05장 연관 관계 매핑

● 학습 목표

- 연관 관계 매핑의 종류와 설정방법을 알아본다.
- 매핑된 엔티티 조회 시 즉시 로딩과 지연 로딩 차이점을 이해한다.

5.1 연관 관계 매핑 종류

5.1.1. 연관 관계 매핑 종류

- DB에서 테이블은 외래키를 통해 연관 관계를 맺듯이, 대부분 엔티티는 다른 엔티티와 연관 관계를 맺고 있다.
  예를 들어, 장바구니와 회원과의 관계는 회원 하나가 장바구니를 하나 갖는 일대일 관계다. 그런데 하나의 장바구니에는 여러 상품이 들어갈 수 있기에 장바구니와 상품 관계는 일대다 관계다.

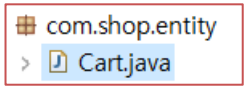
  . 일대일(1:1) @OneToOne
  . 일대다(1:N) @OneToMany
  . 다대일(N:1) @ManyToOne
  . 다대다(N:N) @ManyToMany

- 엔티티를 매핑할 때 방향성을 고려해야 하는데, 테이블에서 관계는 양방향이지만 객체에선 단방향과 양방향이 존재한다.

5.1 연관 관계 매핑 종류

5.1.2. 일대일 단방향 매핑하기

- 회원 엔티티는 이미 만들었기 때문에 장바구니(Cart)엔티티를 만들고 회원 엔티티와 연관 관계 매핑을 설정하겠다.




```
package com.shop.entity;

import lombok.Getter;
import lombok.Setter;
import lombok.ToString;
import javax.persistence.*;

@Entity
@Table(name = "cart")
@Getter @Setter
@ToString
public class Cart {

    @Id
    @Column(name = "cart_id")
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    @OneToOne   // 일대일 연관 관계 설정
    @JoinColumn(name="member_id")  // 외래키 설정
    private Member member;
}
```

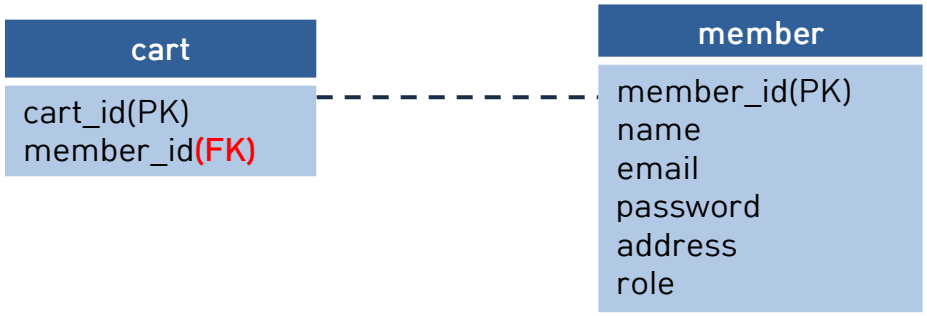
5.1 연관 관계 매핑 종류

5.1.2. 일대일 단방향 매핑하기

- 장바구니 엔티티가 일방적으로 회원 엔티티를 참조하고 있다.
- 장바구니와 회원은 일대일 매핑돼 있으며 장바구니 엔티티가 회원 엔티티를 참조하는 일대일 단방향 매핑이다.
- 코드를 실행하면 콘솔에 아래와 같이 나타나는 것을 볼 수 있다.

```
Hibernate:

    create table cart (
       cart_id bigint not null,
        member_id bigint,
        primary key (cart_id)
    )
```

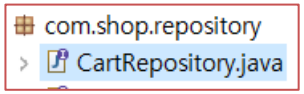
```
Hibernate:

    alter table cart
       add constraint FKix170nytunweovf2v9137mx2o
       foreign key (member_id)
       references member
```



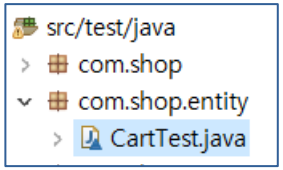
5.1 연관 관계 매핑 종류

5.1.3. 일대일 단방향 매핑하기

- CartRepository 생성하고, CartTest 테스트 케이스를 만들어 테스트해보자.




```
package com.shop.repository;

import com.shop.entity.Cart;
import org.springframework.data.jpa.repository.JpaRepository;

public interface CartRepository extends JpaRepository<Cart, Long> {

}
```



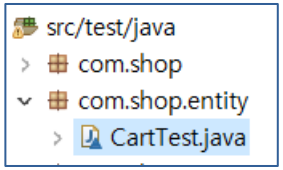

```
package com.shop.entity;

import com.shop.dto.MemberFormDto;
import com.shop.repository.CartRepository;
import com.shop.repository.MemberRepository;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.test.context.TestPropertySource;
import org.springframework.transaction.annotation.Transactional;

import javax.persistence.EntityManager;
import javax.persistence.EntityNotFoundException;
import javax.persistence.PersistenceContext;

import static org.junit.jupiter.api.Assertions.assertEquals;
```

5.1 연관 관계 매핑 종류

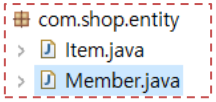
5.1.3. 일대일 단방향 매핑하기

- CartRepository 생성하고, CartTest 테스트 케이스를 만들어 테스트해보자.

```
@SpringBootTest
@Transactional
@TestPropertySource(locations="classpath:application-test.properties")

class CartTest {

    @Autowired
    CartRepository cartRepository;

    @Autowired
    MemberRepository memberRepository;

    @Autowired
    PasswordEncoder passwordEncoder;

    @PersistenceContext
    EntityManager em;

   public Member createMember() {       // 회원 엔티티를 생성하는 메소드를 만든다.
       MemberFormDto memberFormDto = new MemberFormDto();
       memberFormDto.setEmail("test@email.com");
       memberFormDto.setName("홍길동");
       memberFormDto.setAddress("서울시 마포구 합정동");
       memberFormDto.setPassword("1234");
       return Member.createMember(memberFormDto, passwordEncoder);
   }
```




```
public static Member createMember(MemberFormDto memberFormDto, PasswordEncoder passwordEncoder){
        Member member = new Member();
        member.setName(memberFormDto.getName());
        member.setEmail(memberFormDto.getEmail());
        member.setAddress(memberFormDto.getAddress());
        String password = passwordEncoder.encode(memberFormDto.getPassword());
        member.setPassword(password);
        member.setRole(Role.ADMIN);
        return member;
    }
```

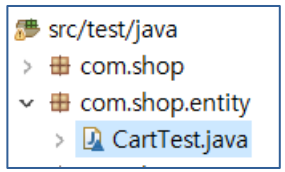


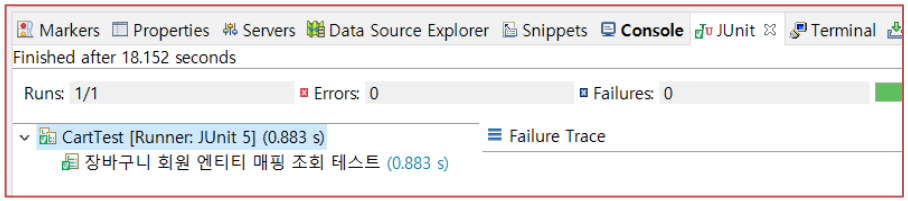
5.1 연관 관계 매핑 종류

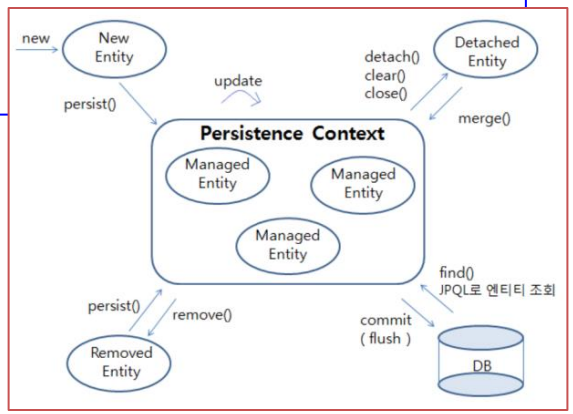
5.1.3. 일대일 단방향 매핑하기

- CartRepository 생성하고, CartTest 테스트 케이스를 만들어 테스트해보자.

```
@Test
@DisplayName("장바구니 회원 엔티티 매핑 조회 테스트")
public void findCartAndMemberTest() {
    Member member = createMember();
    memberRepository.save(member);
    Cart cart = new Cart();
    cart.setMember(member);
    cartRepository.save(cart);

    em.flush();     // 트랜잭션이 끝날 때  영속성 컨텍스트에 저장 후 flush()를 호출하여 DB에 반영한다.
    em.clear();     // 장바구니 엔티티를 가지고 올 때 회원 엔티티도 함께 가져오는지 보기 위해서 영속성 컨텍스트를 비워준다.

    Cart savedCart = cartRepository.findById(cart.getId())      //  저장된 장바구니 엔티티를 조회한다.
            .orElseThrow(EntityNotFoundException::new);
    assertEquals(savedCart.getMember().getId(), member.getId());   // 처음에 저장한 member엔티티의 id와
                                                                   // savedCart에 매핑된 member엔티티의 id를 비교한다.
}

}
```







5.1 연관 관계 매핑 종류

5.1.3. 일대일 단방향 매핑하기

- 조인

```
Cart savedCart = cartRepository.findById(cart.getId())      // 저장된 장바구니 엔티티를 조회한다.
```

상기 코드를 실행할 때는 장바구니 테이블과 회원 테이블을 조인(join)해서 가져오는 쿼리문이 실행된다.
장바구니 엔티티를 조회하면서 회원 엔티티도 동시에 가져오는 것이다. 이처럼 엔티티를 조회할 때 해당 엔티티와 매핑된 엔티티도 한 번에 조회하는 것을 '즉시 로딩'이라고 한다. 일대일이나 다대일로 매핑할 경우 즉시 로딩을 기본 Fetch전략으로 설정한다. 따로 옵션을 주지 않으면 다음과 같은 설정과 동일하다.

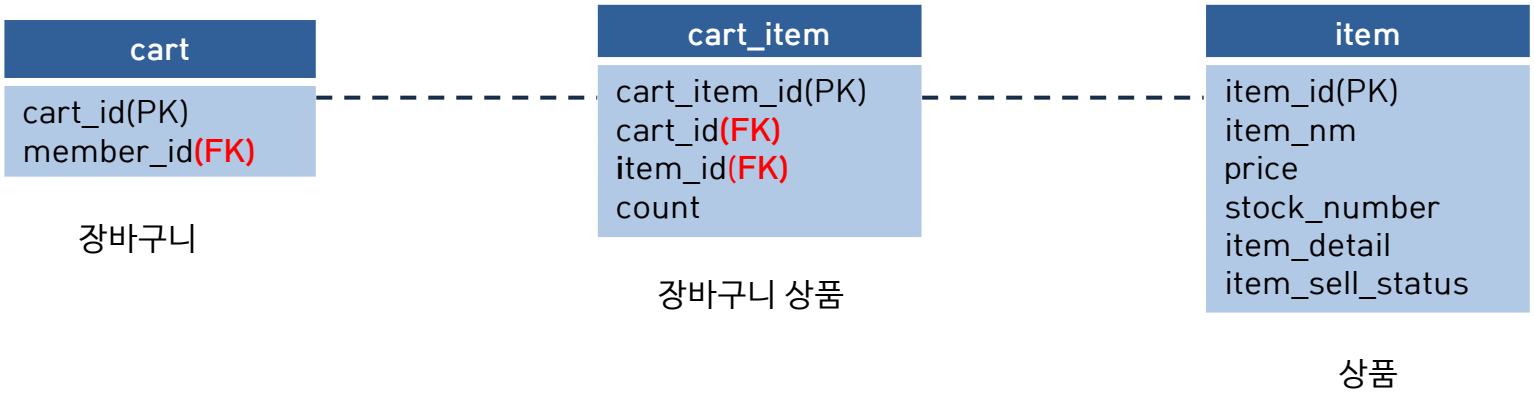
```
@OneToOne(fetch = FetchType.EAGER)
```

```
Hibernate:
    select
        cart0_.cart_id as cart_id1_0_0_,
        cart0_.member_id as member_i2_0_0_,
        member1_.member_id as member_i1_2_1_,
        member1_.address as address2_2_1_,
        member1_.email as email3_2_1_,
        member1_.name as name4_2_1_,
        member1_.password as password5_2_1_,
        member1_.role as role6_2_1_
    from
        cart cart0_
    left outer join
        member member1_
            on cart0_.member_id=member1_.member_id
    where
        cart0_.cart_id=?
```

5.1 연관 관계 매핑 종류

5.1.4. 다대일 단방향 매핑하기

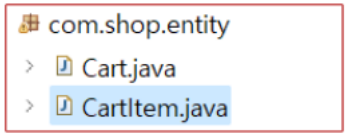
- **장바구니에는 고객이 관심이 있거나 나중에 사려는 상품들을 담아둘 것이다. 하나의 장바구니에는 여러 개의 상품들이 들어갈 수 있다. 또한 같은 상품을 여러 개 주문할 수 있으므로 몇 개를 담을 것인지 설정해줘야 한다.**



5.1 연관 관계 매핑 종류

5.1.4. 다대일 단방향 매핑하기

- 장바구니 상품 엔티티 CartItem.java 생성한다.




```
package com.shop.entity;

import lombok.Getter;
import lombok.Setter;
import javax.persistence.*;

@Entity
@Getter @Setter
@Table(name="cart_item")
public class CartItem {

    @Id
    @GeneratedValue   // 기본값은 AUTO 따라서  @GeneratedValue(strategy = GenerationType.AUTO)와 동일
    @Column(name = "cart_item_id")
    private Long id;

    @ManyToOne
    @JoinColumn(name="cart_id")     // 하나의 장바구니 상품 엔티티에는 여러 상품을 담을 수 있기에 다대일 연관 관계
    private Cart cart;

    @ManyToOne
    @JoinColumn(name = "item_id")  // 장바구니 상품 엔티티에 담을 상품 정보를 알아야 하므로 상품 엔티티를 매핑해준다.
    private Item item;

    private int count;     // 같은 상품을 장바구니에 몇 개 담을지 저장한다.
}
```

5.1 연관 관계 매핑 종류

5.1.4. 다대일 단방향 매핑하기

- **@GeneratedValue 전략**

@GeneratedValue(strategy = GenerationType.IDENTITY)

IDENTITY 전략은 기본 키 생성을 **데이터베이스에 위임**하는 전략이다. 주로 MySQL, PostgreSQL, SQL Server에서 사용한다.

예를 들어 MySQL의 **AUTO_INCREMENT** 기능은 데이터베이스가 기본 키를 자동으로 생성해준다.

IDENTITY 전략은 AUTO_INCREMENT처럼 데이터베이스에 값을 저장하고 나서야 기본 키 값을 구할 수 있을 때 사용한다.

엔티티가 영속 상태가 되기 위해서는 식별자가 필수이다.

그런데 IDENTITY 전략을 사용하면 식별자를 데이터베이스에서 지정하기 전까지는 알 수 없기 때문에, **em.persist()**를 하는 **즉시 INSERT SQL**이 데이터베이스에 전달된다. 따라서 이 전략은 트랜잭션을 지원하는 **쓰기 지연이 동작하지 않는다.**

@GeneratedValue(strategy = GenerationType.SEQUENCE)

데이터베이스 시퀀스는 유일한 값을 순서대로 생성하는 특별한 데이터베이스 오브젝트이다. SEQUENCE 전략은 이 **시퀀스를 사용해서 기본 키를 생성한다.** 이 전략은 **시퀀스를 지원**하는 **오라클**, **PostgreSQL**, **H2** 데이터베이스에서 사용할 수 있다. 시퀀스 전략을 사용하기 위해서는 우선 사용할 데이터베이스 시퀀스를 매핑해야 한다. @SequenceGenerator를 사용하여 시퀀스 생성기를 등록한 후, @GeneratedValue의 generator 속성으로 시퀀스 생성기를 선택한다.

뭔가 두 전략이 비슷해 보이지만 사실 차이가 있다. 시퀀스 전략은 em.persist()를 호출할 때 먼저 데이터베이스 시퀀스를 사용해서 식별자를 조회한다. 그리고 조회한 식별자를 엔티티에 할당한 후, 해당 엔티티를 영속성 컨텍스트에 저장한다. 이후 트랜잭션 커밋 시점에 플러시가 발생하면 엔티티를 데이터베이스에 저장한다. IDENTITY 전략은 먼저 엔티티를 데이터베이스에 저장한 후에 식별자를 조회하여, 엔티티의 식별자에 할당한 후, 영속성 컨텍스트에 저장한다. 그리고 **GenerationType.TABLE** 은 시퀀스 전략과 흡사한데 시퀀스 대신 테이블을 사용한다는 것이 차이점이다.

