

# (10월 29일) 스프링 애플리케이션과 영속성 컨텍스트

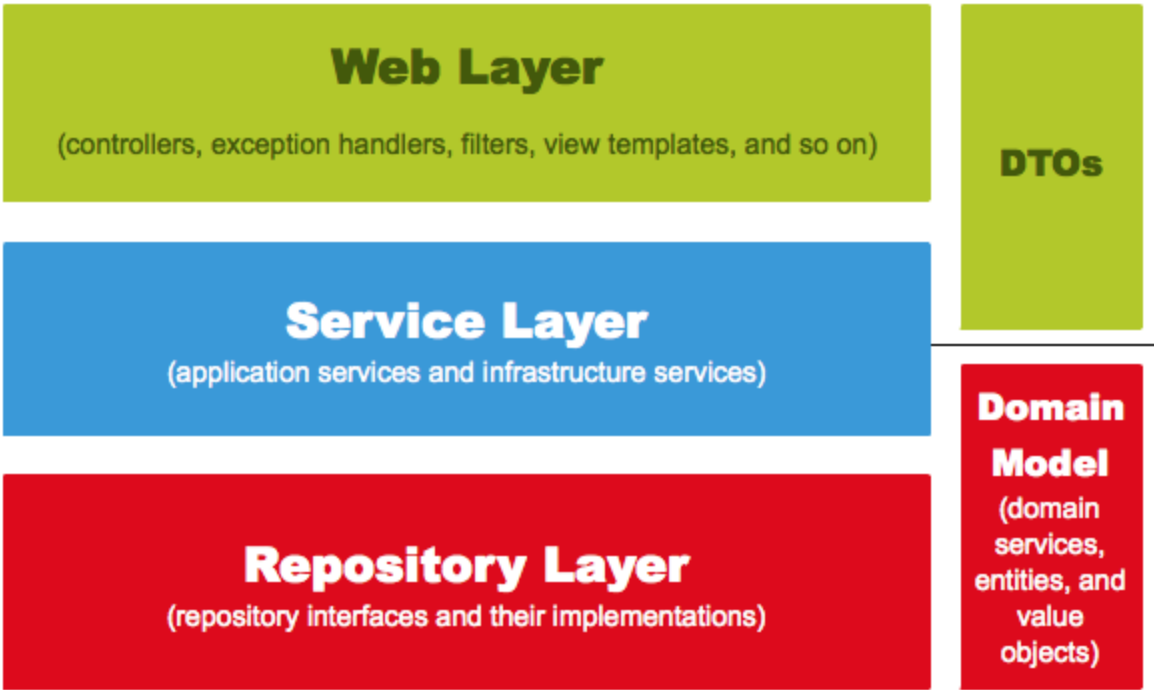
## J2EE, 스프링 컨테이너 환경 속 JPA

- 컨테이너가 트랜잭션, 영속성 컨텍스트를 관리
- 개발자가 직접 EntityManagerFactory 혹은 EntityManager를 생성할 필요 X
- 컨테이너가 제공하는 전략에 맞게 JPA를 사용해야 한다.



컨테이너가 제공하는 영속성 관리 전략을 알아야 적은 삼잘 오류를 내고 기능을 십분 활용할 수 있다.

## 웹 계층 분리와 영속성 컨텍스트



- 스프링의 웹 계층은 위와 같이 3가지로 나뉜다
  - Web layer
  - Service Layer
  - Repository Layer
- Web Layer는 Presentation Layer라고도 부르며 서버에서 가공된 데이터를 전달하는 (보여주는) 책임만 담당
- Service Layer는 Application Layer로써 여러 도메인 레이어(비즈니스 로직이 존재하는 곳)를 호출하여 요청을 가공하는 서버의 작업 수행의 시작점의 책임을 담당
- Repository Layer는 Data Source에 접근하는 계층을 의미하며 필요한 데이터를 가공해서 비즈니스 로직을 수행할 수 있도록 넘겨주는 책임을 담당

각 레이어의 책임에 맞게 컨테이너가 영속성 컨텍스트를 관리!

