) 자바 엔터프라이즈 환경 : 대량 네트워크 환경, 분산 환경, 웹 환경

=> 대량의 request를 받아 처리할 수 있는 높은 성능이 요구되는 환경.

각각의 객체를 생성하게 되면 부하가 높아진다.

(서블릿 또한 대부분 멀티스레드 환경에서 싱글톤으로 동작한다.)

\* 디자인 패턴에서의 싱글톤 패턴 단점

pricate 생성자 -> 상속, 다형성 사용할 수 없음

테스트의 어려움

전역변수이기 때문에 접근, 수정 시 의도치 않은 구현 발생 가능

싱글톤 레지스트리

스프링은 서버환경에서 싱글톤이 만들어져서 서비스오브젝트 방식으로 사용되는 것을 적극 지지한다.

하지만 자바의 기본적인 싱글톤 패턴의 구현 방식은 여러가지 단점이 있기 때문에, 스프링은 직접 싱글톤 형태의 오브젝트를 만들고 관리하는 기능을 제공한다.

그것이 바로 싱글톤 레지스트리다.

스태틱 메소드와 private 생성자를 사용해야 하는 비정상적인 클래스가 아니라 평범한 자바 클래스라도 IoC 방식의 컨테이터를 사용해서 생성, 관계설정, 사용등에 대한 제어권을 컨테이너에게 넘기면 손쉽게 싱글톤 방식으로 만들어져 관리되게 할 수 있다.

스프링의 싱글톤 레지스트리 덕분에 싱글톤 방식으로 사용될 애플리케이션 클래스라도 public 생성자를 가질 수 있다.

=> 스프링은 IoC 컨테이너일 뿐만 아니라, 고전적인 싱글톤 패턴을 대신해서 싱글톤을 만들고 관리해주는 싱글톤 레지스트리이다.

싱글톤과 오브젝트의 상태

싱글톤은 멀티스레드라면 여러스레드가 동시에 접근이 가능하다.

기본적으로 싱글톤이 멀티스레드 환경에서 서비스 형태의 오브젝트로 사용되는 경우에는 상태정보를 내부에 갖고있지 않은 무상태 방식으로 만들어져야 한다.

다중 사용자의 요청을 한꺼번에 처리하는 스레드들이 동시에 싱글톤 오브젝트의 인스턴스 변수를 수정하는 것은 매우 위험하다. (저장공간이 하나뿐이니 덮어 쓸 수 있음)

따라서 싱글톤은 기본적으로 인스턴스 필드의 값을 변경하고 유지하는 상태유지 방식으로 만들지 않는다.

정보(변수)들은 파라미터 변수, 로컬 변수, 리턴 값 등을 이용하면 된다.

p.110 코드 참조

Connection c;

User user;

=> 로컬 변수로 저장. new 할 때 마다 매번 새로운 값으로 바뀔 수 있음. 개별적으로 사용하거나 파라미터로 받아서 사용 (p.76)

그렇다면 connectionMarker는? 인스턴스 변수로 저장되어 있다! 사용이 가능한가?

=> 그렇다. connectionMarker는 읽기 전용 정보이기 때문이다.

@Bean을 만들어 생성. 스프링이 관리하는 Bean! 싱글톤으로 하나의 객체만 생성

이렇게 자신이 사용하는 다른 싱글톤 빈을 저장하려는 용도라면 인스턴스 변수를 사용해도 좋다.

출처: <http://limmmee.tistory.com/28?category=654011> [심플하게 개발]

스프링 프레임워크 (Spring Framework) 1 - POJO에 대하여

프레임워크란?

소프트웨어 프레임워크 : 복잡한 문제를 해결하거나 서술하는 데 사용되는 기본 개념 구조 (by 위키피디아)

개발자에게 프레임워크란 맨땅의 헤딩으로 개발하지 않을 수 있도록 제공된 큰 틀이라고 볼 수 있다. 다수의 개발자가 함께 개발하는 환경이 되면서 소프트웨어는 점점 거대해졌고 이 소프트웨어를 구현하는 코드를 작성하는 방식은 제마다 방식이 다르다. 그런 다양한 방식 속에서 일정한 형태를 가질 수 있도록 유고하는 것이 프레임워크가 가지고 있는 힘(or 강제력)이다.