5.1 연관 관계 매핑 종류

5.1.4. 다대일 단방향 매핑하기

- 장바구니 상품 도메인 설계가 끝났으므로 애플리케이션을 재실행하여 콘솔에 출력되는 쿼리문을 살펴 보면 @JoinColumn을 사용하여 foreign key로 추가되는 것을 알 수 있다.

```
Hibernate:

    create table cart_item (
       cart_item_id bigint not null,
        count integer not null,
        cart_id bigint,
        item_id bigint,
        primary key (cart_item_id)
    )
```

```
Hibernate:

    alter table cart_item
       add constraint FK1uobyhgl1wvgt1jpccia8xxs3
       foreign key (cart_id)
       references cart
Hibernate:

    alter table cart_item
       add constraint FKdljf497fwm1f8eb1h8t6n50u9
       foreign key (item_id)
       references item
```

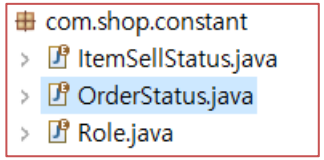
5.1 연관 관계 매핑 종류

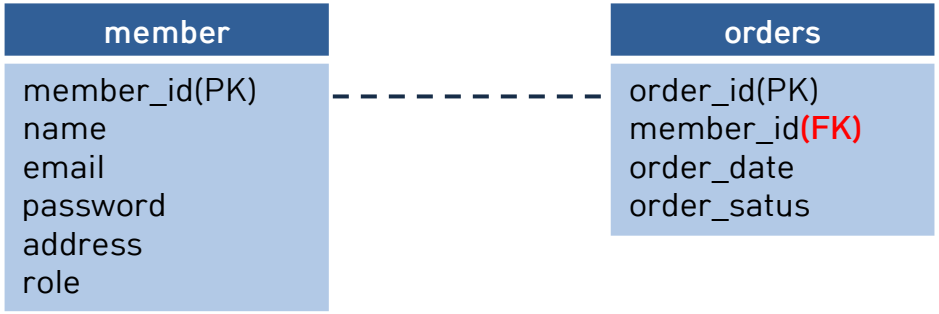
5.1.5. 다대일 / 일대다 양방향 매핑하기

- 양방향 매핑이란 단방향 매핑이 2개 있다고 생각하면 된다.

  현재는 장바구니 상품 엔티티가 장바구니를 참조하는 단방향 매핑이다. 장바구니 엔티티에 장바구니 상품 엔티티를 일대다 관계로 매핑해준다면 양방향 매핑이 된다.
- 주문과 주문 상품의 매핑을 통해 양방향 매핑을 알아보자.
- 주문 엔티티를 먼저 설계해보자.

```
package com.shop.constant;

public enum OrderStatus {
    ORDER, CANCEL
}
```

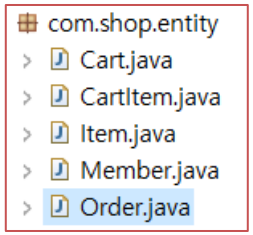




5.1 연관 관계 매핑 종류

5.1.6. 다대일 / 일대다 양방향 매핑하기

- Order 엔티티를 생성한다.

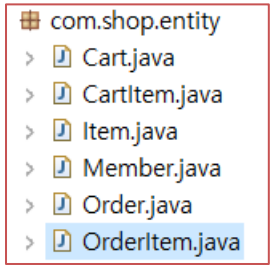



```
package com.shop.entity;

import com.shop.constant.OrderStatus;
import lombok.Getter;
import lombok.Setter;
import javax.persistence.*;
import java.time.LocalDateTime;

@Entity
@Table(name = "orders")  // 정렬할 때 사용하는 키워드 'order'가 있기 때문에 엔티티에 매핑되는 테이블로 'orders'로 지정
@Getter @Setter
public class Order {

    @Id @GeneratedValue
    @Column(name = "order_id")
    private Long id;

    @ManyToOne     // 한 명의 회원은 여러 번 주문을 할 수 있으므로 주문 엔티티 기준에서 다대일 단방향 매핑을 한다.
    @JoinColumn(name = "member_id")
    private Member member;

    private LocalDateTime orderDate; //주문일

    @Enumerated(EnumType.STRING)
    private OrderStatus orderStatus; //주문상태
    private LocalDateTime regTime;
    private LocalDateTime updateTime;
}
```

5.1 연관 관계 매핑 종류

5.1.7. 다대일 / 일대다 양방향 매핑하기

- OrderItem 엔티티를 생성한다.




```
package com.shop.entity;

import lombok.Getter;
import lombok.Setter;
import java.time.LocalDateTime;
import javax.persistence.*;

@Entity
@Getter @Setter
public class OrderItem {

    @Id @GeneratedValue
    @Column(name = "order_item_id")
    private Long id;

    @ManyToOne   // 하나의 상품은 여러 주문 상품으로 들어갈 수 있으므로 주문 상품 기준으로 다대일 매핑을 설정한다.
    @JoinColumn(name = "item_id")
    private Item item;

    @ManyToOne   // 한 번의 주문에 여러 개의 상품을 주문할 수 있으므로 주문 상품 엔티티와 주문 엔티티를 다대일 단방향 매핑
    @JoinColumn(name = "order_id")
    private Order order;

    private int orderPrice; //주문가격
    private int count; //수량
    private LocalDateTime regTime;
    private LocalDateTime updateTime;
}
```

5.1 연관 관계 매핑 종류

5.1.8. 다대일 / 일대다 양방향 매핑하기

- 엔티티는 양방향 연관 관계로 설정하면 객체 참조는 둘인데 외래키는 하나이므로 둘 중 누가 외래키를 관리할지를 결정해야 한다.
  . 연관 관계의 주인은 외래키가 있는 것으로 설정한다.
  . 연관 관계의 주인이 외래키를 관리한다(등록, 수정, 삭제).
  . 주인이 아닌 쪽은 연관 관계 매핑 시 mappedBy 속성의 값으로 연관 관계의 주인을 설정한다.
  . 주인이 아닌 쪽은 읽기만 가능하다.

5.1 연관 관계 매핑 종류

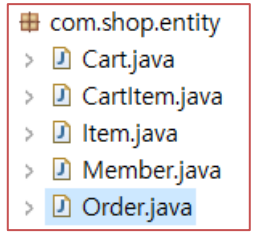
5.1.9. 다대일 / 일대다 양방향 매핑하기

- 아래 코드를 통해 상기 내용을 적용해 보자.

  Order엔티티에 OrderItem 엔티티와의 연관 관계 매핑을 추가한다. OrderItem 엔티티에서 이미 다대일 단방향 매핑을 했으므로 양방향 매핑이 된다.

- 양방향 매핑은 단방향 매핑 설계 후 필요에 따라 양방향을 설정하는 것이 복잡한 연관 관계를 피할 수 있다.

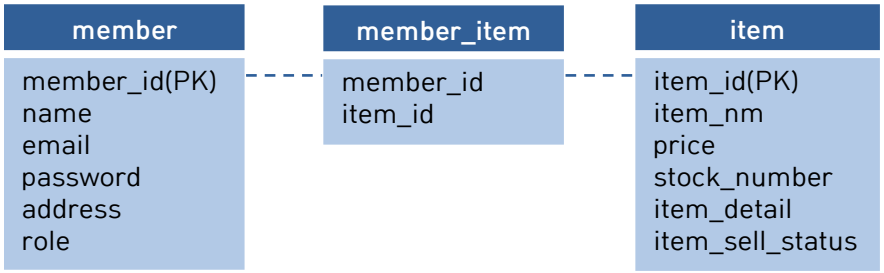
```
import java.util.ArrayList;
import java.util.List;

        :       :

 @ManyToOne
  @JoinColumn(name = "member_id")
  private Member member;
        :       :

@Enumerated(EnumType.STRING)
  private OrderStatus orderStatus; //주문상태

@OneToMany(mappedBy = "order") // 주문 상품 엔티티와 일대다 매핑한다. 외래키가 있는 주문 상품이 연관관계의 주인이다.
  private List<OrderItem> orderItems = new ArrayList<>();  // 하나의 주문에 여러 개 주문 상품을 갖기에 List 자료형 사용
```

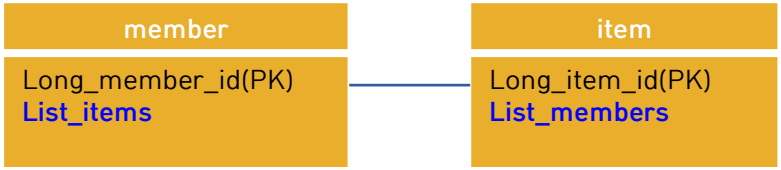


5.1 연관 관계 매핑 종류

5.1.10. 다대다 매핑하기

- 실무에서 사용하지 않는 매핑이다. 관계형 DB는 정규화된 테이블 2개로 다대다를 표현할 수 없다.
- 따라서 연결 테이블을 생성해서 다대다 관계를 일대다, 다대일 관계로 풀어낸다.
- 객체는 테이블과 다르게 컬렉션을 사용해 다대다 관계를 표현할 수 있다. (아래 코드는 설명용)





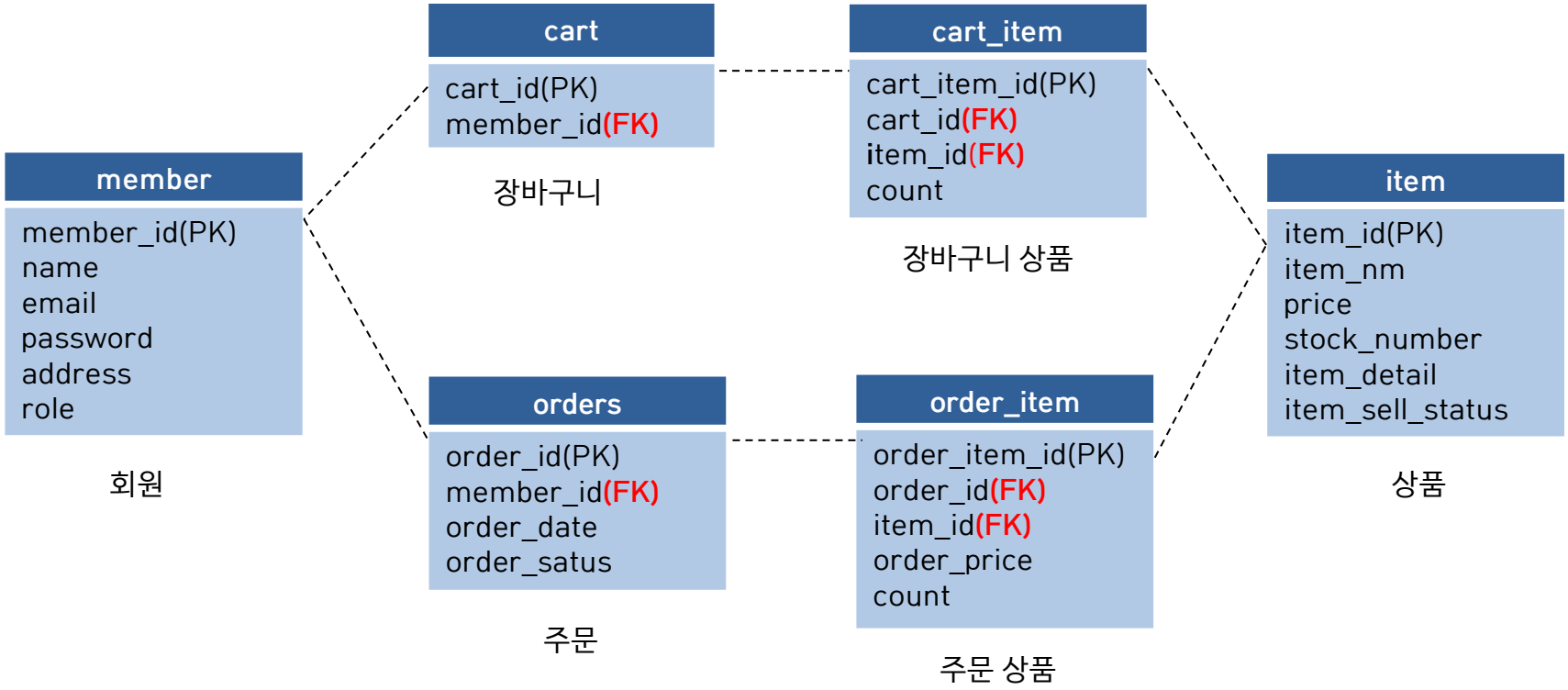

```
public class Item {

  @ManyToOne
   @JoinTable(
              name = "member_item",
              joinColumns = @JoinColumn(name = "member_id"),
              inverseJoinColumns = @JoinColumn(name = "item_id")
   )
   private List<Member> member;
}
```

다른 방식으로 구현할 수 있는
지난 자료인 [스프링부트 핵심 가이드]
3 번째 PDF 파일 97쪽
@ManytoMany 참고

5.1 연관 관계 매핑 종류

5.1.11. 연관관계

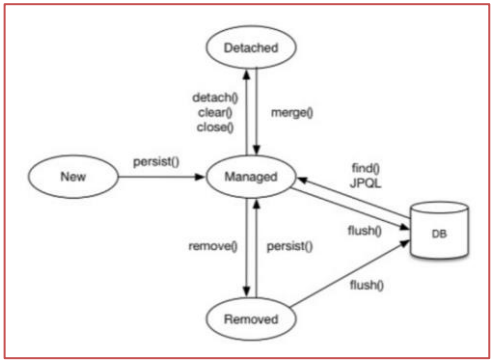
■ 전체 흐름도



5.2 영속성 전이(CASCADE)

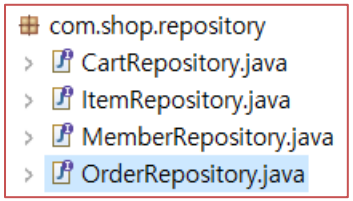
5.2.1. 영속성 전이란?

- 엔티티의 상태를 변경할 때 해당 엔티티와 연관된 엔티티 상태 변화를 전파시키는 옵션이다.
- 이때 영속성 전이에 있어서, 부모는 One에 해당되고 자식은 Many에 해당한다.
- CASCADE 종류: PERSIST, MERGE, REMOVE, REFRESH, DETACH, ALL
- 영속성 전이 옵션을 무분별하게 사용할 경우 삭제되지 말아야 할 데이터가 삭제될 수 있으니 조심해야 한다.
- 부모 엔티티와 자식 엔티티의 라이프사이클이 유사할 때 활용하는 것이 좋다.
- OrderRepository를 생성하고, Order 엔티티를 수정한다.

