**💡 Spring 3대 핵심요소**

Spring은 Spring Triangle이라고 부르는 핵심 3대요소를 제공해준다. 이는 각각 IoC , AOP , PSA를 일컫는다.

**💡 Spring 프레임워크가 무엇인지 설명해주세요.**

"스프링이란 IoC와 AOP를 지원하는 경량의 컨테이너 프레임워크이다."

스프링 프레임워크는 자바 플랫폼을 위한 오픈 소스 어플리케이션 프레임워크로 POJO 기반을 바탕으로, DI와 AOP, PSA를 지원하는 경량의 컨테이너를 제공하는 프레임워크를 의미합니다.

동적인 웹을 개발하기 위한 여러가지 서비스를 제공합니다.

**💡 DI(Dependency Injection, 의존관계 주입)가 무엇인지 설명해주세요.**

어떤 객체가 사용하는 의존 객체를 직접 생성하는 것이 아니라, 주입 받아 사용하는 것이다.

IoC가 디자인 원칙이라면, DI는 IoC를 구현할 때 자주 등장하는 '디자인 패턴' 중 하나입니다. DI는 의존 관계 주입으로, 의존 객체를 직접 생성하는 것이 아니라, 생성자나 setter를 통해 넣어 줌으로써 의존 관계를 주입하는 것을 의미합니다.

**수정자 주입**

* 대부분 의존 관계 주입은 한번 일어나면 종료시점까지 변경할 일이 거의 없다.
* Setter를 통해 주입하게 되면 변경될 위험이 존재
* setter을 public으로 열어야 함.

**생성자 주입**

* 생성자 주입을 권장
* 생성자 호출 시점에 딱 1번만 호출되는 것을 보장
* final 키워드를 통해 불변하게 설계 가능
* 의존성 주입이 누락되는 것을 방지할 수 있음(IDE에서 컴파일 오류로 알려줌)

**💡 '의존관계'이란 무엇인가?**

'두 개의 클래스 혹은 모듈이 의존관계에 있다'라는 것은, 누가 누구에게 의존하는 관계에 있다는 식으로 방향성이 있다.

대표적으로 A에서 B에 정의된 메소드를 호출하는 경우 의존관계가 있다고 표현할 수 있는 것이다.

방향성이 있기 때문에, B는 A의 변화에 영향을 받지 않는다는 것도 알아 두어야 할 점이다.

장난감은 배터리에 의존하고 있다.

즉, 방향성으로 표기하면 장난감 → 배터리로 표기할 수 있다.

**💡 DI로 얻을 수 있는 장점은 무엇인가?**

종속 객체 내부적으로 수정 및 변경 사항이 있는 경우 강력한 결합을 하고 있는 클래스라면 해당 클래스 내부의 코드 역시 수정이 필요하다.

반면 종속 객체 내부의 수정 및 변경이 있더라도(있을 예정이라도) 느슨한 결합을 이루고 있는 클래스라면 setter method나, 생성자를 통해 유기적인 종속 관계 주입이 가능하다.

즉 디자인 패턴인 DI(Dependency Injection)는 모듈 간의 결합도를 낮추기 위한 전략적인 방법이라고 볼 수 있다.

**💡 IoC(Inversion of Control, 제어의 역전)이 무엇인지 설명해주세요.**

IoC는 객체지향 설계에서 클래스 간의 결합도를 느슨하게 하기 위해 다양한 종류의 제어를 반전시킬 것을 권장하는 디자인 원칙이다.

객체의 생성부터 생명주기의 관리까지 모든 객체에 대한 제어권이 바뀐 것을 의미합니다.

정확히 설명하면, 디자인 원칙 중 하나로 클래스 간의 결합도를 줄이기 위해 다양한 종류의 제어를 반전시킬 것을 권장하는 것을 의미합니다.

* 깊게는 종속성을 가진 클래스의 객체 생성 제어를 역전하는 것(제어를 다른 클래스에 위임)
* 얕게는 모든 객체에 대한 제어권을 개발자가 아닌 IoC 컨테이너에게 위임하는 것
* 종속성을 가진 클래스의 객체 생성 제어를 역전하는 것(제어를 다른 클래스에 위임)
* 모든 객체에 대한 제어권을 개발자가 아닌 IoC 컨테이너에게 위임하는 것

**💡 Spring에서의 DI와 IoC / DI(IoC) Container의 장점**

스프링에서 DI와 IoC가 따라다닌 이유는, IoC 컨테이너가 IoC 원칙을 따른 다른 패턴을 대신해주는 것이 아니라 IoC, DIP, DI에 대한 구현을 우리 대신 제공하기 때문이다.

이처럼 IoC가 가지고 있는 말의 범위가 너무 넓기 때문에, DI Container라고 표기하는 것을 권장한다고 한다.

인스턴스 생성부터 소멸까지의 인스턴스 생명주기 관리를 개발자가 아닌 컨테이너가 대신해주는 것은, 개발자에게 굉장히 매력적인 장점이다. 객체관리 주체가 프레임워크(Container)가 되기 때문에 개발자는 로직에만 집중할 수 있기 때문이다.

