Spring Boot 실전! 스프링 부트와 JPA 활용 1

김영한
2022.07.29

서버 재실행 문제

계속 재실행 하기 싫다.

```
implementation 'org.springframework.boot:spring-boot-devtools'
```

build.gradle 에 라이브러리 추가

build - rebuild(ctrl + shift + f9) 사용

`@Transactional`

테스트 코드속에 들어있다면 트렌젝션이 롤백된다.

쿼리 파라미터 로그남기기

spring-boot-data-source-decorator 라이브러리 사용

도메인 설계

1대 다 관계에서는 FK 는 무조건 다에 있다.

외래키가 있는 곳을 연관 관계의 주인으로 정한다. (외래키를 누가 관리하냐의 문제)
비즈니스상 우위에 있다고 정하면 안된다. 자동차 - 바퀴 (4개) 일 대 다 관계일 때 자동차가 관리하지 않는 바퀴 테이블의 외래 키 값이 업데이트 되므로 관리와 유지보수가 어렵고, 추가 업데이트 쿼리가 필요할 수 있다.

오더의 연관관계 - 오더에 있는 멤버를 바꾸면 자기가 변경된다 (주인) .

-검색

일 대 일 관계에서는 엑세스가 많은 곳에 FK 를 준다.

엔티티 클래스 개발

실무에서는 Getter는 열어두고 Setter는 가급적 닫아둔다.

다대다관계는 사용하지 말자.

embeddable? :

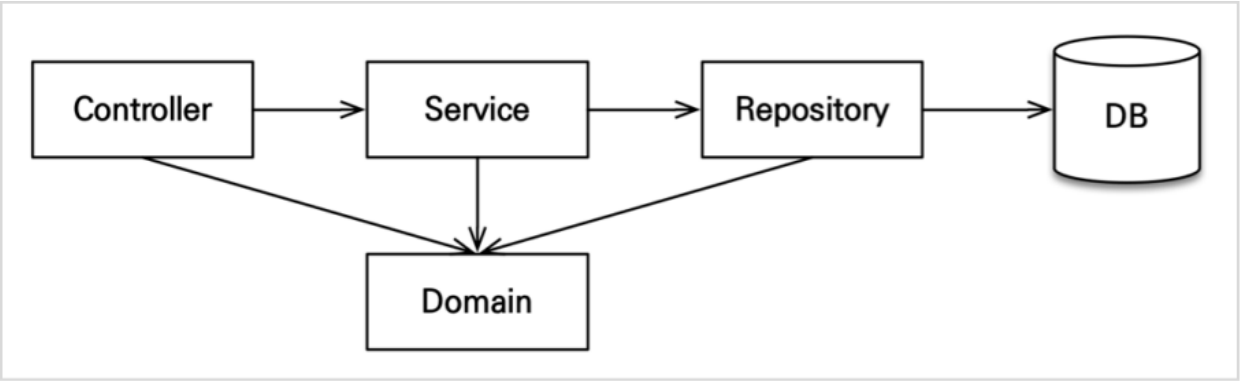
엔티티 설계시 주의점

- 가급적 Setter를 사용하지 말자
- 모든 연관관계는 지연로딩으로 설정
    - 즉시로딩 (EAGER) 은 예) 맴버 조회할때 오더도 로딩해서 조회 준비 (연관된 데이터를 모두 끌고온다.) 그러므로 지연로딩 ('LAZY') 으로 설정해야 한다.
    - 연관된 엔티티를 DB에서 조회하려면 fetch join또는 엔티티 그래프 기능을 사용
    - JPQL을 실행할 때 1 + n 문제가 생길수 있다.
    - ManyToOne은 기본이 fatch = FetchType.EAGER 이기 때문에 모두 찾아서 변경해 줘야한다.
- 컬렉션은 필드에서 초기화 하자. 하이버네이트가 내장 컬렉션으로 변화시켜서 관리한다.

```
cascade = CascadeType.ALL
```

캐스케이드로 묶어두면 연관 테이블이 같이 동작한다.

애플리케이션 아키텍쳐



JPQL 차이

엔티티 객체에 대해 select를 진행한다.

```
"select m from Member m"
```

MemberService 개발

@Transactional 을 꼭 선언해줘야한다.

```
@Transactional(readOnly = true)
```

조회 영역에 대해 readOnly 속성을 추가하면 최적화해준다.

의존 주입

필드 주입 : 변경이 힘들다. 테스트가 힘들다.

세터 주입 : 조립이 끝난후에 setter로 내부가 변경될 수도 있다.

생성자 주입

메모리 **DB** 사용법

test 디렉토리에 resources 디렉토리 생성 application.yml 생성후

h2에서 제공하는 jdbc:h2:mem:test url 로 사용가능

주문 도메인 개발

주문 검색 **JPA** 동적쿼리

```
public List<Order> findAll(OrderSearch orderSearch){

   return em.createQuery("select * from Order o join o.member m " +
                   "where o.status = :status" +
                   "and m.name like :name", Order.class)
           .setParameter("status",orderSearch.getOrderStatus())
           .setParameter("name", orderSearch.getMemberName())
           .setMaxResults(1000)
           .getResultList();
}
```

서치 조건에 따른 동적 쿼리를 만들어 줘야 한다.

