3_jpa_apiAndOptimization

▼ API test environment

- postman

An application that allows the testing of web APIs.

▼ init

@Transactional doesn’t work in @PostConstruct due to spring life cycle.

Declare initializing methods separate and call them.

```
@Component
@RequiredArgsConstructor
public class InitDB {

    private final InitService initService;

    @PostConstruct
    public void init() {
        initService.dbInit1();
        initService.dbInit2();
    }


    @Component
    @Transactional
    @RequiredArgsConstructor
    static class InitService{

      private final EntityManager em;
      public void dbInit1() {
          Member member = createMember("userA", "서울", "1", "1111");
          em.persist(member);

          Book book1 = createBook("JPA1 BOOK", 10000, 100);
          em.persist(book1);

          OrderItem orderItem1 = OrderItem.createOrderItem(book1, 10000, 1);

          Delivery delivery = createDelivery(member);

          Order order = Order.createOrder(member, delivery, orderItem1, orderItem2);
          em.persist(order);
        }

      public void dbInit2() {   ...   }

      private Member createMember(... ) {
```

```
        ...
        return member;
      }

      private Book createBook(String name, int price, int stockQuantity) {
        ...
        return book1;
      }

      private Delivery createDelivery(Member member) {
        ...
        return delivery;
      }
    }
}
```

▼ basic rule and optimization procedure

▼ 1. Use DTO for request and response.

DO NOT expose entity to the outside

Entity is only used in logical process.

Entity change doesn’t affect API spec

▼ @JsonIgnore

add if return entity directly (But, DO NOT return entity!!!)

If not, both class called each other continuously → infinite loop

▼ 2. Different API, different DTO.

Reasons

1. Each API requires different fields.
2. To avoid adding presentation logic in entity.
3. To avoid adding API validation logic in entity.

   (@JsonIgnore, @NotEmpty)
4. To avoid exposing all entity fields.
5. It’s difficult to include all logic for API in entity.

6. Entity changes → API spec changes.

▼ 3. Set FetchType.Lazy

- Why should EAGER loading be avoided?
  1. It can cause performance deterioration

     by fetching unnecessary data even from the mapped relation when not needed.

  2. makes it harder for performance tuning

▼ 4. Use fetch join for performance

Use fetch join when relation mapped entity has to be called.

Or else, more queries will dispatched duet to LAZY loadng

→ 1 + N (different from N + 1)

▼ 1. Apply fetch join only to xToOne relation.

then find collections as LAZY loading.

Duplicate data are not selected

even though more queries dispatched than ‘fetch join on xToMany’.

```
public List<Order> findAllWithMemberDelivery(int offset, int limit) {
     /* << Recommended >>
     * xToOne 관계만 fetch join 적용 */
     String query1 = "select o from Order o " +
           " join fetch o.member m" +
           " join fetch o.delivery d";

     /* 생략해도 가능 ( Lazy join으로 가져옴 )
     * 네트워크를 더 많이 탐 */
     String query2 = "select o from Order o ";

     return em.createQuery(
           query1, Order.class)
           .setFirstResult(offset)
           .setMaxResults(limit)
           .getResultList();
}
```

▼ fetch join on xToMany

1. sql distinct → not filtered because data are different.

   For this reason, paging doesn't work.

   `WARN : HHH000104: firstResult/maxResults specified with collection fetch; applying in memory!`

2. Entity distinct → duplicates are filtered here.

▼ 5. Optimize for xToMany

▼ 1. Apply fetch join only to xToOne relation.

then find collections as LAZY loading.

Duplicate data are not selected

even though more queries dispatched than 'fetch join on xToMany'.

▼ NOTE: fetch join on xToMany

1. sql distinct → not filtered because data are different.

   For this reason, paging doesn't work.