* IoC 컨테이너는 객체의 생성을 책임지고, 의존성을 관리한다.
* POJO의 생성, 초기화, 서비스, 소멸에 대한 권한을 가진다.
* 개발자들이 직접 POJO를 생성할 수 있지만 컨테이너에게 맡긴다.
* 개발자는 비즈니스 로직에 집중할 수 있다.
* 객체 생성 코드가 없으므로 TDD가 용이하다

**💡 스프링 빈(Spring Bean) 이란?**

"스프링 컨테이너가 생성해준 자바 객체 "

Spring Bean은 DI 컨테이너가 관리하는 자바 객체를 의미한다.

우리가 흔히 스프링 부트에서는 @Component , @Controller , @Service 등의 어노테이션을 통해 특정 객체를 빈으로 등록해 사용한다.

이 스프링 빈으로 등록된 객체를 DI 컨테이너가 의존 관계를 주입시킨다.

사용자의 시간 절약과 반복 구현에 대한 불편함을 해소하기 위해 프레임워크가 DI를 대신해준다는 것은, 내가 만든 클래스에 대해서도 Spring Bean으로 등록할 수 있다는 것이다.

따라서 이 Spring Bean으로 등록하는 과정에서 의존관계에 대한 메타데이터를 Spring에게 제공하는 것이다.

이 메타 데이터는 XML을 통해서, 어노테이션을 통해서, 자바를 통해서 설정이 가능하다.

**💡 스프링 컨테이너?**

컨테이너는 객체의 생성 및 관리를 담당하며 이외의 여러 다양한 기능을 제공한다.

스프링 또한 객체를 생성하고 객체간 의존관계를 관리하므로 컨테이너라고 말할 수 있으며, 이를 스프링 컨테이너라고 부른다.

스프링에서는 크게 BeanFactory 라는 bean 객체를 관리하는 컨테이너와, 이를 상속한 ApplicationContext 라는 컨테이너를 제공한다.

ApplicationContext 는 bean 객체를 관리하는 기능 이외에도 다양한 기능을 지원한다. 따라서 대부분의 스프링 프로젝트는 이 유형의 컨테이너를 이용한다.

**💡 왜 Spring은 '프레임워크'인가?**

'프레임워크'와 '라이브러리'의 차이를 설명하는 핵심 키워드는 IoC이다.

라이브러리를 사용하는 애플리케이션 코드는 애플리케이션 흐름을 직접 제어한다. 단지 동작하는 중에 필요한 기능이 있을 때 능동적으로 라이브러리를 사용할 뿐이다.

반면에 프레임워크는 거꾸로 애플리케이션 코드가 프레임워크에 의해 사용된다. 보통 프레임워크 위에 개발한 클래스를 등록해두고, 프레임워크가 흐름을 주도하는 중에 개발자가 만든 애플리케이션 코드를 사용하도록 만드는 방식이다.

프레임워크에는 분명한 제어의 역전 개념이 적용되어 있어야 한다. 애플리케이션 코드는 프레임워크가 짜 놓은 틀에서 수동적으로 동작해야 한다.

**💡 Spring에서 IoC와 DI를 어떻게 지원해 주는지에 대해 설명해주세요.**

스프링은 IoC와 DI를 DI(IoC) Container로써 제공합니다. 클래스 사이의 의존 관계를 Bean 설정 정보를 바탕으로 컨테이너가 자동으로 연결해주는 방식으로 제공합니다.

**장점**

* 스프링 자체에서 설정을 통해 연관 관계를 맺어 줌으로써 객체간 결합도를 낮춰준다.
* 클래스의 재사용성을 높이고, 유지보수가 편리 해진다.
* 의존성 주입으로 인해 stub, mock 객체를 사용해 unit 테스트의 이점이 생긴다.

**단점**

* 의존성 주입을 위한 선행 작업이 필요해 간단한 프로그램에서는 번거롭다.
* 코드 추적이 어렵다.

DI 컨테이너는, 자바 객체의 생명 주기를 관리하며, 생성된 자바 객체들에게 추가적인 기능을 제공합니다.

종류는, BeanFactory와 ApplicationContext가 있습니다. ApplicationContext가 BeanFactory의 빈 관리 기능을 상속 받고 국제화 등의 추가기능을 갖고 있어 ApplicationContext를 사용합니다.

**💡 스프링 IOC / DI 핵심 요약**

- IoC는 소프트웨어 개발에서 모듈 간의 결합도를 줄이기 위해 제어의 반전을 권장하는 디자인 원칙에 해당하고, DI는 IoC 원칙을 구현하는 디자인 패턴 중 하나로 생성자와 setter method를 통해 의존 관계를 주입함으로써 모듈 간의 결합도를 줄인다.

- 스프링은 이러한 IoC 원칙과 DI 구현을 대신 해주는 IoC 컨테이너를 제공하는 프레임워크이다.

- 인스턴스 생성부터 소멸까지의 인스턴스 생명주기 관리를 개발자가 아닌 컨테이너가 대신 해주는 것은, 개발자에게 굉장히 매력적인 장점이다. 객체관리 주체가 프레임워크(Container)가 되기 때문에 개발자는 로직에만 집중할 수 있기 때문이다.