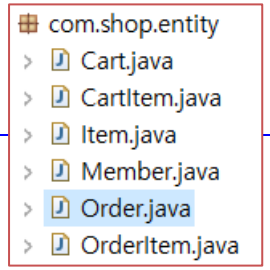


https://victorydntmd.tistory.com/207

```
package com.shop.repository;

import com.shop.entity.Order;
import org.springframework.data.jpa.repository.JpaRepository;

public interface OrderRepository extends JpaRepository<Order, Long> {

}
```

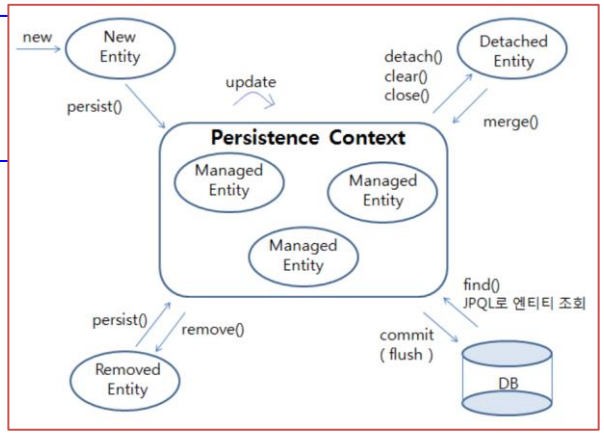



```
@OneToMany(mappedBy = "order", cascade = CascadeType.ALL)
private List<OrderItem> orderItems = new ArrayList<>();
```



orders
order_id(PK)
member_id(FK)
order_date
order_satus

order_item
order_item_id(PK)
order_id(FK)
item_id(FK)
order_price
count



5.2 영속성 전이(CASCADE)

5.2.2. 영속성 전이 테스트하기

- 영속성 전이 테스트 코드 OrderTest.java를 작성한다.

```
package com.shop.entity;

import com.shop.constant.ItemSellStatus;
import com.shop.repository.ItemRepository;
import com.shop.repository.OrderRepository;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.TestPropertySource;
import org.springframework.transaction.annotation.Transactional;

import javax.persistence.EntityManager;
import javax.persistence.EntityNotFoundException;
import javax.persistence.PersistenceContext;
import java.time.LocalDateTime;

import static org.junit.jupiter.api.Assertions.assertEquals;
import com.shop.repository.MemberRepository;

@SpringBootTest
@TestPropertySource(locations="classpath:application-test.properties")
@Transactional
class OrderTest {
```

5.2 영속성 전이(CASCADE)

5.2.2. 영속성 전이 테스트하기

- 영속성 전이 테스트 코드 OrderTest.java를 작성한다.

```
@Autowired
OrderRepository orderRepository;

@Autowired
ItemRepository itemRepository;

@PersistenceContext
EntityManager em;

@Autowired
MemberRepository memberRepository;

public Item createItem() {
    Item item = new Item();
    item.setItemNm("테스트 상품");
    item.setPrice(10000);
    item.setItemDetail("상세설명");
    item.setItemSellStatus(ItemSellStatus.SELL);
    item.setStockNumber(100);
    item.setRegTime(LocalDateTime.now());
    item.setUpdateTime(LocalDateTime.now());
    return item;
}
```

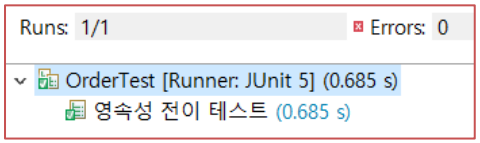
5.2 영속성 전이(CASCADE)

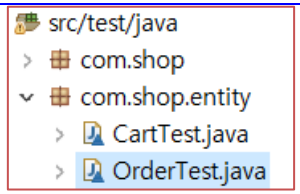
5.2.2. 영속성 전이 테스트하기

- 영속성 전이 테스트 코드 OrderTest.java를 작성한다.
- 테스트하면 콘솔에 Order에 담아 둔 OrderItem이 3번 insert되는 것을 확인할 수 있다.

```
@Test
@DisplayName("영속성 전이 테스트")
public void cascadeTest() {

    Order order = new Order();

    for(int i=0;i<3;i++) {
        Item item = this.createItem();
        itemRepository.save(item);
        OrderItem orderItem = new OrderItem();
        orderItem.setItem(item);
        orderItem.setCount(10);
        orderItem.setOrderPrice(1000);
        orderItem.setOrder(order);
        order.getOrderItems().add(orderItem); // 아직 영속성 컨텍스트에 저장되지 않은 OrderItem엔티티를 Order엔티티에 저장
    }

    orderRepository.saveAndFlush(order); // Order엔티티를 저장하면서 강제로 flush를 호출하여 영속성 컨텍스트에 있는 객체들을 DB에 반영한다.
    em.clear();    // 영속성 상태를 초기화한다.

    Order savedOrder = orderRepository.findById(order.getId()) // DB에서 주문 엔티티를 조회한다.
            .orElseThrow(EntityNotFoundException::new);
    assertEquals(3, savedOrder.getOrderItems().size());  // 주문상품 엔티티 3개가 실제로 DB에 저장되었는지 검사한다.
  }
}
```

persistence context
managed 상태






```
Hibernate:
    insert
    into
        orders
        (member_id, order_date, order_status, reg_time, update_time, order_id)
    values
        (?, ?, ?, ?, ?, ?)
```

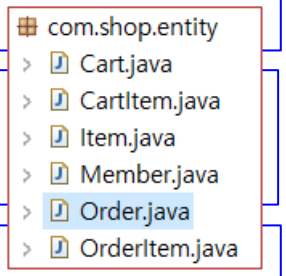
```
Hibernate:
    insert
    into
        order_item
        (count, item_id, order_id, order_price, reg_time, update_time, order_item_id)
    values
        (?, ?, ?, ?, ?, ?, ?)
```

5.2 영속성 전이(CASCADE)

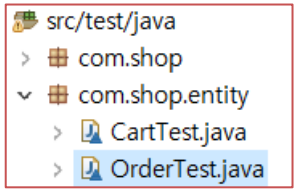
5.2.3. 고아 객체 제거하기

- 부모 엔티티와 연관 관계가 끊어진 자식 엔티티를 고아 객체라 한다.
- 고아 객체를 제거하기 위해서는 참조하는 곳이 하나일 경우에만 가능하다. 다른 곳에서 참조하고 있는 엔티티를 삭제하면 안 된다.
- @OneToOne, @OneToMany 에서 옵션으로 사용하면 된다. (직전 테스트를 주석처리 후 실행하기)

```
@OneToMany(mappedBy = "order", cascade = CascadeType.ALL ,
                orphanRemoval = true)
private List<OrderItem> orderItems = new ArrayList<>();
```




```
public Order createOrder() {       // 주문 데이터를 생성해서 저장하는 메서드를 만든다.
    Order order = new Order();
    for(int i=0;i<3;i++) {
        Item item = createItem();
        itemRepository.save(item);
        OrderItem orderItem = new OrderItem();
        orderItem.setItem(item);
        orderItem.setCount(10);
        orderItem.setOrderPrice(1000);
        orderItem.setOrder(order);
        order.getOrderItems().add(orderItem);
    }
    Member member = new Member();
    memberRepository.save(member);
    order.setMember(member);
    orderRepository.save(order);
    return order;
}
```




```
@Test
@DisplayName("고아객체 제거 테스트")
public void orphanRemovalTest() {
    Order order = this.createOrder();
    order.getOrderItems().remove(0);    // Order 엔티티에서 관리하고 있는 OrderItem 리스트의 0번째 인덱스 요소를 제거한다.
    em.flush();                         // 부모 객체와 자식 객체의 연관 관계를 끊으면 delete(삭제)가 이뤄진다.
}
```

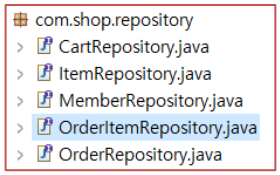
```
Hibernate:
    delete
    from
        order_item
    where
        order_item_id=?
```

5.3 지연 로딩

5.3.1. 지연 로딩

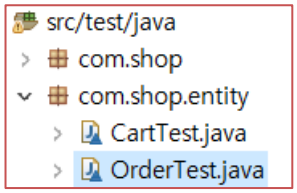
- 앞서 Fetch전략 중 Fetch.EAGER (즉시 로딩)에 대해 다뤘다. 그 반대 개념이 Fetch.LAZY(지연 로딩)이다.
- OrderItem을 조회하기 위해서 OrderItemRepository 인터페이스를 생성한다.
- 일대일 다대일 매핑 경우 기본 전략인 즉시 로딩으로 인해 해당 연관 관계 데이터를 조인해서 비효율적이게도 다 가져 온다.

```
package com.shop.repository;

import com.shop.entity.OrderItem;
import org.springframework.data.jpa.repository.JpaRepository;

public interface OrderItemRepository extends JpaRepository<OrderItem, Long> {

}
```




```
select
    orderitem0_.order_item_id as order_it1_4_0_,
    orderitem0_.count as count2_4_0_,
    orderitem0_.item_id as item_id6_4_0_,
    orderitem0_.order_id as order_id7_4_0_,
    orderitem0_.order_price as order_pr3_4_0_,
    orderitem0_.reg_time as reg_time4_4_0_,
    orderitem0_.update_time as update_t5_4_0_,
    item1_.item_id as item_id1_2_1_,
    item1_.item_detail as item_det2_2_1_,
    item1_.item_nm as item_nm3_2_1_,
    item1_.item_sell_status as item_sel4_2_1_,
    item1_.price as price5_2_1_,
    item1_.reg_time as reg_time6_2_1_,
    item1_.stock_number as stock_nu7_2_1_,
    item1_.update_time as update_t8_2_1_,
    order2_.order_id as order_id1_5_2_,
    order2_.member_id as member_i6_5_2_,
    order2_.order_date as order_da2_5_2_,
    order2_.order_status as order_st3_5_2_,
    order2_.reg_time as reg_time4_5_2_,
    order2_.update_time as update_t5_5_2_,
    member3_.member_id as member_i1_3_3_,
    member3_.address as address2_3_3_,
    member3_.email as email3_3_3_,
    member3_.name as name4_3_3_,
    member3_.password as password5_3_3_,
    member3_.role as role6_3_3_
from
    order_item orderitem0_
left outer join
    item item1_
        on orderitem0_.item_id=item1_.item_id
left outer join
    orders order2_
        on orderitem0_.order_id=order2_.order_id
left outer join
    member member3_
        on order2_.member_id=member3_.member_id
```

```
import com.shop.repository.OrderItemRepository;
        :               :
    @Autowired
    OrderItemRepository orderItemRepository;
        :               :

    @Test
    @DisplayName("지연 로딩 테스트")
    public void lazyLoadingTest() {
        Order order = this.createOrder();
        Long orderItemId = order.getOrderItems().get(0).getId();    // 기존 만들었던 주문 생성 메소드를 이용하여 주문 데이터를 저장한다.
        em.flush();
        em.clear();
        OrderItem orderItem = orderItemRepository.findById(orderItemId)  // Order엔티티에 저장했던 주문 상품 아이디를 이용하여 OrderItem을
                .orElseThrow(EntityNotFoundException::new);                                          // DB에서 조회한다.
        System.out.println("Order class : " + orderItem.getOrder().getClass());  // orderItem에 있는 order객체의 클래스들을 출력한다.
    }                                                                              // 출력 결과 -> Order class : class com.shop.entity.Order
```

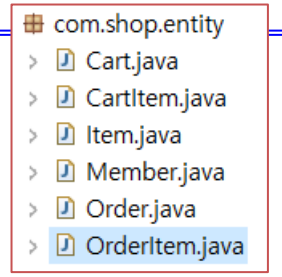


5.3 지연 로딩

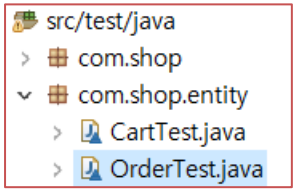
5.3.2. 지연 로딩

- 따라서 Fetch.EAGER (즉시 로딩)을 Fetch.LAZY(지연 로딩)으로 변경하여 관리하는 것이 효율적이다.
- 실무에선 거의 즉시 로딩을 사용하지 않는다.
- 지연 로딩으로 설정하면 실제 엔티티 대신에 프록시 객체를 넣어둔다. 프록시 객체는 실제로 사용되기 전까지 데이터를 로딩하지 않고, 실제 사용 시점에서 조회 쿼리문이 실행된다. (수정하고 실행해 본다)

```
@ManyToOne(fetch = FetchType.LAZY)
@JoinColumn(name = "item_id")
private Item item;

@ManyToOne(fetch = FetchType.LAZY)
@JoinColumn(name = "order_id")
private Order order;
```




```
        :               :
OrderItem orderItem = orderItemRepository.findById(orderItemId)
        .orElseThrow(EntityNotFoundException::new);
System.out.println("Order class : " + orderItem.getOrder().getClass());
System.out.println("============================");
orderItem.getOrder().getOrderDate();
System.out.println("============================");
```




```
Order class : class com.shop.entity.Order$HibernateProxy$E57Ye9cA
============================
Hibernate:
    select
        order0_.order_id as order_id1_5_0_,
        order0_.member_id as member_i6_5_0_,
        order0_.order_date as order_da2_5_0_,
        order0_.order_status as order_st3_5_0_,
        order0_.reg_time as reg_time4_5_0_,
        order0_.update_time as update_t5_5_0_,
        member1_.member_id as member_i1_3_1_,
        member1_.address as address2_3_1_,
        member1_.email as email3_3_1_,
        member1_.name as name4_3_1_,
        member1_.password as password5_3_1_,
        member1_.role as role6_3_1_
    from
        orders order0_
    left outer join
        member member1_
            on order0_.member_id=member1_.member_id
    where
        order0_.order_id=?
```

ManyToOne 의 default FetchType 은 EAGER
OneToMany 의 default FetchType 은 LAZY

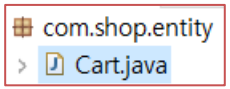
http://jaynewho.com/post/39

5.3 지연 로딩

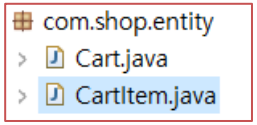
5.3.3. 지연 로딩

- Cart, CartItem, Order 엔티티 등을 Fetch.LAZY(지연 로딩)으로 변경하여 관리하는 것이 좋다.

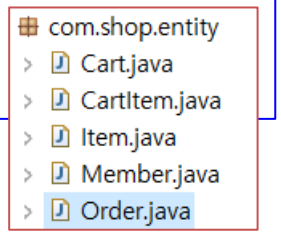
```
@OneToOne(fetch = FetchType.LAZY)
  @JoinColumn(name="member_id")
  private Member member;
```




```
@ManyToOne(fetch = FetchType.LAZY)
  @JoinColumn(name="cart_id")
  private Cart cart;

  @ManyToOne(fetch = FetchType.LAZY)
  @JoinColumn(name = "item_id")
  private Item item;
```




```
@OneToMany(mappedBy = "order", cascade = CascadeType.ALL,
    orphanRemoval = true, fetch = FetchType.LAZY)
private List<OrderItem> orderItems = new ArrayList<>();
```

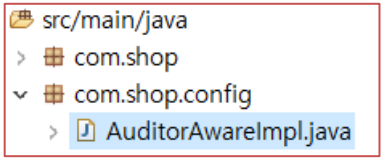


5.4 Auditing을 이용한 엔티티 공통 속성 공통화

5.4.1. 공통 속성 공통화

- Item, Order, OrederItem 엔티티에 공통으로 들어가는 멤버변수 regTime, updateTime가 있는데, 실제 서비스를 운영할 때는 보통 등록시간과 수정시간, 등록자, 수정자를 테이블에 넣어두고 활용한다. 그리고 데이터가 생성되거나 수정될 때 시간을 기록해 주고 어떤 사용자가 등록했는지 아이디를 남긴다. 이 컬럼들은 버그가 있거나 문의가 들어 왔을 때 활용이 가능하다.
- Spring Data Jpa에서는 Auditing 기능을 제공하여 엔티티가 저장 또는 수정될 때 자동으로 등록일, 수정일, 등록자, 수정자를 입력해 준다. 즉 엔티티의 생성과 수정을 감시하고(audit)있는 셈이다.
- 이런 공통 멤버 변수들을 추상 클래스로 만들고 해당 추상 클래스를 상속받는 형태로 엔티티를 리팩토링 하겠다.

```
package com.shop.config;

import org.springframework.data.domain.AuditorAware;
import org.springframework.security.core.Authentication;
import org.springframework.security.core.context.SecurityContextHolder;
import java.util.Optional;

public class AuditorAwareImpl implements AuditorAware<String> {

    @Override
    public Optional<String> getCurrentAuditor() {
        Authentication authentication = SecurityContextHolder.getContext().getAuthentication();
        String userId = "";
        if(authentication != null){
            userId = authentication.getName(); // 현재 로그인한 사용자의 정보를 조회하여 사용자의 이름을 등록자와 수정자로 등록한다.
        }
        return Optional.of(userId);
    }
}
```

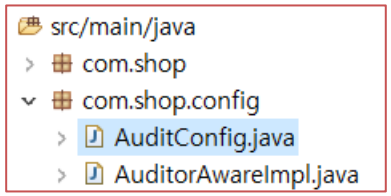


인증된 사용자 정보 Principal을 Authentication이 관리하고, Authentication은 SecutityContext가 관리하고, SecutityContext는 SecurityContextHolder가 관리한다.

5.4 Auditing을 이용한 엔티티 공통 속성 공통화

5.4.2. 공통 속성 공통화

- Auditing 기능을 사용하기 위해서 Config파일을 생성한다.