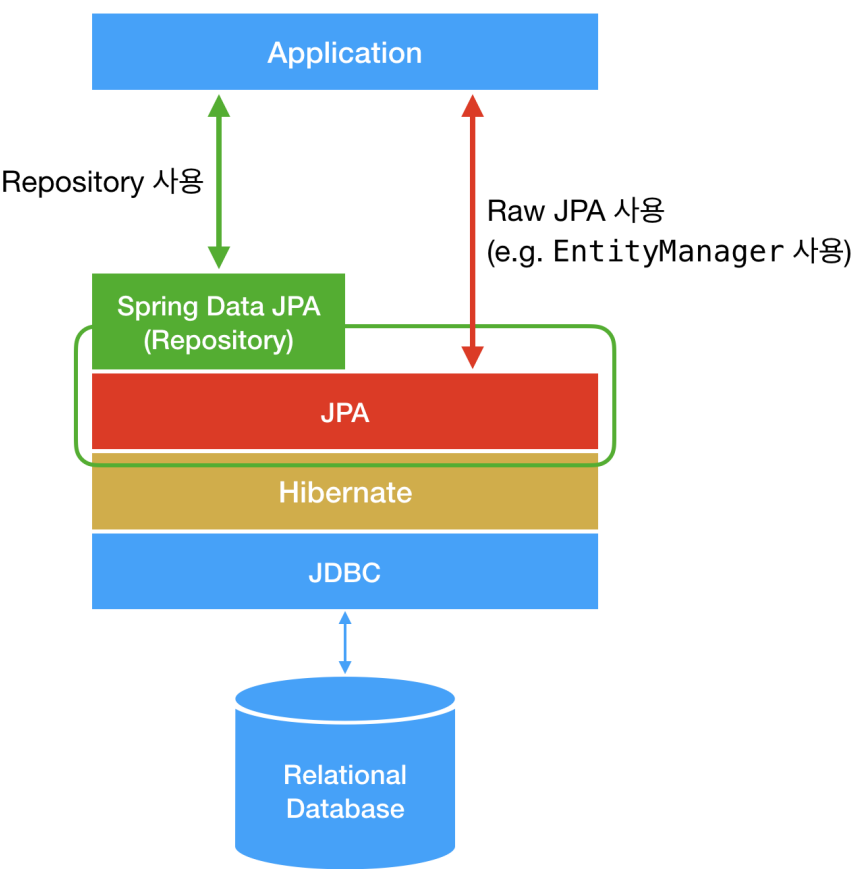
## Spring Data JPA

```
@Slf4j
@Service
@RequiredArgsConstructor
@Transactional
public class OrderServiceImpl implements OrderService {
    private final OrderRepository orderRepository;
    private final UserRepository userRepository;
    private final CustomerRepository customerRepository;
    private final ProductRepository productRepository;
    private final ModelMapper modelMapper;
```

```
package com.mcfuturepartners.crm.api.order.repository;

import ...

@Repository
public interface OrderRepository extends JpaRepository<Order, String> {
    Optional<Order> findById(long id);
    List<Order> findAllByCustomer(Customer customer);
    void deleteById(long id);
    List<Order> findAllByRegDateIsAfter(LocalDateTime localDateTime);
}
```



- 실제 Spring Data JPA 가장 큰 장점이 Entity Manager 및 commit을 직접 명시해줄 필요가 없다는 것
- 도메인의 Repository가 JpaRepository 인터페이스를 상속받게 하면 해당 도메인의 기본적인 CRUD를 설정없이 할 수 있음
  - 실제 Spring Data JPA의 Repository 구현부에서 JPA에 대한 설정을 해주고 있다는 뜻

```
1 package org.springframework.data.jpa.repository.support;
2
3 import ...
4
5 public class SimpleJpaRepository<T, ID> implements JpaRepositoryImplementation<T, ID> {
6
7     private final EntityManager em;
8
9     public Optional<T> findById(ID id) {
10
11         Assert.notNull(id, ID_MUST_NOT_BE_NULL);
12
13         Class<T> domainType = getDomainClass();
14
15         if (metadata == null) {
16             return Optional.ofNullable(em.find(domainType, id));
17         }
18
19         LockModeType type = metadata.getLockModeType();
20
21         Map<String, Object> hints = getQueryHints().withFetchGraphs(em).asMap();
22
23         return Optional.ofNullable(type == null ? em.find(domainType, id, hints) : em.find(domainType, id, type, hints));
24     }
25
26     // Other methods...
27 }
```