프레임워크를 기반으로 한 개발은 일정한 개발형식을 띄게 된다.

애플리케이션 프레임워크는 일관된 개발을 지원하고, 프레임워크에 정의된 경계에 따라 설정을 강제하고 재사용을 가능하게 하고 확장가능하며 단순한 형태를 유지하면서 유지관리성의 이익을 얻을 수 있는 장점이 있다.

스프링 프레임워크는?

스프링 프레임워크에 들어가기 앞서, POJO 를 먼저 알아야한다.

POJO (Plain Old Java Object)

POJO (Plain Old Java Object) 란 번역하면 '평범한 구식 자바 객체'이다. 도대체 평범하고 구식인 자바 오프젝트가 뭐가 다르고 특별해서 POJO라고 부르는 것일까? 그럼 평범하지 않은 최신의 자바 오브젝트는 무엇인가?

POJO를 이해하려면 POJO라는 단어가 만들어진 역사적 배경을 살펴볼 필요가 있다. POJO는 마틴 파울러가 2000년 가을에 열렸던 어느 컨퍼런스의 발표를 준비하면서 처음 만들어낸 말이다. 마틴 파울러는 EJB(Enterprise JavaBean)보다는 단순한 자바 오브젝트에 도메인 로직을 넣어 사용하는 것이 여러가지 장점이 있는데 왜 사람들이 EJB가 아닌 '평범한 자바 오브젝트'를 사용하기를 꺼려하는지에 대해 의문을 가졌다. 그리고 그는 단순한 오브젝트에는 EJB와 같은 그럴듯한 이름이 없어서 그 사용을 주저하는 것이라고 결론을 내렸고, POJO라는 용어를 만들었다.

EJB와 엔터프라이즈 서비스

자바에서 EJB 기술의 등장은 필연적인 것이었다. 기업의 IT 시스템은 점점 그 중요성이 증대되고 그에 따라 점점 기술이 요구되었으며 자바의 기초적인 JDK만으로는 그것을 충족시킬 수 없었다. 서버 기반의 자바 기술인 J2EE(Java2 Enterprise Edition)가 등장했지만 Servlet, JSP 레벨의 최소한의 서버 프로그래밍 인터페이스만 가지고는 복잡한 엔터프라이즈 애플리케이션을 제작하는데 부담이 적지 않았다.

기업업무처리의 IT시스템에 대한 의존도가 높아지면서 시스템이 다뤄야 하는 비즈니스 로직 자체가 점차 복잡해졌다. 애플리케이션 로직의 복잡도와 상세 기술의 복잡함을 개발자들이 한 번에 다룬다는 것은 쉬운일이 아니었다. 예를 들어, 한 개발자가 보험업무와 관련된 계산 로직을 자바로 어떻게 구현해야 하는지에 집중하면서 동시에 시스템 레벨에서 멀티 DB로 확장 가능한 트랙잭션 처리와 보안 기능을 멀티스레드에 세이프하게 만드는 것에 신경써야한다면 여간 부담되는게 아닐 것이다.

많은 사용자의 처리요구를 빠르게 안정적이면서 확장 가능한 형태로 유지하기위해 필요한 로우레벨의 기술적인(트랜젝션 처리, 상태관리, 멀티쓰레딩, 리소스풀링, 보안등) 처리가 요구되었다.

EJB는 이런 문제를 다루기 위해 등장했다. EJB 1.0의 스펙이 제시한 EJB의 비전은 'EJB는 애플리케이션 개발을 쉽게 만들어준다. 애플리케이션 개발자는 로우레벨의 기술들에 관심을 가질 필요도 없다.' 였다. 애플리케이션 개발자들은 다뤄야하는 해당 도메인과 비즈니스 로직에만 집중하면 된다는 것이다. 게다가 EJB는 독립적으로 개발한 컴포넌트들을 서버에 자유롭게 배포하고 서로 연동해 사용하게하는 컴포넌트 기반의 개발 모델을 제시할 뿐더러, 여러 대의 서버에 분산되어있는 모듈간의 리모팅 처리도 개발자들이 거의 신경쓰지 않고 개발 할 수 있게 했다. 더 나아가 벤더별로 제각각 발전시켜 혼란에 빠지기 쉬운 자바의 서버 기술을 일관성있게 구현할 수 있도록 지원하므로 특정 서버에 종속되지 않고 서버간의 이동성을 보장해준다고 약속했다.

