API 개발과 성능 최적화

API 개발 기본

회원 등록 API

V1 - 엔티티를 Request Body에 직접 매핑

```
@RestController
@RequiredArgsConstructor
public class MemberApiController{
    private final MemberService memberService;

    @PostMapping("/api/v1/members")
    public CreateMemberResponse saveMemberV1(@RequestBody @Valid Member member){
        Long id = memberService.join(member);
        return new CreateMemberResponse(id);
    }

    @Data
    static class CreateMemberRequest {
        private String name;
    }

    @Data
    static class CreateMemberResponse {
        private Long id;
```

```
        public CreateMemberResponse(Long id) {
            this.id = id;
        }
    }
}
```

문제점

- 엔티티에 프레젠테이션 계층을 위한 로직이 추가된다.
- 엔티티에 API 검증을 위한 로직이 들어간다(@NotEmpty 등등)
- 회원 엔티티를 위한 다양한 API가 만들어지는데, 한 엔티티에 각각의 API를 위한 모든 요청 요구사항을 담기 어렵다.
- **엔티티가 변경되면 API 스펙이 변한다.**

결론

- API요청 스펙에 맞추어 별도의 DTO를 파라미터로 받도록 한다.

V2 - 엔티티 대신에 DTO를 RequestBody에 매핑

```
@PostMapping("/api/v2/members")
public CreateMemberResponse saveMemberV2(@RequestBody @Valid CreateMemberRequest
request){

        Member member = new Member();
        member.setName(request.getName());

        Long id = memberService.join(member);
        return new CreateMemberResponse(id);
}

@Data
static class CreateMemberRequest{
    private String name;
}
```

- CreateMemberRequest를 Member 엔티티 대신에 RequestBody와 매핑한다.
- 엔티티와 프레젠테이션 계층을 위한 로직을 분리할 수 있다.
- 엔티티와 API 스펙을 명확하게 분리할 수 있다.

회원 수정 API

```
@PutMapping("/api/v2/members/{id}")
public UpdateMemberResponse updateMemberV2(@PathVariable("id") Long id,
@RequestBody @Valid UpdateMemberRequest request)
{
    memberService.update(id, request.getName());
    Member findMember = memberService.findOne(id);
    return new UpdateMemberResponse(findMember.getId(), findMember.getName());
}

@Data
static class UpdateMemberRequest{
```

```
    private String name;
}

@Data
@AllArgsConstructor
static class UpdateMemberResponse {
    private Long id;
    private String name;
}
```

- 회원 수정도 DTO를 요청 파라미터에 매핑한다.

```
public class MemberService {

    private final MemberRepository memberRepository;

        @Transactional
        public void update(Long id, String name) {
            Member member = memberRepository.findOne(id);
            member.setName(name);
    }
}
```

- 변경 감지를 사용해서 데이터를 수정한다.

회원 조회 API

V1 - 응답 값으로 엔티티를 직접 외부에 노출

```
package jpabook.jpashop.api;

@RestController
@RequiredArgsConstructor
public class MemberApiController {

    private final MemberService memberService;

    @GetMapping("/api/v1/members")
    public List<Member> membersV1() {
        return memberService.findMembers();
    }
}
```

문제점

- 기본적으로 엔티티의 모든 값이 노출된다.
- 응답 스펙을 맞추기 위해 로직이 추가된다.(@JsonIgnore, 별도의 뷰 로직 등등)
- 엔티티가 변경되면 API 스펙이 변한다.
- 추가로 컬렉션을 직접 반환하면 향후 API 스펙을 변경하기 어렵다.

결론

- 어떤 API는 name 필드가 필요하지만, 어떤 API는 name필드가 필요없을 수 있다.

  ---> API 응답 스펙에 맞추어 별도의 DTO를 반환한다.

V2 - 응답 값으로 엔티티가 아닌 별도의 DTO사용

```
@GetMapping("/api/v2/members")
public Result membersV2() {

    List<Member> findMembers = memberService.findMembers();
    //엔티티 -> DTO 변환
    List<MemberDto> collect = findMembers.stream()
            .map(m -> new MemberDto(m.getName()))
            .collect(Collectors.toList());

    return new Result(collect);
}

@Data
@AllArgsConstructor
static class Result<T> {
    private T data;
}
@Data
@AllArgsConstructor
static class MemberDto {
    private String name;
}
```

- 엔티티를 DTO로 변환해서 반환한다.
- 엔티티가 변해도 API 스펙이 변경되지 않는다.
- 추가로 Result 클래스로 컬렉션을 감싸서 향후 필요한 필드를 추가할 수 있다.