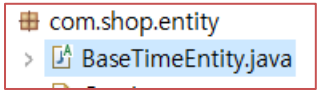

```
package com.shop.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.data.domain.AuditorAware;
import org.springframework.data.jpa.repository.config.EnableJpaAuditing;

@Configuration
@EnableJpaAuditing          // Jpa의 Auditing 기능 활성화
public class AuditConfig {

    @Bean
    public AuditorAware<String> auditorProvider() {    // 등록자와 수정자를 처리해주는 AuditorAware를 빈으로 등록한다.
        return new AuditorAwareImpl();
    }

}
```

스프링부트 핵심 가이드에서는 '등록일과 수정일'에 대한 Audit를 수행했고,
스프링부트 쇼핑몰에서는 '등록일과 수정일'뿐만 아니라 '등록자와 수정자'를 Audit도 포함 했다.

5.4 Auditing을 이용한 엔티티 공통 속성 공통화

5.4.3. 공통 속성 공통화

- 보통 테이블에 등록일, 수정일, 등록자, 수정자를 모두 다 넣어주지만, 어떤 테이블은 등록자, 수정자를 넣지 않는 테이블도 있을 수 있다. 그런 엔티티는 BaseTimeEntity만 상속받을 수 있도록 BaseTimeEntity 클래스를 생성한다.




```
package com.shop.entity;

import lombok.Getter;
import lombok.Setter;
import org.springframework.data.annotation.CreatedDate;
import org.springframework.data.annotation.LastModifiedDate;
import org.springframework.data.jpa.domain.support.AuditingEntityListener;

import javax.persistence.Column;
import javax.persistence.EntityListeners;
import javax.persistence.MappedSuperclass;
import java.time.LocalDateTime;

@EntityListeners(value = {AuditingEntityListener.class})
@MappedSuperclass  // 공통 매핑 정보가 필요할 때 사용하는 어노테이션으로, 부모 클래스를 상속 받는 자식 클래스에 매핑정보만 제공한다.
@Getter @Setter
public abstract class BaseTimeEntity {

    @CreatedDate   // 엔티티가 생성되어 저장될 때 시간을 자동으로 저장한다.
    @Column(updatable = false)
    private LocalDateTime regTime;

    @LastModifiedDate    // 엔티티 값을 변경할 때 시간을 자동으로 저장한다.
    private LocalDateTime updateTime;

}
```

5.4 Auditing을 이용한 엔티티 공통 속성 공통화

5.4.4. 공통 속성 공통화

- BaseTimeEntity 클래스를 상속한 BaseEntity 클래스를 생성한다.
- 등록일, 수정일, 등록자, 수정자를 모두 갖는 엔티티는 BaseEntity 클래스를 상속받으면 된다.

com.shop.entity
BaseEntity.java

```
package com.shop.entity;

import lombok.Getter;
import org.springframework.data.annotation.CreatedBy;
import org.springframework.data.annotation.LastModifiedBy;
import org.springframework.data.jpa.domain.support.AuditingEntityListener;

import javax.persistence.Column;
import javax.persistence.EntityListeners;
import javax.persistence.MappedSuperclass;

@EntityListeners(value = {AuditingEntityListener.class})
@MappedSuperclass
@Getter
public abstract class BaseEntity extends BaseTimeEntity{

    @CreatedBy
    @Column(updatable = false)
    private String createdBy;

    @LastModifiedBy
    private String modifiedBy;

}
```

@EntityListeners
spring data-jpa 사용시 데이터 변경시 알림을 받는 방법이다.
JPA Entity에서 이벤트가 발생할 때마다 특정 로직을 실행시킬 수 있 어노테이션이다.

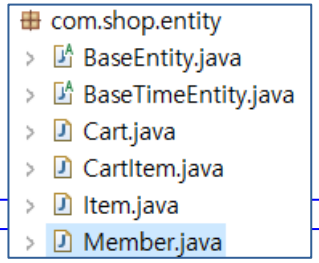
JPA Auditing 기능과 @CreatedBy, @LastModifiedBy 를 사용하여, 데이터가 생성되거나 수정될 때 유저의 ID가 DB에 저장되게 기능을 구현한다.

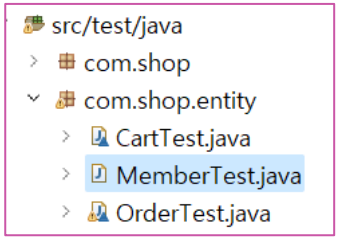
5.4 Auditing을 이용한 엔티티 공통 속성 공통화

5.4.5. 공통 속성 공통화

- Member 엔티티는 BaseEntity 클래스를 상속한다.
- 회원 엔티티 저장 시 자동으로 등록자, 수정자, 등록시간, 수정시간이 저장되는지 테스트 코드를 작성한다.

```
@Entity
@Table(name="member")
@Getter @Setter
@ToString
public class Member extends BaseEntity {
```





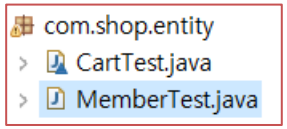

```
package com.shop.entity;

import com.shop.repository.MemberRepository;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.security.test.context.support.WithMockUser;
import org.springframework.test.context.TestPropertySource;
import org.springframework.transaction.annotation.Transactional;
import javax.persistence.EntityManager;
import javax.persistence.EntityNotFoundException;
import javax.persistence.PersistenceContext;

@SpringBootTest
@Transactional
@TestPropertySource(locations="classpath:application-test.properties")
public class MemberTest {

    @Autowired
    MemberRepository memberRepository;

    @PersistenceContext
    EntityManager em;
```

5.4 Auditing을 이용한 엔티티 공통 속성 공통화

5.4.6. 공통 속성 공통화

- 테스트 코드를 작성 후 테스트한다.
- Member 엔티티를 저장할 때 등록자나 등록일을 지정해 주지 않았지만 저장시간과 현재 로그인된 계정의 이름으로 저장된 것을 확인할 수 있다.

```
@Test
@DisplayName("Auditing 테스트")
@WithMockUser(username = "gildong", roles = "USER")
public void auditingTest(){
    Member newMember = new Member();
    memberRepository.save(newMember);

    em.flush();
    em.clear();

    Member member = memberRepository.findById(newMember.getId())
            .orElseThrow(EntityNotFoundException::new);

    System.out.println("register time : " + member.getRegTime());
    System.out.println("update time : " + member.getUpdateTime());
    System.out.println("create member : " + member.getCreatedBy());
    System.out.println("modify member : " + member.getModifiedBy());
}

}
```



@WithMockUser 오류 발생하면 pom.xml 추가

```
<dependency>
  <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-test</artifactId>
    <scope>test</scope>
</dependency>
```

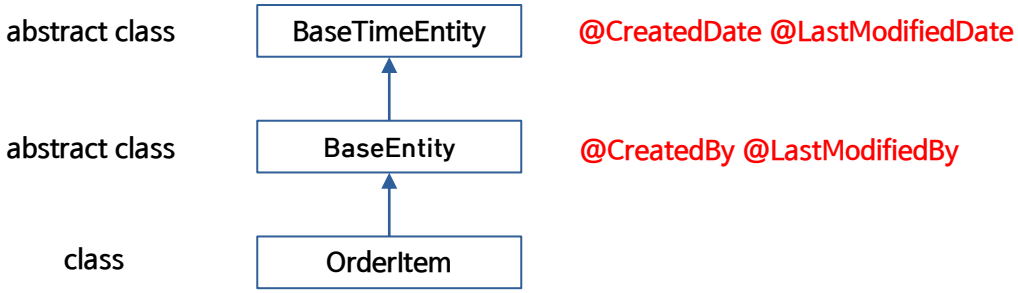
```
register time : 2022-03-25T14:50:50.249379
update time : 2022-03-25T14:50:50.249379
create member : gildong
modify member : gildong
```

5.4 Auditing을 이용한 엔티티 공통 속성 공통화

5.4.7. 공통 속성 공통화

- 모든 Cart, CartItem, Item, Order, OrederItem 엔티티들은 BaseEntity를 상속받도록 수정한다.
- Item, Order, OrederItem 에 코딩된 기존의 regTime, updateTime 변수를 삭제한다.

```
public class OrderItem extends BaseEntity {

        :       :

  private LocalDateTime regTime;

  private LocalDateTime updateTime;
```



06장 상품 등록 및 조회하기

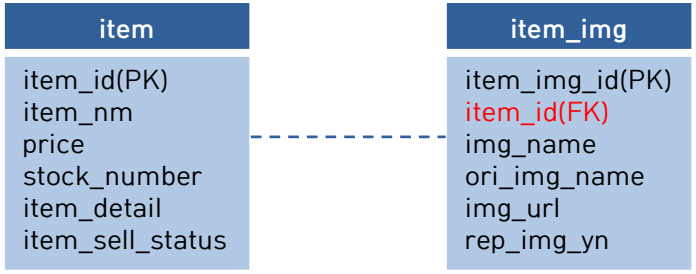
● 학습 목표

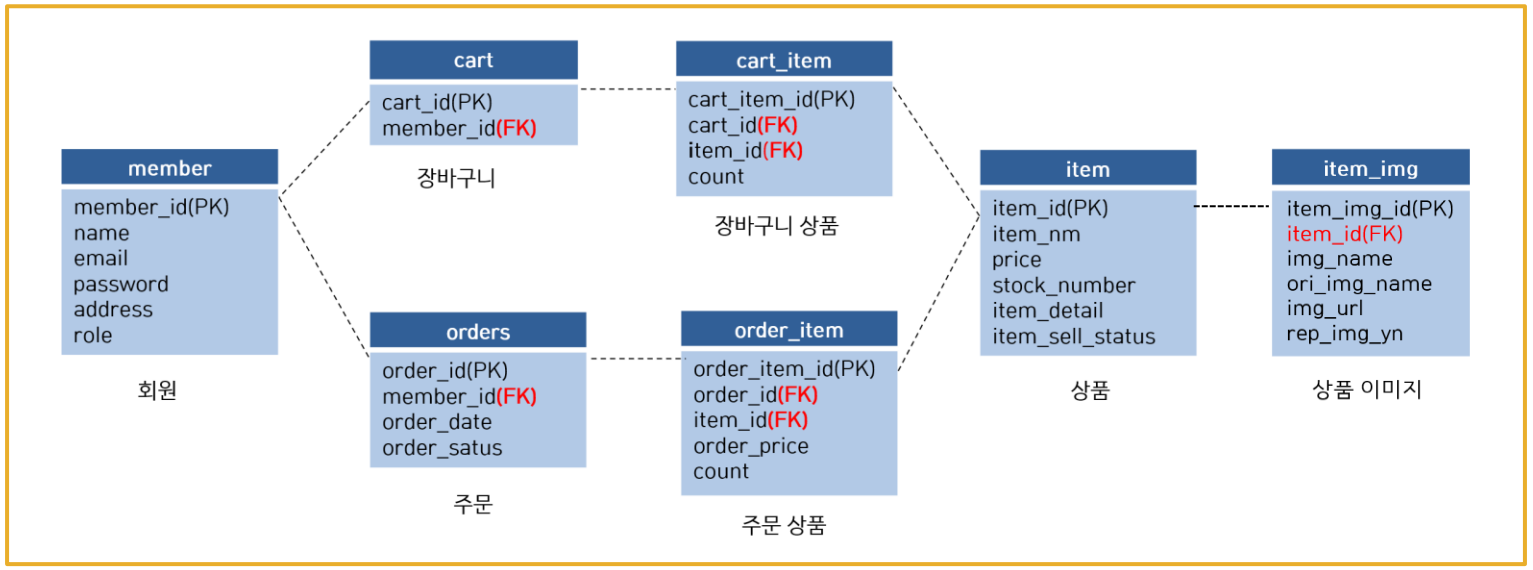
- 상품 등록 및 수정 기능을 구현하면서 Spring Data JPA를 이용해 데이터 처리하는 방법을 학습한다.
- Querydsl을 이용해 등록한 상품 데이터를 다양한 조건에 따라 조회하는 방법을 학습한다.

6.1 상품 등록하기

6.1.1. 상품 등록 구현하기

- 상품 이미지 엔티티 설계하기: 이미지 파일명, 원본 이미지 파일명, 이미지 조회경로, 대표 이미지 여부(Y일 경우, 메인 페이지 상품 보여줄 때 사용)



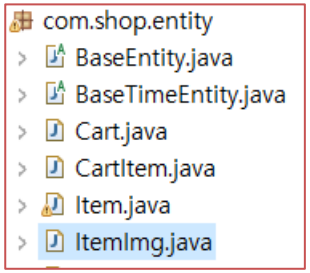


6.1 상품 등록하기

6.1.2. 상품 등록 구현하기

- ItemImg 엔티티

```
package com.shop.entity;

import lombok.Getter;
import lombok.Setter;
import javax.persistence.*;

@Entity
@Table(name="item_img")
@Getter @Setter
public class ItemImg extends BaseEntity{

    @Id
    @Column(name="item_img_id")
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    private String imgName; //이미지 파일명
    private String oriImgName; //원본 이미지 파일명
    private String imgUrl; //이미지 조회 경로
    private String repimgYn; //대표 이미지 여부

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "item_id")
    private Item item;

    public void updateItemImg(String oriImgName, String imgName, String imgUrl){
        this.oriImgName = oriImgName;
        this.imgName = imgName;
        this.imgUrl = imgUrl;
    }
}
```



원본 이미지 파일명, 업데이트할 이미지 파일명, 이미지 경로를 파라미터로 입력 받아 이미지 정보를 업데이트하는 메소드

6.1 상품 등록하기

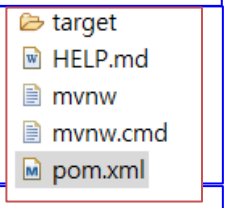
6.1.3. 상품 등록 구현하기

- ItemImgDto 생성하기: 상품 저장 후 상품 이미지에 대한 데이터를 전달할 DTO를 만든다.
- 엔티티 자체를 화면으로 반환할 수도 있지만, 그럴 때 엔티티 클래스에 화면에서만 사용하는 값이 추가된다. 따라서 DTO를 생성하여 구현하는 것이 좋다. 또한 실제 쇼핑몰에서 상품 등록 페이지는 정말 많은 데이트를 입력해야 상품을 등록할 수 있다. 특히 상품 등록 시 화면에서 DTO로 전달 받은 데이터를 엔티티로 전환, 상품검색 시 엔티티에서 화면으로 이동하기 위해 DTO로 전환 작업을 수행해야 한다. 멤버변수가 많을 때 상당시간이 걸린다. 이를 도와주는 modelmapper 라이브러리를 사용하면 좋다.

```
<dependency>
    <groupId>org.modelmapper</groupId>
    <artifactId>modelmapper</artifactId>
    <version>2.3.9</version>
</dependency>
```

만약 오류가 나면 최신 버전으로 맞춰 주거나 <version> 태그 라인 삭제

서로 다른 클래스의 값을 필드의 이름과 자료형이 같으면 getter, setter를 통해 값을 복사해서 객체를 반환해 준다

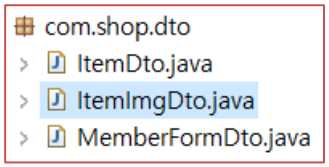



```
package com.shop.dto;

import com.shop.entity.ItemImg;
import lombok.Getter;
import lombok.Setter;
import org.modelmapper.ModelMapper;

@Getter @Setter
public class ItemImgDto {

    private Long id;
    private String imgName;
    private String oriImgName;
    private String imgUrl;
    private String repImgYn;
    private static ModelMapper modelMapper = new ModelMapper();  // 멤버 변수로 ModelMapper 객체 추가

    public static ItemImgDto of(ItemImg itemImg) {
        return modelMapper.map(itemImg, ItemImgDto.class);
    }
}
```

이미지 이름이 imgName과 oriImgName 2개인 것은 중복 이름을 처리하기 위한 것

ItemImg Entity 필드명과 데이터타입이 일치

ItemImg 엔티티 객체를 파라미터로 받아서 ItemImg 객체의 자료형과 멤버변수의 이름이 같을 때 ItemImgDto로 값을 복사해서 반환한다. static 메소드로 선언해 ItemImgDto객체를 생성하지 않아도 호출할 수 있도록 한다.



ItemService.java

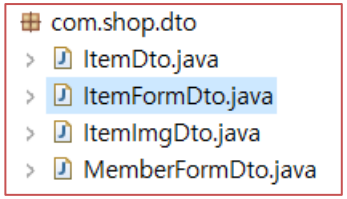
```
ItemImgDto itemImgDto = ItemImgDto.of(itemImg);
```

of(entity) : 엔티티를 DTO로 변환하는 작업을 위해 만든 메소드입니다.