그러나, EJB는 불필요할만큼 과도한 엔지니어링으로 실패한 대표적인 케이스였다.

EJB에서는 현실에서 1% 미만의 애플리케이션에서만 필요한 멀티 DB를 분산 트랜잭션을 위해 나머지 99%의 애플리케이션에서도 무거운 JTA기반의 글로벌 트랙잰션 관리 기능을 사용해야 했다. EJB의 혜택을 얻기 위해 모든 기능이 다 필요하지도 않은 고가의 WAS를 구입해야했고, 고급 IDE(Intergrated Development Environment)의 도움없이는 손쉽게 다룰 수 없는 복잡한 설정 파일 속에서 허우적대야 했다. EJB 컴포넌트는 컨테이너 밖에서는 정상적으로 동작할 수 없으므로 개발자들은 끝도없이 반복되는 수정-빌드-배포-테스트의 지루한 과정으로 많은 시간을 낭비해야했다. 가장 최악의 문제점은 EJB 스펙을 따르는 비즈니스 오브젝트들은 객체지향적인 특징과 장점을 포기해야했다는 것이다. EJB 빈은 상속과 다형성등의 혜택을 제대로 누릴 수 없었다.

그럼에도 EJB가 계속 사용되었던 이유는, 엔터프라이즈 애플리케이션에서 반드시 필요로하는 주요한 엔터프라이즈 서비스들을 애플리케이션 코드와 분리해서 독립적인 서비스로 사용할 수 있게 만들어줬다는 점이다. 비록 불완전하고 불필요한 복잠도가 남아있긴 했지만 선언적인 트랜잭션 관리나 역할 기반의 보안 기능들을 제공했다. 한편으로는 '개발자들이 로우레벨의 기술적인 문제에 신경쓰지 않고 비즈니스 로직에 충실히 개발하게 함으로써 애플리케이션 개발을 손쉽게 만들어 준다'는 처음 약속을 어느 정도 지켰다고 볼 수 있었다. 하지만 EJB 문제는 앞서 지적한 것처럼 한편으로는 애플리케이션 개발의 복잡도를 제거하면서 다른 한편으로는 더 많은 문제와 복잡성을 가지고 왔다는 것이다.

결국 EJB는 형편없는 생산성과 느린 성능, 불필요한 기술적인 복잡도등으로 자반의 엔터프라이즈 개발에 대한 불신을 가중시켰다. 마틴 파울러는 EJB와 같은 잘못 설계된 과도한 기술을 피하고, 객체지향 원리에 따라 만들어진 자바 언어의 기본에 충실하게 비즈니스 로직을 구현하는 일명 POJO 방식으로 돌아서야 한다고 지적했다. POJO 방식의 개발은 EJB가 잃어버린 소중한 가치인 객체지향적인 설계와 자동화된 테스트의 편의성, 개발생산성 등을 회복시켜 줄 수 있는 길이기 때문이다.

결국 POJO를 정리하자면,

 특정 규약(contract)에 종속되지 않는다. (Java 언어와 꼭 필요한 API 외에 종속되지 않는다.)

 특정 환경에 종속되지 않는다.

 객체지향원리에 충실해야 한다.

POJO를 사용하는 이유

코드의 간결함 (비즈니스 로직과 특정 환경/low 레벨 종속적인 코드를 분리하므로 단순하다.)

 자동화 테스트에 유리 (환경 종속적인 코드는 자동화 테스트가 어렵지만, POJO는 테스트가 매우 유연하다.

 객체지향적 설계의 자유로운 사용

앞에서 강조한것처럼 EJB를 사용하지 말고 POJO를 쓰자는 것이 EJB 이전의 방식으로 돌아가는 것을 의미한다면 또 다른 문제가 발생할 수밖에 없다. 여전히 복잡한 로우레벨의 API를 이용해 코드를 작성해야하고, 많은 기술적인 문제를 애플리케이션 코드에 그대로 노출시켜 개발해야 한다면 기껏 POJO로의 복귀 덕분에 얻은 많은 장점을 놓칠 수 밖에 없다. 그래서 등장한 것이 바로 POJO 기반의 프레임워크이다.

