



Spring&JPA

# Index	0
📅 CreatedAt	
👤 Person	 Ally Hyeseong Kim
⚙️ Status	TODO
☰ Tags	Spring
📅 UpdatedAt	

References

면접 시리즈2 - Spring & JPA

프레임워크란 응용 프로그램이나 소프트웨어 솔루션 개발을 수월하기 위해 구조, 틀이 제공된 소프트웨어 환경 이다.스프링 프레임워크는 자바 플랫폼을 위한 오픈 소스 애플리케이션 프레임워크 로서 간단히 스

 <https://velog.io/@backtony/면접-시리즈2-Spring-JPA>

면접 시리즈2 - Spring & JPA

References

1. 프레임워크
2. Spring
3. DI (Dependency Injection)
 - 3.1. 주입 방식
4. IoC (Inverse of Control 제어의 역전)
5. 스프링 컨테이너
6. Bean
 - 6.1. Bean 생명주기
 - 6.2. 빈 스코프
 - 6.3. 싱글톤 vs 스프링 싱글톤
 - 6.4. 스프링 싱글톤
7. Annotation
 - 7.1. 각종 애노테이션
8. 웹 서버와 웹 애플리케이션 서버
9. 서블릿과 MVC 패턴
 - 9.1. 서블릿
 - 9.2. 서블릿 컨테이너
 - 9.3. 요청 시 동작 과정

- 10. MVC 패턴
 - 10.1. MVC 패턴 장단점
 - 10.2. 어떻게 하나의 컨트롤러로 여러 요청을 받을까?
 - 10.3. 특별한 설정이 없다면 싱글톤 패턴인데 멀티스레드 환경에 어떤 문제가 생길까?
- 11. Thread-safe하게 싱글톤 구현(LazyHolder)
- 12. AOP(Aspect Oriented Programming)
 - 12.1. AOP 적용 방식
- 13. POJO
- 14. DAO, DTO
- 15. Filter, Interceptor
 - 15.1. 실행 과정
 - 15.2. Filter vs Interceptor
 - 15.3. AOP, Interceptor
- 16. 레이어드 아키텍처
- 17. OSIV
- 18. 커넥션 풀
- 19. DataSource
 - 19.1. 트랜잭션을 추상화하는 이유
 - 19.2. 트랜잭션 동기화 매니저
 - 19.3. 선언적 트랜잭션 vs 프로그래밍 방식 트랜잭션
 - 19.4. @Transactional
 - 19.5. 내부 호출 문제
 - 19.6. Propagation 전파단계
- 17. ORM
- 18. JPA
 - 18.1. 영속성 컨텍스트
 - 18.2. 영속성 컨텍스트의 이점
 - 18.3. 저장 동작 과정
 - 18.4. 수정 과정
 - 18.5. N+1 문제
 - 18.6. OneToOne 양방향 관계 Lazy 로딩 주의
 - 18.7. OneToMany fetch join 데이터 뺏기기 문제
 - 18.8. MultipleBagFetchException
 - 18.9. fetch join 한계
 - 18.10. 상속관계 매핑
 - 18.11. Id 값을 Long으로 사용하는 이유
 - 18.12. QueryDsl을 사용하는 이유
- 19. spring Application을 구동할 때 메서드를 실행시키는 법
- 20. 순환참조
- 21. Spring batch
 - 21.1. 청크기반 방식
 - 21.2. hibernateItemReader 와 jpaPaingReader의 차이

1. 프레임워크

- 프레임워크란 응용 프로그램이나 소프트웨어 솔루션 개발을 수월하기 위해 구조, 틀이 제공된 소프트웨어 환경이다.
-

2. Spring

- 스프링 프레임워크는 자바 플랫폼을 위한 오픈 소스 애플리케이션 프레임워크로서 간단히 스프링이라고도 한다. 동적인 웹 사이트를 개발하기 위한 여러 가지 서비스를 제공하고 있다.
 - 장점
 - POJO 기반의 구성으로 자바 코드를 이용해서 객체를 구성하는 방식 그대로 스프링에서 사용할 수 있다.
 - 덕분에 높은 생산성과 유연한 테스트를 할 수 있다.
 - DI(의존성 주입)을 통한 객체 관계 구성을 지원한다.
 - AOP(횡단 관심사 분리) 지원
 - MVC 구조로 계층이 분리되어 관리하기 수월하다.
 - 배치 애플리케이션 스프링 배치가 있다.
-

3. DI (Dependency Injection)

- DI는 스프링 프레임워크에서 지원하는 IoC의 형태이다. 클래스 사이의 의존관계를 빈 설정 정보를 바탕으로 컨테이너가 자동으로 연결해주는 것이다.
- 장점
 - 스프링 자체에서 설정을 통해 연관 관계를 맺어줌으로써 객체간 결합도를 낮춰준다.
 - 클래스의 재사용성을 높이고, 유지보수가 편리해진다.
 - 의존성 주입으로 인해 stub, mock 객체를 사용해 unit 테스트의 이점이 생긴다.
- 단점

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

3.1. 주입 방식

- 수정자 주입
 - 대부분 의존 관계 주입은 한번 일어나면 종료시점까지 변경할 일이 거의 없다.
 - Setter를 통해 주입하게 되면 변경될 위험이 존재
 - setter을 public으로 열어야함
 - 필드 주입
 - 외부에서 변경이 불가능해서 테스트하기 어렵다.
 - 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)
-

4. IoC (Inverse of Control 제어의 역전)

- 객체의 생성부터 생명주기의 관리까지 모든 객체에 대한 제어권이 바뀐 것을 의미 한다. 개발자는 프레임워크에 필요한 부품을 개발하고 조립하는 방식으로 개발을 하고 최종 호출은 개발자가 아니고 프레임워크의 내부에서 결정된 대로 이뤄지게 되는데 이런 현상을 제어의 역전이라고 한다.
-

5. 스프링 컨테이너

- 스프링 컨테이너는 자바 객체의 생명 주기를 관리하며, 생성된 자바 객체들에게 추가적인 기능을 제공하는 역할을 한다.스프링 컨테이너의 종류에는 BeanFactory 와 ApplicationContext 가 있다.둘 다 빈을 등록하고 생성하고 조회하고 돌려주는 등 빈을 관리하는 역할을 한다.ApplicationContext가 BeanFactory의 빈 관리 기능들을 상속받았고, 그 외에 국제화 등의 추가적인 기능을 갖고 있어 스프링 컨테이너라고 하면 보통 ApplicationContext라고 한다.IoC와 DI의 원리가 이 스프링 컨테이너에 적용된다.
-

6. Bean

- 컨테이너 안에 들어있는 객체
- 컨테이너에 담겨있으며, 필요할 때 컨테이너에서 가져와서 사용
- @Bean을 사용해 등록하거나 xml을 사용해 등록하고, Bean으로 등록된 객체는 쉽게 주입하여 사용 가능

6.1. Bean 생명주기

- 스프링 컨테이너 생성 -> 스프링 빈 생성 -> 의존 관계 주입 -> 초기화 콜백 -> 사용 -> 소멸전 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 객체 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
- 초기화와 소멸 메서드는 애노테이션으로 @PostConstruct, @PreDestroy 를 사용하는 것이 권장된다.

6.2. 빈 스코프

- 싱글톤
 - spring 프레임워크의 기본이 되는 스코프
 - 스프링 컨테이너 시작과 종료까지 1개의 객체로 유지
- 프로토타입
 - 빈의 생성, 의존관계 주입, 초기화까지만 관여하고 이후에는 컨테이너에서 관리하지 않는 스코프
 - 따라서 매번 요청마다 새로 만들어짐
 - 싱글톤은 스프링이 뜰때 생성되는데 반해, 프로토타입은 요청할때 생성됨
- 웹 스코프

- request : 각 요청이 들어오고 나갈때까지 유지
- session : 세션이 생성되고 종료될때까지 유지
- application : 웹의 서블릿 컨텍스트와 동일한 생명주기를 갖는 스코프
 - 서블릿 컨텍스트는 web application내에 있는 모든 서블릿들을 관리하며 정보 공유할 수 있게 도와 주는 역할 을 하는데, 톰캣 컨테이너가 실행 시 애플리케이션 하나당 한개의 서블릿컨텍스트가 생성된다.
 - 생명 주기는 보통 톰캣의 시작과 종료와 일치한다.

6.3. 싱글톤 vs 스프링 싱글톤

- 싱글톤

```
public class Person {

    private static Person instance;

    private Person() {
        throw new IllegalStateException("Private Constructor");
    }

    public static Person getInstance() {
        if (instance == null) {
            instance = new Person();
        }
        return instance;
    }
}

public class Singleton {
    private Singleton(){}

    public static Singleton getInstance() {
        return LazyHolder.INSTANCE;
    }

    private static class LazyHolder {
        private static final Singleton INSTANCE = new Singleton();
    }
}
```

- 싱글톤 패턴은 전역 상태를 이용할 수 있다는 장점이 있지만 다음과 같은 문제점으로 인해 안티 패턴으로도 불린다.
 - private 생성자를 갖고 있어 상속이 불가능하다.