API 개발 고급 - 지연로딩과 성능 최적화

V1: 엔티티를 직접 노출

- 간단한 주문 조회

```
@RestController
@RequiredArgsConstructor
public class OrderSimpleApiController{
    private final OrderRepository orderRepository;

    @GetMapping("/api/v1/simple-orders")
    public List<Order> ordersV1{
        List<Order> all = orderRepository.findAllByString(new OrderSearch());
        for(Order order : all){
            order.getMember().getName(); // Lazy 강제 초기화
            order.getDelivery().getAddress(); // Lazy 강제 초기화
        }
        return all;
    }
}
```

- 엔티티를 직접 노출하는 것은 좋지 않다.

- order -> member 와 order -> address는 지연 로딩이다. 따라서 실제 엔티티 대신 프록시가 존재
- jackson 라이브러리는 기본적으로 이 프록시 객체를 json으로 어떻게 생성해야 하는지 모름 -> 예외 발생
- Hibernate5Module을 스프링 빈에 등록하면 해결된다.

V2: 엔티티를 DTO로 변환

```
@GetMapping("api/v2/simple-orders")
public List<SimpleOrderDto> ordersV2(){
    List<Order> orders = orderRepository.findAll();

    List<SimpleOrderDto> result = orders.stream()
            .map(o -> new SimpleorderDto(o))
            .collect(toList());

    return result;
}

@Data
static class SimpleOrderDto{

    private Long orderId;
    private String name;
    private LocalDateTime orderDate;
    private OrderStatus orderStatus;
    private Address address;

    public SimpleOrderDto(Order order) {
        orderId = order.getId();
        name = order.getMember().getName();
        orderDate = order.getOrderDate();
        orderStatus = order.getStatus();
        address = order.getDelivery().getAddress();
    }

}
```

- 엔티티를 DTO로 변환하는 일반적인 방법이다.
- 쿼리가 총 1 + N + N번 실행 된다.
    - order 조회 1번
    - order -> member 지연 로딩 조회 N번
    - order -> delivery 지연 로딩 조회 N번

V3: 엔티티를 DTO로 변환 - 페치 조인 최적화

```
@GetMapping("api/v3/simple-orders")
public List<SimpleOrderDto> oredersV3(){
    List<order> orders = orderRepository.findAllWithMemberDelivery();
    List<SimpleOrderDto> result = orders.stream()
            .map(o -> new SimpleOrderDto(o))
            .collect(toList());
    return result;
}
```

- ORderRepositroy 추가

```
public List<Order> findAllWithMemberDelivery(){
    return em.createQuery(
            "select o from Order o" +
                    "join fetch o.member m" +
                    "join fetch o.delivery d", Order.class)
            .getResultList();
}
```

- 엔티티를 페치 조인(fetch join)을 사용해서 쿼리 1번에 조회한다.
    - 쿼리 한개로 연관된 엔티티까지 같이 가져온다.
- 패치 조인으로order -> member, order -> delivery는 이미 조회 된 상태 이므로 지연로딩 x

V4: JPA에서 DTO로 바로 조회

[OrderSimpleApiController]

```
@RestController
@RequiredArgsConstructor
public class OrderSimplApiController{
    private final OrderSimpleQueryRepository orderSimpleQueryRepository; // 의존
관계 주입

    @GetMapping("/api/v4/simple-orders")
    public List<OrderSimpleQueryDto> ordersV4(){
        return orderSimpleQueryRepository.findOrderDtos();
    }
}
```

[OrderSimpleQueryRepository]

```
@Repository
@RequiredArgsConstructor
public class OrderSimpleQueryRepository{

    private final EntityManger em;

    public List<OrderSimpleQueryDto> findOrderDtos(){
        return em.createQuery(
        "select new
jpabook.jpashop.repository.order.simplequery.OrderSimpleQueryDto(o.id, m.name,
o.orderDate, o.status, d.address)" +
                        " from Order o" +
```

```
                        " join o.member m" +
                        " join o.delivery d", OrderSimpleQueryDto.class)
                        .getResultList();
    }
}
```