POJO 프레임워크

POJO를 이용한 애플리케이션 개발이 가진 특징과 장점을 그대로 살리면서 EJB에서 제공하는 엔터프라이즈 서비스와 기술을 그대로 사용할 수 있도록 도와주는 프레임워크, 나아가 기존의 EJB에서보다 훨씬 더 세련되고 나은 방법을 제공한다.

(많은 POJO 프레임워크가 있지만 그 중에서 가장 대표적인 것을 꼽으라면 하이버네이트와 스프링을 들 수 있다.)

스프링 프레임워크

스프링 : POJO 프레임워크 중 하나이며, 자바 애플리케이션 개발을 위한 포괄적인 인트라 스트럭처를 제공하는 자바 플랫폼이다.

스프링을 사용하면 POJO로 어플리케이션을 만들고 엔터프라이즈 서비스를 비침투적으로 POJO에 적용할 수 있다.

스프링 플랫폼의 이점에 대한 예제

 트랜잭션 API를 사용하지 않고도 데이터베이스 트랜잭션에서 자바메소드를 실행하도록 만든다.

 원격 API를 사용하지 않고도 로컬 자바메소드를 원격 프로시저로 만든다

 JMS API를 사용하지 않고도 로컬 자바메소드를 메시지 핸들러로 만든다

스프링 프레임워크 2 - 컨테이너와 빈의 생명주기

자 그럼 이제 본격적으로 스프링 프레임워크에 대해서 얘기를 해 볼까?

어마어마한 두께를 자랑하며 한 장만 봐도 눈물을 흘린다는 스프링 입문서. 토비의 스프링에서는 이렇게 말한다.

스프링은 거대한 컨테이너임과 동시에 IoC/DI를 기반으로 하고 있는 거룩한 존재이며 서비스 추상화를 통해 삼위일체로 분리되는 3단 변신로봇이다. (응? 한글인데 무슨말인지 1도 모르겠다.)

음. 대충 보니, 컨테이너, IoC/DI 가 키워드인것 같군.

컨테이너란?

Spring은 하나의 프레임워크이다. 그런데 왜 Spring 컨테이너, IoC 컨테이너라는 말을 사용할까? 그렇다면 컨테이너의 정의는 무엇인가?

Servlet 컨테이너, EJB 컨테이너라고 말하는 것을 종종 들어봤을 것이다. 컨테이너는 보통 인스턴스의 생명주기를 관리하며, 생성된 인스턴스들에게 추가적인 기능을 제공하도록하는 것이라 할 수 있다.

다시말해, 컨테이너란 당신이 작성한 코드의 처리과정을 위임받은 독립적인 존재라고 생각하면 된다. 컨테이너는 적절한 설정만 되어있다면 누구의 도움없이도 프로그래머가 작성한 코드를 스스로 참조한 뒤 알아서 객체의 생성과 소멸을 컨트롤해준다.

Servlet 컨테이너는 Servlet의 생성, 생성 후 초기화, 서비스 실행, 소멸에 관한 모든 권한을 가지고 있다. 개발자들이 직접 Servlet을 생성하고 서비스하지는 않는다. 이처럼 Servlet 인스턴스에 대한 생명주기를 관리하는 기능을 가진다. 또한, Servlet 컨테이너의 web.xml을 보면 JSP/Servlet 접근 권한에 대한 추가적인 서비스도 지원하고 있다. 이는 Servlet의 구현과는 별도로 각 JSP/Servlet에 대한 Security를 관리해주는 기능을 한다.

스프링 컨테이너는 스프링 프레임워크의 핵심부에 위치하며, 종속객체 주입을 이용하여 애플리케이션을 구성하는 컴포넌트들을 관리한다.

다시 말하면, 프로그래머가 작성한 코드는 컨테이너를 사용하게 됨으로서 프로그래머의 손을 떠나 컨테이너의 영역으로 떠나버리게 된다. (정확히 말하자면 컨테이너가 맘대로 객체를 생성하는 게 아니라 프로그램을 이용하는 이용자의 호출에 의해 컨테이너가 동작하게 되는 구조이다.)