- 싱글톤은 자신만이 객체를 생성할 수 있도록 `private`으로 제한한다. 하지만 상속을 통해 다형성을 적용하기 위해서는 기본 생성자가 필요하므로 `private`으로 인해 객체지향의 장점을 적용할 수 없다. 또한 싱글톤을 구현하기 위해서는 객체지향적이지 못한 `static` 필드와 `static` 메서드를 사용해야 한다.
- 테스트하기 힘들다.
 - 싱글톤은 생성 방식이 제한적이기 때문에 Mock 객체로 대체하기 어려우며, 동적으로 객체를 주입하기도 힘들다.
- 서버 환경에서는 싱글톤이 1개만 생성됨을 보장하지 못한다.
 - 서버에서 클래스 로더를 어떻게 구성하느냐에 따라 싱글톤 클래스임에도 불구하고 1개 이상의 객체가 만들어질 수 있다. 따라서 자바 언어를 이용한 싱글톤 기법은 서버 환경에서 싱글톤이 꼭 보장된다고 볼 수 없다. 또한 여러 개의 JVM에 분산돼서 설치되는 경우 독립적으로 객체가 생성된다.
- 전역 상태를 만들 수 있기 때문에 바람직하지 못하다.
 - 싱글톤의 정적 메서드를 사용하면 언제든지 해당 객체를 사용할 수 있고, 전역 상태로 사용되기 쉽다. 아무 객체나 자유롭게 접근하고 수정하며 공유되는 전역 상태는 객체지향 프로그래밍에서 권장하지 않는다.

6.4. 스프링 싱글톤

- 객체의 생성을 스프링에 위임함으로써 스프링 컨테이너가 관리하여 자바 언어 레벨에서 직접 구현하기 위한 내용들이 모두 제거되어 앞선 싱글톤의 모든 단점들이 제거된다.
 - `private` 생성자가 필요 없어 상속이 가능해진다.
 - 테스트하기 편하다.
 - 프레임워크를 통해 1개의 객체 생성을 보장받을 수 있다.
 - 객체지향적으로 개발할 수 있다.

7. Annotation

- Annotation은 프로그램에게 추가적인 정보를 제공하는 메타데이터이다.
- 자바 코드에 특별한 의미를 부여한 주석으로 컴파일러를 위한 정보를 제공하기 위한 용도
- 동작 순서
 - 애노테이션 정의

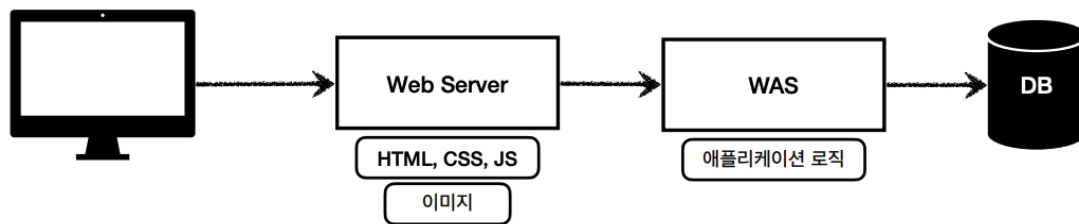
- 원하는 위치에 배치
- 코드가 실행되는 중에 Reflection을 이용하여 추가 정보를 획득하여 기능 실시
- Reflection
 - Reflection을 사용하면 컴파일 타임에 인터페이스, 필드, 메소드의 이름을 알지 못해도 실행 중에 클래스, 인터페이스, 필드 및 메소드에 접근할 수 있다. 또한 새로운 객체의 인스턴스화 및 메소드 호출을 허용한다.
 - Annotation 자체는 아무런 동작을 가지지 않는 단순한 표식일 뿐이지만, Reflection을 이용하면 Annotation의 적용 여부와 엘리먼트 값을 읽고 처리할 수 있다.
 - Spring 컨테이너(BeanFactory)에서 객체가 호출되면 객체의 인스턴스를 생성하게 되는데 이 때 필요하게 된다. 즉, 프레임워크에서 유연성있는 동작을 위해 쓰인다.
 - Reflection을 이용하면 Annotation 지정만으로도 원하는 클래스를 주입할 수 있다.

7.1. 각종 애노테이션

- @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 라이브러리등을 bean으로 만들려할 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 @Qualifier을 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
-

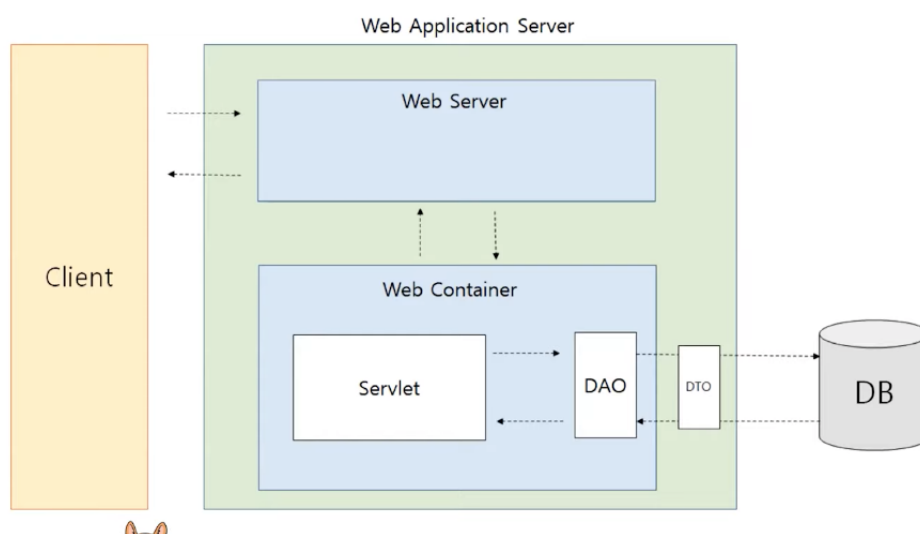
8. 웹 서버와 웹 애플리케이션 서버



- 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.

9. 서블릿과 MVC 패턴

9.1. 서블릿

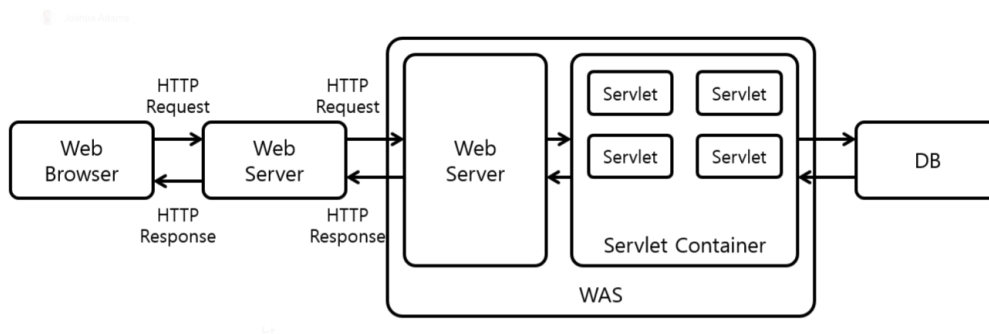


- 서블릿은 WAS 안에서 동적인 페이지를 만드는데 사용되는 서버 프로그램이다. 서블릿이 존재하기 전에는 요청이 들어오면 HTTP 요청 메시지를 파싱하는 것부터 여러 부가 작업을 개발자가 수행해야 했다. 하지만 서블릿이 나오면서 부가적인 작업을 대신해주게 되었고, 개발자는 실질적인 메인 로직에만 집중 할 수 있게 되었다.

9.2. 서블릿 컨테이너

- 앞서 스프링 컨테이너와 비슷하게 서블릿 컨테이너는 서블릿의 생명주기를 관리한다.
- init: 서블릿 초기화
- service: HTTP 요청 유형을 확인하고 맞게 doGet, doPost, doPut 등 메서드를 호출하여 요청 처리
- destroy: 서블릿 제거
- 서블릿 객체도 싱글톤 으로 관리되기 때문에 최초 요청 시점에 서블릿 객체를 초기화해서 서블릿 컨테이너에 보관하고 이후에는 같은 서블릿을 공유해서 사용한다.