[OrderSimpleQueryDto]

```
@Data
public class OrderSimpleQueryDto {

    private Long orderId;
    private String name;
    private LocalDateTime orderDate; //주문시간
    private OrderStatus orderStatus;
    private Address address;

    public OrderSimpleQueryDto(Long orderId, String name, LocalDateTime
orderDate, OrderStatus orderStatus, Address address) {
        this.orderId = orderId;
        this.name = name;
        this.orderDate = orderDate;
        this.orderStatus = orderStatus;
        this.address = address;
    }
}
```

- 일반적인 SQL을 사용할 때 처럼 원하는 값을 선택해서 조회
- new 명령어를 사용해서 JPQL의 결과를 DTO로 즉시 변환
- SELECT절에서 원하는 데이터를 직접 선택하므로 DB 네트웍 용량 최적화
- **단점: 리포지토리 재사용성이 떨어짐, API 스펙에 맞춘 코드가 리포지토리에 들어감**

정리 - 어떤것을 주로 사용해야 할까?

1. V2 - 엔티티를 DTO로 변환하는 방법을 선택한다.
2. V3 - 성능이슈가 생겼을 때 패치 조인으로 성능을 최적화 한다.

==========

3. V4 - 그래도 해결이 안될때 DTO로 직접 조회하는 방법을 선택한다.
4. 최후의 방법은 JPA가 제공하는 네이티브 SQL이나 스프링 JDBC 템플릿을 이용하여 SQL을 직접사용한다.

API 개발 고급 - 컬랙션 조회 최적화

- 주문내역에서 추가로 주문한 상품 정보를 추가 조회

V1: 엔티티 직접 노출

```
@RestController
@RequiredArgsConstructor
public class OderApiController{

    private final OrderRepository orderRepository;

    @GetMapping("/api/v1/orders")
    public List<Order> ordersV1(){
        List<Order> all = orderRepository.findAll();
        for(Order order : all){
            order.getMember().getName(); // Lazy 강제 초기화
            order.getDelivery().getAddress(); // Lazy 강제 초기화
            List<OrderItem> orderItems = order.getOrderItem(); // Lazy 강제 초기화
            orderItems.stream().forEach(o -> o.getItem().getName());
        }
        return all;
    }
}
```

- orderItem, item 관계를 직접 초기화하면 Hibernate5Module 설정에 의해 엔티티를 JSON으로 생성한다.
- 양방향 연관관계면 무한 루프에 걸리지 않게 한곳에 @JsonIgnore를 추가해야 한다.
- 엔티티를 직접노출하므로 좋은 방법은 아니다.

V2: 엔티티를 DTO로 변환

```
@GetMapping("/api/v2/orders")
public List<OrderDto> orderV2(){
    List<Order> orders = orderRepository.findAll();
    List<OrderDto> result = orders.stream()
            .map(o -> new OrderDto(o))
            .collect(toList());

    return result;
}
```

```
@Data
static class OrderDto{
    private Long orderId;
    private String name;
    private LocalDateTime orderDate; //주문시간
    private OrderStatus orderStatus;
    private Address address;
    private List<OrderItemDto> orderItems;

    public OrderDto(Order order) {
            orderId = order.getId();
            name = order.getMember().getName();
            orderDate = order.getOrderDate();
            orderStatus = order.getStatus();
            address = order.getDelivery().getAddress();
```

```
            orderItems = order.getOrderItems().stream()
                    .map(orderItem -> new OrderItemDto(orderItem))
                    .collect(toList());
    }
}

@Data
static class OrderItemDto{
    private String itemName;//상품 명
    private int orderPrice; //주문 가격
    private int count;      //주문 수량

    public OrderItemDto(OrderItem orderItem) {
        itemName = orderItem.getItem().getName();
        orderPrice = orderItem.getOrderPrice();
        count = orderItem.getCount();
    }
}
```

- 지연 로딩으로 너무 많은 SQL을 실행한다.
- SQL 실행수
  - order - 1번
  - member, address - N번(order 조회 수 만큼)
  - orderItem - N번(order 조회 수 만큼)
  - item - N번(orderItem 조회 수 만큼)
- 만약 영속성 컨텍스트에서 이미 로딩한 회원 엔티티를 추가로 조회하면 SQL을 실행하지 않는다.