**💡 Bean 객체와 Bean 생명주기에 대해 설명해주세요.**

DI 컨테이너 내부에 존재하며 관리를 받고 있는 객체를 의미합니다. @Bean 어노테이션을 등록해 사용하거나, xml을 통해 등록할 수 있습니다. Bean으로 등록된 객체는 쉽게 주입하여 사용이 가능합니다.

Bean의 생명 주기는 다음과 같습니다.

스프링 컨테이너 생성 → 스프링 빈 생성 → 의존 관계 주입 → 초기화 콜백 → 사용 → 소멸 전 콜백 → 스프링 종료

* 스프링 컨테이너에 의해 생명주기 관리
* 스프링 컨테이너 초기화 시 빈 객체 생성, 의존 객체 주입 및 초기화
* 생성과 의존관계 주입과 초기화 분리
* 의존관계 주입(생성자 주입)은 필수정보를 받고 메모리 할당을 통해 객채 생성 책임
* 초기화는 생성된 값들을 활용해 외부 커넥션을 연결하는 등 무거운 작업 수행
* 명확하게 분리하는 것이 유지보수 관점에서 좋다.
* 싱글톤 빈들은 컨테이너가 종료되기 직전에 소멸전 콜백이 발생

**💡 Annotation에 대해 설명해주세요.**

프로그램에게 추가적인 정보를 제공하는 메타데이터. annotation을 정의하고, 원하는 위치에 배치한 후, 코드가 실행되는 도중 자바 리플렉션을 이용해 추가 정보를 획득해 기능을 실시하는 방식으로 동작됩니다.

Spring 컨테이너에서 객체가 호출되면, 객체의 인스턴스를 생성하게 되는데 이때 자바 리플렉션이 필요하게 됩니다.

**💡 Spring에서 제공하는 대표적인 Annotation 몇 가지만 예를 들어주세요.**

- 프로그램에 추가적인 정보를 제공하는 메타데이터.

- 리플렉션을 이용해 추가정보를 획득해 기능한다.

@ComponentScan

@Component, @Service, @Repository, @Controller, @Configuration이 붙은 클래스 Bean들을 찾아서 Context에 bean을 등록해주는 애노테이션

전부 다 @Component를 사용하지 않고 @Repository 등으로 분리해서 사용하는 이유는, 예를 들어 @Repository는 DAO에서 발생할 수 있는 unchecked exception들을 스프링의 DataAccessException으로 처리할 수 있기 때문이다.

또한 가독성에서도 해당 애노테이션을 갖는 클래스가 무엇을 하는지 단 번에 알 수 있다.

@EnableAutoConfiguration

autoConfiguration도 Configuration중 하나에 해당한다.

spring.factories 내부에 여러 Configuration들이 있고 조건에 따라 Bean이 등록되게 되는데 메인 클래스 @SpringBootApplication을 실행하면 @EnableAutoConfiguration에 의해 spring.factories 안에 있는 수많은 자동 설정들이 조건에 따라 적용되어 수 많은 Bean들이 생성된다.

간단하게 정리하면, Application Context를 만들 때 자동으로 빈설정이 되도록 하는 기능이다.

@Component

개발자가 직접 작성한 class를 Bean으로 등록하기 위한 애노테이션

@Bean

개발자가 직접 제어가 불가능한 외부 라이브러리 등을 bean으로 만들려 할 때 사용되는 애노테이션

@Configuration

@Configuration을 클래스에 적용하고 @Bean을 해당 class의 메서드에 적용하면 @autowired로 Bean을 부를 수 있다.

@Autowired

스프링이 Type에 따라 알아서 Bean을 주입해준다.

Type을 먼저 확인한 후 못 찾으면 Name에 따라 주입한다.

강제로 주입하고자 하는 경우 @Qulifier을 같이 명시

@Qualifier

같은 타입의 빈이 두 개 이상 존재하는 경우 스프링이 어떤 빈을 주입해야할 지 알 수 없어서 스프링 컨테이너를 초기화하는 과정에서 예외가 발생한다.

@Qualifier는 @Autowired와 함께 사용하여 정확히 어떤 bean을 사용할지 지정하여 특정 의존 객체를 주입할 수 있다.

@Resource

@Autowired와 마찬가지로 Bean 객체를 주입해주는데 차이점은 Autowired는 타입으로, Resource는 이름으로 연결해준다.

애노테이션 사용으로 인해 특정 Framework에 종속적인 애플리케이션을 구성하지 않기 위해서 @Resource 사용을 권장한다.

@Controller

API와 view를 동시에 사용하는 경우에 사용

보통 view 화면 return을 목적으로 사용한다.

@RestController

view가 필요 없이 API만 지원하는 서비스에서 사용

@SpringBootApplication

@Configuration, @EnableAutoConfiguration, @ComponentScan 3가지를 하나로 합친 애노테이션

Annotation은 적당히 내가 설명할 수 있을 정도로만 알아가도록 하자..너무 많아서 다 설명하기 힘듬.

**💡 리플렉션 (Reflection)이란?**

구체적인 클래스 타입을 알지 못해도, 그 클래스의 메소드, 타입, 변수들에 접근할 수 있도록 해주는 자바 API

자바는 정적인 언어라 부족한 부분이 많은데 이 동적인 문제를 해결하기 위해서 리플렉션을 사용합니다.