6.1 상품 등록하기

6.1.4. 상품 등록 구현하기

- ItemFormDto 생성하기: (1) 상품 정보 데이터를 전달하는 DTO




```
package com.shop.dto;

import com.shop.constant.ItemSellStatus;
import com.shop.entity.Item;
import lombok.Getter;
import lombok.Setter;
import org.modelmapper.ModelMapper;
import javax.validation.constraints.NotBlank;
import javax.validation.constraints.NotNull;
import java.util.ArrayList;
import java.util.List;

@Getter @Setter
public class ItemFormDto {

    private Long id;

    @NotBlank(message = "상품명은 필수 입력 값입니다.")
    private String itemNm;

    @NotNull(message = "가격은 필수 입력 값입니다.")
    private Integer price;

    @NotBlank(message = "상품 상세는 필수 입력 값입니다.")
    private String itemDetail;

    @NotNull(message = "재고는 필수 입력 값입니다.")
    private Integer stockNumber;

    private ItemSellStatus itemSellStatus; // 상품 판매상태
```

6.1 상품 등록하기

6.1.4. 상품 등록 구현하기

▪ ItemFormDto 생성하기: (2) 상품 저장 후 상품 이미지에 대한 데이터를 전달할 DTO

```
    private List<ItemImgDto> itemImgDtoList = new ArrayList<>(); // 상품 저장 후 수정할 때 상품 이미지 정보를 저장하는 리스트

    private List<Long> itemImgIds = new ArrayList<>();


    private static ModelMapper modelMapper = new ModelMapper();

    public Item createItem(){
        return modelMapper.map(this, Item.class);
    }          // dto -> entity

    public static ItemFormDto of(Item item){
        return modelMapper.map(item, ItemFormDto.class);
    }          // entity -> dto

}
```

상품 이미지 아이디를 저장하는 리스트. 상품 등록 시에는 아직 상품의 이미지를 저장하지 않았기 때문에 아무 값도 들어가 있지 않고 수정 시에 이미지 아이디를 담아둘 용도로 사용한다.

modelMapper를 이용하여 엔티티 객체와 DTO객체 간의 데이터를 복사하여 복사한 객체를 반환해주는 메소드

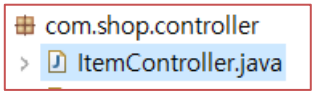
2개의 메서드 차이점은 createItem()은 ItemFormDto 객체를 생성해서 메서드를 호출하여 매핑하고, of() 메서드는 스태틱에 저장하는 메서드로서 객체생성 없이도 클래스.메서드를 호출하여 사용하게 된다.

- `toEntity(dto)` : dto를 엔티티로 변환하는 작업을 위해 만든 메소드입니다.
- `of(entity)` : 엔티티를 DTO로 변환하는 작업을 위해 만든 메소드입니다.

6.1 상품 등록하기

6.1.5. 상품 등록 구현하기

- 상품 등록 페이지로 접근할 수 있도록 ItemController 클래스를 수정한다.

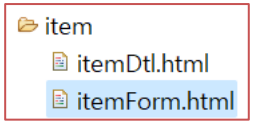



```
import org.springframework.ui.Model;
import com.shop.dto.ItemFormDto;

@Controller
public class ItemController {

    @GetMapping(value = "/admin/item/new")
    public String itemForm(Model model) {
        model.addAttribute("itemFormDto", new ItemFormDto());
        return "item/itemForm";
    }

}
```

6.1 상품 등록하기

6.1.6. 상품 등록 구현하기

▪ 상품 등록 페이지 itemForm.html를 수정한다.




```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org"
    xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout"
    layout:decorate="~{layouts/layout1}">

<!-- 사용자 스크립트 추가 -->
<th:block layout:fragment="script">

    <script th:inline="javascript">
        $(document).ready(function(){
            var errorMessage = [[${errorMessage}]]; <!--상품등록 시 실패 메시지를 받아서 상품등록 재진입 시 alert를 통해서 실패 사유를 보여준다.-->
            if(errorMessage != null){
                alert(errorMessage);
            }

            bindDomEvent();

        });

        function bindDomEvent(){
            $(".custom-file-input").on("change", function() {
                var fileName = $(this).val().split("₩₩").pop(); //이미지 파일명
                var fileExt = fileName.substring(fileName.lastIndexOf(".")+1); // 확장자 추출
                fileExt = fileExt.toLowerCase(); //소문자 변환(이미지 확장자를 소문자로 통일하여 비교하기 위함)

                if(fileExt != "jpg" && fileExt != "jpeg" && fileExt != "gif" && fileExt != "png" && fileExt != "bmp"){
                    alert("이미지 파일만 등록이 가능합니다."); <!-- 파일첨부 시 이미지 파일인지 검사한다. -->
                    return;
                }

                $(this).siblings(".custom-file-label").html(fileName); <!-- 라벨 태그 안의 내용을 jquery의 .html()을 이용하여 파일명을 입력해준다. -->
            });
        }

    </script>
</th:block>
```

객체가 아닌 string 값은 스크립트에 잘 전송이 된다. 그러나 컨트롤러에서 받은 객체 값을 전송하는 것은 CDATA로 처리해야 한다.

 # CDATA 객체 전달

```
<script th:inline="javascript">
  /*<![CDATA[*/
var query = [[${query}]];
console.log(query);
  /*]]>*/
</script>
```

6.1 상품 등록하기

6.1.6. 상품 등록 구현하기

- 상품 등록 페이지 itemForm.html를 수정한다.

```
<!-- 사용자 CSS 추가 -->
<th:block layout:fragment="css">
    <style>
        .input-group {
            margin-bottom : 15px
        }
        .img-div {
            margin-bottom : 10px
        }
        .fieldError {
            color: #bd2130;
        }
    </style>
</th:block>

<div layout:fragment="content">

    <form role="form" method="post" enctype="multipart/form-data" th:object="${itemFormDto}">

        <p class="h2">
            상품 등록
        </p>

        <input type="hidden" th:field="*{id}">

        <div class="form-group">
            <select th:field="*{itemSellStatus}" class="custom-select">
                <option value="SELL">판매중</option>
                <option value="SOLD_OUT">품절</option>
            </select>
        </div>
```

상품 판매 상태의 경우, '판매 중'과 '품절 상태'가 있다. 상품 주문이 많이 들어와서 재고가 없을 경우 주문 시 품절 상태로 바꿔줄 것이다. 또한 상품 등록만 먼저 해놓고 팔지 않을 경우에도 이용할 수 있다.

6.1 상품 등록하기

6.1.6. 상품 등록 구현하기

- 상품 등록 페이지 itemForm.html를 수정한다.

```
<div class="input-group">
    <div class="input-group-prepend">
        <span class="input-group-text">상품명</span>
    </div>
    <input type="text" th:field="*{itemNm}" class="form-control" placeholder="상품명을 입력해주세요">
</div>
<p th:if="${#fields.hasErrors('itemNm')}" th:errors="*{itemNm}" class="fieldError">Incorrect data</p>

<div class="input-group">
    <div class="input-group-prepend">
        <span class="input-group-text">가격</span>
    </div>
    <input type="number" th:field="*{price}" class="form-control" placeholder="상품의 가격을 입력해주세요">
</div>
<p th:if="${#fields.hasErrors('price')}" th:errors="*{price}" class="fieldError">Incorrect data</p>

<div class="input-group">
    <div class="input-group-prepend">
        <span class="input-group-text">재고</span>
    </div>
    <input type="number" th:field="*{stockNumber}" class="form-control" placeholder="상품의 재고를 입력해주세요">
</div>
<p th:if="${#fields.hasErrors('stockNumber')}" th:errors="*{stockNumber}" class="fieldError">Incorrect data</p>

<div class="input-group">
    <div class="input-group-prepend">
        <span class="input-group-text">상품 상세 내용</span>
    </div>
    <textarea class="form-control" aria-label="With textarea" th:field="*{itemDetail}"></textarea>
</div>
<p th:if="${#fields.hasErrors('itemDetail')}" th:errors="*{itemDetail}" class="fieldError">Incorrect data</p>
```

```
@PostMapping(value = "/admin/item/new")
public String itemNew(@Valid ItemFormDto itemFormDto, BindingResult bindingResult,
        Model model, @RequestParam("itemImgFile") List<MultipartFile> itemImgFileList) {
```

ItemController.java

th:errors는 th:field="*{itemNm}"을 이용해 th:errors:"*{itemNm}"으로 오류가 있을 경우 bindingResult에 담긴 해당 필드의 오류 메세지를 text로 보여줄 수도 있다.

6.1 상품 등록하기

6.1.6. 상품 등록 구현하기

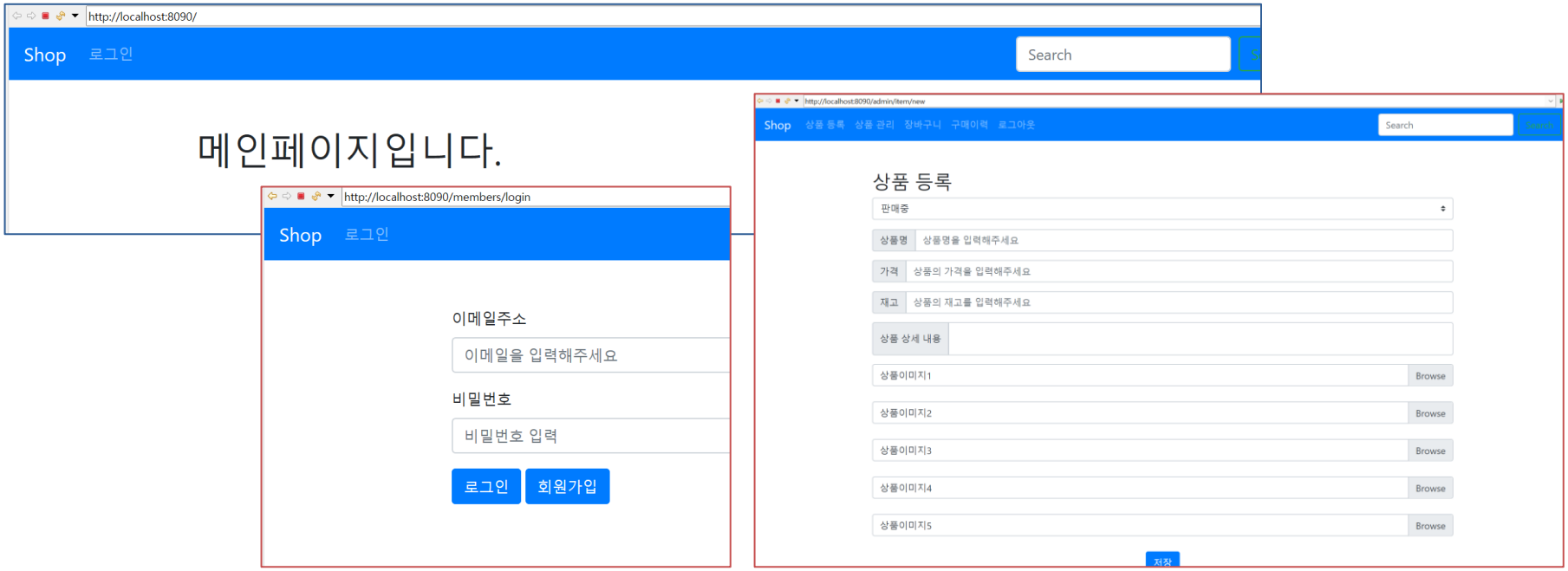
▪ 상품 등록 페이지 itemForm.html를 수정한다.

```
    <div th:if="${#lists.isEmpty(itemFormDto.itemImgDtoList)}"> <!-- 상품 이미지 정보를 담고 있는 리스트가 비어 있다면 상품 등록한다. -->
      <div class="form-group" th:each="num: ${#numbers.sequence(1,5)}">
        <div class="custom-file img-div">
          <input type="file" class="custom-file-input" name="itemImgFile">
          <label class="custom-file-label" th:text="상품이미지 + ${num}"></label>
        </div>
      </div>
    </div>

    <div th:if = "${not #lists.isEmpty(itemFormDto.itemImgDtoList)}">
      <div class="form-group" th:each="itemImgDto, status: ${itemFormDto.itemImgDtoList}">
        <div class="custom-file img-div">
          <input type="file" class="custom-file-input" name="itemImgFile">
          <input type="hidden" name="itemImgIds" th:value="${itemImgDto.id}">
          <label class="custom-file-label" th:text="${not #strings.isEmpty(itemImgDto.oriImgName)} ? ${itemImgDto.oriImgName} : '상품이미지' +
${status.index+1}"></label> <!--타임리프 유틸리티 객체인 #strings.isEmpty(string)을 이용하여 저장이미지 정보가 있다면 파일명을 보여주고 없다면 상품이미지번호출력-->
        </div>
      </div>
    </div>

    <div th:if="${#strings.isEmpty(itemFormDto.id)}" style="text-align: center">
      <button th:formaction="@{/admin/item/new}" type="submit" class="btn btn-primary">저장</button>
    </div>
    <div th:unless="${#strings.isEmpty(itemFormDto.id)}" style="text-align: center">
      <button th:formaction="@{'/admin/item/' + ${itemFormDto.id} }" type="submit" class="btn btn-primary">수정</button>
    </div>
    <input type="hidden" th:name="${_csrf.parameterName}" th:value="${_csrf.token}">

  </form>

</div>

</html>
```

6.1 상품 등록하기

6.1.7. 상품 등록 구현하기

- application.properties 수정한다.
  spring.jpa.hibernate.ddl-auto=validate
  설정이 validate이면 애플리케이션 실행시점에 테이블을 삭제한 후 재생성하지 않으며 엔티티와 테이블이 매핑이 정상적으로 되어 있는지만 확인한다. 엔티티를 추가가 필요할 경우 create와 validate를 번갈아 가면서 사용하면 편리하다.
  application.properties의 설정이 validate이면 테스트 코드 실행 시 테이블이 자동으로 생성되지 않으므로 테스트 환경에선 create로 설정한다.
- application-test.properties 수정한다.
  spring.jpa.hibernate.ddl-auto=create

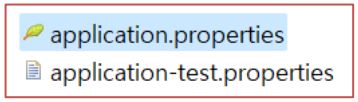


6.1 상품 등록하기

6.1.8. 상품 등록 구현하기

- 이제 상품 등록을 위한 로직을 구현한다. 상품을 저장하는 로직과 수정하는 로직을 만든다.
- 이미지 파일을 등록할 때 서버에서 각 파일의 최대 사이즈와 한 번에 다운 요청할 수 있는 파일의 크기를 지정할 수 있다.
- application.properties에서 설정한다.
- C:₩shop₩item 폴더를 만든다.

```
#파일 한 개당 최대 사이즈
spring.servlet.multipart.maxFileSize=20MB
#요청당 최대 파일 크기
spring.servlet.multipart.maxRequestSize=100MB
#상품 이미지 업로드 경로
itemImgLocation=C:/shop/item
#리소스 업로드 경로
uploadPath=file:///C:/shop/
```

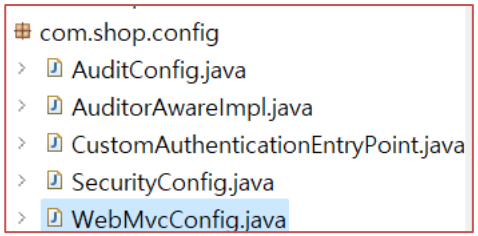


6.1 상품 등록하기

6.1.9. 상품 등록 구현하기

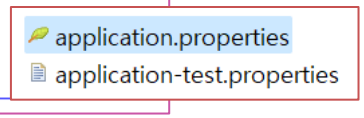
- 업로드한 파일을 읽어 올 경로를 설정한다.
- WebMvcConfigurer인터페이스를 구현한 WebMvcConfig.java를 작성한다.
- addResourceHandlers메소드를 통해 자신의 로컬 컴퓨터에 업로드한 파일을 찾을 위치를 설정한다.

```
package com.shop.config;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.ResourceHandlerRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

@Configuration
public class WebMvcConfig implements WebMvcConfigurer {

    @Value("${uploadPath}")
    String uploadPath;

    @Override
    public void addResourceHandlers(ResourceHandlerRegistry registry) {
        registry.addResourceHandler("/images/**")  // 브라우저 url에 대해 /images로 시작하는 경우 uploadPath에 설정한 폴더를 기준으로
                .addResourceLocations(uploadPath);  // 파일을 읽어 오도록 한다.
    }

}
```