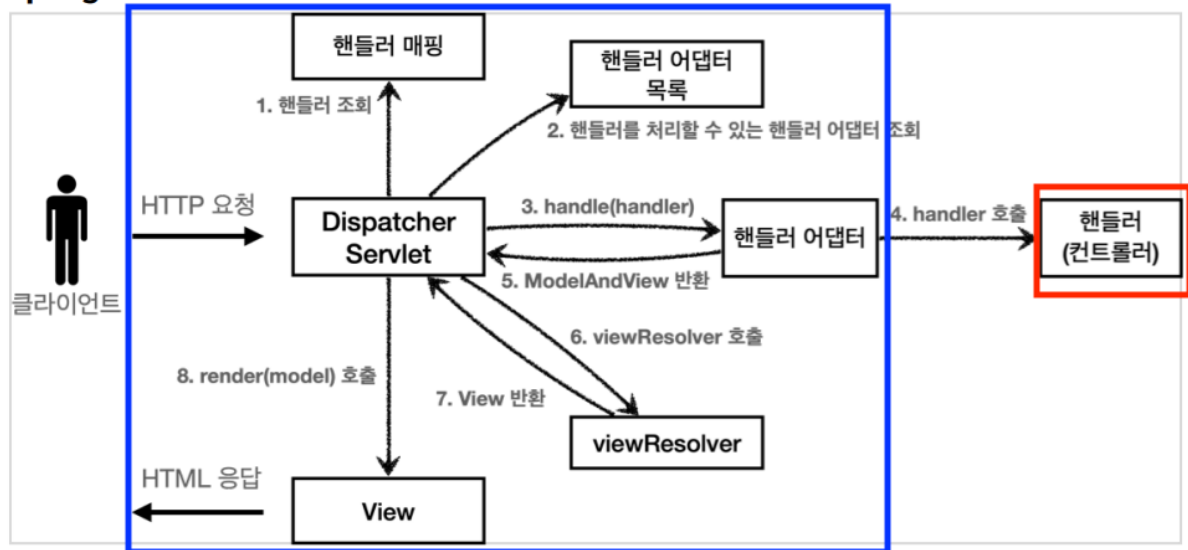
9.3. 요청 시 동작 과정



1. 사용자가 URL을 클릭하면 HTTP Request를 Servlet Container로 보낸다.
2. Servlet Container는 스레드 풀에서 스레드를 꺼내 할당해주고 HttpServletRequest, HttpServletResponse 두 객체를 생성한다.
3. 사용자가 요청한 URL을 분석하여 어느 서블릿에 대한 요청인지 찾는다.
4. 서블릿 컨테이너에 존재하지 않으면 초기화하고 있다면 가져와서 service() 메서드를 호출한다.
 - Spring MVC의 경우 DispatcherServlet이 초기화되고 호출된다.
5. service 메서드가 수행이 끝나면 HttpServletResponse 객체에 응답을 보낸다.

10. MVC 패턴

SpringMVC 구조



1. 핸들러 조회

- 핸들러 매핑을 통해 요청 URL에 매핑된 핸들러(컨트롤러)를 조회한다.

2. 핸들러 어댑터 조회

- 핸들러를 실행할 수 있는 핸들러 어댑터를 조회한다.

3. 핸들러 어댑터 실행

- 조회한 핸들러(컨트롤러)를 인자로 핸들러 어댑터에 넘겨서 핸들러를 실행시킨다.

4. ModelAndView 반환

- 핸들러(컨트롤러)가 로직을 수행하고 반환하는 정보로 ModelAndView로 변환해서 반환한다.

5. viewResolver 호출

- 적절한 viewResolver를 찾고 해당 viewResolver를 호출한다.
- RestController라면 이 과정과 이후 과정 없이 컨버터를 이용해 바로 결과값을 리턴한다.

6. View 반환

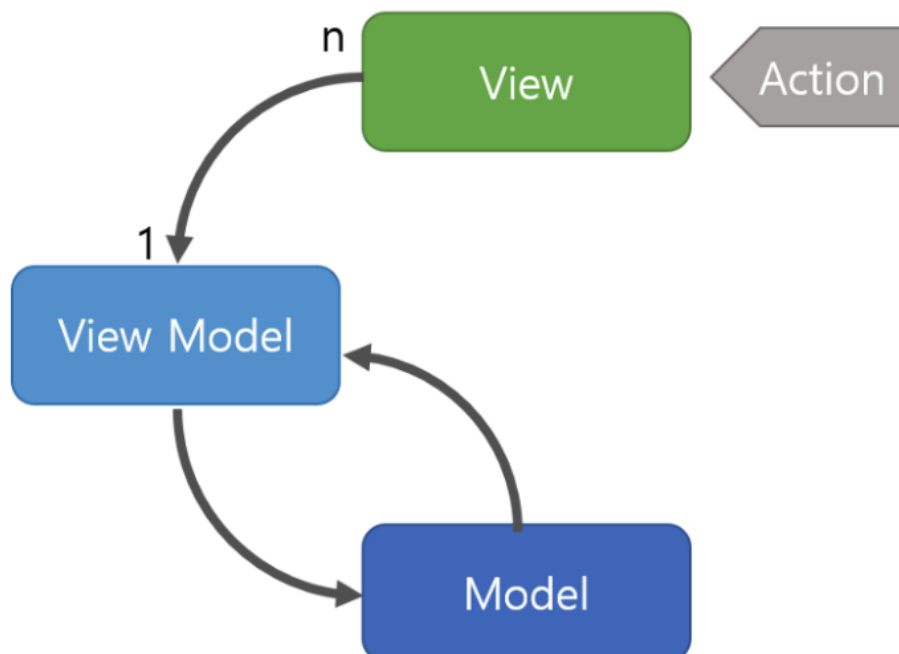
- viewResolver는 뷰의 논리 이름을 물리 이름으로 바꾸고, 랜더링 역할을 담당하는 뷰 객체를 반환한다.

7. 뷰 랜더링

- 뷰를 통해서 뷰를 렌더링한다.

10.1. 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 도입되었다.
- MVVM 패턴



- View : 사용자에게 보여지는 UI 부분
 - View Model : View를 표현하기 위해 만든 Model

- View를 나타내주기 위한 Model이면서 동시에 View를 나타내기 위한 데이터를 처리하는 부분
 - Model : 애플리케이션에서 사용되는 데이터와 그 데이터를 처리하는 부분
- 동작 과정
 1. 사용자의 Action들은 View를 통해 들어온다.
 2. View에 Action이 들어오면, Command 패턴으로 View Model에 Action을 전달한다.
 3. View Model은 Model에게 데이터를 요청한다.
 4. Model은 View Model에게 요청받은 데이터를 응답한다.
 5. View Model은 응답 받은 데이터를 가공하여 저장한다.
 6. View는 View Model과 Data Binding을 하여 화면에 나타낸다.
- MVC 패턴은 View와 Model 사이의 의존성이 높기 때문에 애플리케이션이 커질수록 복잡해지고 유지보수가 어려워지게 되는데, MVVM 패턴은 View와 Model 사이에 의존성이 없다.

10.2. 어떻게 하나의 컨트롤러로 여러 요청을 받을까?

- 컨트롤러는 기본적으로 컴포넌트 스캔되면서 스프링 빈 컨테이너에 올라가있고 싱글톤 패턴으로 구현되어있기 때문에 여러 스레드의 요청이 들어와도 하나의 컨트롤러 객체를 공유하면서 처리한다. 여기서 주의할 점은 싱글톤 패턴으로 구현되어 있어 있다는 것은 Thread-Safe하지 않다는 의미이므로 상태를 저장하는 코드가 없게 Stateless하게 설계해야 한다. 결과적으로 내부에는 상태가 존재하지 않으니 메서드에 대한 정보만 같이 공유해서 쓰는 것이다.

10.3. 특별한 설정이 없다면 싱글톤 패턴인데 멀티스레드 환경에 어떤 문제가 생길까?

- 멀티스레드 환경에서 싱글톤에서 문제가 생겼다면 메서드를 호출하는 환경이 스레드 세이프하게 구현되지 않았던가, 싱글톤 패턴으로 생성되는 객체가 전역변수를 가졌기 때문이다. 싱글톤 패턴은 하나의 객체를 공유하기 때문에 전역변수 같은 것은 되도록이면 사용하지 않아야 한다. 변수의 공유로 인한 문제라면 지역변수로 해결 수 있다면 지역변수로 해결한다. 반면에 지역 변수로 해결할 수 없는 경우라면 ThreadLocal 을 사용해 해결한다. 메서드 자체에 접근을 막아야 한다면 synchronized 키워드 로 묶어서 동기화시킬 수 있다. 하지만 synchronized는 성능상 이슈가 있기 때문에 특정 블록만 잡도록 하는 것이 좋다.