방법

1. JPQL 을 조건에 따라 쿼리를 만들어준다 : 에러가 많다.
2. JPA Criteria 를 사용 : 유지보수가 어렵다. 명시적으로 보이지 않는다.
3. QueryDSL 사용 !

JPA 쓸 때는 **Entity**는 핵심 비즈니스 로직만 가질 수 있게 설계하고 화면 관련 로직을 넣으면 유지 보수가 힘들다.

DTO 를 사용해서 화면에 뿌려지는 내용을 전달하는게 좋다.
API를 설계할 때는 절대 외부로 **ENTITY**를 반환해서는 안된다!!

데이터 수정

변경 감지와 병합**(merge)**

준영속 엔티티**?**

- 영속성 엔티티는 변경 내용을 바로 디비에 반영해준다. 영속성 컨텍스트가 관리하지 않는 엔티티.
- 데이터 베이스로 식별 할 수 있는 멤버를 가지고 있는 엔티티 이다.

```
Book book = new Book();
book.setId(form.getId());
book.setName(form.getName());
book.setPrice(form.getPrice());
book.setStockQuantity(form.getStockQuantity());
book.setAuthor(form.getAuthor());
book.setIsbn(form.getIsbn());
```

생성한 엔티티이지만 디비 생성된 객체**,** 영속 엔티티 처럼 식별자를 가지지만 **JPA** 가 관리하지 못한다**. (** 변경되어도 **JPA** 가 감지 못한다**)**

그러면 수정을 어떻게 해야하나?

변경 감지로 데이터를 변경

```
@Transactional
public void updateItem(Long itemId, Book param){
   Item findItem = itemRepository.findOne(itemId);
   findItem.setPrice(param.getPrice());
   findItem.setName(param.getName());
   findItem.setStockQuantity(param.getStockQuantity());
}
```

병합을 사용한 데이터 변경

```
public void save(Item item){
   if(item.getId() == null){
       em.persist(item);
   } else{
       em.merge(item);
   }}
```

merge가 실행되면 준영속 엔티티의 식별자 값으로 영속성 엔티티를 찾는다.

영속 엔티티에 준영속 엔티티 값을 채워 넣는다.

병합의 주의점 : 병합시 값이 없으면 null 로 업데이트 될 가능성이 크다 .

예) 변경할 수 있는 파라미터를 name 이라고 했더니 다른 값을 기존 값으로 추가 해주지 않고 merge를 호출해 버리면 나머지 값들이 전부 null 처리 될 수도 있다.

실무에선 **merge**로 전부 처리하기는 힘들다 **.** 그러기 때문에 변경감지로 영속성 엔티티를 조정한다고 생각하고 사용하자**. [** 변경 감지 사용하자 **]**

변경 감지 사용시

1. 컨트롤러에서 어설프게 **Entity**를 생성하지 않는다**.**

```
@PostMapping("items/{itemId}/edit")
   public String updateItem(@PathVariable("itemId") Long itemId, BookForm form){
//
//        Book book = new Book();
//        book.setId(form.getId());
//        book.setName(form.getName());
//        book.setPrice(form.getPrice());
//        book.setStockQuantity(form.getStockQuantity());
//        book.setAuthor(form.getAuthor());
//        book.setIsbn(form.getIsbn());
//
//        itemService.saveItem(book);
       itemService.updateItem(itemId,form.getName(), form.getPrice(),
form.getStockQuantity());
       return "redirect:/items";
   }
```

어설프게 Book 객체 (엔티티) 를 생성하지 않는다.

2. 트랜잭션 서비스 계층에서 직접 수정을 진행한다.
3. 트랜잭션 커밋 시점에 변경 감지가 실행된다.

따로 공부

도메인 주도 개발

[도메인 모델 패턴] < — > [트랜잭션 스크립트 패턴]

> 참고: 주문 서비스의 주문과 주문 취소 메서드를 보면 비즈니스 로직 대부분이 엔티티에 있다. 서비스 계층 은 단순히 엔티티에 필요한 요청을 위임하는 역할을 한다. 이처럼 엔티티가 비즈니스 로직을 가지고 객체 지 향의 특성을 적극 활용하는 것을 도메인 모델 패턴(http://martinfowler.com/eaaCatalog/ domainModel.html)이라 한다. 반대로 엔티티에는 비즈니스 로직이 거의 없고 서비스 계층에서 대부분 의 비즈니스 로직을 처리하는 것을 트랜잭션 스크립트 패턴(http://martinfowler.com/eaaCatalog/ transactionScript.html)이라 한다

TDD 테스트 주도 개발