   `WARN : HHH000104: firstResult/maxResults specified with collection fetch; applying in memory!`

2. Entity distinct → duplicates are filtered here.

▼ 2. Set batch size (the number of 'in queries')

- global (recommended)

```
// yml
spring:
  jpa:
    properties:
      hibernate:
        format_sql: true
        default_batch_fetch_size: 100 # the number of 'in query'
```

- separate

  @BatchSize(size=1000)

```
@BatchSize(size=1000)
@GetMapping("/api/v3.1/orders")
public List<OrderDto> ordersV3_page(
  ...
}
```

▼ Additonal - Check if performance is still bad after applying fetch join

Next steps

▼ 1. Use cache

▼ 2. (Not recommended) Search DTO directly

- Advantage

  minimizing network capacity (but, not so great)

- Disadvantage

  Lack of repository reusability

  Repository will contain code that corresponded to API spec

▼ 3. (Not recommended) Use native SQL that JPA provide or make query directly using spring jdbc template.

▼ Examples

▼ Create (sign-in)

```
@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;
}

@Data
static class CreateMemberResponse {
  private Long id;

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

▼ Update

전체 업데이트: put

부분 업데이트: post, patch

```
// @PatchMapping("/api/v2/members/{id}")
@PostMapping("/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;
}
```

```
public class MemberService {
  private final MemberRepository memberRepository;

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

▼ Read (Searching)

▼ Simple

Use Result class for adding additional fields required.

```
@GetMapping("/api/v2/members")
public Result membersV2() {
    List<Member> findMembers = memberService.findMembers();

    // Entity -> DTO
    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;
}
```

▼ xToOne

```
@RestController
@RequiredArgsConstructor
public class OrderSimpleApiController {

    private final OrderRepository orderRepository;
    private final OrderSimpleQueryRepository orderSimpleQueryRepository;
```

```
    // Fetch Join
    @GetMapping("/api/v3/simple-orders")
    public List<SimpleOrderDto> ordersV3() {
        List<Order> orders = orderRepository.findAllWithMemberDelivery();

        List<SimpleOrderDto> result = orders.stream()
            .map(o -> new SimpleOrderDto(o))
            .collect(Collectors.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();
        }
    }
}
```

▼ xToMany (collections)

- controller

```
public List<Order> findAllWithMemberDelivery(int offset, int limit) {
     /* << Recommended >>
     * xToOne 관계만 fetch join 적용 */
     String query1 = "select o from Order o " +
           " join fetch o.member m" +
           " join fetch o.delivery d";

     /* 생략해도 가능 ( Lazy join으로 가져옴 )
     * 네트워크를 더 많이 탐 */
     String query2 = "select o from Order o ";

     return em.createQuery(
           query1, Order.class)
           .setFirstResult(offset)
           .setMaxResults(limit)
           .getResultList();
}
```

- repository

```
public List<Order> findAllWithMemberDelivery(int offset, int limit) {
   /* << Recommended >>
   * xToOne 관계만 fetch join 적용 */
   String query1 = "select o from Order o " +
         " join fetch o.member m" +
         " join fetch o.delivery d";

   /* 생략해도 가능 ( Lazy join으로 가져옴 )
   * 네트워크를 더 많이 탐 */
   String query2 = "select o from Order o ";

   return em.createQuery(
         query1, Order.class)
         .setFirstResult(offset)
         .setMaxResults(limit)
         .getResultList();
}
```

▼ 2. Set batch size (the number of ‘in queries’)

- global (recommended)

```
// yml
spring:
  jpa:
    properties:
      hibernate:
        format_sql: true
        default_batch_fetch_size: 100 # the number of 'in query'
```

- separate

  @BatchSize(size=1000)

```
@BatchSize(size=1000)
@GetMapping("/api/v3.1/orders")
public List<OrderDto> ordersV3_page(
  ...
}
```

▼ OSIV

Open Session In view (Hibernate)

Open EntityManager In View (JPA)