V3: 엔티티를 DTO로 변환 - 페치 조인 최적화

```
@GetMapping("/api/v3/orders")
public List<OrderDto> ordersV3(){
    List<Order> orders = orderRepoistory.findAllWithItem();
    List<OrderDto> result = orders.stream()
            .map(o -> new OrderDto(o))
            .collect(toList());

    return result;
}
```

```
public List<Order> findAllWithItem(){
    return em.createQuery(
            "select distinct o from Order o" +
                    "join fetch o.member m" +
                    "join fetch o.delivery d" +
                    "join fetch o.orderItems oi" +
                    "join fetch oi.item i", Order.class)
            .getResultList();
}
```

- 패치 조인으로 SQL이 1번만 실행됨
- distinct를 사용한 이유는 One-to-Many 조인이 있으므로 데이터베이스 row가 증가한다.

  -> order엔티티의 조회수 증가

같은 엔티티가 조회되면, 애플리케이션에서 중복을 걸러준다.

- 페이징이 불가능 하다.

V3.1: 엔티티를 DTO로 변환 - 페이징과 한계 돌파

컬렉션을 페치하면 페이징이 불가능하다.

- 컬렉션을 페치 조인하면 One-to-Many 조인이 발생하므로 데이터가 예측할 수 없이 증가한다.
- One을 기준으로 페이징 하는것이 목적이지만, 데이터는 Many를 기준으로 row가 생성된다.
- Order를 기준으로 페이징을 하고 싶은데, Many(N)인 OrderItem을 조인하면 OrderItem이 기준이 된다.
- 하이버네이트는 모든 DB데이터를 읽어서 메모리에서 페이징을 시도한다 -> 장애로 이어질 수 있음

페이징 + 컬렉션 엔티티 함께 조회 해결방법

- 먼저 ToOne(One-To-One, Many-To-One) 관계를 모두 페치 조인한다. row수를 증가시키지 않으므로 페이징 쿼리에 영향을 주지 않는다.
- 컬렉션은 지연 로딩으로 조회한다.
- 지연 로딩 성능 최적화를 위해 hibernate.default_batch_fetch_size, @BatchSize 를 적용한다.
  - hibernate.default_batch_fetch_size: 글로벌 설정
  - @BatchSize: 개별 최적화
  - 이 옵션을 사용하면 컬렉션이나, 프록시 객체를 한꺼번에 설정한 size만큼 IN쿼리로 조회한다.

```
public List<Order> findAllWithMemberDelivery(int offset, int limit){
    return em.createQuery(
            "select o from Order o" +
                    " join fetch o.member m" +
                    " join fetch o.delivery d", Order.class)
            .setFirstResult(offset)
            .setMaxResults(limit)
            .getResultList();

}
```

```
 @GetMapping("/api/v3.1/orders")
public List<OrderDto> ordersV3_page(@RequestParam(value = "offset", defaultValue
= "0") int offset,
                                    @RequestParam(value = "limit", defaultValue =
"100") int limit) {

        List<Order> orders = orderRepository.findAllWithMemberDelivery(offset,
limit);
        List<OrderDto> result = orders.stream()
                .map(o -> new OrderDto(o))
                .collect(toList());

        return result;
    }
```

[최적화 옵션]

```
spring:
    jpa:
        properties:
            hibernate:
                default_batch_fetch_size: 1000
```

- 개별로 설정하려면 @BatchSize를 적용하면 된다.(컬렉션은 컬렉션 필드에, 엔티티는 엔티티 클래스에 적용)

장점

- 쿼리 호출 수가 1 + N -> 1 + 1 로 최적화 된다.
- 조인보다 DB데이터 전송량이 최적화 된다.
- 페이징이 가능하다.

결론

- ToOne 관계는 페치 조인해도 페이징에 영향을 주지 않는다. 따라서 ToOne관계는 패치조인으로 쿼리수를 줄이고 해결하고, 나머지는 hibernate.default_batch_fetch_size 로 최적화 하자.

hibernate.default_batch_fetch_size

- 100 ~ 1000 사이의 값으로 권장
- 1000으로 잡으면 한번에 1000개를 DB에서 애플리케이션에 불러오므로 DB에 순간 부하가 증가할 수 있다.