스프링 컨테이너의 두 종류

1. 빈팩토리 BeanFactory (org.springframework.beans.factory.BeanFactory)

DI의 기본사항을 제공하는 가장 단순한 컨테이너

팩토리 디자인 패턴을 구현한 것. Bean(이하 빈) 팩토리는 빈을 생성하고 분배하는 책임을 지는 클래스

빈 팩토리가 빈의 정의는 즉시 로딩하는 반면, 빈 자체가 필요하게 되기 전까지는 인스턴스화를 하지 않는다 (lazy loading, 게으른 호출)

|  |
| --- |
| BeanFactory factory = new XmlBeanFactory(new FileInputStream("bean.xml"));  MyBean myBean = (Mybean) factory.getBean("myBean"); |

getBean()이 호출되면, 팩토리는 의존성 주입을 이용해 빈을 인스턴스화하고 빈의 특성을 설정하기 시작. 여기서 빈의 일생이 시작된다.

2. 어플리케이션 컨텍스트 ApplicationContext (org.springframework.context.factory.BeanFactory)

빈팩토리와 유사한 기능을 제공하지만 좀 더 많은 기능을 제공하는 어플리케이션 컨텍스트

빈팩토리보다 더 추가적으로 제공하는 기능

 국제화가 지원되는 텍스트 메시지를 관리해 준다.

 이미지같은 파일 자원을 로드 할 수 있는 포괄적인 방법을 제공해준다.

 리너스로 등록된 빈에게 이벤트 발생을 알려준다.

따라서 대부분의 애플리케이션에서는 빈팩토리보다는 어플리케이션 컨텍스트를 사용하는 것이 좋다.

가장 많이 사용되는 어플리케이션 컨텍스트 구현체

 ClassPathXmlApplicationContext : 클래스패스에 위치한 xml 파일에서 컨텐스트 정의 내용을 읽어들인다.

 FileSystemxmlApplicationContext : 파일 경로로 지정된 xml 파일에서 컨텐스트 정의 내용을 읽어들인다.

 XmlWebApplicationContext : 웹 어플리케이션에 포함된 xml 파일에서 컨텐스트 정의 내용을 읽어들인다.

|  |
| --- |
| ApplicationContext context = new ClassPathXmlApplicationContext("conf/bean.xml");  MyBean bean = context.getBean("myBean"); |

빈 팩토리와 애플리케이션컨텍스트의 기능상의 차이점 말고 또 다른 차이점은 다음과 같다.

빈 팩토리 : 처음으로 getBean()이 호출된 시점에서야 해당 빈을 생성(lazy loading)

애플리케이션 컨텍스트 : 컨텍스트 초기화 시점에 모든 싱글톤 빈을 미리 로드한 후 애플리케이션 기동 후에는 빈을 지연 없이 얻을 수 있음(미리 빈을 생성해 놓아 빈이 필요할 때 즉시 사용할 수 있도록 보장)

IoC (Inversion of Control, 제어의 역전)

IoC란 Inversion of Control의 약어이다. inversion은 사전적 의미로는 '도치, 역전'이다. 보통 IoC를 제어의 역전, 의존적 주입이라 번역한다.

역시, 한글인데 무슨말인지 이해가 안간다.

IoC 란 제어의 역전 즉 외부에서 제어를 한다는 것이다. 음 감이 딱 온다면, 당신은 천재!

맞다. IoC는 바로 컨테이너이다.

위에서 컨테이너를 설명하면서 말했던 것 처럼 기존에 자바 기반으로 어플리케이션을 개발할 때 자바 객체를 생성하고 서로간의 의존관계를 연결시키는 작업에 대한 제어권은 보통 개발되는 어플리케이션에 있었다. 그러나 Servlet, EJB등을 사용하는 경우 Servlet Container, EJB Container에게 제어권이 넘어가서 객체의 생명주기(라이프사이클)을 Container들이 전담하게 된다. 이처럼 IoC에서 이야기하는 제어권의 역전이란 객체의 생성에서부터 생명주기의 관리까지 모든 객체에 대한 제어권이 바뀌었다는 것(IoC 컨테이너)을 의미 한다.