- 이미 정의되어 있는 구현부에 EntityManager에 대한 내용 그리고 제네릭 타입으로 데이터를 동적으로 매핑할 수 있다는 것을 알 수 있다.
- Repository별로 EntityManager를 생성하는 것을 알 수 있다.

```
@Repository
public interface OrderRepository extends JpaRepository<Order, String> {
    Optional<Order> findById(long id);
    List<Order> findAllByCustomer(Customer customer);
    void deleteById(long id);
    List<Order> findAllByRegDateIsAfter(LocalDateTime localDateTime);
}
```

```
@Repository
public interface CustomerRepository extends JpaRepository<Customer, Long>{
    //create
    Customer save(Customer customer);
    //research
    Optional<Customer> findById(Long id);
    Optional<Customer> findByNameAndPhone(String name, String phone);
    List<Customer> findByManager(String managerNo);
    List<Customer> findByRegDate(String regDate);
    // List<Customer> findByProduct(String Product);
    List<Customer> findByFunnel(String Funnel);
    List<Customer> findByPhone(String Funnel);
}
```

```

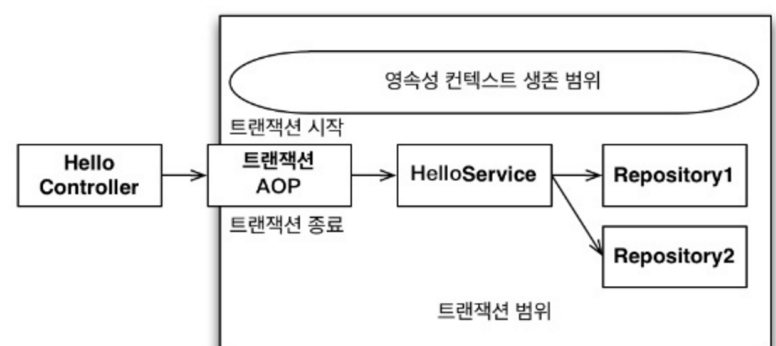
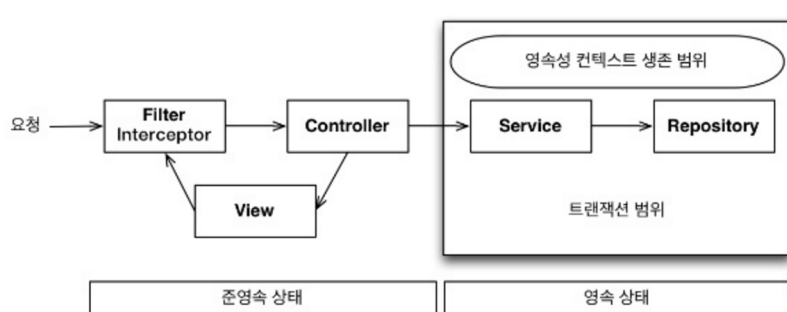
@Repository
public interface UserRepository extends JpaRepository<User, String> {
    List<User> findAll();
    User getByUsername(String username);
    User save(User entity);
    Optional<User> findByUsername(String username);
    Optional<User> findById(long id);
    boolean existsById(long id);
    boolean existsByUsername(String username);
}