  하지만, 결국 전체 데이터를 로딩해야하므로 100이나 1000이나 메모리 사용량은 같다.

  부하를 견딜 수 있는 수준으로 설정을 하면 된다.

V4: JPA에서 DTO 직접 조회

```
private final OrderQueryRepository orderQueryRepository;

@GetMapping("/api/v4/orders")
public List<OrderQueryDto> ordersV4() {
    return orderQueryRepository.findOrderQueryDtos();
}
```

[orderQueryRepository]

```
@Repository
@RequiredArgsConstructor
public class OrderQueryRepository {

    private final EntityManager em;

    public List<OrderQueryDto> findOrderQueryDtos() {
    //루트 조회(toOne 코드를 모두 한번에 조회)
        List<OrderQueryDto> result = findOrders();
    //루프를 돌면서 컬렉션 추가(추가 쿼리 실행)
        result.forEach(o -> {
```

```
            List<OrderItemQueryDto> orderItems = findOrderItems(o.getOrderId());
            o.setOrderItems(orderItems);
        });
    return result;
    }

    /**
     * 1:N 관계(컬렉션)를 제외한 나머지를 한번에 조회
     */
private List<OrderQueryDto> findOrders() {
    return em.createQuery(
            "select new
            jpabook.jpashop.repository.order.query.OrderQueryDto(o.id, m.name,
o.orderDate, o.status, d.address)" +
            " from Order o" + " join o.member m" +
            " join o.delivery d", OrderQueryDto.class)
       .getResultList();
}

    /**
     * 1:N 관계인 orderItems 조회
     */
private List<OrderItemQueryDto> findOrderItems(Long orderId) {
    return em.createQuery(
            "select new

 jpabook.jpashop.repository.order.query.OrderItemQueryDto(oi.order.id, i.name,
oi.orderPrice, oi.count)" +
            " from OrderItem oi" +
            " join oi.item i" +
            " where oi.order.id = : orderId",OrderItemQueryDto.class)
            .setParameter("orderId", orderId) .getResultList();
    }
}
```

[OrderQueryDto]

```
@Data
@EqualsAndHashCode(of = "orderId") public class OrderQueryDto {

    private Long orderId;
    private String name;
    private LocalDateTime orderDate; //주문시간
    private OrderStatus orderStatus;
    private Address address;
    private List<OrderItemQueryDto> orderItems;

    public OrderQueryDto(Long orderId, String name, LocalDateTime orderDate,
OrderStatus orderStatus, Address address) {
    this.orderId = orderId;
    this.name = name;
    this.orderDate = orderDate;
    this.orderStatus = orderStatus;
    this.address = address;
    }
}
```

[OrderItemQueryDto]

```
@Data
public class OrderItemQueryDto {

    @JsonIgnore
    private Long orderId; //주문번호
    private String itemName;//상품 명
    private int orderPrice; //주문 가격
    private int count;      //주문 수량

    public OrderItemQueryDto(Long orderId, String itemName, int orderPrice, int
count) {
    this.orderId = orderId;
    this.itemName = itemName;
    this.orderPrice = orderPrice;
    this.count = count;
    }
}
```

- Query: 루트 1번, 컬렉션 N번 실행
- ToOne 관계들을 먼저 조회하고, ToMany관계는 별도로 처리한다.
    - ToMany관계는 조인하면 row수가 증가하므로
- row수가 증가하지 않는 ToOne관계는 조인으로 최적화 하기 쉬우므로 한번에 조회하고, ToMany 관계는 최적화 하기 어려우므로 findOrderItems()같은 별도의 메서드로 조회한다.