IoC와 DI의 관계

스프링 프레임워크의 가장 큰 장점으로 IoC 컨테이너 기능이 부각되어 있으나, IoC 기능은 스프링 프레임워크가 탄생하기 훨씬 이전부터 사용되던 개념이었다. 그러므로 "IoC 기능을 스프링 프레임워크의 장점이라고 이야기하는 것은 적합하지 않다"고 반론을 제기하면서 "새로운 개념을 사용하는 것이 적합하다"고 주장한 사람이 마틴 파울러이다. 경량 컨테이너들이 이야기하는 IoC를 DI라는 용어로 사용하는 것이 더 적합하다고 이야기하고 있다. 마틴 파울러의 이 같은 구분 이후 IoC 개념을 개발자들마다 다양한 방식으로 분류하고 있으나 다음 그림과 같이 IoC와 DI 간의 관계를 분류하는 것이 일반적이다.

+) 자바빈 VS 스프링 빈

자바 빈 : 데이터를 표현하는 것을 목적으로 하는 자바클래스이다. 컴포넌트와 비슷한 의미로 사용되기도 한다.

클래스에는 값을 저장하는 속성필드, get,set 메소드, 기본 생성자등을 포함하고 있다.

+) java.io.Serializable 인터페이스는 생략 가능하나 빈즈 규약에 명시된 내용으로 자바 빈즈에 저장된 프로퍼티를 포함한 채로 파일 시스템에 저장되거나 네트워크로 전송될 수 있도록 객체 직렬화를 제공해야 하므로 implements 한다.

스프링 빈 : 스프링이 제어권을 가지고 직접 만들고 관계를 부여하는 오브젝트

#참고 URL

http://blog.daum.net/\_blog/BlogTypeView.do?blogid=09mzN&articleno=135

http://egloos.zum.com/springmvc/v/487497

http://flyburi.com/277

http://ooz.co.kr/178

http://www.javajigi.net/pages/viewpage.action?pageId=535

http://www.javajigi.net/pages/viewpage.action?pageId=281

http://dev.anyframejava.org/docs/anyframe/4.1.0/reference/html/ch06.htm

출처: <http://limmmee.tistory.com/13?category=654011> [심플하게 개발]

스프링 프레임워크 3 - DI (Dependency Injection, 의존성 주입)

스프링 프레임워크에서 DI는 아래 URL에 정말 자세하게 잘 나와있다.

아래 URL부터 10개의 강의를 보면 DI에 대해서 어느정도 충분한 이해가 가능할 것이다.

http://expert0226.tistory.com/189

(이 포스팅은 아래 강의를 개인적인 목적으로 정리한 것이기 때문에 집약적이다. 가능하면 아래 URL의 강의를 보기 바란다.)

DI . Dependcy Injection 의존성 주입. DI를 들어가기 전에, 의존성이라는 말부터 한번 보자

|  |
| --- |
| \* 의사 코드  운전자가 자동차를 생산한다.  자동차는 내부적으로 타이어를 생산한다.  \* Java 표현  new Cat();  Car 객체 생성자에서 new Tire();  \* 정말 간단히 의존성이란?  의존성은 new 이다.  new를 실행하는 Car와 Tire 사이에서 Car가 Tire에 의존한다라고 한다. |

|  |
| --- |
| \* 의사 코드  운전자가 타이어를 생산한다.  운전자가 자동차를 생산하며 타이어를 장착한다.  \* Java 표현 (생성자의 인자 이용)  Tire tire = new KoreanTire();  Car car = new Car(tire);  \* 주입이란?  주입이란 말은 외부에서라는 뜻을 내포하고 있는 단어이다.  결국 자동차 내부에서 타이어를 생산하는 것이 아니라, 외부에서 생산된 타이어를 자동차에 장착하는 작업이 주입이다. |

\* 소스 코드는 해당 강의 참조 (스프링 없이 생성자, 속성을 이용한 의존성 주입)

기존 코드에서는 Car가 구체적으로 KoreaTire를 생산할지 AmericaTire를 생산할지를 결정했었다 (new를 사용해서) 그러한 코드를 유연성이 떨어진다고 한다.

현실 세계를 비유로 하자면 자동차가 자신이 생산될 때 스스로 어떤 타이어를 생산/장착할까 고민하지 않고 운전자가 차량을 생산할 때, 운전자가 어떤 타이어를 장착할까를 고민하게 하는것이다. 자동차는 더이상 어떤 타이어를 장착할까 고민하지 않아도 된다.

자. 그럼 여기서 결론. 과연 이런 방식의 코드 작성은 어떤 점에서 이점이 있을까?