```



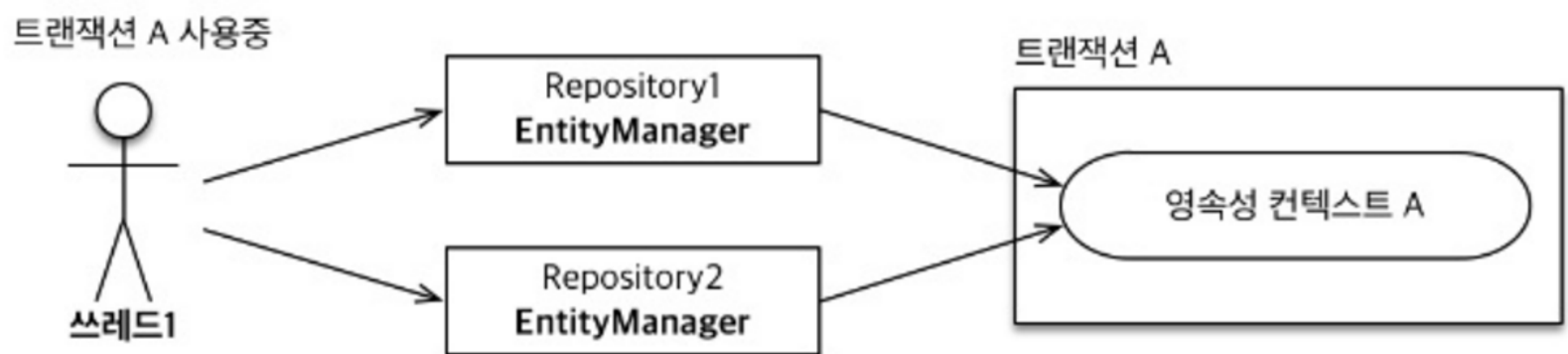
직접 EntityManager나 설정, transaction 단계를 구현하지 않아도 되서 매우 편리하지만, 동작 방법을 이해하지 않고 쓰면 오류 디버깅에 어려움을 느낄 수 있다.

## 스프링 컨테이너의 기본 전략



### Transaction의 범위와 영속성 컨텍스트의 생존 범위가 같다

- 스프링 컨테이너는 트랜잭션 범위를 영속성 컨텍스트의 생존 범위로 본다
- @Transactional Annotation을 붙인 메소드부터 트랜잭션 AOP가 동작
- Transaction의 시작점으로 볼 수 있는 Service에서 부터 영속성 컨텍스트가 수행
- Service 시점부터 트랜잭션을 시작하고, 정상적으로 수행하였을 때 트랜잭션 커밋
  - 트랜잭션 커밋이 되면 영속성 컨텍스트 1차 캐시 내 저장되어 있던 엔티티들이 DB에 flush
  - 비정상적으로 커밋이 된 경우 Rollback을 수행하여 flush를 수행하지 않아 영속성 컨텍스트 반영 X
- 트랜잭션의 시작 시점인 Service layer의 외부로 벗어난 경우 영속(persist) 상태였던 엔티티는 준영속(detach) 상태가 된다.
  - 해당 영속성 컨텍스트는 close() 메소드를 수행하며 참여했던 모든 엔티티를 준영속 상태로 만듦



## Transaction이 같으면 같은 영속성 컨텍스트를 사용한다

- Service Layer는 많은 비즈니스 로직들이 순차적으로 수행할 수 있도록 정리해주고 결합해주는 Transaction의 시작점
- 비즈니스 로직의 복잡도에 따라 Service Layer가 주입받는 Repository의 갯수가 상이
- Repository별로 EntityManager 생성
  - 1개의 Transaction에서 N개의 EntityManager 생성 가능
  - 기존 J2SE(Standard Edition)에선 EntityManager와 영속성 컨텍스트는 1:1 관계 유지
  - 동일 Transaction에서 다른 영속성 컨텍스트를 사용한다면?
    - 트랜잭션의 기본 원칙을 지키지 못하고, commit되지 않으면 롤백을 정상적으로 수행하지 못함

⇒ 이를 위해 Spring은 Transaction이 같을 때 다른 EntityManager를 통해서도 같은 영속성 컨텍스트를 사용할 수 있도록 한다.

동일 트랜잭션인 건 어떻게 압니까?

