Spring Boot 04

[도메인 개발]

실전! 스프링 부트와 JPA 활용 1

김영한

2022.08.11

회원 도메인 개발

회원 리포지토리

```
@Repository
@RequiredArgsConstructor      → 생성자 자동 주입해준다.
public class MemberRepository {

   @PersistenceContext        → 스프링이 자동 주입해준다.
                              → 스프링부트에서 @Autowired로 사용가능하게 해주고
                                생성자 주입이 가능해 진다.
   private final EntityManager em;

   public void sava(Member member){
       em.persist(member);
   }

   public Member findOne(Long id){
       return em.find(Member.class, id);
   }

   public List<Member> findAll(){
       return em.createQuery("select m from Member m", Member.class)
               .getResultList();
   }

   public List<Member> findByName(String name){
       return em.createQuery("select m from Member m where m.name =
:name",Member.class)
               .setParameter("name", name)
               .getResultList();
   }
}
```

리포지토리 예시 JPA 사용해서 개발

JPQL 사용→ 엔티티 객체를 대상으로 쿼리를 함 ⇒ JPA 가 SQL로 번역

회원 서비스

```
@Service
@Transactional(readOnly = true)    → 트랜잭션을 읽기전용으로 설정 후 업데이트 부분에
@RequiredArgsConstructor
public class MemberService {

    private final MemberRepository memberRepository;

    /**
     * 회원 가입
     */
    @Transactional                  → 트랜잭션을 필요한 부분에서 사용
    public Long join(Member member){
        validateDuplicateMember(member);
        memberRepository.sava(member);
        return member.getId();
    }

    private void validateDuplicateMember(Member member) {
        List<Member> findMembers = memberRepository.findByName(member.getName());
        if(!findMembers.isEmpty()){
            throw new IllegalStateException("이미 존재하는 회원입니다.");
        }
    }

    /**
     * 회원 조회
     */
    public List<Member> findMembers(){
        return memberRepository.findAll();
    }
    public Member findOnd(Long memberId){
        return memberRepository.findOne(memberId);
    }
}
```

회원 도메인 테스트

```
@SpringBootTest
@Transactional
class MemberServiceTest {

   @Autowired
   MemberService memberService;

   @Autowired
   MemberRepository memberRepository;

   @Autowired
   EntityManager em;

   @Test
   public void 회원가입() throws Exception {
       //given
       Member member = new Member();
       member.setName("kim");

       //when
       Long savedId = memberService.join(member);

       //then
       em.flush();
       assertEquals(member, memberRepository.findOne(savedId));
   }

   @Test
   public void 중복_회원_예외() throws Exception {
       //given
       Member member1 = new Member();
       member1.setName("kim");
       Member member2 = new Member();
       member2.setName("kim");

       //when
       memberService.join(member1);

       //then
       assertThrows(IllegalStateException.class, ()->
memberService.join(member2));
   }
}
```

상품 도메인 개발

상품 도메인 비즈니스 로직

```
//== 비즈니스 로직 ==//

/**
* stock 증가
* @param quantity : 추가량
*/
public void addStock(int quantity){
   this.stockQuantity += quantity;
}

/**
* stock 감소
* @param quantity : 감소 변경수량
*/
public void removeStock(int quantity){
   int restStock = this.stockQuantity - quantity;
   if(restStock < 0){
       throw new NotEnoughStockException("need more stock");
   }
   this.stockQuantity = restStock;
}
```

도메인 안에 비즈니스 로직을 넣어주면 응집력이 좋아지고 유지보수하기 좋아진다.

주문 도메인 개발

주문 엔티티 개발

오더 연관관계 메소드를 걸면서 생성을 하게 만드는 메소드를 엔티티에 개발을 한다.

```
//연관관계 편의 메서드//
public void setMember(Member member){
   this.member = member;
   member.getOrders().add(this);
}
public void addOrderItem(OrderItem orderItem){
   orderItems.add(orderItem);
   orderItem.setOrder(this);
}
```

```
public void setDelivery(Delivery delivery){
   this.delivery = delivery;
   delivery.setOrder(this);
}

//==생성 메서드==//
public static Order createOrder(Member member, Delivery delivery, OrderItem...
orderItems){
   Order order = new Order();
   order.setMember(member);
   order.setDelivery(delivery);
   for (OrderItem orderItem : orderItems) {
       order.addOrderItem(orderItem);
   }
   order.setStatus(OrderStatus.ORDER);
   order.setOrderDate(LocalDateTime.now());
   return order;
}
```

오더를 취소하는 메소드도(비즈니스 로직) 도 도메인에 만든다.

```
// == 비즈니스 로직 == //
// 바뀐 내역이 업데이트 내역이 바뀌게 된다.
/**
* 주문 취소
*/
public void cancel(){
   if(delivery.getStatus() == DeliveryStatus.COMP){
       throw new IllegalStateException("이미 배송안되로딘 상품은 취소가 불가능
합니다.");
   }
   this.setStatus(OrderStatus.CANCEL);
   for (OrderItem orderItem : orderItems) {
       orderItem.cancel();  → 오더 아이템에 연결된 아이템들의 잔여수량을 올려준다.
   }
}
```