11. Thread-safe하게 싱글톤 구현(LazyHolder)

```
public class Singleton {
    private Singleton(){}

    public static Singleton getInstance() {
        return LazyHolder.INSTANCE;
    }

    private static class LazyHolder {
        private static final Singleton INSTANCE = new Singleton();
    }
}
```

- 이 방법은 static영역에 초기화를 하지만 객체가 필요한 시점까지 초기화를 미루는 방식이다. Singleton 클래스 로딩 시 LazyHolder 클래스는 변수가 없기 때문에 초기화하지 않는다. Singleton 클래스의 getInstance 메서드에서 LazyHolder.INSTANCE를 참조하는 순간 클래스가 로딩되며 초기화가 진행되는데 클래스를 로딩하고 초기화하는 시점은 Thread-safe를 보장한다.

12. AOP(Aspect Oriented Programming)

- 관점 지향 프로그래밍으로 공통 관심 사항과 핵심 관심 사항을 분리 하는 것을 의미한다. 소스 코드에서 여러 번 반복해서 사용하는 코드(흩어진 관심사)를 Aspect로 모듈화 하여 핵심 로직에서 분리해 재사용하는 것이라고 볼 수 있다. 여러 객체에 공통으로 적용할 수 있는 기능을 구분함으로써 재사용성을 높여주는 프로그래밍 기법이다. 특정 로직(로그, 성능테스트, 권한)을 모든 메서드에 적용하고 싶을 때, 일일이 추가하는 것이 아니라 로직을 만들어서 적용할 수 있다. 따라서, 비즈니스 로직 앞/뒤에 공통 관심 사항을 수행해 중복 코드를 줄인다.

- Aspect

- 흩어진 관심사를 모듈화 한 것

- 모듈 : 외부에서 재사용할 수 있는 패키지들을 묶은 것

- advice + pointcut을 모듈화 한 것

- Target

- advice의 대상이 되는 객체

- Pointcut으로 결정

- Advice

- 실질적인 부가 기능 로직을 정의하는 곳
- 특정 조인 포인트에서 Aspect에 의해 취해지는 조치
- Join point
 - 추상적인 개념 으로 advice가 적용될 수 있는 모든 위치
 - ex) 메서드 실행 시점, 생성자 호출 시점, 필드 값 접근 시점 등등..
 - 스프링 AOP는 프록시 방식을 사용하므로 조인 포인트는 항상 메서드 실행 지점
- Pointcut
 - 조인 포인트 중에서 advice가 적용될 위치를 선별하는 기능
 - 스프링 AOP는 프록시 기반이기 때문에 조인 포인트가 메서드 실행 시점 뿐이 없고 포인트컷도 메서드 실행 시점만 가능
- Advisor
 - 스프링 AOP에서만 사용되는 용어로 advice + pointcut 한 쌍
- Weaving
 - pointcut으로 결정한 타겟의 join point에 advice를 적용하는 것
- AOP 프록시
 - AOP 기능을 구현하기 위해 만든 프록시 객체
 - 스프링에서 AOP 프록시는 JDK 동적 프록시 또는 CGLIB 프록시

12.1. AOP 적용 방식

- 컴파일 시점
 - .java 파일을 컴파일러를 통해 .class를 만드는 시점에 부가 기능 로직을 추가
 - 모든 지점에 적용 가능
 - AspectJ가 제공하는 특별한 컴파일러를 사용해야 하기 때문에 특별할 컴파일러가 필요한 점과 복잡하다는 단점이 있다.
- 클래스 로딩 시점
 - .class 파일을 JVM 내부의 클래스 로더에 보관하기 전에 조작하여 부가 기능 로직 추가
 - 모든 지점에 적용 가능

- 특별한 옵션과 클래스 로더 조작기를 지정해야하므로 운영하기 어렵다.
 - 런타임 시점
 - 스프링이 사용하는 방식
 - 컴파일이 끝나고 클래스 로더에 이미 다 올라가 자바가 실행된 다음에 동작하는 런타임 방식
 - 실제 대상 코드는 그대로 유지되고 프록시를 통해 부가 기능이 적용
 - 프록시는 메서드 오버라이딩 개념으로 동작하기 때문에 메서드에만 적용 가능 - > 스프링 빈에만 AOP를 적용 가능
 - 특별한 컴파일러나, 복잡한 옵션, 클래스 로더 조작기를 사용하지 않아도 스프링만 있으면 AOP를 적용할 수 있기 때문에 스프링 AOP는 런타임 방식을 사용
-

13. POJO

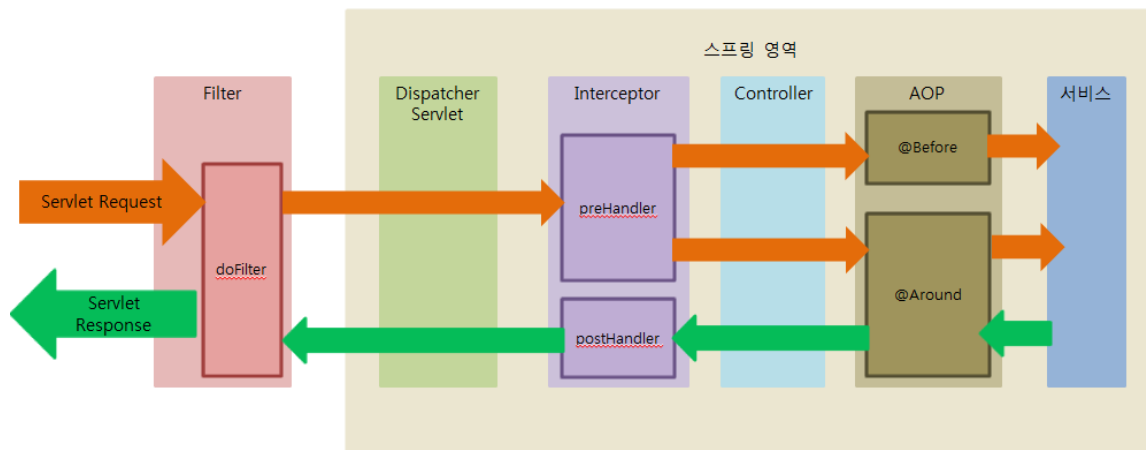
- 평범한 구식 자바 객체로, 프레임워크 인터페이스나 클래스를 구현하거나 확장하지 않은 단순 클래스를 의미한다. JAVA에서 제공하는 API외에는 종속되지 않아 코드가 간결하고 테스트 자동화에 유리하다. Spring에서는 도메인과 비즈니스 로직을 수행하는 대상이 POJO 대상이 될 수 있다.
 - 도메인이란 화면, UI, 기술 인프라 등등의 영역을 제외한 시스템이 구현해야 하는 핵심 비즈니스 업무 영역을 말한다. 컨트롤러는 도메인이 아니고, 엔티티와 리포지토리 등이 도메인으로 볼 수 있다.
-

14. DAO, DTO

- DAO
 - DB 데이터를 조회하거나 조작하는 기능을 전담하는 객체
 - DB 접근 로직과 비즈니스 로직을 분리하기 위해서 사용
- DTO
 - 계층간의 데이터 교환을 위한 객체
 - 로직을 갖지 않는 순수 데이터 객체로 getter, setter만 포함
 - VO
 - DTO와 동일한 개념

- Read Only로 수정 불가
- getter, setter 이외의 추가 로직 포함 가능

15. Filter, Interceptor



- Filter
 - Dispatcher Servlet에 요청이 전달되기 전/후에 url 패턴에 맞는 모든 요청에 대해 부가작업을 처리하는 기능을 제공하는 것
 - 톰캣과 같은 웹 컨테이너(웹 애플리케이션 WAS 단)에서 동작 하기 때문에 Spring 과 무관한 자원에 대해 동작
 - Spring Context 외부에서 동작하므로 ErrorController 에서 예외 처리
 - init
 - 필터 객체를 초기화하고 서비스에 추가하기위한 메서드
 - 웹 컨테이너(WAS 단)에서 1회 init 메서드를 호출하여 필터 객체를 초기화하면 이후 요청들은 doFilter를 통해 전/후 처리가 된다.
 - doFilter
 - url-pattern에 맞는 모든 HTTP 요청이 디스패처 서블릿으로 전달되기 전/후에 웹 컨테이너에 의해 실행되는 메서드
 - doFilter의 파라미터로 FilterChain이 있는데, FilterChain의 doFilter 를 통해 다음 대상으로 요청을 전달한다.
 - destroy

- 필터 객체를 서비스에서 제거하고 사용하는 자원을 반환하는 메서드
 - 웹 컨테이너에 의해 1번 호출된다.
- 참고로 필터를 추가하기 위해서는 javax.servlet의 Filter 인터페이스를 구현하면 된다.
- **Interceptor**
 - Spring이 제공하는 기술로, Dispatcher Servlet이 컨트롤러를 호출하기 전과 후에 요청과 응답을 참조하거나 가공할 수 있는 기능을 제공하는 것
 - 스프링 컨텍스트에서 동작
 - Spring Context 내부에서 동작하므로 @ControllerAdvice 을 사용하여 예외 처리
 - preHandle
 - 컨트롤러가 호출되기 전에 실행되어 컨트롤러 이전에 처리해야 하는 전처리 작업이나 요청 정보를 가공하거나 추가하는 경우에 사용할 수 있다.
 - postHandle
 - 컨트롤러 호출된 후에 실행되어 컨트롤러 이후에 처리해야하는 후처리 작업이 있을 때 사용할 수 있다.
 - 보통 컨트롤러가 반환하는 ModelAndView 타입의 정보가 제공되는데, 최근에는 Json 형태로 데이터를 제공하는 REST API 기반의 컨트롤러가 사용되면서 잘 사용하지 않는다.
 - afterCompletion
 - 모든 뷰에서 최종 결과를 생성하는 일을 포함해 모든 작업이 완료된 후에 실행된다.
 - 요청 처리 중에 사용한 리소스를 반환할 때 사용하기 적합하다.
 - 참고로 인터셉터를 추가하기 위해서는 org.springframework.web.servlet의 HandlerInterceptor 인터페이스를 구현하면 된다.