- 예전 발표(Project Reactor)에서 Spring Web Project는 하나의 요청을 하나의 스레드에서 처리
- 따라서, 동일 스레드에서 처리되는 Transaction은 모두 1개의 Transaction의 일부

## Transaction이 다르면 다른 영속성 컨텍스트를 사용한다

- Spring Container의 가장 큰 장점은 Transaction과 멀티 스레드 상황을 컨테이너가 알아서 처리해줌
- 위 상황과 반대로 같은 Entity를 사용하는 2개의 Transaction이 있을 때
  - 다른 트랜잭션은 다른 스레드에서 처리되기 때문에 EntityManager를 공유하고 있더라도 다른 영속성 컨텍스트를 사용한다



스프링 컨테이너에선 영속성 컨텍스트와 트랜잭션의 범위가 거의 일치한다. ⇒ 영속성 컨텍스트 하나를 하나의 트랜잭션 단위로 보는 것

# 우리는 그럼 어떤 문제를 조심해야 하는가?

## JPA 연관 관계

- 회원과 팀이 있다.
- 회원은 하나의 팀에만 소속할 수 있고, 팀은 여러명의 회원의 소속일 수 있다

⇒ 팀과 회원은 어떤 관계에 있나?

이를 JPA 매핑 방식으로 구현하면 이렇게 된다

```
@Entity
public class Team {

    @OneToMany(mappedBy = user)
    private List<Member> members = new ArrayList();

}

@Entity
public class Member{
    @ManayToOne
    @JoinColumn(name = "TEAM_ID")
    private Team team;
    ...
}
```

- 이런식으로 기존 join table로 엔티티의 관계를 나타냈던 SQL과 달리 JPA는 객체 사이 매핑을 이용하여 엔티티의 관계를 표현한다.
- 그렇다면 Team에서 가장 마지막에 들어온 회원을 뽑아서 response를 보내주는 코드는 아래와 같다.

```
@RestController
public class TeamController {

    public String findLatestMemberName(Long teamId){
        Team team = teamService.findOne(teamId);
        Member member = team.getMember();
        return member.getName();
    }
}
```

- 이러면 결과는 어떨까???

## 지연로딩

- 실제 하나의 필드가 아닌 객체들의 관계도를 애플리케이션에서 탐색하기 위해선 모든 객체들의 정보를 애플리케이션에 로딩을 해야한다
- 한, 두개의 매핑 단계에서 혹은 단방향의 상황에서는 가능할 수 있지만 복잡한 도메인 환경에선 불가능이다.
- JPA는 매핑 된 객체를 '필요한 시점'에 사용할 수 있도록 프록시 패턴을 사용
- 실제 엔티티를 매핑하지 않고, 프록시 객체를 제공한 후 실제 엔티티가 사용되는 시점에 데이터베이스 조회를 수행하는 것이 지연 로딩
- JPA는 디폴트로 지연 로딩을 지원하고 있다

## 스프링 컨테이너에서 준영속 상태

## 준영속 상태

### detach

- 특정 엔티티를 준영속 상태로 만들 때 사용

### detach(엔티티)가 수행 되면 어떤 일이 일어나나?

- 1차 캐시에서 대상 엔티티가 제거
- 쓰기 지연 SQL 저장소에서 해당 엔티티와 관련된 모든 SQL들이 제거

⇒ 더 이상 영속성 컨텍스트가 관리하지 않는 엔티티가 되었기 때문에 준영속 상태가 됨

### clear

- detach가 특정 엔티티를 준영속 상태를 만드는 것이라면, clear는 영속성 컨텍스트를 초기화 하는 것
- 영속성 컨텍스트를 제거하고 새로 만든 것과 동일

### close

- 영속성 컨텍스트를 종료
- 종료된 영속성 컨텍스트가 관리하던 엔티티들은 모두 준영속 상태

### merge

- 준영속 상태의 엔티티를 다시 영속상태로 만들 기 위해 사용

💡 준영속 상태의 엔티티는 영속성 컨텍스트에 관리되지 않기 때문에 1차 캐시, 쓰기 지연, 변경 감지 등 영속성 컨텍스트의 특징을 하나도 이용하지 못한다.