가격 조회하는 로직도 만들어준다.

```
//==조회 로직==// 전체 가격조회
public int getTotalPrice(){
   return orderItems.stream().mapToInt(OrderItem::getTotalPrice).sum();
}
```

주문 서비스 개발

```
@Service
@Transactional(readOnly = true)
@RequiredArgsConstructor
public class OrderService {

    private final OrderRepository orderRepository;
    private final MemberRepository memberRepository;
    private final ItemRepository itemRepository;

    /**
     *주문
     */
    @Transactional
    public Long order(Long memberId, Long itemId, int count){

        //엔티티 조회
        Member member = memberRepository.findOne(memberId);
        Item item = itemRepository.findOne(itemId);

        //배송정보 생성
        Delivery delivery = new Delivery();
        delivery.setAddress(member.getAddress());

        //주문상품 생성
        OrderItem orderItem = OrderItem.createOrderItem(item, item.getPrice(),
count);

        //주문 생성
        Order order = Order.createOrder(member, delivery, orderItem);

        //주문 저장
        orderRepository.save(order);

        return order.getId();

    }

    /**
     * 취소
     */
    @Transactional
    public void cancelOrder(Long orderId){
        //주문 엔티티 조회
        Order order = orderRepository.findOne(orderId);
```

```
        //주문 취소
        order.cancel();
    }

    //검색
    public List<Order> findOrders(OrderSearch orderSearch){
        return orderRepository.findAllByString(orderSearch);
    }
}
```

사실 createOrder에서 여러개의 오더아이템을 넣었을 때 작동을 해야하지만 우선 한개만 실햄되게 새팅

캐스케이드 → 단일 소유일 경우에만 사용해야 한다.

엔티티 안에서 변경된 값을 Dirty checking 해서 변경된 내용이 업데이트 쿼리가 날아간다.

영속성 컨텍스트가 관리될 때 사용된다.

도메인 모델 패턴

비즈니스 로직이 대부분 엔티티에 있다. 복잡한 비즈니스 로직을 가지고 객체 지향의 특징을 적극 활용한다.

반대로 트랜잭션 스크립트 패턴에 엔티티에는 비즈니스 로직이 거의 없고 서비스 계층에서 대부분의 비즈니스 로직을 구현하는 방식을 가지고 있는 것도 있다. 어느 쪽이 좋다고 할 수 없고 양립 할 수 있다.

주문 검색 기능 개발

JPA에서 동적쿼리를 날리기 위해서 구현하는 기능

회원을 넣거나 주문상태를 확인해서 둘 중 하나로 구현 되게 사용하는 것

아래와 같이 생성

JPQL 동적쿼리 직접 생성

```
//동적 쿼리
public List<Order> findAllByString(OrderSearch orderSearch) {

   //language=JPQL
   String jpql = "select o From Order o join o.member m";
   boolean isFirstCondition = true;
   //주문 상태 검색
   if (orderSearch.getOrderStatus() != null) {
       if (isFirstCondition) {
           jpql += " where";
           isFirstCondition = false;
       } else {
           jpql += " and";
       }
       jpql += " o.status = :status";
   }
   //회원 이름 검색
   if (StringUtils.hasText(orderSearch.getMemberName())) {
       if (isFirstCondition) {
           jpql += " where";
           isFirstCondition = false;
       } else {
           jpql += " and";
       }
       jpql += " m.name like :name";
   }
   TypedQuery<Order> query = em.createQuery(jpql, Order.class)
```

```
            .setMaxResults(1000); //최대 1000건
    if (orderSearch.getOrderStatus() != null) {
        query = query.setParameter("status", orderSearch.getOrderStatus());
    }
    if (StringUtils.hasText(orderSearch.getMemberName())) {
        query = query.setParameter("name", orderSearch.getMemberName());
    }
    return query.getResultList();
}
```

→ 굉장히 복잡하고 문자열을 만드는 거기 때문에 오류가 생길 여지가 충분히 있다.

JPA Criteria 동적쿼리 생성

```
public List<Order> findAllByCriteria(OrderSearch orderSearch){
    CriteriaBuilder cb = em.getCriteriaBuilder();
    CriteriaQuery<Order> cq = cb.createQuery(Order.class);
    Root<Order> o = cq.from(Order.class);
    Join<Object, Object> m = o.join("member", JoinType.INNER);

    List<Predicate> criteria = new ArrayList<>();

    if(orderSearch.getOrderStatus() != null){
        Predicate status = cb.equal(o.get("status"), orderSearch.getOrderStatus());
        criteria.add(status);
    }

    //회원 이름 검색
    if (StringUtils.hasText(orderSearch.getMemberName())) {
        Predicate name =
                cb.like(m.<String>get("name"), "%" +
                        orderSearch.getMemberName() + "%");
        criteria.add(name);
    }
    cq.where(cb.and(criteria.toArray(new Predicate[criteria.size()])));
    TypedQuery<Order> query = em.createQuery(cq).setMaxResults(1000);

    //최대 1000건
    return query.getResultList();
}
```

→ 가시성이 떨어지고 유지보수가 힘들다 사용하기 힘들다.

QueryDSL 사용! [결론]