15.1. 실행 과정

1. 서버 실행 시 Servlet이 올라오는 동안 init 후 doFilter 실행
2. Dispatcher Servlet을 지나쳐 Interceptor의 PreHandler 실행
3. 컨트롤러를 거쳐 내부 로직 수행 후, Interceptor의 PostHandler 실행
4. doFilter 실행

5. Servlet 종료 시 destory

15.2. Filter vs Interceptor

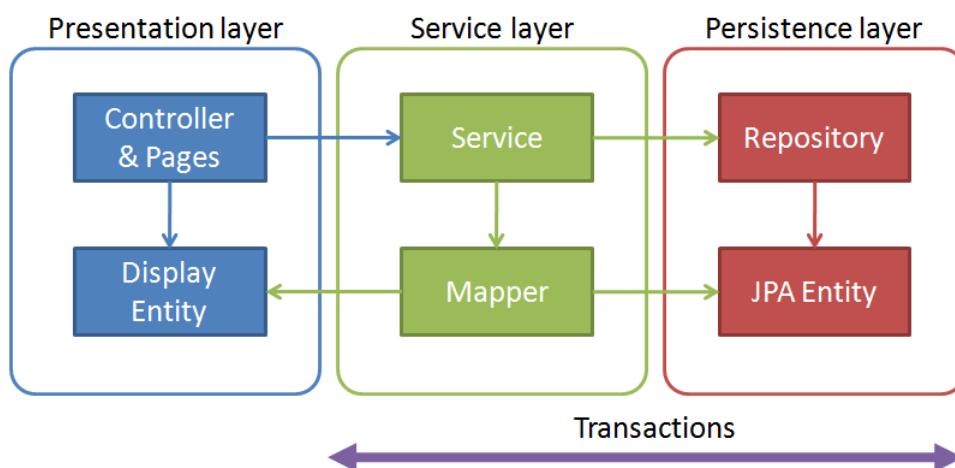
대상	필터(Filter)	인터셉터(Interceptor)
관리 컨테이너	웹 컨테이너	스프링 컨테이너
Request/Response 조작 여부	가능	불가능
용도	+ 보안 관련 공통 작업+ 이미지/데이터 압축 및 문자열 인코딩+ 모든 요청에 대한 로깅 또는 감사	+ 인증/인가 등과 같은 공통 작업+ Controller로 넘겨주는 정보의 가공+ API 호출에 대한 로깅 또는 감사

- 인터셉터가 조작 여부가 불가능하다는 것은 HttpServletRequest, HttpServletResponse 객체를 제공받으므로 객체 자체는 조작할 수 없다는 의미이고, 내부 값들은 조작할 수 있다.

15.3. AOP, Interceptor

- AOP와 Interceptor가 비슷한 기능을 수행한다고 할 수 있는데 이들의 사용을 구분 짓는 방법은 파라미터 이다. 모든 메서드의 파라미터와 타입은 제각각이기 때문에 이에 대해 AOP를 적용하게 되면 부가 작업들이 생기게 된다. 반면에 Interceptor의 경우 HttpServletRequest, HttpServletResponse를 파라미터로 사용하여 부가 작업이 필요하지 않다. 또한, 인터셉터는 Controller 앞에서 동작하고 AOP는 Service 앞에서 동작한다.

16. 레이어드 아키텍처



- Spring은 레이어드 아키텍처로 이루어져있다. 하나의 레이어는 자신의 고유 역할을 수행하고, 인접한 다른 레이어에 무언가를 요청하거나 응답한다. 그밖의 다른 레이어는 신경 쓸 필요가 없기 때문에 각 레이어는 자신의 역할에 충실할 수 있다. 따라서 시스템 전체를 수정하지 않고 특정한 레이어의 기능을 개선하거나 교체할 수 있기 때문에 재사용성이 좋고 유지 보수하기에도 유리하다. 또한, 레이어별로 테스트 구현이 편해지고 코드가독성도 높아진다.
 - 프레젠테이션 레이어 : Controller
 - view를 담당하는 부분으로, 클라이언트와 직접적으로 맞는 부분
 - 서비스 레이어 : Service
 - 비즈니스 핵심 로직을 처리하는 부분
 - Service 객체라는 것은 하나의 트랜잭션으로 구성되어 작동
 - Persistence Layer
 - 데이터 관련 처리를 담당하는 부분
-

17. OSIV

OSIV는 Open Session In View의 약자로 영속성 컨텍스트를 뷰단까지 열어준다 는 뜻이다.스프링의 OSIV는 프레젠테이션 계층에는 트랜잭션이 없기 때문에 엔티티를 수정할 수 없지만 영속성 컨텍스트가 살아있기 때문에 지연로딩이 가능합니다.언뜻보면 만능처럼 보이지만 단점이 있다.

- 같은 영속성 컨텍스트를 여러 트랜잭션이 공유할 수 있다.
 - 프레젠테이션 계층에서 엔티티를 수정하고 트랜잭션(서비스계층)으로 들어오면 엔티티가 수정된다.
 - 프레젠테이션 계층에서 지연로딩에 의한 SQL이 실행되기 때문에 성능 튜닝시 확인해야할 부분이 넓어진다.
-

18. 커넥션 풀

- 애플리케이션이 데이터베이스를 사용하기 위해서는 커넥션을 맺어야 한다.커넥션을 생성하고 소멸시키는 비용이 크기 때문에 커넥션 풀을 세팅해두고(기본 10) 애플리케이션이 시작하는 시점에 커넥션을 미리 다 만들어 놓고 이를 재활용하면서 사용한다.스프링 부트 2.0부터는 hikariCP를 기본 커넥션 풀로 사용한다.
-

19. DataSource

- 커넥션 관련 기술이 여러 개 등장하면서 코드레벨에서는 서로 다르지만 논리적으로는 커넥션을 획득하는 역할을 하기 때문에 이를 추상화 시킨 것이 DataSource이다. 실질적인 로직은 DataSource에 의존하도록 하고 구현 기술이 바뀔때마다 DataSource의 구현체만 바꾸면 되므로 재사용성과 확장성을 높일 수 있다. 커넥션 관련 기술은 커넥션을 계속 신규 생성하는 DriverManager, DBCP2 커넥션 풀, HikariCP 커넥션 풀 등이 있다. DriverManager는 DataSource를 구현하지 않아서 스프링에서 DriverManagerDataSource라는 구현 클래스를 제공한다.

19.1. 트랜잭션을 추상화하는 이유

- DataSource와 같은 맥락이다. 다양한 데이터 접근 기술이 등장하면서 코드레벨에서는 서로 다르지만 논리적으로는 같은 기능을 수행하기 때문에 트랜잭션을 추상화했다. 스프링 트랜잭션 추상화 클래스는 PlatformTransactionManager이다. 보통 트랜잭션 매니저 라고 부른다. 다양한 접근 기술로는 JDBC, JPA, 하이버네이트 등이 있다.

19.2. 트랜잭션 동기화 매니저

- 보통 서비스 단에서 트랜잭션을 시작하고 끝낸다. 그렇다면 하나의 트랜잭션 내에서는 같은 커넥션을 사용 해야 하는데 과정이 다음과 같다.
 1. 서비스단에서 트랜잭션이 시작하면 트랜잭션 매니저가 커넥션을 생성하고(풀을 사용하면 풀에서 가져오고) autoCommit을 false로 세팅한 뒤 트랜잭션 동기화 매니저의 스레드 로컬에 커넥션을 보관한다.
 2. 이후 리포지토리 계층에서는 트랜잭션 동기화 매니저의 스레드 로컬에서 해당 커넥션을 가져와서 사용한다.
 3. 서비스 단에서 트랜잭션을 종료할 때는 트랜잭션 동기화 매니저에서 해당 커넥션을 가져와 커밋 또는 롤백을 수행하고 리소스를 정리하고 커넥션을 커넥션 풀에 반환한다.
- 하나의 트랜잭션에서 같은 커넥션을 사용하도록 도움을 주는 기능을 제공한다고 보면 된다.