Spring Boot에서 외부 파일(ex.application.properties, application.yml)에 있는 값들을 소스 코드에 주입해서 사용하는 경우 사용하는 어노테이션이다.

```
#파일 한 개당 최대 사이즈
spring.servlet.multipart.maxFileSize=20MB
#요청당 최대 파일 크기
spring.servlet.multipart.maxRequestSize=100MB
#상품 이미지 업로드 경로
itemImgLocation=C:/shop/item
#리소스 업로드 경로
uploadPath=file:///C:/shop/
```



6.1 상품 등록하기

6.1.9. 상품 등록 구현하기

- WebMvcConfigurer인터페이스 추상메서드 기능 소개
- 1. CORS 설정 기능 / 2. ViewResolver 기능 / 3. 정적 Resource 매핑 기능

1. CORS 설정 기능
CORS(Cross-Origin Resource Sharing,CORS) 란 다른 출처의 자원을 공유할 수 있도록 설정하는 권한 체제를 말하는데,
WebMvcConfigurer addCorsMappings 메서드로 global corss origin request process를 설정한다.

```
@Configuration
public class CorsConfig implements WebMvcConfigurer {

@Override
public void addCorsMappings(CorsRegistry registry) {
  registry.addMapping("/**")
    .allowedOrigins("http://127.0.0.1:8080")
    .allowedOrigins("http://localhost:8080");
  }
}
```

2. ViewResolver 기능

```
@Override
public void configureViewResolvers(ViewResolverRegistry registry) {
registry.jsp("/WEB-INF/",".jsp");
}
```

6.1 상품 등록하기

6.1.9. 상품 등록 구현하기

- WebMvcConfigurer인터페이스 추상메서드 기능 소개
- CORS 설정 기능 / 2. ViewResolver 기능 / 3. 정적 Resource 매핑 기능

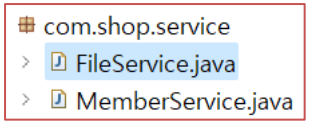
3. 정적 Resource 매핑 기능

```
@Configuration
@EnableWebMvc
public class MvcConfig implements WebMvcConfigurer {
// 개발 시점에 사용 가능한 코드.
@Override
public void addResourceHandlers(ResourceHandlerRegistry registry) {
  registry
    .addResourceHandler("/resources/**")
    .addResourceLocations("/resources/");
}
// 배포 시점에 사용 가능한 코드.
@Override
public void addResourceHandlers(ResourceHandlerRegistry registry) {
  registry
    .addResourceHandler("/files/**")
    .addResourceLocations("file:/opt/files/");
// 윈도우라면
.addResourceLocations("file:///C:/opt/files/");
  }
}
```

6.1 상품 등록하기

6.1.10. 상품 등록 구현하기

- 파일을 처리하는 FileService 클래스를 만든다.
- 파일을 업로드하는 메소드와 삭제하는 메소드를 작성한다.



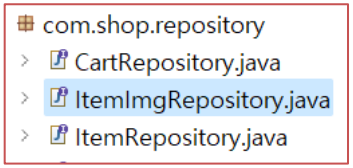

```
package com.shop.service;

import lombok.extern.java.Log;
import org.springframework.stereotype.Service;
import java.io.File;
import java.io.FileOutputStream;
import java.util.UUID;

@Service
@Log
public class FileService {

    public String uploadFile(String uploadPath, String originalFileName, byte[] fileData) throws Exception{
        UUID uuid = UUID.randomUUID(); // Universally Unique Identifier는 서로 다른 객체들을 구별하기 위해서 이름을 부여할 때 사용한다.(파일명 중복문제 해결)
        String extension = originalFileName.substring(originalFileName.lastIndexOf("."));
        String savedFileName = uuid.toString() + extension; // UUID로 받은 값과 원래 파일의 이름의 확장자를 조합해서 저장될 파일 이름을 만든다.
        String fileUploadFullUrl = uploadPath + "/" + savedFileName;
        FileOutputStream fos = new FileOutputStream(fileUploadFullUrl); // 바이트 단위의 출력을 내보내는 클래스로서 생성자로 파일이 저장될 위치와 파일의 이름을 넘겨
        fos.write(fileData);  // fileData를 파일 출력 스트림에 입력한다.                                                    파일에 쓸 파일 출력 스트림을 만든다.
        fos.close();
        return savedFileName;  // 업로드된 파일의 이름을 반환한다.
    }

    public void deleteFile(String filePath) throws Exception{
        File deleteFile = new File(filePath);  // 파일이 저장된 경로를 이용하여 파일 객체를 생성한다.
        if(deleteFile.exists()) {                  // 해당 파일이 존재하면 파일을 삭제한다.
            deleteFile.delete();
            log.info("파일을 삭제하였습니다.");
        } else {
            log.info("파일이 존재하지 않습니다.");
        }
    }
}
```

6.1 상품 등록하기

6.1.11. 상품 등록 구현하기

- 상품의 이미지 정보를 저장하기 위한 ItemImgRepository인터페이스를 만든다.

```
package com.shop.repository;

import com.shop.entity.ItemImg;
import org.springframework.data.jpa.repository.JpaRepository;

public interface ItemImgRepository extends JpaRepository<ItemImg, Long> {


}
```



6.1 상품 등록하기

6.1.12. 상품 등록 구현하기

▪ 상품 이미지를 업로드하고 상품 이미지 정보를 저장하는 ItemImgService 클래스를 만든다.

```
package com.shop.service;

import com.shop.entity.ItemImg;
import com.shop.repository.ItemImgRepository;
import lombok.RequiredArgsConstructor;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
import org.springframework.web.multipart.MultipartFile;
import org.thymeleaf.util.StringUtils;

@Service
@RequiredArgsConstructor
@Transactional
public class ItemImgService {

    @Value("${itemImgLocation}") // applicatin.properties에 등록한 itemImgLocation값을 불러와
    private String itemImgLocation;  // 변수  itemImgLocation 에 넣는다.

    private final ItemImgRepository itemImgRepository;
    private final FileService fileService;

    public void saveItemImg(ItemImg itemImg, MultipartFile itemImgFile) throws Exception{
        String oriImgName = itemImgFile.getOriginalFilename();
        String imgName = "";
        String imgUrl = "";

        //파일 업로드
        if(!StringUtils.isEmpty(oriImgName)){
            imgName = fileService.uploadFile(itemImgLocation, oriImgName,
                    itemImgFile.getBytes());  // 사용자가 상품 이미지를 등록했다면 저장할 경로, 파일 이름, 파일 바이트수를 파라미터로 하는 uploadFile 매소드를 호출한다.
            imgUrl = "/images/item/" + imgName; // 저장한 상품 이미지를 불러올 경로를 설정한다.  C:/shop/images/item/
        }

        //상품 이미지 정보 저장
        itemImg.updateItemImg(oriImgName, imgName, imgUrl); // 업로드했던 상품 이미지 파일의 원래 이름, 실제 로컬에 저장된 상품 이미지 파일의 이름,
        itemImgRepository.save(itemImg);                    // 업로드 결과 로컬에 저장된 상품  이미지 파일을 불러오는 경로 등의 상품 이미지 정보를 저장한다
    }
}
```

com.shop.service
- FileService.java
- ItemImgService.java
- MemberService.java

application.properties

```
itemImgLocation=C:/shop/item
uploadPath=file:///C:/shop/
```

Spring Boot 프로젝트가 커지면 공통으로 사용되는 글로벌 값을 별도로 관리할 필요가 생긴다. @Value 어노테이션은 properties 파일에 세팅한 내용을 Spring 변수에 주입하는 역할을 한다.

@Value 어노테이션의 사용법
1. @Value("문자열") 사용 - "문자열" 을 해당 변수에 대입

2. @Value("${...}") 사용 - application.properties 에 정의한 내용을 가져와서 해당 변수에 대입

6.1 상품 등록하기

6.1.13. 상품 등록 구현하기

- 상품을 등록하는 ItemService 클래스를 만든다.

```
package com.shop.service;

import com.shop.dto.ItemFormDto;
import com.shop.entity.Item;
import com.shop.entity.ItemImg;
import com.shop.repository.ItemImgRepository;
import com.shop.repository.ItemRepository;
import lombok.RequiredArgsConstructor;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
import org.springframework.web.multipart.MultipartFile;
import java.util.List;

@Service
@Transactional
@RequiredArgsConstructor
public class ItemService {

    private final ItemRepository itemRepository;
    private final ItemImgService itemImgService;
    private final ItemImgRepository itemImgRepository;

    public Long saveItem(ItemFormDto itemFormDto, List<MultipartFile> itemImgFileList) throws Exception{
        //상품 등록
        Item item = itemFormDto.createItem(); // 상품 등록 폼으로부터 입력 받은 데이터를 이용하여 item 객체를 생성한다.
        itemRepository.save(item);    // 상품 데이터를 저장한다.
        //이미지 등록
        for(int i=0;i<itemImgFileList.size();i++) {
            ItemImg itemImg = new ItemImg();
            itemImg.setItem(item);

            if(i == 0)    // 첫 번째 이미지일 경우 대표 상품 이미지 여부 값을 Y로 세팅한다. 나머지 상품 이미지는 N으로 설정한다.
                itemImg.setRepimgYn("Y");
            else
                itemImg.setRepimgYn("N");
            itemImgService.saveItemImg(itemImg, itemImgFileList.get(i)); // 상품 이미지 정보를 저장한다.
        }

        return item.getId();
    }
}
```

com.shop.service
- FileService.java
- ItemImgService.java
- ItemService.java

ItemFormDto.java

```
public Item createItem(){
    return modelMapper.map(this, Item.class);
}
```

ItemFormDto -> Item

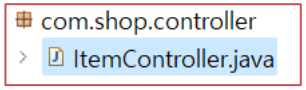
ItemImgService.java

```
public void saveItemImg(ItemImg itemImg, MultipartFile itemImgFile)
throws Exception{
        String oriImgName = itemImgFile.getOriginalFilename();
        String imgName = "";
        String imgUrl = "";
```

6.1 상품 등록하기

6.1.14. 상품 등록 구현하기

- 상품을 등록하는 url을 ItemController 클래스에 추가한다.




```
package com.shop.controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.ui.Model;
import com.shop.dto.ItemFormDto;
import com.shop.service.ItemService;
import lombok.RequiredArgsConstructor;
import org.springframework.web.bind.annotation.PostMapping;
import javax.validation.Valid;
import org.springframework.validation.BindingResult;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.multipart.MultipartFile;
import java.util.List;


@Controller
@RequiredArgsConstructor
public class ItemController {

    private final ItemService itemService;

    @GetMapping(value = "/admin/item/new")
    public String itemForm(Model model) {
        model.addAttribute("itemFormDto", new ItemFormDto());
        return "item/itemForm";
    }
```

6.1 상품 등록하기


```
<form role="form" method="post" enctype="multipart/form-data" th:object="${itemFormDto}">
```

itemForm.html

6.1.14. 상품 등록 구현하기

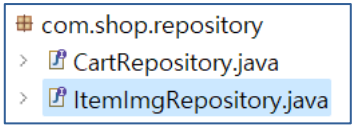
- 상품을 등록하는 url을 ItemController 클래스에 추가한다.

```
    @PostMapping(value = "/admin/item/new")
    public String itemNew(@Valid ItemFormDto itemFormDto, BindingResult bindingResult,
                          Model model, @RequestParam("itemImgFile") List<MultipartFile> itemImgFileList) {

        if(bindingResult.hasErrors()) {   // 상품 등록시 필수 값이 없다면 다시 상품 등록 페이지로 전환한다.
            return "item/itemForm";
        }

        if(itemImgFileList.get(0).isEmpty() && itemFormDto.getId() == null) {
            model.addAttribute("errorMessage", "첫번째 상품 이미지는 필수 입력 값 입니다.");
            return "item/itemForm";  // 상품 등록시 첫 번째 이미지가 없다면 에러 메시지와 함께 상품등록 페이지로 전환한다.
        }                            // 상품 첫번째 이미지는 메인  페이지에서 보여줄 상품 이미지를 사용하기 위해 필수 값으로 지정한다.

        try {
            itemService.saveItem(itemFormDto, itemImgFileList);  // 상품 저장 로직을 호출. 상품정보와 상품이미지정보를 넘긴다.
        } catch (Exception e) {
            model.addAttribute("errorMessage", "상품 등록 중 에러가 발생하였습니다.");
            return "item/itemForm";
        }

        return "redirect:/";  // 정상적으로 등록되었다면 메인페이지로 이동한다.
    }

}
```

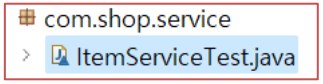
6.1 상품 등록하기

6.1.15. 상품 등록 구현하기

- 이미지가 잘 저장됐는지 테스트 코드를 작성하기 위해서 ItemImgRepository 인터페이스에 findByItemIdOrderByIdAsc 메소드를 추가한다.
- 이제 상품저장 로직 테스트를 작성한다. 소스를 수정할 일이 많아질수록 테스트 케이스를 잘 만드는 것이 중요하다.




```
package com.shop.repository;

import com.shop.entity.ItemImg;
import java.util.List;
import org.springframework.data.jpa.repository.JpaRepository;

public interface ItemImgRepository extends JpaRepository<ItemImg, Long> {

    List<ItemImg> findByItemIdOrderByIdAsc(Long itemId);
}
```

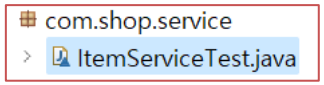



```
package com.shop.service;

import com.shop.constant.ItemSellStatus;
import com.shop.dto.ItemFormDto;
import com.shop.entity.Item;
import com.shop.entity.ItemImg;
import com.shop.repository.ItemImgRepository;
import com.shop.repository.ItemRepository;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.mock.web.MockMultipartFile;
import org.springframework.security.test.context.support.WithMockUser;
import org.springframework.test.context.TestPropertySource;
import org.springframework.transaction.annotation.Transactional;
import org.springframework.web.multipart.MultipartFile;
import javax.persistence.EntityNotFoundException;
import java.util.ArrayList;
import java.util.List;
import static org.junit.jupiter.api.Assertions.assertEquals;
```

6.1 상품 등록하기

6.1.15. 상품 등록 구현하기

- ItemServiceTest




```
@SpringBootTest
@Transactional
@TestPropertySource(locations="classpath:application-test.properties")
class ItemServiceTest {

    @Autowired
    ItemService itemService;

    @Autowired
    ItemRepository itemRepository;

    @Autowired
    ItemImgRepository itemImgRepository;

    List<MultipartFile> createMultipartFiles() throws Exception{

        List<MultipartFile> multipartFileList = new ArrayList<>();

        for(int i=0;i<5;i++){
            String path = "C:/shop/item/";
            String imageName = "image" + i + ".jpg";
            MockMultipartFile multipartFile =
                    new MockMultipartFile(path, imageName, "image/jpg", new byte[]{1,2,3,4});
            multipartFileList.add(multipartFile);
        }

        return multipartFileList;
    }
```