- Controller와 같은 Presentation Layer에선 엔티티들이 준영속 상태로 있다
- 준영속 상태의 엔티티는 영속성 컨텍스트의 특징을 하나도 이용하지 못한다
  - 1차 캐시
  - 쓰기 지연
  - 변경 감지

## Presentation Layer에서 누가 데이터를 쓰고 변경을 해.. 어차피 별 문제 없네

- C,U,D의 경우 데이터를 조작하고 데이터베이스에 반영해야하는 작업이기 때문에 당연히 Presentation Layer에서 걱정할 필요 없음
- 하지만, 지연 로딩이 걸린 Read 작업에선 어떻게 될까?
- 위 예시에서 처럼 Team 도메인에서 Team 데이터를 갖고와 매핑 된 Member 객체를 조회하면?
- 지연로딩 에러!!!!

```
event = {Event@11957}
  id = {Long@11965} 88
  study = {Study@11966}
  createBy = null
  title = "모임을 만들자"
  description = "스프링을 처음부터 공부하기 위한 모임입니다."
  createdDateTime = {LocalDateTime@11969} "2020-09-26T21:12:34.033480"
  endEnrollmentDateTime = {LocalDateTime@11970} "2020-09-26T22:12:34.012220"
  startDateTime = {LocalDateTime@11971} "2020-09-27T00:12:34.012274"
  endDateDateTime = {LocalDateTime@11972} "2020-09-27T04:12:34.012294"
  limitOfEnrollments = {Integer@11973} 2
  enrollments = {PersistentBag@11974} Unable to evaluate the expression Method threw 'org.hibernate.LazyInitializationException' exception.
  eventType = {EventType@11982} "FCFS"
```

## 왜 이러는 걸까요???



실제로 Service Layer, Presentation Layer의 역할에 따른 구현이 모호할 수 있기 때문에 이러한 오류는 잘 알고 지나가야 한다!

# 이를 위한 해결책

## 글로벌 페치(Fetch) 전략 수정

- 결국 지연 로딩 전략 또한 하나의 설정이기 때문에 개발자가 원한다면 Fetch(조회) 전략을 수정할 수 있다

```
@Entity
public class Team {

    @OneToMany(mappedBy = user)
    private List<Member> members = new ArrayList();

}

@Entity
public class Member{
    @ManyToOne(fetch = FetchType.EAGER)
    @JoinColumn(name = "TEAM_ID")
    private Team team;
    ...
}
```

### 단점

- 사용하지 않는 엔티티까지 모두 로딩하여 효율성이 떨어지고 성능 저하 유발
- N+1 문제가 생길 수 있다. (이 주제에 대해서 한참 얘기를 할 수 있고 공부할 수 있지만, 시간 관계상 이건 빼야할 것 같다)

## 강제 초기화

- Presentation Layer에서 필요한 엔티티를 Service Layer에서 강제로 초기화 하여 전달하는 것

```
@Service @Transactional
public class TeamService {

    public Team findOne(Long teamId){
        Team team = teamRepository.findById(teamId);
        team.getMember().getName();
        return team;
    }
}
```

### 단점

- Presentation Layer에서 필요한 엔티티가 계속 바뀌는 상황이라면??
  - Service Layer의 로직이 계속 변경되고, 추가적인 메소드가 필요해진다
  - 이는 Presentation Layer를 위한 일을 Service Layer가 해주고 있는 것