정적 언어, 동적 언어 ?

* 정적 언어: 컴파일 시점에 타입을 결정 ex) Java, C, C++ 등..
* 동적 언어: 런타임 시점에 타입을 결정 ex) Javascript, Python, Ruby 등..

리플렉션은 애플리케이션 개발에서 보다는 프레임워크, 라이브러리에서 많이 사용됩니다.

프레임워크, 라이브러리는 사용하는 사람이 어떤 클래스를 만들지 모릅니다. 이럴 때 동적으로 해결해주기 위해서 리플렉션을 사용합니다.

대표적인 사용 예로는 스프링의 DI(dpendency injection), Proxy, ModelMapper 등이 있습니다.

**💡 웹 서버와 웹 애플리케이션 서버가 어떻게 다른지 알고 계시나요?**

스크린샷, 도표, 라인, 직사각형이(가) 표시된 사진

자동 생성된 설명

**웹 서버**

* 정적 리소스 파일을 제공하는 서버

**웹 애플리케이션 서버(WAS)**

* 웹 서버가 하는 일 + 애플리케이션 로직(DB 연결, 동작 수행, 데이터 제공)까지 제공하여 동적인 처리를 하는 서버
* 자바 진영에서는 서블릿 컨테이너 기능을 제공하면 WAS 라고 한다.
* 위 그림에는 없지만 WAS 안에도 웹 서버가 따로 존재한다.

**💡 서블릿과 서블릿 컨테이너에 대해서 설명해주세요.**

웹 서버와 웹 애플리케이션 서버의 차이를 보면, '동적인 처리'에 대한 차이가 있습니다.

WAS안에서 이러한 동적인 처리를 하는데 사용하는 서버 프로그램을 서블릿이라고 합니다.

서블릿이 존재하기 전에는 HTTP 요청 메시지를 파싱하는 작업 등 여러 부가적인 작업을 개발자가 해야 하나, 서블릿의 등장이후로는 이러한 부가 작업을 서블릿이 대신 하고, 개발자는 실질적인 메인 로직만 집중할 수 있게 되었습니다.

DI Container와 비슷하게, 서블릿 컨테이너는 서블릿의 생명주기를 관리합니다.

* 사용자가 URL을 클릭하면 HTTP Request를 Servlet Container로 보낸다.
* Servlet Container는 쓰레드 풀에서 쓰레드를 꺼내 할당 해주고 HttpServletRequest, HttpServletResponse 두 객체를 생성한다.
* 사용자가 요청한 URL을 분석하여 어느 서블릿에 대한 요청인지 찾는다.
* 서블릿 컨테이너에 존재하지 않으면 초기화하고 있다면 가져와서 service() 메서드를 호출한다.
* Spring MVC의 경우 DispatcherServlet이 초기화되고 호출된다.
* service 메서드가 수행이 끝나면 HttpServletResponse 객체에 응답을 보낸다.
* 응답이 완료되면 HttpServletRequest, HttpServletResponse 객체를 소멸시킨다.

**💡 Spring MVC에 대해 설명해주세요.**

Spring에서 제공하는 웹 모듈로, Model / View / Controller 세 가지 구성요소를 사용해 사용자의 다양한 HTTP Request를 처리하고 단순한 텍스트 형식의 응답부터 REST 형식의 응답은 물론, View를 표시하는 HTML을 Retrun하는 응답까지 다양한 응답을 할 수 있도록 도와주는 Framework입니다.

**MVC 패턴 장단점**

정의

* Model, View, Controller로 분리하는 아키텍처

**장점**

* 과거에는 Controller에 다 담아두고 처리했다.
* 기능 별로 코드를 분리하여, 가독성을 높이고 재사용성을 증가시킨다.

**단점**

* view와 model 사이에 의존성이 높아서 애플리케이션이 커질수록 복잡해지고 유지보수가 어렵다.
* 대규모의 프로그램에서 Controller에 다수의 Model과 View가 복잡하게 연결되어 코드 분석과 테스트가 어려워 질 수 있다.
* 이런 의존성 문제를 해결하기 위해 MVVM, MVP 구조가 등장했다.

**💡 그렇다면, 어떻게 하나의 컨트롤러로 여러 요청을 받을 수 있을까요?**

컨트롤러는 컴포넌트 스캔이 되면서 스프링 빈 컨테이너에 올라갑니다.

또한 빈 객체는 싱글톤 패턴으로 구현 되어있기 때문에 여러 쓰레드의 요청이 들어와도 하나의 컨트롤러 객체를 공유하면서 처리합니다.

즉, 여러 쓰레드가 메서드에 대해 공유 자원으로써 접근해 사용합니다.

주의할 점은, 싱글톤 패턴으로 구현 되어있다는 점은, Thread-Safe하지 않다는 의미이므로, 상태를 공유하거나 저장하는 코드가 없도록 Stateless하게 정의해야 합니다.

**💡 + 그럼 싱글톤 패턴은 무엇인가요?**

컨트롤러는 컴포넌트 스캔이 되면서 스프링 빈 컨테이너에 올라갑니다.