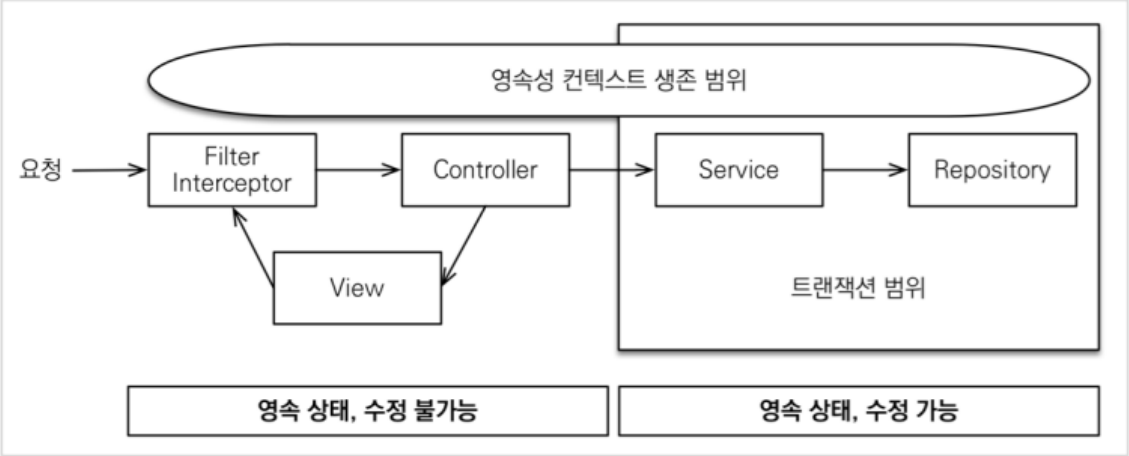
Range that DB connection remains

Let’s et it default then refactor(change)

when needed!

▼ ON (default)

Applicable for APIs that doesn’t require many connections like ADMIN



Connection returns when API result returns

Connection life cycle finishes when API result returns

- Advantage

  LAZY loading is able in controller

- Disadvantage

  Use DB connection resource too long

  → connection can be in short (dried)

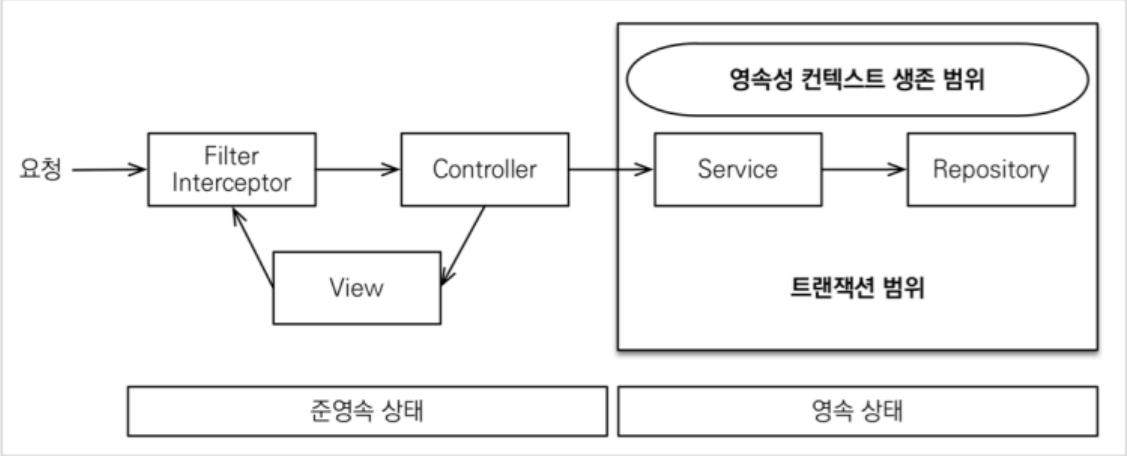
→ Service failure (in applications that traffic is important)

▼ yml

```
spring:
  jpa:
    open-in-view: true
```

▼ OFF

Applicable for realtime service APIs that require many connections



Connection returns when the transaction ends.

Connection life cycle finishes when the transactions ends.

Codes that LAZY loading applied should be done in service layer.

- Advantage

  Connection resource are not wasted

- Disadvantage

  Bit more complicate

→ code in cotroller should be separeted in two sector below.

However, it is better in aspect to maintenance.

▼ Separate command and query

OrderService ← business logic

OrderQueryService ← service that tailered to the view or API

ref: https://en.wikipedia.org/wiki/Command–query_separation

▼ yml

```
spring:
  jpa:
    open-in-view: false
```