이를 위해 프록시 객체의 초기화만 담당하는 Facade 계층을 추가하여 Presentation Layer, Service Layer의 경계를 명확히 한다.



```
@Data
@Builder
@Table(name = "users")
@AllArgsConstructor
@NoArgsConstructor
public class User {

    @Id
    @Column(name = "user_id")
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id;

    @Column(name = "username", unique = true, nullable = false)
    private String username;

    @JsonIgnore
    @Size(min = 20, max = 40, message = "8자리 이상 입력하십시오")
    @Column(name = "password", nullable = false)
    private String password;

    @Column(name = "name", nullable = false)
    private String name;

    @Column(name = "phone", nullable = false)
    private String phone;

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name="department_id")
    private Department department;

    @OneToMany(mappedBy = "user")
    private List<Order> orders = new ArrayList<>();
}
```

```
public class UserRevenueDto {

    public static UserRevenue salesRevenue(User user){
        return UserRevenue.builder()
            .name(user.getName())
            .dailySales(user.getOrders().stream()
                .filter(order -> order
                    .getRegDate()
                    .isAfter(LocalDate.now().atStartOfDay()))
                .map(order -> order.getProduct().getPrice())
                .reduce( identity: 0, Integer::sum))
            .monthlySales(user.getOrders().stream()
                .filter(order -> order
                    .getRegDate()
                    .isAfter(LocalDate.of(LocalDate.now().getYear(), LocalDate.now().getMonth().getValue(), dayOfMonth: 1).atStartOfDay()))
                .map(order -> order.getProduct().getPrice())
                .reduce( identity: 0, Integer::sum))
            .build();
    }
}
```

- User는 사원, 한 사원은 여러 개의 주문을 발행할 수 있음
  - user entity를 get 했을 때, orders entity는 프록시 객체로 옴
  - 각 user의 매출을 계산해야 하는 데, 지연 로딩의 문제점으로 인해 문제가 지속적으로 발생
  - 실제로 지연 로딩의 문제점을 느끼고 Dto Layer를 추가적으로 만들어 계산
- ⇒ DTO는 만들어진 데이터를 전달하는 객체이기 때문에 로직 수행은 적절치 않음, Domain level에서 수행할 수 있도록 refactoring 예정



# Facade 계층 구현, OSIV(Open Session In View=영속성 컨텍스트 생존 범위를 늘리는 것) 등이 있지만...

## 중요하지만 자세히 알아야 의미있다고 생각되는 부분...


그래서...



## 참고자료


백엔드 서버 아키텍처-Application Layer 1. 개요와 기본 Variation

시리즈내의 다른 포스팅 목록은 → 이쪽에 Application Layer의 개요/역할/Variation들을 설명합니다. Variation으로는 Domain Facade, Operation Script, 생략이 존재합니다. 이 Layer를 부르는 명칭은 이 Layer를 부르는 맥락마다 달라지기 때문에, 헷갈리는 경우가 많습니다. 우선 일반적인 Layered Architecture나, DDD에서는 이

 <https://tech.junhabaek.net/%EB%B0%B1%EC%97%94%EB%93%9C-%EC%84%9C%EB%B2%84-%EC%95%84%ED%82%A4%ED%85%8D%EC%B2%98-application-layer-1-%EA%B0%9C%EC%9A%94%EC%99%80-%EA%B8%B0%EB%B3%B8-variation-9fac801ddba8#e1b4>


[JPA] 다양한 연관관계 매핑 (1) 다대일, 일대다

엔티티의 연관관계를 매핑할 때는 3가지를 고려해야한다. 먼저 두 엔티티가 일대일 관계일지 일대다 관계인지 다중성을 고려한다. 다음으로, 두 엔티티가 단방향과 양방향중 어떤 방식으로 참조하는지 결정한다. 마지막으로 양방향 관계라면 연관관계의 주인을 결정한다. 다대일(@ManyToOne) 일대다(@OneToMany) 일대일

 <https://jgrammer.tistory.com/entry/JPA-%EB%8B%A4%EC%96%91%ED%95%9C-%EC%97%B0%EA%B4%80%EA%B4%80%EA%B3%84-%EB%A7%A4%ED%95%91-1-%EB%8B%A4%EB%8C%80%EC%9D%BC-%EC%9D%BC%EB%8C%80%EB%8B%A4-1>


LazyLoading의 value를 단위테스트에서 확인하는 방법

spring-boot-starter-web:2.3.1 spring-boot-starter-data-jpa:2.3.1 spring-boot-postgresql:4.2.2.14 junit:junit:4.13 안녕하세요. 테스트 코드를 작성하면서 다음과 같은 hibernate 예외를 만났습니다. unable to evaluate the expression Method threw 'org.hibernate.LazyInitializationException' exception 테스트 코드