또한 빈 객체는 싱글톤 패턴으로 구현 되어있기 때문에 여러 쓰레드의 요청이 들어와도 하나의 컨트롤러 객체를 공유하면서 처리합니다.

즉, 여러 쓰레드가 메서드에 대해 공유 자원으로써 접근해 사용합니다.

메모리 측면에서 최초 한 번의 new 연산자를 통해 고정된 메모리 영역을 사용하기 때문에 해당 객체에 접근할 때 메모리 낭비를 방지할 수 있고, 이미 생성된 인스턴스를 활용하니 속도 측면에도 이점이 있습니다.

다른 이점으로는 다른 클래스 간에 데이터 공유가 쉽습니다. 싱글톤 인스턴스는 전역으로 사용되는 인스턴스이기 때문에 다른 클래스의 인스턴스들이 접근해 사용할 수 있습니다. 그렇기 때문에 '동시성 문제'가 일어나지 않도록 유의해 설계해야 합니다.

다시 말하자면, 싱글톤 패턴으로 구현 되어있다는 점은, Thread-Safe하지 않다는 의미이므로, 상태를 공유하거나 저장하는 코드가 없도록 Stateless하게 정의해야 합니다.

**💡 AOP에 대해 설명해주세요.**

관점 지향 프로그래밍의 약어로, 공통 관심사항과 핵심 관심사항을 분리하는 것을 의미합니다. 각 소스코드에서 여러 번 반복해서 사용하는 코드를 흩어진 관심사라고 부르고, 이것을 Aspect로 모듈화해 핵심 로직에서 분리해 재사용하는 것이라고 볼 수 있습니다.

예를 들어, 로그, 성능테스트, 권한, 트랜잭션 등을 모든 메소드에 적용하고 싶을 때 일일이 추가하는 것이 아니라 AOP를 활용해 적용할 수 있다는 것입니다.

**💡 POJO란 무엇인가요?**

Plain Old Java Object의 약어로, 평범한 구식 자바 객체를 의미합니다. 프레임워크 인터페이스나 클래스를 구현하거나 확장하지 않은 단순한 클래스를 의미하는 것으로, 타 프레임워크에 종속되지 않아 코드가 간결하고 테스트 자동화에 유리합니다. Spring에서는 도메인과 비즈니스 로직을 수행하는 대상이 POJO 대상이 될 수 있습니다.

**💡 DAO, DTO, BO, VO의 차이를 알고 계시나요? 아는 대로 설명해주세요.**

**- DAO (Data Access Object)**

DB 데이터를 조회하거나 조작하는 기능을 전담하는 객체

DB 접근 로직과 비즈니스 로직을 분리하기 위해서 사용

**- DTO (Data Transfer Object)**

계층 간의 데이터 교환을 위한 객체

로직을 갖지 않는 순수 데이터 객체로 getter, setter만 포함

**- VO (Value Object)**

DTO와 동일한 개념

Read Only로 수정 불가

getter, setter 이외의 추가 로직 포함 가능

**- BO (Business Object)**

비즈니스 로직을 포함하는 오브젝트로, 여러 DAO를 사용해 데이터를 처리

**💡 Spring의 레이어드 아키텍처에 대해 설명해주세요**.

Spring은 레이어드 아키텍처로 이루어져 있습니다.

하나의 레이어는 자신의 고유 역할을 수행하고, 인접한 다른 레이어에 무언가를 요청하거나 응답합니다.

그 밖의 다른 레이어는 신경 쓸 필요가 없기 때문에 각 레이어는 자신의 역할에 충실할 수 있습니다.

따라서 시스템 전체를 수정하지 않고 특정한 레이어의 기능을 개선하거나 교체할 수 있기 때문에 재사용성이 좋고 유지 보수에도 유리합니다. 또한, 레이어 별로 테스트 구현이 편해지고 코드 가독성도 높아집니다.

Presentation Layer

* view를 담당하는 부분으로, 클라이언트와 직접적으로 맞닿는 부분

Application Layer

* 비즈니스 핵심 로직을 처리하는 부분
* Service 객체라는 것은 하나의 트랜잭션으로 구성되어 작동

Persistence Layer

* 데이터 관련 처리를 담당하는 부분

**💡 커넥션 풀이 무엇인지, Springboot에서의 커넥션 풀로는 어떤 것을 사용하는지 설명해주세요.**

어플리케이션이 DB를 사용하기 위해서는 Connection이 필요한데, 이 생성 및 소멸 비용이 크기 때문에, 커넥션 풀을 미리 생성하고 애플리케이션이 시작하는 시점에 커넥션을 미리 다 생성하고 이것을 재활용하며 사용하게 됩니다.

**💡 트랜잭션을 추상화해서 사용하는 이유를 알고 계시나요?**

다양한 데이터 접근 기술이 등장하면서, 코드레벨에서는 다르지만 논리적으로는 같은 기능을 수행하기 때문에 트랜잭션을 추상화했습니다.

다양한 접근 기술로는 JDBC, JPA, 하이버네이트 등이 있습니다.

**💡 트랜잭션 동기화 매니저의 역할에 대해 안다면, 아는대로 설명해주세요.**

보통 서비스 단에서 트랜잭션을 시작하고 끝내게 됩니다.