6.1 상품 등록하기

6.1.15. 상품 등록 구현하기

- ItemServiceTest 실행하기

```
@Test
@DisplayName("상품 등록 테스트")
@WithMockUser(username = "admin", roles = "ADMIN")
void saveItem() throws Exception {
    ItemFormDto itemFormDto = new ItemFormDto();
    itemFormDto.setItemNm("테스트상품");
    itemFormDto.setItemSellStatus(ItemSellStatus.SELL);
    itemFormDto.setItemDetail("테스트 상품 입니다.");
    itemFormDto.setPrice(1000);
    itemFormDto.setStockNumber(100);

    List<MultipartFile> multipartFileList = createMultipartFiles();
    Long itemId = itemService.saveItem(itemFormDto, multipartFileList);

    List<ItemImg> itemImgList = itemImgRepository.findByItemIdOrderByIdAsc(itemId);

    Item item = itemRepository.findById(itemId)
            .orElseThrow(EntityNotFoundException::new);

    assertEquals(itemFormDto.getItemNm(), item.getItemNm());  // 입력한 상품 데이터와 실제 저장된 상품 데이터가 같은지 확인한다.
    assertEquals(itemFormDto.getItemSellStatus(), item.getItemSellStatus());
    assertEquals(itemFormDto.getItemDetail(), item.getItemDetail());
    assertEquals(itemFormDto.getPrice(), item.getPrice());
    assertEquals(itemFormDto.getStockNumber(), item.getStockNumber());
    assertEquals(multipartFileList.get(0).getOriginalFilename(), itemImgList.get(0).getOriImgName());
}

}
```

com.shop.service
ItemServiceTest.java

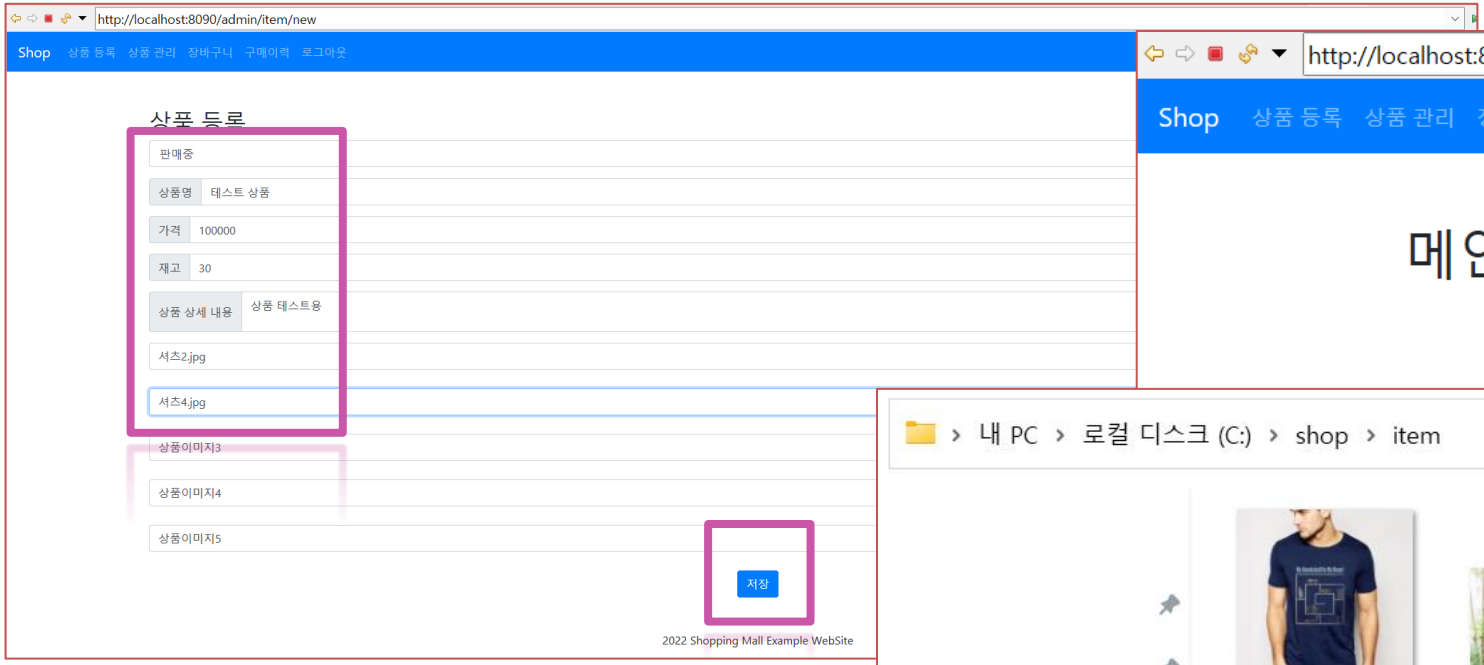
Runs: 1/1 Errors: 0

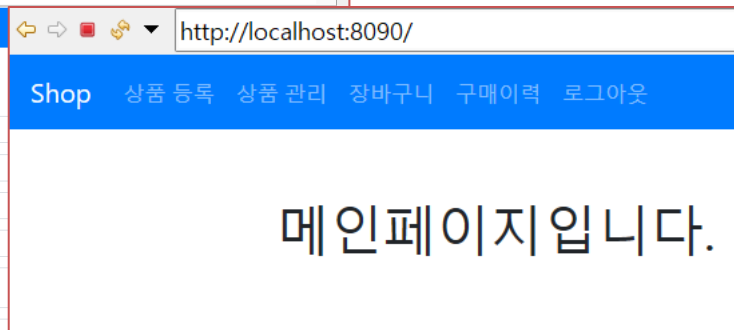
ItemServiceTest [Runner: JUnit 5] (0.409 s)
상품 등록 테스트 (0.409 s)

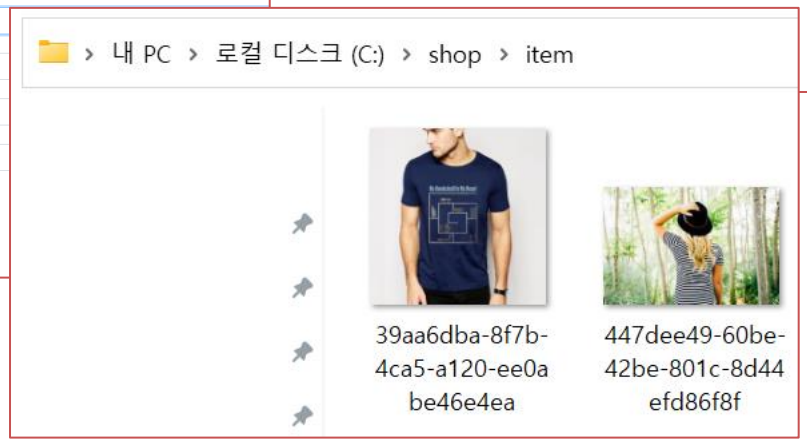
6.1 상품 등록하기

6.1.16. 상품 등록 구현하기

- 테스트를 마쳤으니 이제 상품등록 페이지에서 데이터를 입력 후 실제로 등록되는지 확인한다.
- 상품 등록 시 사용하는 이미지가 중요한 것이 아니기 때문에 이미지를 인터넷에서 퍼오든지 하자.
- 애플리케이션을 실행 후 C:₩shop₩item 폴더 안에 이미지가 올라와 있는지 확인해 보자.
- 상품등록 페이지처럼 복잡한 구조는 자바스크립트와 JQuery만으로 처리하는 것보다 Vue.js 프레임워크 활용이 편리하다.







6.2 상품 수정하기

6.2.1. 상품 수정하기

- 등록한 상품 정보를 볼 수 있는 상세 페이지 진입 및 상품 데이터를 수정할 수 있도록 기능을 구현한다.
- 상품 등록 후 콘솔을 보면 insert into item 쿼리문이 실행된 것을 볼 수 있다.

  해당 상품 아이디를 이용해서 상품 수정 페이지에 진입한다. / item_id 값은 그동안의 실행 상황에 따라 다르다.

```
Hibernate:
    insert
    into
        item
        (reg_time, update_time, created_by, modified_by, item_detail, item_nm, item_sell_status, price, stock_number, item_id)
    values
        (?, ?, ?, ?, ?, ?, ?, ?, ?, ?)
□[2m2022-03-28 10:58:59.406□[0;39m □[32mTRACE□[0;39m □[35m16108□[0;39m □[2m---□[0;39m □[2m[nio-8090-exec-4]□[0;39m □[36mo.h.ty
□[2m:□[0;39m binding parameter [1] as [TIMESTAMP] - [2022-03-28T10:58:59.345240900]
□[2m2022-03-28 10:58:59.407□[0;39m □[32mTRACE□[0;39m □[35m16108□[0;39m □[2m---□[0;39m □[2m[nio-8090-exec-4]□[0;39m □[36mo.h.ty
□[2m:□[0;39m binding parameter [2] as [TIMESTAMP] - [2022-03-28T10:58:59.345240900]
□[2m2022-03-28 10:58:59.407□[0;39m □[32mTRACE□[0;39m □[35m16108□[0;39m □[2m---□[0;39m □[2m[nio-8090-exec-4]□[0;39m □[36mo.h.ty
□[2m:□[0;39m binding parameter [3] as [VARCHAR] - [daewonpk@daum.net]
□[2m2022-03-28 10:58:59.408□[0;39m □[32mTRACE□[0;39m □[35m16108□[0;39m □[2m---□[0;39m □[2m[nio-8090-exec-4]□[0;39m □[36mo.h.ty
□[2m:□[0;39m binding parameter [4] as [VARCHAR] - [daewonpk@daum.net]
□[2m2022-03-28 10:58:59.408□[0;39m □[32mTRACE□[0;39m □[35m16108□[0;39m □[2m---□[0;39m □[2m[nio-8090-exec-4]□[0;39m □[36mo.h.ty
□[2m:□[0;39m binding parameter [5] as [CLOB] - [상품 테스트용]
□[2m2022-03-28 10:58:59.414□[0;39m □[32mTRACE□[0;39m □[35m16108□[0;39m □[2m---□[0;39m □[2m[nio-8090-exec-4]□[0;39m □[36mo.h.ty
□[2m:□[0;39m binding parameter [6] as [VARCHAR] - [테스트 상품]
□[2m2022-03-28 10:58:59.414□[0;39m □[32mTRACE□[0;39m □[35m16108□[0;39m □[2m---□[0;39m □[2m[nio-8090-exec-4]□[0;39m □[36mo.h.ty
□[2m:□[0;39m binding parameter [7] as [VARCHAR] - [SELL]
□[2m2022-03-28 10:58:59.415□[0;39m □[32mTRACE□[0;39m □[35m16108□[0;39m □[2m---□[0;39m □[2m[nio-8090-exec-4]□[0;39m □[36mo.h.ty
□[2m:□[0;39m binding parameter [8] as [INTEGER] - [100000]
□[2m2022-03-28 10:58:59.415□[0;39m □[32mTRACE□[0;39m □[35m16108□[0;39m □[2m---□[0;39m □[2m[nio-8090-exec-4]□[0;39m □[36mo.h.ty
□[2m:□[0;39m binding parameter [9] as [INTEGER] - [30]
□[2m2022-03-28 10:58:59.415□[0;39m □[32mTRACE□[0;39m □[35m16108□[0;39m □[2m---□[0;39m □[2m[nio-8090-exec-4]□[0;39m □[36mo.h.ty
□[2m:□[0;39m binding parameter [10] as [BIGINT] - [8]
```

6.2 상품 수정하기

6.2.2. 상품 수정하기

▪ 먼저 등록한 상품을 불러오는 메소드를 ItemService 클래스에 추가한다.

```
import javax.persistence.EntityNotFoundException;
import java.util.ArrayList;
import com.shop.dto.ItemImgDto;

@Service
@Transactional
@RequiredArgsConstructor
public class ItemService {
        :
  @Transactional(readOnly = true) // 상품데이터를 읽어오는 트랜잭션 읽기전용 설정한다. 이럴 경우 JPA가 변경감지(더티체킹)를 수행하지 않아서 성능향상할 수 있다.
   public ItemFormDto getItemDtl(Long itemId){
      List<ItemImg> itemImgList = itemImgRepository.findByItemIdOrderByIdAsc(itemId);  // 해당 상품 이미지 조회
      List<ItemImgDto> itemImgDtoList = new ArrayList<>();
      for (ItemImg itemImg : itemImgList) {                    // 조회한 ItemImg 엔티티를 ItemImgDto 객체로 만들어서 리스트에  추가한다.
         ItemImgDto itemImgDto = ItemImgDto.of(itemImg);
         itemImgDtoList.add(itemImgDto);
      }

      Item item = itemRepository.findById(itemId)  // 상품 아이디를 통해 상품 엔티티를 조회한다. 존재하지 않을 땐 예외를 발생시킨다.
            .orElseThrow(EntityNotFoundException::new);
      ItemFormDto itemFormDto = ItemFormDto.of(item);
      itemFormDto.setItemImgDtoList(itemImgDtoList);
      return itemFormDto;
   }
```

findById(id) / JpaRepository 디폴트 메소드
예: Member 테이블에서 기본키 필드 값이 id인 레코드를 조회
보통 Optional<Member> 타입의 객체로 리턴한다.
이 객체의 get 메서드를 호출하면 Member 객체가 리턴
예: Member m = memberRepository.findById(id).get();

```
@Id
@Column(name="item_id")
```

Snake case

com.shop.service
- FileService.java
- ItemImgService.java
- ItemService.java
- MemberService.java

```
public static ItemImgDto of(ItemImg itemImg) {
    return modelMapper.map(itemImg, ItemImgDto.class);
}
```

ItemImgDto

Camel case

```
public static ItemFormDto of(Item item) {
    return modelMapper.map(item, ItemFormDto.class);
}
```

ItemFormDto

readOnly=true가 성능 향상이 되는 이유
트랜잭션에 readOnly=true 옵션을 주면 하이버네이트의 Session Flush Mode를 FlushMode.MANUAL로 설정한다. MANUAL로 설정이 되면 강제로 Flush를 호출하지 않는 한 flush가 일어나지 않게 된다. 따라서 트랜잭션을 commit하여도 영속성 콘텍스트가 Flush 되지 않아서 엔티티의 등록, 수정, 삭제가 동작하지 않으며 추가적으로 영속성 콘텍스트는 변경 감지를 위한 스냅샷을 보관하지 않으므로 성능이보다 향상이 된다.

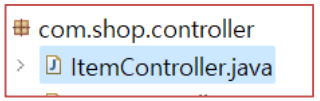
더티 체킹(Dirty Checking):
더티체킹은 Transaction 안에서 엔티티의 변경이 일어나면, 변경 내용을 자동으로 데이터베이스에 반영하는 JPA 특징이다. Dirty Checking 즉 변경 감지를 통해 DB에 반영한다.

데이터베이스에 변경 데이터를 저장하는 시점
.Transaction Commit
.EntityManager Flush
.JPQL 사용

6.2 상품 수정하기

6.2.3. 상품 수정하기

- 상품 수정 페이지로 진입하기 위해서 ItemController 클래스에 코드를 추가한다.



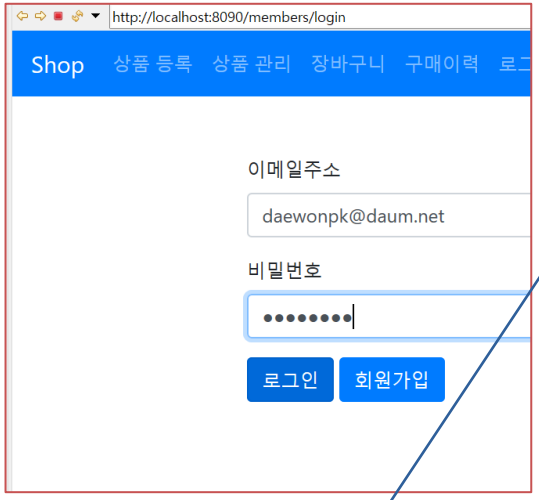

```
import org.springframework.web.bind.annotation.PathVariable;
import javax.persistence.EntityNotFoundException;

@Controller
@RequiredArgsConstructor
public class ItemController {
    :

  @GetMapping(value = "/admin/item/{itemId}")    // url 경로 변수는 { } 표현한다.
   public String itemDtl(@PathVariable("itemId") Long itemId, Model model) {

      try {
         ItemFormDto itemFormDto = itemService.getItemDtl(itemId);  // 조회한 상품 데이터를 모델에 담아 뷰로 전달한다.
         model.addAttribute("itemFormDto", itemFormDto);
      } catch(EntityNotFoundException e) {    // 상품 엔티티가 존재하지 않을 경우 에러 메시지를 담아 상품 등록 페이지로 이동한다.
         model.addAttribute("errorMessage", "존재하지 않는 상품 입니다.");
         model.addAttribute("itemFormDto", new ItemFormDto());
         return "item/itemForm";
      }

      return "item/itemForm";
   }
```