19.3. 선언적 트랜잭션 vs 프로그래밍 방식 트랜잭션

- 선언적 트랜잭션은 @Transactional을 의미한다. 프로그래밍 방식 트랜잭션은 트랜잭션 매니저나 트랜잭션 템플릿 등을 직접 사용해서 프로그래밍 코드를 작성하는 방식이다. @Transactional을 사용하면 프록시(메서드 오버라이딩 개념)를 사용하기 때문에

추가적인 코드를 작성할 필요 없이 간편하게 사용할 수 있으므로 대부분 선언적 트랜잭션을 사용한다.

19.4. @Transactional

- @Transactional AOP로 구성되어 있다. 즉, 프록시로 동작하므로 오버라이딩 개념으로 동작한다. 메서드에 @Transactional을 붙이면 해당 클래스가 빈으로 등록될 때 @Transactional이 붙은 메서드만 트랜잭션 처리되는 메서드로 오버라이딩 한 프록시 객체가 빈으로 등록된다. 클래스에 붙으면 클래스의 전체 public 메서드에 트랜잭션 처리가 된 프록시가 빈으로 등록된다. public이 아닌 다른 접근제한자가 붙은 메서드의 경우는 트랜잭션처리가 되지 않는데 이유는 프록시가 오버라이딩 개념이기 때문에 public으로 열려있지 않고 private 메서드 같은 경우에는 적용이 불가능하다.

19.5. 내부 호출 문제

- @Transactional이 붙은 클래스는 프록시로 빈으로 등록된다. 따라서 주입받은 객체를 사용할 경우 프록시가 들어오게 되고 접근 시 프록시 객체를 통한 호출이 이뤄진다.

```
public class UserService {

    @Transactional
    public void createUserListWithTrans(){
        for (int i = 0; i < 10; i++) {
            createUser(i);
        }

        throw new RuntimeException();
    }

    @Transactional
    public User createUser(int index){
        User user = User.builder()
            .name("testname::"+index)
            .email("testemail::"+index)
            .build();

        userRepository.save(user);
        return user;
    }
}
```

- createUserListWithTrans에서 createUser를 호출한다. 둘다 @Transactional이 붙어 있다. 서로 트랜잭션이 붙어있기에 createUser에서 save 처리를 했으므로 예외가 발생하더라도 다 저장되었을 것이라고 생각할 수 있지만 틀렸다. 이유는 트랜잭션이 붙은 상태로 동작하려면 프록시를 통해 접근해야 하는데 위는 그냥 코드 자체를 호출한 것이기

때문이다. 즉, 프록시를 사용하려면 userService.XXX 형식으로 호출해야된다는 뜻이다. 따라서 createUserListWithTrans를 호출할 경우 아래와 프록시 객체로 동작한다.

```
public void createUserListWithTrans(){
    EntityTransaction tx = em.getTransaction();
    tx.begin();

    super.createUserListWithTrans();

    tx.commit();
}
```

- createUserListWithTrans에 붙은 @Transactional만 동작하게 된다. 즉, 하나의 트랜잭션 안에서 동작하게 되는 것이다. 만약 트랜잭션이 붙은 프록시를 호출해서 사용하다가 다른 트랜잭션이 붙은 프록시 클래스를 호출할 경우 이때부터는 트랜잭션 전파 속성에 따라 트랜잭션이 동작한다.

```
public class UserService {

    public void createUserListWithTrans(){
        for (int i = 0; i < 10; i++) {
            createUser(i);
        }

        throw new RuntimeException();
    }

    @Transactional
    public User createUser(int index){
        User user = User.builder()
            .name("testname:"+index)
            .email("testemail:"+index)
            .build();

        userRepository.save(user);
        return user;
    }
}

// AopApplication.java
userService.createUserListWithoutTrans();
```

- Userservice는 createUser에 붙은 @Transactional 때문에 프록시가 빈으로 등록된다. userService는 프록시이지만 createUserListWithTrans는 트랜잭션이 붙어있지 않으므로 트랜잭션이 처리되지 않는 것을 호출하게 되고 내부적으로 createUser를 호출하게 되어 트랜잭션 없이 호출하게 된다. 실행결과를 보면 user가 10개 생성되게 되는데 이는 @Transactional이 없기 때문에 createUser가 각각 insert하면서 DB의 기본

설정대로 auto commit으로 인한 동작 결과이다.(Transactional은 auto commit을 false로 하고 마지막에 commit한다.)

19.6. Propagation 전파단계

- 전파 단계는 트랜잭션 동작 도중 다른 트랜잭션을 호출하는 상황에서 선택할 수 있는 옵션이다. 디폴트는 required로 트랜잭션이 없으면 새로 생성하고, 부모 트랜잭션 내에 실행하면 부모 트랜잭션 내에서 수행한다.
-

17. ORM

- JDBC API
 - JAVA 진영 Database 연결 표준 인터페이스
- Spring JDBC
 - template을 통해 데이터를 꺼내면서 한단계 더 추상화
- MyBatis
 - SQL 분리를 목적으로 XML로 관리하는 방식
- ORM(Object Relational Mapping)
 - 객체지향 코드와 데이터 중심 데이터베이스의 패러다임 불일치를 해결하기 위해 나온 기술
 - 객체와 관계형 데이터베이스를 맵핑하는 기술
- JPA
 - 자바 ORM의 표준 API 명세를 JPA 인터페이스라고 한다.
- Hibernate
 - JPA 인터페이스의 구현체
- Spring Data JPA
 - JPA에 Repository를 추가하여 한단계 더 추상화한 것
- 추상화
 - 사물들의 공통적인 특징을 파악해서 하나의 개념(집합)으로 다루는 것
 - 장점
 - 공통 사항이 한 곳에서 관리되기 때문에 개발 및 유지보수에 좋다.

- 자식 클래스에서 추상 메서드를 반드시 구현하도록 강요한다.
 - 단점
 - 추상 클래스의 경우 상속으로 인한 단점을 가져오게 된다. 실질적으로 부모 클래스의 기능을 사용하지 않는 부분이 있어도 자식 클래스는 이를 갖고 있어야 한다.
 - 상속이 깊어질수록 계층이 많아지기 때문에 이해하기 어려울 수 있다.
-

18. JPA

18.1. 영속성 컨텍스트

- 영속성 컨텍스트란 엔티티를 영구 저장하는 환경을 의미 한다.
- 생명 주기
 - 영속
 - 영속성 컨텍스트에 저장된 상태
 - 준영속
 - 영속성 컨텍스트에 저장되었다가 분리된 상태
 - 비영속
 - 영속성 컨텍스트와 전혀 관계없는 상태
 - 삭제
 - 삭제된 상태

18.2. 영속성 컨텍스트의 이점

- 1차 캐시
 - 조회가 가능하며 1차 캐시에 없으면 DB에서 조회하여 1차 캐시로 가져온다.
- 동일성 보장
 - == 비교가 가능하다.
- 쓰기 지연
 - 트랜잭션 커밋 전까지 SQL을 바로 보내지 않고 모아서 보낼 수 있다.
- 변경 감지(더티 체크)

- 1차 캐시에 들어온 데이터를 스텝샷 찍어두고 커밋시점에 비교하여 update SQL을 생성한다.
- 지연 로딩
 - 엔티티안에서 엔티티를 불러올 때 사용 시점에 쿼리를 날려 가져올 수 있다.

18.3. 저장 동작 과정

1. JPA는 트랜잭션 실행 단위안에서 동작한다.
2. 객체를 영속성 컨텍스트에 등록한다.(1차 캐시) insert 쿼리의 경우 쓰기 지연으로 SQL 저장소에 저장된다.
3. 트랜잭션이 끝나는 시점에 쓰기 지연 SQL 저장소에 있는 쿼리문들이 flush되고 트랜잭션이 커밋된다.

18.4. 수정 과정

1. 1차 캐시에 등록된 엔티티와 스냅샷을 비교해서 변경내역을 확인하고 update 쿼리를 쓰기 지연 저장소에 저장한다.
2. 트랜잭션이 끝나는 시점에 쿼리들이 flush되고 트랜잭션이 커밋된다.