그렇다면 하나의 트랜잭션 내에서는 같은 커넥션을 사용 해야 하는데 과정이 다음과 같습니다.

서비스단에서 트랜잭션이 시작하면 트랜잭션 매니저가 커넥션을 생성하고(풀을 사용하면 풀에서 가져오고) autoCommit을 false로 세팅한 뒤 트랜잭션 동기화 매니저의 스레드 로컬에 커넥션을 보관

이후 리포지토리 계층에서는 트랜잭션 동기화 매니저의 스레드 로컬에서 해당 커넥션을 가져와서 사용

서비스 단에서 트랜잭션을 종료할 때는 트랜잭션 동기화 매니저에서 해당 커넥션을 가져와 커밋 또는 롤백을 수행하고 리소스를 정리하고 커넥션을 커넥션 풀에 반환

하나의 트랜잭션에서 같은 커넥션을 사용하도록 도움을 주는 기능을 제공한다고 보면 됩니다.

**💡 Spring을 사용해 트랜잭션을 적용하는 방법 2가지에 대해 설명해주세요.**

**1) 선언적 트랜잭션**

@Transactional 어노테이션을 사용함으로써 트랜잭션을 적용시킬 수 있습니다. 추가적인 코드 작성이 없어 대부분 사용하는 방식입니다.

@Transactional 어노테이션은 AOP로 구성되어 있습니다. AOP는 프록시 패턴으로 동작하기 때문에, 오버라이딩 개념으로 동작합니다. 메서드에 @Transactional을 붙이면, 해당 클래스가 빈으로 등록될 때, @Transactional이 붙은 메소드만 트랜잭션 처리되는 메소드로 오버라이딩 한 프록시 객체가 빈으로 등록되게 됩니다.

**2) 프로그래밍 방식 트랜잭션**

트랜잭션 매니저나 트랜잭션 템플릿을 사용해 프로그래밍 코드로 작성해 적용하는 방식입니다.

**💡 Spring Framework의 특징**

1. DI(Dependency Injection)
2. IoC(Invesion of Control)
3. AOP(Aspect Oriented Programming)
4. 중복 코드 제거
5. 다른 프레임워크와의 통합

**💡 Spring Boot Framework와 Spring Framework의 차이점**

스프링 부트는 단독적이고, 상용화 수준의, 스프링 기반 애플리케이션을 단지 실행할 수 있을 정도로 쉽게 만들 수 있다.

1. **Dependency**

* Spring Framework의 경우 dependency를 설정해줄 때 설정 파일이 매우 길고, 모든 dependency에 대해 버전 관리도 하나하나 해줘야 합니다.
* Spring Boot Framework의 경우 dependency를 Spring Framework보다 쉽게 설정해 줄 수 있습니다. 버전 관리도 자동으로 해줍니다.

1. **Configuration**

* Spring Framework의 경우 configuration설정을 할 때도 매우 길고, 모든 어노테이션 및 빈 등록 등을 설정해 줘야 합니다.
* Spring Boot Framework는 application.properties파일이나 application.yml파일에 설정하면 됩니다.

1. **편리한 배포**

* Spring Framework로 개발한 애플리케이션의 경우, war파일을 Web Application Server에 담아 배포했습니다.
* Spring Boot Framework의 경우, Tomcat 이나 Jetty 같은 내장 WAS를 가지고 있기 때문에 jar 파일로 간편하게 배포할 수 있습니다.
* Spring Framework로 WAS를 정하고, 모든 설정을 마쳐 배포를 하는 것보다 훨씬 간단한 배포 방법입니다.

**💡 메타데이터란?**

"데이터를 위한 데이터"

'속성정보'라고도 불리는 메타데이터는 '데이터에 관한 구조화된 데이터', '다른 데이터를 설명해 주는 데이터'이다.

대량의 정보 가운데에서 찾고 있는 정보를 효율적으로 찾아내서 이용하기 위해 일정한 규칙에 따라 콘텐츠에 대하여 부여되는 데이터이다. >> (인스타그램의 해시태그('#')와 유사한 역할)

- 콘텐츠의 위치와 내용, 작성자에 관한 정보, 권리 조건, 이용 조건, 이용 내력 등이 기록된다.

- 목적 :

1. 데이터를 표현하기 위한 목적

2. 데이터를 빨리 찾기 위한 목적 ( 정보의 인덱스 역할 )

+ 메타데이터 주 사용언어

- HTML : 계층적 구조 but 새로운 태그 사용X, 문서양식에 사용되는 속성 표현 어려움, 문서 재사용 불가능

- XML : HTML 단점 보완, 대용량의 복잡한 구조 문서 작성에 용이함 but 문서작성이 복잡해짐

- 출력형식의 언어 : HTML -> CSS // XML -> CSS/XSL

**💡 Hibernate란?**

텍스트, 스케치, 화이트, 도표이(가) 표시된 사진

자동 생성된 설명

하이버네이트는 자바 언어를 위한 ORM 프레임워크. JPA의 구현체로, JPA 인터페이스를 구현하며, 내부적으로 JDBC API를 사용.

JPA는 관계형 데이터베이스와 객체의 패러다임 불일치 문제를 해결할 수 있다는 점과 영속성 컨텍스트(엔티티를 영구 저정하는 환경) 제공이 큰 특징.