기존 방식에서라면 Car는 KoreaTire, AmericaTire에 대해 정확히 알고 있어야만 그에 해당하는 객체를 생성할 수 있다. 그러나 이렇게 의존성 주입을 통하게 되면 Car는 그저 Tire 인터페이스를 구현한 어떤 객체가 들어오기만 하면 정상 작동하게 된다. 그렇게 되면 확장성이 좋아지게 된다고 하는데 후에 ChinaTire, JapenTire 등등 어떤 새로운 타이어 브랜드가 생겨도 각 타이어 브랜드들이 Tire 인터페이스를 구현하기만 했다면 Car.java 소스의 변경없이 (컴파일 없이) 사용할 수 있다는 것이다.

이것은 인터페이스를 구현(준수)했기에 얻는 이점이라고 보면 된다. 현실 세계에서는 인터페이스라는 말보다 표준화했다는 말이 더욱 와닿을 것이다. 대표적인 표준화 사례는 PET 병마개를 생각하면 된다. 여러 회사의 음료수 PET병이 있다면 서로 뚜껑을 바꾸어서 닫아 보아라. 잘 맞을 것이다. 바로 표준화 된 규격에 맞추어 생산했기 때문이다.

현실세계 표준 규격 준수 = 프로그래밍 세계의 인터페이스 구현

스프링을 통한 의존성 주입 - XML 사용

|  |
| --- |
| \* 의사 코드  운전자가 종합 쇼핑몰에서 타이어를 구매한다.  운전자가 종합 쇼핑몰에서 자동차를 구매한다.  운전자가 자동차에 타이어를 장착한다.  \* Java 표현  AplicationContext context = new FilesystemXmlApplication("/src/main/java/test.xml" );  Tire tire = (Tire) context.getBean("tire");  Cat car = (Car) context.getBean("car");  car.setTire(tire); |

종합쇼핑몰을 SpringFramework라고 생각해보자.

자바 표현에서 두,세째 줄은 종합쇼핑몰(Spring)에서 Car와 Tire를 구매하는 코드이다. 그럼 입점된 상품에 대한 내역은 어디에 숨어있을까? 바로 위쪽 xml 파일에 정의되어있다.

|  |
| --- |
| <bean id="tire" class="expert002.KoreaTire"></bean>  <bean id="americaTire" class="expert002.AmericaTire"></bean>  <bean id="car" class="expert002.Car"></bean> |

해당 xml에서는 Car와 Tire를 각각 위와 같이 정의하였다. 상품을 등록할때는 bean이라는 태그 안에 등록하고 각각의 상품을 구분하기 위한 id 속성과 그 상품을 어떤 클래 통해 생산해야하는지를 나타내는 class 속성을 사용한다.

자, 그럼 이렇게 스프링(종합쇼핑몰)을 도입함으로 말미암아 얻게 된 잇점은 무엇인가?

자동차의 타이어 브랜드를 바꾸고자 할 때 이제는 그 무엇도 재컴파일 없이 xml 파일만 수정하면 실행 결과를 바꾸어 줄 수 있다.

스프링을 통한 의존성 주입 - XML에서 속성 주입

그런데, Car를 생성할때 기본 defualt로 국산 타이어를 장착하고 싶다면? xml에서 default 속성 셋팅이 가능하다.

|  |
| --- |
| <bean id="koreaTire" class="expert003.KoreaTire"></bean>  <bean id="americaTire" class="expert003.AmericaTire"></bean>  <bean id="car" class="expert003.Car">  <property name="tire" ref="koreaTire"></property>  </bean> |

결국 car.setTire(tire) 하던 부분을 xml 파일의 property 태그를 이용해 대체해 버리는 것이다.

|  |
| --- |
| ApplicationContext context =  new FileSystemXmlApplicationContext("/src/main/java/test.xml");  Car car = (Car)context.getBean("car"); |

기존 코드와 비교했을때, Tire 객체를 만드는 것과, setTire한 부분이 사라졌다.

기본적으로 koreaTire를 셋팅해 준다는 것이다.

스프링을 통한 의존성 주입 - @Autowired를 통한 속성 주입

그런데 객체가 늘어나고 xml에서 빈 객체를 설정하는 것이 늘어날수록 개발자는 귀찮아지고 소스코드도 늘어나게 될 것이다.