18.5. N+1 문제

- N+1 문제는 하위 엔티티들을 첫 쿼리 실행 시 한 번에 가져오지 않고, 지연 로딩으로 프록시가 들어온 상태에서 후에 이것을 사용하면서 조회 쿼리가 다시 나가게 되어 발생하는 문제이다. 예를 들어, 학생(N)과 팀(1)에서 양방향관계를 갖고 DB에서 팀을 10개를 꺼낸다고 가정해보자. 첫 쿼리는 팀 10개를 꺼내는 쿼리가 하나의 쿼리가 나가게 되고 이 때, 팀 기준 OneToMany이므로 Lazy로 동작하여 학생은 프록시로 들어오게 된다. 이게 가져온 각각의 팀에 대해 학생들에게 접근하는 로직이 있다면 각 팀마다 학생들을 조회 하는 쿼리가 1개씩 더 나가게 된다. 즉, 10개의 쿼리가 더 나가게 된다. 그래서 1개의 쿼리가 나가고 이후에 N개의 쿼리가 더 나간다고 해서 N+1 문제라고 한다. 이에 대한 해결책은 Fetch Join과 Batch Size가 있다. Fetch Join을 사용하면 Lazy로딩으로 프록시로 들어오던 것을 join으로 한 번에 땡겨올 수 있다. Batch Size는 N+1문제가 발생하던 것 처럼 프록시로 가져오고 학생들 가져오게 될 때 쿼리가 한번 더 나가게 되는데 이 때 in쿼리로 Batch size 개수만큼 가져온다. 가져온 팀이 10개이고 Batch size가 5라면, 최초에 학생을 가져오는 쿼리에서 where 조건문 in 쿼리로 5개의 team id값을 넣어서 쿼리를 날린다. 이렇게 되면 결과적으로 학생을 가져오는 쿼리는 2번이 나가게 되어 총 쿼리는 3(팀 가져오는 쿼리 + 학생 가져오는 쿼리)개의 쿼리가 나가게 된다. 참고로 @EntityGraph를 사용해도 Fetch join으로 가져올 수 있다. 위에서는 지연로딩으로 설

명했지만 팀을 가져올 때, 학생들을 즉시 로딩으로 설정해둬도 팀 땡겨오고 각 팀에 대한 학생도 땡겨오기 때문에 N+1 문제가 발생한다.

18.6. OneToOne 양방향 관계 Lazy 로딩 주의

- 이유는 프록시는 null을 감쌀 수 없기 때문에 프록시의 한계로 나타나는 문제이다.

USER_ID	CART_ID
1	5
2	6
3	null
4	7

USER 테이블

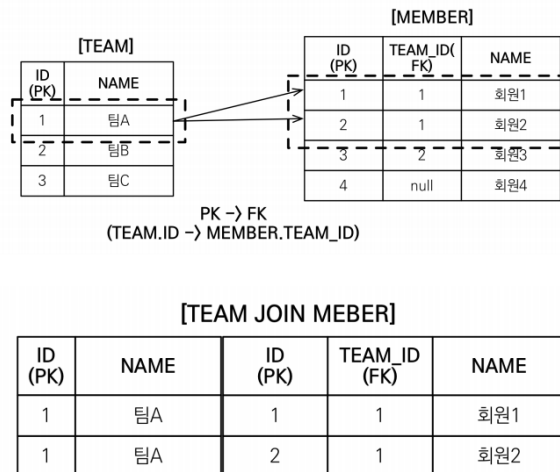
CART_ID
1
2
3
4

CART 테이블

- User과 Cart가 일대일 양방향관계이고 연관관계 주인은 User라고 가정해보자. Cart 테이블을 보면 Cart는 User_Id 값을 알지 못한다. 알기 위해서는 User 테이블을 조회해야 하는데 이렇게 되면 User 테이블을 조회해야하기 때문에 Lazy의 의미가 없어진다. 그래서 Lazy로 설정해도 Eager로 동작하는 것이다. 그렇다면 OneToMany의 경우에도 마찬가지로 아닐까 라고 생각할 수 있지만 OneToMany는 Lazy가 정상 동작한다. 이유는 컬렉션의 경우는 비어있다고 isEmpty로 표현이 가능하지만, OneToOne은 없다면 Null값이 들어가기 때문이다. 이러한 이유 때문에 OneToOne에서는 보통 optional = false로 지정할 수 있는(nullable이 허용되지 않는) 일대일 단방향 관계로 설계해서 Lazy로딩을 사용하는 것이 좋다.

18.7. OneToMany fetch join 데이터 뺏기기 문제

- ManyToOne의 경우 애초에 Many에 One을 끼워넣기 때문에 최대 Many의 데이터 개수만큼만 조회된다. 반면에 OneToMany의 경우 One에 Many를 끼워넣기 때문에 기존 One의 개수 만큼의 데이터가 아니라 더 많은 데이터가 조회된다.(뺏기기 된다.)



- 팀A에 학생1, 학생2가 연관된 데이터가 DB에 있다고 했을 때, 개발자의 의도는 팀A에 연결된 모든 Member를 모두 꺼내오는 식으로 페이징을 1로 줘서 쿼리를 oneToMany에서 fetch Join을 날리게 되면 DB단에서는 OneToMany이므로 Team쪽 데이터가 뺏겨지면서 (팀A, 회원1), (팀A, 회원2)로 구성되고 여기서 페이징하게 되면 (팀A, 회원1)의 데이터만 나오게 된다. 따라서 JPA는 이를 판단할 수 없기 때문에 애초에 나가는 쿼리를 살펴보면 페이징 쿼리가 제거되어 나가고 연관된 데이터를 전부다 끌고오는 쿼리가 나가게 된다. 그리고 JPA는 이를 다 메모리에 적재해서 페이징을 시작합니다. 결과적으로는 의도대로 동작할 지라도 엄청난 성능에 악영향을 주기 때문에 사용해서는 안 된다. 그리고 이를 경고하는 메시지가 콘솔에 찍힌다. 따라서 OneToMany에서 페이징 쿼리를 날리고 싶다면 batch size를 사용해야 한다.
- 페이징 쿼리를 사용하지 않고 OneToMany를 사용할 때 주의할 점도 있다. 앞서 설명했듯이 OneToMany에서 fetch Join을 하면 DB에서는 데이터가 뺏겨진 상태로 넘어온다. 이걸 애플리케이션 단에서 받으면 당연히 아무처리 없이 뺏겨진 상태로 받게 된다. 즉, (팀A, 회원1), (팀A, 회원2) 이렇게 받게 된다는 것이다. 하지만 쿼리에서 distinct를 명시하면 DB단에서는 당연히 행이 서로 다르기 때문에 distinct가 먹지 않지만 애플리케이션 단으로 데이터가 넘어오면 JPA에서 distinct로 식별자가 같은 것을 걸러서 컬렉션으로 꽂아주는 기능을 제공해준다. 따라서 OneToMany 관계에서 fetch join을 사용한다면 반드시 distinct를 명시해야 한다.

18.8. MultipleBagFetchException

- OneToOne, ManyToOne과 같이 단일 관계의 자식 테이블에는 Fetch Join을 써도 된다. 하지만 2개 이상의 OneToMany 자식 테이블에 Fetch Join을 선언했을 때 MultipleBagFetchException이 발생한다. 쉽게 말하면, One을 가져올 때 Fetch join으로 컬렉션을 2개 이상 같이 땡겨오면 문제가 발생한다. 해결책은 Batch Size를 이용하는 것이다. 데이터가 많은 쪽에는 Fetch join을 걸어주고, 나머지는 어쩔 수 없이 Lazy

Loading으로 땡겨오고 한 번 땡겨올 때 Batch Size로 인해 in쿼리로 많이 땡겨오는 식으로 처리한다.

18.9. fetch join 한계

- fetch join 대상에는 별칭을 줄 수 없다.
 - 하이버네이트는 허용하지만 가급적이면 사용하지 않는게 좋다.
 - fetch join은 나의 연관된 것들을 다 끌고오겠다는 의미로 설계된 것이기 때문에 대상을 where문과 on에서 사용하게 되면 필터링이 되므로 의도된 설계와 맞지 않는다.
- 둘 이상의 컬렉션은 fetch join 할 수 없다. 하나의 컬렉션과만 fetch join이 가능하다.
 - 둘 이상 컬렉션과 진행시 곱하기 곱하기가 되어 정합성이 맞지 않는다.
- OneToMany의 경우 페이징 불가능

18.10. 상속관계 매핑

- 상속관계 매핑
 - 객체간의 상속관계를 DB에 적용시키기 위한 작업
 - 부모 클래스에 @Inheritance 애노테이션 붙여서 상속관계 명시
 - JOINED 전략이 제일 합리적, 매우 간단하면 SINGLE_TABLE 전략 사용
 - 부모, 자식 클래스 모두 @Entity
 - 전략에 따라 부모테이블이 생성될 수도, 안될 수도 있다.
- @MappedSuperclass
 - 객체끼리 공통되는 속성(프로퍼티)를 뽑아서 만든 클래스(BaseEntity 처럼 시간 뽑을 때)
 - Item을 부모로 book, album 같이 포함되는 상속관계가 아니라 createDate같이 객체끼리 공통되는 필드가 겹칠 때 사용
 - 자바에서는 상속으로 사용하나 상속관계 매핑과 다르게 엔티티가 아니라 DB상에 올라가지 않음
- 임베디드 타입
 - MappedSuperclass와 유사하게 공통되는 속성을 뽑아서 만든 클래스로 @Embeddable 애노테이션을 붙인다.