**JPA**

자바 애플리케이션에서 관계형 데이터베이스를 사용하는 방식을 정의한 인터페이스. 라이브러리가 아닌 인터페이스이므로 특정 기능을 하진 않는다.

**JDBC**

자바 프로그래밍 언어와 다양한 데이터베이스 SQL 또는 테이블 형태의 데이터 사이에 독립적인 연결을 지원하는 표준이에요. 즉, DB 작업을 위한 표준이라 볼 수 있다.

**하이버네이트의 장점**

1. 생산성

* Hibernate는 SQL을 직접 사용하지 않고, 메서드 호출만으로 쿼리가 수행돼요. 즉, SQL 반복 작업을 하지 않음으로 생산성이 높아진다.
* SQL을 몰라도 되는 건 아니다. (내부 동작에 대해 알아야 하기 때문)

2. 유지보수

* 테이블 컬럼이 변경되었을 때, 테이블과 관련된 DAO의 파라미터, 결과, SQL 등을 대신 수행. 이로 인해 유지보수 측면에서 높다.

3. 특정 벤더에 종속적이지 않음

* JPA는 추상화된 데이터 접근 계층을 제공하기 때문에 특정 벤더에 종속적이지 않다.
* 설정 파일에서 JPA에게 어떤 DB를 사용하고 있는지를 알려주기만 하면 얼마든지 DB를 바꿀 수 있다.

4. 패러다임 불일치 해결

* 상속, 연관 관계, 객체 그래프 탐색, 비교 등 객체와 관계형 데이터베이스와의 패러다임 불일치를 해결할 수 있다.

**하이버네이트의 단점**

1. 성능

* 메서드 호출만으로 쿼리를 수행하는 것은 직접 SQL을 작성하는 것보다는 성능상 좋지 않다.

2. 세밀함

* 메서드 호출만으로 DB 데이터를 조작하기에는 한계가 있다. 이를 보완하기 위해 JPQL을 지원.
* NativeQuery를 지원하여 SQL 자체 쿼리도 작성할 수 있다.

3. 러닝커브

* 공부해야 할 것이 많다.

**💡 ORM (Object-Relational Mapping)?**

* Class와 RDB (Relational DataBase)의 테이블을 매핑한다는 뜻으로 객체를 RDB 테이블에 자동으로 영속화 해주는 것

**💡 JPA vs MyBatis 정리 (특징, 장점, 단점)**

JPA, MyBatis 등장배경

* 기존에 JDBC를 사용했을 때는 sql문에 코드에 섞여 있어 만들어 요청하는 과정에서 sql문 생성시 String을 붙이고 자르는 등의 작업이 필요해서 sql문이 조금만 길어져도 번거롭고 관리도 힘들었다.
* 그래서 코드와 sql문을 분리해서 관리하기 위해서 JPA, MaBatis 등을 사용하기 시작했다.

**MyBatis란?**

**장점**

* 접근이 쉽고 코드가 간결하다.
* SQL문과 프로그래밍 코드가 분리되어 있어서 SQL문에 변경이 있을 때마다 자바 코드를 수정하거나 컴파일 하지 않아도 된다.
* 다양한 프로그래밍 언어로 구현이 가능하다. (이식성이 뛰어나다)

**단점**

* 스키마 변경시 SQL 쿼리를 직접 수정해주어야 한다.
* 반복된 쿼리가 발생하여 반복 작업이 있다.
* 쿼리를 직접 작성하기 때문에 데이터베이스에 종속된 쿼리문이 발생할 수 있다.
* 데이터베이스 변경시 로직도 함께 수정해주어야 한다.

**JPA란?(Java Presistence API)**

- Java ORM 기술에 대한 API 표준 명세.

- 구현된 클래스와 매핑을 해주기 위해 사용되는 프레임워크이다.

- 구현체로는 Hibernate 가 가장 대중적이다.

**장점**

* 쿼리를 하나하나 작성할 필요가 없어 코드 량이 줄어든다.
* 가독성이 좋다.
* 간편하게 수정이 가능하다.(유지보수, 리팩토링 용이)
* 동일한 쿼리에 대한 캐시 기능을 사용하기 때문에 더욱 높은 성능을 낼 수 있다.

**단점**

* 매핑 설계를 잘못했을 때 성능 저하가 발생.
* JPA를 제대로 사용하려면 알아야할 것이 많아서 학습하는데 시간이 오래 걸린다.
* 다수의 테이블 조인시 신경써야 할게 많다.

**JPA vs MyBatis**

- JPA는 객체 중심의 개발 방식을 지원하며, 개발자가 직접 SQL을 작성하지 않고도 객체를 관리할 수 있다는 장점

- MyBatis는 SQL을 직접 작성할 수 있어 개발자가 더욱 자유롭게 데이터베이스를 다룰 수 있다는 장점

- 따라서, 프로젝트의 목적과 상황에 따라 적절한 ORM기술을 선택해야한다.

**💡 PSA(Portable Service Abstraction)란?**

PSA란 환경의 변화와 관계없이 일관된 방식의 기술로의 접근 환경을 제공하는 추상화 구조를 말합니다.