V5: JPA에서 DTO 직접 조회 - 컬렉션 조회 최적화

```
@GetMapping("/api/v5/orders")
public List<OrderQueryDto> ordersV5() {
    return orderQueryRepository.findAllByDto_optimization();
}
```

```
public List<OrderQueryDto> findAllByDto_optimization() {

    //루트 조회(toOne 코드를 모두 한번에 조회)
    List<OrderQueryDto> result = findOrders();

    //orderItem 컬렉션을 MAP 한방에 조회
    Map<Long, List<OrderItemQueryDto>> orderItemMap
=findOrderItemMap(toOrderIds(result));

    //루프를 돌면서 컬렉션 추가(추가 쿼리 실행X)
    result.forEach(o -> o.setOrderItems(orderItemMap.get(o.getOrderId())));

    return result;
}

private List<Long> toOrderIds(List<OrderQueryDto> result) {
        return result.stream()
                .map(o -> o.getOrderId())
                .collect(Collectors.toList());
}
```

```
private Map<Long, List<OrderItemQueryDto>> findOrderItemMap(List<Long> orderIds)
{
    List<OrderItemQueryDto> orderItems = em.createQuery(
            "select new
jpabook.jpashop.repository.order.query.OrderItemQueryDto(oi.order.id, i.name,
oi.orderPrice, oi.count)" +
            " from OrderItem oi" +
            " join oi.item i" +
            " where oi.order.id in :orderIds", OrderItemQueryDto.class)
            .setParameter("orderIds", orderIds)
            .getResultList();
    return orderItems.stream()
            .collect(Collectors.groupingBy(OrderItemQueryDto::getOrderId));
}
```

- Query: 루트 1번, 컬렉션 1번
- ToOne 관계들을 먼저 조회하고, 여기서 얻은 식별자 orderId로 ToMany관계인 OrderItem을 한꺼번에 조회
- MAP을 사용해서 매칭 성능 향상(O(1))

V6: JPA에서 DTO로 직접 조회 - 플랫 데이터 최적화

정리 - 어떤것을 주로 사용해야 할까?

- 엔티티 조회
  - 엔티티를 조회해서 그대로 반환: V1
  - 엔티티 조회 후 DTO로 변환: V2
  - 페치 조인으로 쿼리 수 최적화: V3
  - 컬렉션과 페이징과 한계 돌파: V3.1
- DTO 직접 조회
  - JPA에서 DTO를 직접 조회: V4
  - 컬렉션 조회 최적화 - 일대다 관계인 컬렉션은 IN절을 활용해서 메모리에 미리 조회해서 최적화: V5
  - 플랫 데이터 최적화 - JOIN 결과를 그대로 조회 후 애플리케이션에서 원하는 모양으로 직접 변환: V6

  - 특정 주문 한건만 조회할 때는 V4도 성능이 잘 나온다.

    ex) Order 데이터가 1건이면 OrderItem을 찾기 위한 쿼리도 1번만 실행하면 된다.
  - 여러 주문을 한꺼번에 조회하는 경우 최적화된 V5를 사용해야 된다.

1. 엔티티 조회 방식으로 우선 접근 : V2
2. 페이징 필요x - 페치 조인으로 쿼리 수 최적화 : V3
3. 페이징 필요: V3.1

==========

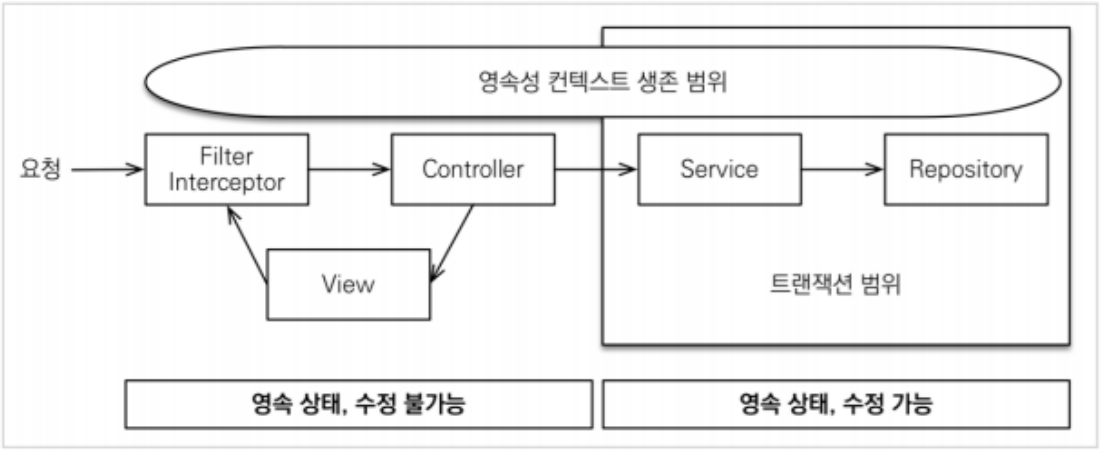
4. 해결이 안되면 DTO방식 사용 (V4/ V5)

API 개발 고급 -실무 필수 최적화

OSIV와 성능 최적화

- Open Session In View : 하이버네이트
- Open EntityManager In View : JPA
- 관례상 OSIV로 부른다.