- 사용하는 곳에서는 상속이 아니라 필드로 선언하고 위에 @Embedded 애노테이션을 붙인다.
- 테이블로 올라가지 않는다.
- MappedSuperclass와 임베디드 타입은 거의 똑같다고 보면 되는데 임베디드 타입은 위임이고, MappedSuperclass는 상속이다. 보통 상속보다는 위임이 좋기 때문에 위임을 선택하지만 편의상 경우에 따라 상속이 좋은 선택이 될 수도 있다.

```
-- 임베디드 타입
select m from Member m where m.traceDate.createdDate > ?

-- 상속
select m from Member m where m.createdDate > ?
```

- 위임의 경우 한 번 더 접근해야하는데 상속은 바로 접근할 수 있다. 또한 임베디드 타입은 서로 다른 엔티티가 공유하게 되면 SideEffect가 발생할 수 있다. (한 쪽에서 고치면 다른 쪽에서도 바뀜)

18.11. Id 값을 Long으로 사용하는 이유

- 범위
- Null값 허용
- 공식문서에서 nullable 한 값 사용 권장

18.12. QueryDsl을 사용하는 이유

- QueryDsl을 사용하면 컴파일 타임에 오류를 잡을 수 있고, 동적 쿼리를 쉽게 작성할 수 있다.
- 원하는 필드만 뽑아서 DTO로 만드는 기능도 지원한다.

19. spring Application을 구동할 때 메서드를 실행시키는 법

- CommandLineRunner, ApplicationRunner를 구현한 클래스를 만들어 실행시킨다.

20. 순환참조

- 순환참조가 발생되면 컴포넌트 간의 명확한 경계가 사라지고 연쇄적으로 변경에 의한 영향이 발생할 수 있다. 이로 인하여 개발과 유지보수 속도에 영향을 끼치고 예상치 못한 문제점을 만들어 낼 가능성도 높다. 이 후에 컴포넌트들을 분리해내도 어려워진다. 컴포넌트를 분리해내기 어렵다면 단기적으로 테스트하기가 어려워질 것이고, 장기적으로 협업하기 어려워지고 DDD나 MSA와 같은 개발 아키텍처를 구성할 수 없게 된다.

21. Spring batch

- 단발성으로 대용량 데이터를 처리하는 애플리케이션을 배치 애플리케이션이라고 하고 스프링 진영에는 스프링 배치가 있다.스프링 배치는 레이어 구조 3개로 구분된다.
- 애플리케이션 레이어
 - 개발자가 작성한 모든 배치 작업과 사용자 정의 코드를 포함한다.
- 코어 레이어
 - 배치 작업을 시작하고 제어하는데 필요한 핵심 런타임 클래스들을 포함 (jobLauncher, job, step, flow 등)한다.
- 인프라 레이어
 - 잡을 실행의 흐름과 처리를 위한 틀을 제공
 - 애플리케이션과 코어 모두 인프라 위에서 빌드된다.

21.1. 청크기반 방식

- itemReader, itemProcessor, itemWriter로 구성된다.
 - itemReader
 - Custor 기반 처리
 - 데이터를 호출하면 다음 커서로 이동하는 스트리밍 방식으로 데이터를 한 건씩 처리
 - 모든 결과를 메모리에 할당하기 때문에 메모리 사용량 증가
 - 모든 데이터를 처리할 때까지 커넥션 유지
 - 멀티스레드 환경에서 동시성 이슈 발생하므로 동기화 처리가 필요
 - 페이징 기반 처리
 - 페이지 사이즈 만큼 데이터를 한 번에 처리
 - 페이지 사이즈 만큼 커넥션을 맺고 끊음

- 페이지징 결과만 메모리에 할당
- 멀티 스레드 환경에서 스레드 안정성을 보장하기에 별도 동기화 처리 불필요
- 페이지 사이즈와 청크 사이즈를 일치시켜야 하는 이유
 - 청크 사이즈가 50이고 페이지 사이즈가 10이면 5번의 read가 발생하면서 한 번의 트랜잭션 처리를 위해서 5번의 조회 쿼리를 날리는 성능 이슈가 발생할 수 있다.
 - JPA를 사용하는 경우 영속성 컨텍스트가 깨지는 문제가 발생한다.
JpaPagingItemReader의 경우 페이지를 읽을때, 이전 트랜잭션을 flush, clear로 초기화 시켜버린다. 그러다 보니 마지막에 읽은 페이지를 제외한 이전에 조회한 결과들의 트랜잭션 세션이 전부 종료되어 버리면서 processor에서 Lazy 로딩이 불가능하게 되는 현상이 발생한다.
- itemWriter
 - jpaItemWriter
 - JPA 엔티티 기반으로 데이터를 처리하고 엔티티를 하나씩 insert한다.
 - JdbcBatchItemWriter
 - Jpa처럼 단건 처리가 아닌 일괄 bulk insert 처리한다.
 - ChunkSize 만큼 데이터를 커밋하기 때문에 Chunk size가 곧 Commit Interval(커밋 간격)이 된다.
- chunkSize 만큼 데이터를 한 번에 처리하고 다음 chunkSize는 새로운 트랜잭션으로 동작한다.
- repeat, retry, skip을 통해서 반복 및 오류 제어를 할 수 있다.

21.2. hibernateItemReader 와 jpaPaingReader의 차이

- 일반적인 경우에는 차이가 없으나 2개 이상의 컬렉션을 땡겨와야 하는 경우 차이가 발생한다. 2개 이상의 컬렉션을 땡겨오는 경우 MultipleBagFetchException을 피하기 위해 fetch join을 먹이고 다른 쪽에는 batchSize 옵션을 먹이게 된다. 다른 PagingItemReader의 경우 문제가 없으나 JpaPagingItemReader의 경우만 batchSize 옵션이 적용되지 않는다. 이유는 다른 PagingItemReader는 트랜잭션을 Chunk에 맡기지만 JpaPagingItemReader의 경우 트랜잭션이 doReadPage 메서드 안에서 읽기에 관한 트랜잭션을 먼저 처리해버리기 때문이다. doReadPage 메서드를 까보면 트랜잭션을 가져와서 flush, clear 해버리고 데이터를 읽어온 뒤 마지막에 트랜잭션 커밋을 해버리면서 데이터를 읽은 트랜잭션이 종료되어버린다. processor 부분에서

지연 로딩의 N+1문제를 막기 위해서 BatchSize 옵션을 걸어버린 것인데 processor에 오기 전에 해당 트랜잭션이 종료되어 버리므로 processor에서는 batchSize 옵션이 적용되지 않고 N+1 문제가 발생해버린다. 정리를 하자면, 일반적인 경우에는 상관 없으나 컬렉션 2개 이상의 조인을 해야하는 경우 JpaPagingItemReader의 N+1 이슈가 있으니 잘 알고 사용해야 한다.

22. MSA vs Monolithic(모놀리식)

- Monolithic(모놀리식)
 - 장점
 - 개발 환경이 같아서 복잡하지 않다.
 - End-To-End 테스트가 용이하다.(MSA의 경우 필요한 서비스들을 모두 동작시켜야함)
 - 단점
 - 프로젝트가 커지면 빌드, 배포 시간이 오래걸린다.
 - 작은 수정사항이 있어도 전체를 다시 빌드하고 배포해야 한다.
 - 많은 양의 코드가 몰려있어 개발자가 모두를 이해하는데 어렵고 유지보수하기도 어렵다.
 - 일부분의 오류가 전체에 영향을 미친다.
- MSA
 - 장점
 - 서비스 단위로 개발을 진행하기에 해당 부분을 온전히 이해하기 쉽다.
 - 새로 추가되는 부분은 빠르게 수정 및 배포가 가능하다.
 - 해당 기능에 맞는 기술, 언어 등을 선택하여 사용할 수 있다.
 - 문제가 생기면 해당 기능에만 문제가 생긴다.
 - 단점
 - 서비스가 분산되어 있어 관리하기 어렵다.
 - 통신오류가 잦을 수 있다.
 - 테스트가 불편하다.