그래서 Spring 개발팀은 더 창조적으로 게을러지기로 하였다.

|  |
| --- |
| @Autowired  Tire tire; |

|  |
| --- |
| <context:annotation-config />  <bean id="tire" class="expert004.KoreaTire"></bean>  <bean id="americaTire" class="expert004.AmericaTire"></bean>  <bean id="car" class="expert004.Car"></bean> |

음..? @Autowired 라는 것이 무엇인가? 그리고 왜 Car의 속성은 사라졌는가?!

@Autowired를 통해서 자동으로 car의 property를 찾아 줄 수 있음으로 생략이 가능해졌다.

|  |  |
| --- | --- |
| Car.java | test.xml |
| @Autowired  Tire tire; | <bean id="tire" class="expert004.KoreaTire"></bean> |

Car.java 에서 @Autowird가 붙여진 tire 속성과 test.xml 파일에서 bean의 id 속성과 일치하는 것을 찾아 매핑시켰다.

|  |  |
| --- | --- |
| Car.java | test.xml |
| @Autowired  Tire tire; | <bean class="expert004.KoreaTire"></bean> |

그럼 만약 이런 경우라면?? 당연히 에러가 날까?

|  |  |
| --- | --- |
| Car.java | @Autowired Tire tire; |
| test.xml | <bean class="expert004.KoreaTire"></bean> |
| KoreanTire.java | public class KoreaTire implements Tire |

위의 그림은 바로 interface의 구현 여부이다. 스프링의 마법은 type 기준 매핑이 먼저이고, 같은 type을 구현한 여러 개의 클래스가 있다면 그 때 bean id로 구분해서 매핑하게 된다.

그림을 잘 기억해 놓자! @Autowird는 type 기준이 먼저 매핑!

ㅁㄴㅇ

스프링을 통한 의존성 주입 - @Resource를 통한 속성 주입

@Autowired 대신에, @Resource 를 사용할 수도 있다.

|  |
| --- |
| @Resource  Tire tire; |

그럼, @Autowired와 @Resource의 차이점은 무엇일까?

@Autowired는 Spring 만의 어노테이션이다.

@Resource는 Java의 표준 어노테이션이다.

고로, 실전에서는 별반 차이가 없다. 스프링프레임워크를 사용하지 않는다면 @Resource만을 사용해야 한다.

스프링을 통한 의존성 주입 - @Autowired VS @Resource VS <property>

본격적으로 @Autowired와 @Resource의 차이점을 살펴보자.

|  |  |  |
| --- | --- | --- |
|  | @Autowired | @Resource |
| 제공자 | 스프링 프레임워크 | 표준 Java |
| 소속 패키지 | org.springframework  .beans.factory.annotation  .Autowired | javax.annotation.Resource |
| bean 검색 | byType 먼저, 못 찾으면 byName | byName 먼저, 못 찾으면 byType |
| 특이사항  byName  강제하기 | @Qualifier("") 협업  @Autowired  @Qualifier("tire1") | name 어트리뷰트  @Resource(name="tire1") |

@Autowired와 @Resource는 결과가 동일하다. 그럼 어떤 것을 써야할 것인가?

개인적인 의견으로는(여름나라 겨울이야기님) @Resource의 손을 들어 줄 것같다.

자동으로 주입된다는 의미에서는 @Autowired가 명확해 보이지만 실제 Car 입장에서 보면 @Resource라는 표현이 더 어울리기도하고 향후 스프링이 아닌 다른 프레임워크가 들어오거나 제시될 때를 고려해보면 당연히 Java 표준인 @Resource를 써야하는 것이 더 큰 이유일 것이다.

그럼 여기서 고민 @Resource(@Autowired)는 사실 <bean>태그의 자식 태그인 <property> 태그로 해결될 수 있다는 것 기억하는가? 그럼 @Resource VS <property> 에서는 누구의 손을 들어줘야 할까?

<property>의 손을 들어 줄 것이다. xml 파일만 봐도 DI 관계를 확인하기 수월하기 때문이다. 또한 유지보수성이 좋을 것이다.

(그렇지만 개발 생산성은 @Resource가 빠를 것이다.)

프로젝트 범위와 성격을 본 후 정하는 것이 가장 베스트가 아닐까?

#참고 URL

http://expert0226.tistory.com/189

출처: <http://limmmee.tistory.com/14?category=654011> [심플하게 개발]