OSIV ON



- **spring.jpa.open-in-view: true 기본값**

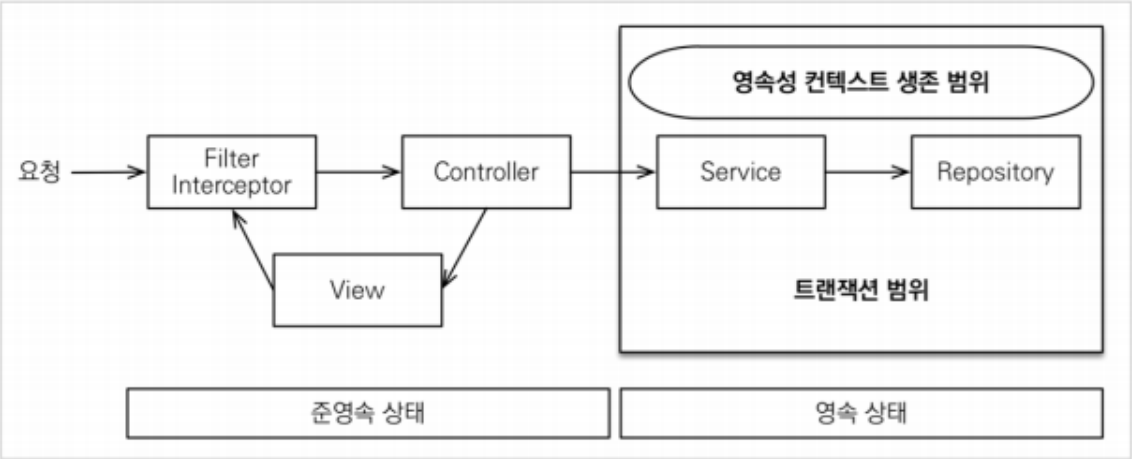
OSIV 전략은 트랜잭션 시작처럼 최초 데이터베이스 커넥션 시작 시점부터 API응답이 끝날 때 까지 영속성 컨텍스트와 데이터베이스 커넥션을 유지한다. 그래서 View Template이나 API컨트롤러에서 지연로딩이 가능했다.

지연 로딩은 영속성 컨텍스트가 살아있어야 가능하고, 영속성 컨텍스트는 기본적으로 데이터베이스 커넥션을 유지한다.

하지만, 너무 오랜시간 데이터베이스 커넥션 리소스를 사용하기 때문에 실시간 트래픽이 중요한 경우 애플리케이션에서는 커넥션이 모자랄 수 있다. -> 장애 발생

ex) 컨트롤러에서 외부 API를 호출하면 외부 API 대기 시간 만큼 커넥션 리소스를 반환하지 못하고 유지해야한다.

OSIV OFF



- **spring.jpa.open-in-view: false OSIV 종료**

OSIV를 끄면 트랜잭션을 종료할 때 영속성 컨텍스트를 닫고, 데이터베이스 커넥션도 반환한다. 따라서

커낵션 리소스를 낭비하지 않는다.

OSIV를 끄면 모든 지연로딩을 트랜잭션 안에서 처리해야 한다. 따라서 지금까지 작성한 많은 지연로딩코드를 트랜잭션 안에 넣어야 한다. 그리고 View Template에서 지연로딩이 동작하지 않는다.

결론적으로 트랜잭션이 끝나기 전에 지연로딩을 강제로 호출해 두어야 한다.

커멘드와 쿼리 분리

실무에서 OSIV를 끈 상태로 복잡성을 관리하는 방법은 Command와 Query를 분리하는 것이다.

OrderService

- OrderService: 핵심 비지니스 로직
- OrderQueryService: 화면이나 API에 맞춘 서비스(주로 읽기 전용 트랜잭션 사용)

이렇게, 관심사를 명확하게 분리

보통 서비스 계층에서 트랜잭션을 유지한다. 두 서비스 모두 트랜잭션을 유지하면서 지연로딩을 사용할 수 있다.

- 고객의 실시간 API : OSIV OFF
- ADMIN처럼 커넥션을 많이 사용하지 않는 경우: OSIV ON