이는 POJO 원칙을 철저히 따른 Spring의 기능으로 Spring에서 동작할 수 있는 Library들은 POJO원칙을 지키게끔 PSA형태의 추상화가 되어있음을 의미합니다.

"잘 만든 인터페이스 하나가 열 클래스 부럽지 않다"

PSA = 잘 만든 인터페이스

PSA가 적용된 코드라면 나의 코드가 바뀌지 않고, 다른 기술로 간편하게 바꿀 수 있도록 확장성이 좋고,

기술에 특화되어 있지 않는 코드를 의미합니다.

Spring은 Spring Web MVC, Spring Transaction, Spring Cache 등의 다양한 PSA를 제공합니다.

하나의 추상화로 여러 서비스를 묶어둔 것을 PSA(Portable Service Abstraction)이라고 합니다.

**💡 @Transactinal이 무엇인가?**

@Transactional 어노테이션은 스프링에서 제공하는 클래스, 메소드 레벨의 트랜잭션 지원 어노테이션이다.

이 어노테이션이 선언되면 스프링은 프록시 객체를 생성하여 자동 commit, rollback 등의 트랜잭션 처리를 맡긴다.

클래스, 메소드 내에서 Persistence layer에 접근하여 데이터를 조작, 조회 할 때 사용한다. 주로 Service layer에서 사용하게된다.

트랜잭션 격리 수준

트랜잭션 어노테이션답게 트랜잭션의 기능들 중 하나인 격리 수준 속성을 가지고 있다.

Default

* 기본 격리 수준(DB의 격리 수준을 따름)

READ\_UNCOMMITTED (Level 0)

* 커밋되지 않은 데이터 읽기 허용
* 변경 중인 데이터에 접근할 수 있다.
* Dirty read 문제가 발생할 수 있다.

READ\_COMMITTED (Level 1)

* 커밋된 데이터만 읽기 허용
* 변경 중인 데이터에 접근할 수 없다.
* Dirty read 문제를 방지하는 레벨

REPEATABLE\_READ (Level 2)

* 한 트랜잭션이 진행되는 동안 다른 트랜잭션이 해당 데이터를 수정할 수 없다.
* 결과적으로 다른 트랜잭션은 일관성 있는 데이터를 가져가게 된다.
* Non-repeatable read 문제를 방지하는 레벨

SERIALIZABLE (Level 3)

* 한 트랜잭션이 진행되는 동안 다른 트랜잭션은 전혀 접근할 수 없다.
* 가장 강한 격리 수준이지만 성능이 크게 저하된다.
* Phantom read 문제를 방지하는 레벨 격리 수준이 올라갈수록 성능은 떨어진다.

**💡 JPA 지연로딩을 사용해야하는 이유, 지연로딩(Lazy)과 즉시로딩(Eager)**

JPA에서는 데이터를 조회할 때 즉시 로딩(EAGER)과 지연 로딩(LAZY) 두 가지 방식이 있다.

이 두 가지 방식을 간단하게 설명하면 즉시 로딩은 데이터를 조회할 때 연관된 데이터까지 한 번에 불러오는 것이고, 지연 로딩은 필요한 시점에 연관된 데이터를 불러오는 것이라고 할 수 있다.

**즉시 로딩(EAGER)**

* @xxToxx(fetch = fetchType.EAGER)

Member 엔티티와 Team 엔티티가 N:1 매핑으로 관계를 맺고 있다.

즉시 로딩(EAGER) 방식을 사용하면 Member를 조회하는 시점에 바로 Team까지 불러오는 쿼리를 날려 한꺼번에 데이터를 불러온다.

**지연 로딩(LAZY)**

* @xxToxx(fetch = fetchType.LAZY)

지연 로딩으로 설정하고 Member를 조회해보면 즉시 로딩 방식과 달리 Team을 조회하는 쿼리가 생성되지 않고 Member를 조회하는 쿼리만 나가고, 실제로 팀을 사용하는 시점에 Team을 조회하는 쿼리가 나간다.

지연 로딩을 사용하면 Member를 조회하는 시점이 아닌 실제 Team을 사용하는 시점에 쿼리가 나가도록 할 수 있다는 장점이 있다.

즉시 로딩에서는 Member와 연관된 Team이 1개 여서 Team을 조회하는 쿼리가 1개 나갔지만, 만약 Member를 조회하는 JPQL을 날렸는데 연관된 Team이 1000개라면? Member를 조회하는 쿼리를 하나 날렸을 뿐인데 Team을 조회하는 SQL 쿼리 1000개가 추가로 나가게 된다.

그렇기 때문에 가급적이면 기본적으로 지연 로딩을 사용하는 것이 좋다.

**💡 TDD(Test Driven Development)란?**

* TDD란 Test Driven Development의 약자로 ‘테스트 주도 개발’이라고 한다.
* 반복 테스트를 이용한 소프트웨어 방법론으로 작은 단위의 테스트 케이스를 작성하고 이를 통과하는 코드를 추가하는 단계를 반복하여 구현한다.
* 짧은 개발 주기의 반복에 의존하는 개발 프로세스이며, 애자일 방법론 중 하나인 eXtream Programming(XP)의 ‘Test-First’ 개념에 기반을 둔 단순한 설계를 중요시한다.