 <https://pinokio0702.tistory.com/194>

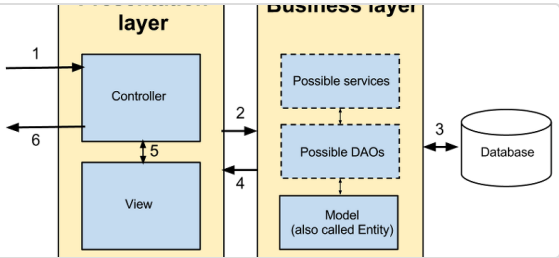
```
15 public @interface OneToMany {
16     Class targetEntity() default void.class;
17
18     CascadeType[] cascade() default {};;
19
20     FetchType fetch() default FetchType.LAZY;
21
22     String mappedBy() default "";
```

3.7 트랜잭션 범위의 영속성 컨텍스트

 <https://ultrakain.gitbooks.io/jpa/content/chapter3/chapter3.7.html>

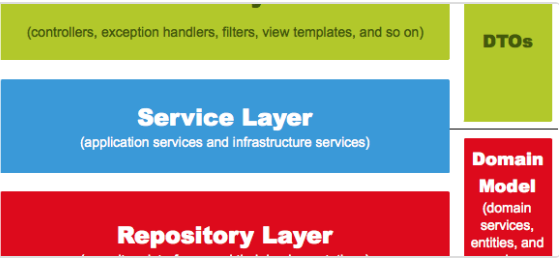
[Spring] MVC : Controller와 Service의 책임 나누기

martinfowler.com/bliki/PresentationDomainDataLayering.html 일반적으로 많이 사용하는 layer 구분 Presentation layer Service layer Business layer Persistence layer \*\*\* 기타 sublayer 들을 포함할 수도 있음. layer를 나눈다는 것은, 무언가를 추상화 하는 것이 그 목적이다. 추상화를 잘 했다면, 각기 다른 계층끼리 부딪히지 않는다. <https://umbum.dev/1066>



[스프링] Spring 웹 계층

1. Web Layer 흔히 사용하는 컨트롤러(Controller)와 JSP/Freemarker 등의 뷰 템플릿 영역이다. 이외에도 필터(Filter), 인터셉터, 컨트롤러 어드바이스@ControllerAdvice 등 외부 요청과 응답에 대한 전반적인 영역을 의미한다. 2. Service Layer @Service에 사용되는 서비스 영역이다. 일반적으로 Controller와 Dao의 중간 영역이다. <https://leveloper.tistory.com/14>



JPA, Hibernate, 그리고 Spring Data JPA의 차이점

Spring 프레임워크는 어플리케이션을 개발할 때 필요한 수많은 강력하고 편리한 기능을 제공해준다. 하지만 많은 기술이 존재하는 만큼 Spring 프레임워크를 처음 사용하는 사람이 Spring 프레임워크에 대한 정확한 이해를 하기는 매우 어렵다. 내가 특히 오랜 기간동안 혼란스러워했던 부분은 JPA와 관련된 개념이었다. <https://suhwan.dev/2019/02/24/jpa-vs-hibernate-vs-spring-data-jpa/>

Hibernate ORM 5.4.32.Final User Guide

```
public class TimestampEpochType extends AbstractSingleColumnStandardBasicType implements VersionType, LiteralType { public static final TimestampEpochType INSTANCE = new TimestampEpochType(); public TimestampEpochType() { super( BigIntTypeDescriptor.INSTANCE, JdbcTimestampTypeDescriptor.INSTANCE ); } @Override public String getName() { return "epoch"; } @Override public Date next( Date current, long increment ) { return current.toInstant().plusSeconds( increment ).toLocalDateTime().toDate(); } }
```

[https://docs.jboss.org/hibernate/orm/5.4/userguide/html\\_single/Hibernate\\_User\\_Guide.html](https://docs.jboss.org/hibernate/orm/5.4/userguide/html_single/Hibernate_User_Guide.html)