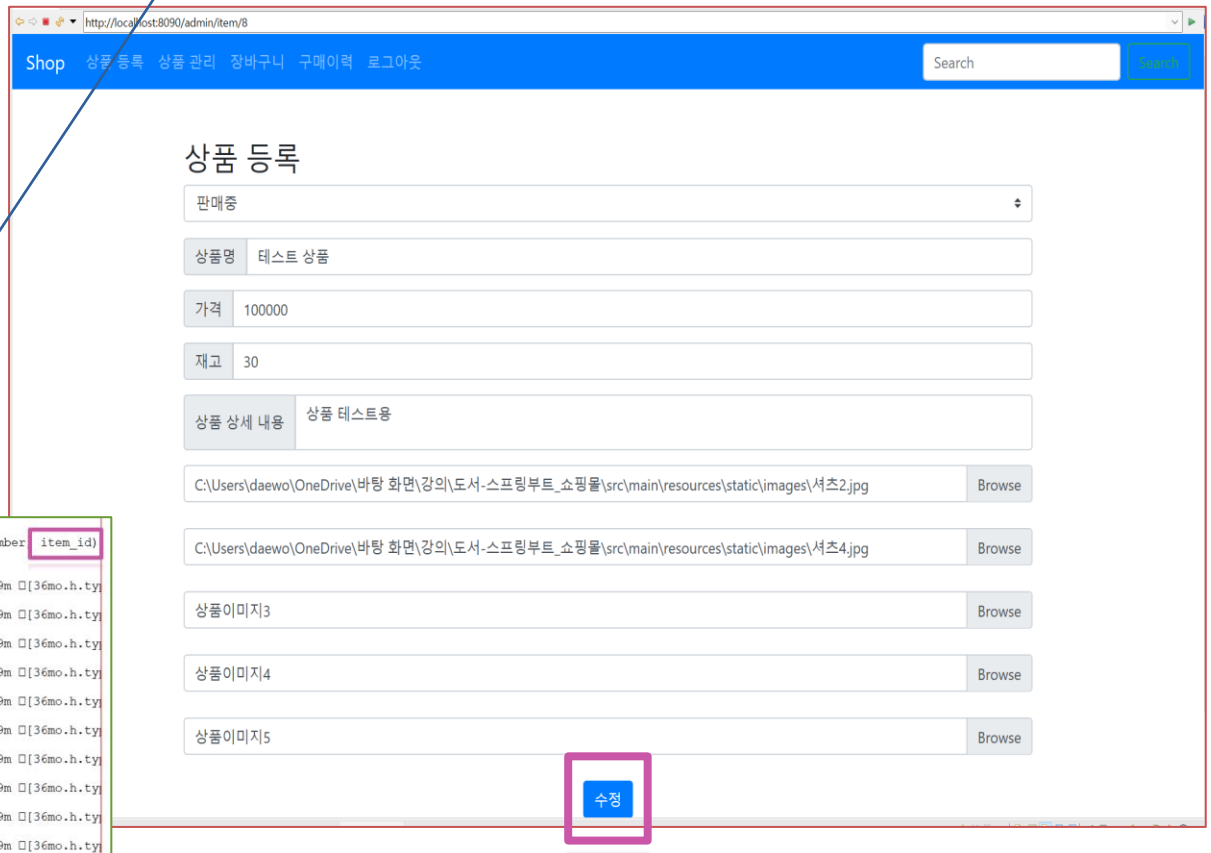
6.2 상품 수정하기

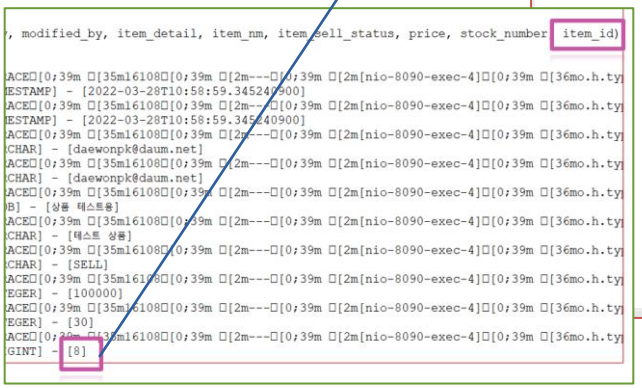
6.2.4. 상품 수정하기

- 로그인 후 실행 http://localhost:8090/admin/item/8 제대로 불러오면 수정 단추도 보인다.

  수정단추를 누르면 아직 이미지 업로드와 Controller에서 @PostMapping을 구현하지 않았기에 오류가 난다.







앞쪽 PT에서 자신의 item_id로 진행한다.

```java
public void updateItem(ItemFormDto itemFormDto){
    this.itemNm = itemFormDto.getItemNm();
    this.price = itemFormDto.getPrice();
    this.stockNumber = itemFormDto.getStockNumber();
    this.itemDetail = itemFormDto.getItemDetail();
    this.itemSellStatus = itemFormDto.getItemSellStatus();
}
```
Item.java

```java
public void updateItemImg(String oriImgName, String imgName, String imgUrl){
    this.oriImgName = oriImgName;
    this.imgName = imgName;
    this.imgUrl = imgUrl;
}
```
ItemImg.java

6.2.5. 상품 수정하기

- 상품 이미지 데이터를 수정을 위해서 전체적인 흐름을 살펴보자.

```java
@PostMapping(value = "/admin/item/{itemId}")
public String itemUpdate(@Valid ItemFormDto itemFormDto, BindingResult bindingResult,
        @RequestParam("itemImgFile") List<MultipartFile> itemImgFileList, Model mod
    if(bindingResult.hasErrors()){
        return "item/itemForm";
    }

    if(itemImgFileList.get(0).isEmpty() && itemFormDto.getId() == null){
        model.addAttribute("errorMessage", "첫번째 상품 이미지는 필수 입력 값 입니다.");
        return "item/itemForm";
    }

    try {
        itemService.updateItem(itemFormDto, itemImgFileList);
    } catch (Exception e){
        model.addAttribute("errorMessage", "상품 수정 중 에러가 발생하였습니다.");
        return "item/itemForm";
    }

    return "redirect:/";
}
```
1 ItemController.java

```java
public Long updateItem(ItemFormDto itemFormDto, List<MultipartFile>
itemImgFileList) throws Exception{
    //상품 수정
    Item item = itemRepository.findById(itemFormDto.getId())
            .orElseThrow(EntityNotFoundException::new);
    item.updateItem(itemFormDto);
    List<Long> itemImgIds = itemFormDto.getItemImgIds();

    //이미지 수정
    for(int i=0;i<itemImgFileList.size();i++){
        itemImgService.updateItemImg(itemImgIds.get(i),
                itemImgFileList.get(i));
    }

    return item.getId();
}
```
2 ItemService.java

```java
public void updateItemImg(Long itemImgId, MultipartFile itemImgFile) throws
Exception{
    if(!itemImgFile.isEmpty()){
        ItemImg savedItemImg = itemImgRepository.findById(itemImgId)
                .orElseThrow(EntityNotFoundException::new);

        //기존 이미지 파일 삭제
        if(!StringUtils.isEmpty(savedItemImg.getImgName())) {
            fileService.deleteFile(itemImgLocation+"/"+
                    savedItemImg.getImgName());
        }

        String oriImgName = itemImgFile.getOriginalFilename();
        String imgName = fileService.uploadFile(itemImgLocation, oriImgName,
itemImgFile.getBytes());
        String imgUrl = "/images/item/" + imgName;
        savedItemImg.updateItemImg(oriImgName, imgName, imgUrl);
    }
}
```
3 ItemImgService.java

```java
public String uploadFile(String uploadPath, String originalFileName, byte[] fileData) throws
Exception{
    UUID uuid = UUID.randomUUID();
    String extension = originalFileName.substring(originalFileName.lastIndexOf("."));
    String savedFileName = uuid.toString() + extension;
    String fileUploadFullUrl = uploadPath + "/" + savedFileName;
    FileOutputStream fos = new FileOutputStream(fileUploadFullUrl);
    fos.write(fileData);
    fos.close();
    return savedFileName;
}

public void deleteFile(String filePath) throws Exception{
    File deleteFile = new File(filePath);
    if(deleteFile.exists()) {
        deleteFile.delete();
        log.info("파일을 삭제하였습니다.");
    } else {
        log.info("파일이 존재하지 않습니다.");
    }
}
```
4 FileService.java

6.2 상품 수정하기

6.2.5. 상품 수정하기

▪ 상품 이미지 데이터를 수정을 위해서 ItemImgService 클래스를 수정한다. 변경감지 기능을 사용한다.

```
import javax.persistence.EntityNotFoundException;

@Service
@RequiredArgsConstructor
@Transactional
public class ItemImgService {
        :
  public void updateItemImg(Long itemImgId, MultipartFile itemImgFile) throws Exception{
     if(!itemImgFile.isEmpty()){     // 상품 이미지를 수정한 경우 상품 이미지를 업데이트한다.
        ItemImg savedItemImg = itemImgRepository.findById(itemImgId)   // 상품 이미지 아이디를 이용하여 기존 저장했던 상품 이미지
              .orElseThrow(EntityNotFoundException::new);              // 엔티티를 조회한다.

        //기존 이미지 파일 삭제
        if(!StringUtils.isEmpty(savedItemImg.getImgName())) {  // 기존에 등록된 상품 이미지 파일이 있을 경우 해당 파일을 삭제한다.
           fileService.deleteFile(itemImgLocation+"/"+
                 savedItemImg.getImgName());
        }

        String oriImgName = itemImgFile.getOriginalFilename();                                          // 업데이트한
        String imgName = fileService.uploadFile(itemImgLocation, oriImgName, itemImgFile.getBytes()); // 상품이미지 파일 업로드
        String imgUrl = "/images/item/" + imgName;
        savedItemImg.updateItemImg(oriImgName, imgName, imgUrl);
     }
  }
```

com.shop.service
- FileService.java
- ItemImgService.java

```
itemImgLocation=C:/shop/item
uploadPath=file:///C:/shop/
```

```
public String uploadFile(String uploadPath, String originalFileName, byte[] fileData)
    :
public void deleteFile(String filePath)
```
FileService

```
public void updateItemImg(String oriImgName, String imgName, String imgUrl){
    this.oriImgName = oriImgName;
    this.imgName = imgName;
    this.imgUrl = imgUrl;
}
```
ItemImg

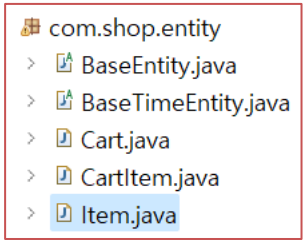
변경된 상품 이미지 정보를 세팅한다. 여기서 중요한 점은 상품 등록 때처럼 itemImgRepository.save()로직을 호출하지 않는다는 것이다. savedItemImg엔티티는 현재 영속상태이므로 데이터를 변경하는 것만으로 변경감지 기능이 동작하여 트랜잭션이 끝날 때 update 쿼리가 실행된다. 주의해야 할 점은 엔티티가 영속 상태이어야 한다.

6.2 상품 수정하기

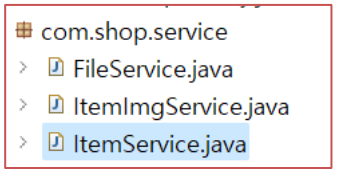
6.2.6. 상품 수정하기

- 상품을 업데이트하는 로직을 구현한다. 먼저 Item 와 itemService 클래스에 상품 데이터를 업데이트하는 로직을 추가한다.
- 상품을 업데이트할 때도 마찬가지로 변경기능 감지를 사용한다.

```
import com.shop.dto.ItemFormDto;

 public void updateItem(ItemFormDto itemFormDto){
     this.itemNm = itemFormDto.getItemNm();
     this.price = itemFormDto.getPrice();
     this.stockNumber = itemFormDto.getStockNumber();
     this.itemDetail = itemFormDto.getItemDetail();
     this.itemSellStatus = itemFormDto.getItemSellStatus();
  }
```




```
@Service
@Transactional
@RequiredArgsConstructor
public class ItemService {
            :
  public Long updateItem(ItemFormDto itemFormDto, List<MultipartFile> itemImgFileList) throws Exception{
     //상품 수정
     Item item = itemRepository.findById(itemFormDto.getId())  // 상품등록화면으로 전달 받은 상품 아이디를 이용 상품엔티티 조회
           .orElseThrow(EntityNotFoundException::new);
     item.updateItem(itemFormDto);  // 상품등록화면으로 전달 받은 ItemFormDto를 통해 상품 엔티티 업데이트

     //이미지 수정
     List<Long> itemImgIds = itemFormDto.getItemImgIds();  // 상품 이미지 아이디 리스트 반환
     for(int i=0;i<itemImgFileList.size();i++){
        itemImgService.updateItemImg(itemImgIds.get(i), // 상품 이미지 아이디를 업데이트하기 위해서 상품이미지 아이디, 상품이미지 파일 정보 전달
              itemImgFileList.get(i));
     }
     return item.getId();
  }
```

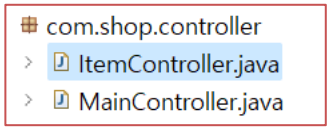


6.2 상품 수정하기

6.2.7. 상품 수정하기

- 상품을 수정하는 URL을 ItemController 클래스에 추가한다.

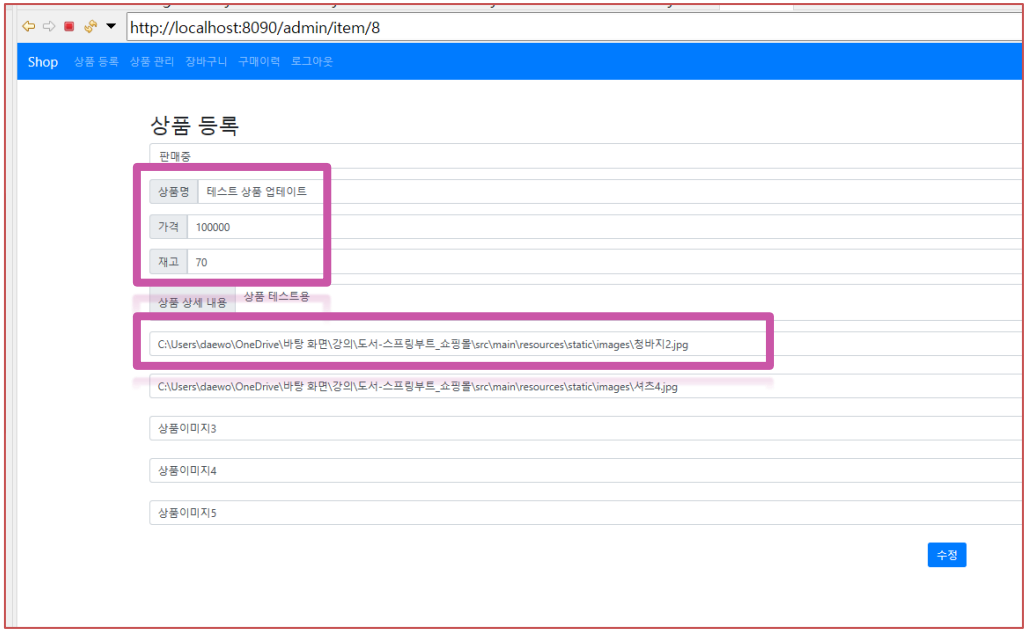
```
@PostMapping(value = "/admin/item/{itemId}")
public String itemUpdate(@Valid ItemFormDto itemFormDto, BindingResult bindingResult,
                         @RequestParam("itemImgFile") List<MultipartFile> itemImgFileList, Model model) {
    if(bindingResult.hasErrors()) {
        return "item/itemForm";
    }

    if(itemImgFileList.get(0).isEmpty() && itemFormDto.getId() == null) {
        model.addAttribute("errorMessage", "첫번째 상품 이미지는 필수 입력 값 입니다.");
        return "item/itemForm";
    }

    try {
        itemService.updateItem(itemFormDto, itemImgFileList);
    } catch (Exception e) {
        model.addAttribute("errorMessage", "상품 수정 중 에러가 발생하였습니다.");
        return "item/itemForm";
    }

    return "redirect:/";
}
```

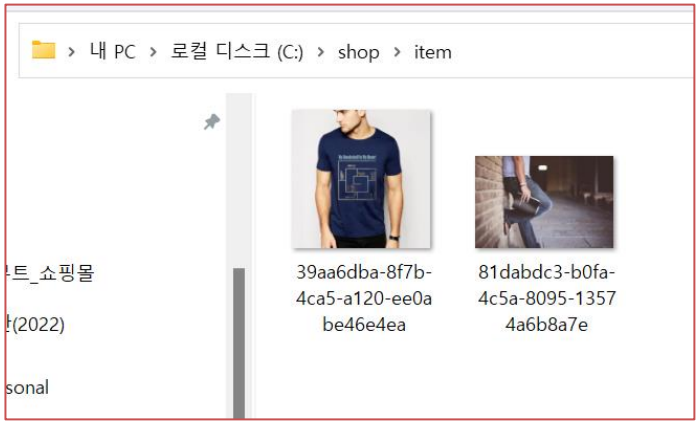


6.2 상품 수정하기

6.2.8 상품 수정하기

- 수정 후 다시 불러오면 제대로 수정된 것을 확인할 수 있다. 만약 이미지 파일도 수정했으면 업로드 폴더를 체크해보자. 기존 이미지가 삭제되고 수정 이미지가 있는 것을 확인할 수 있다.

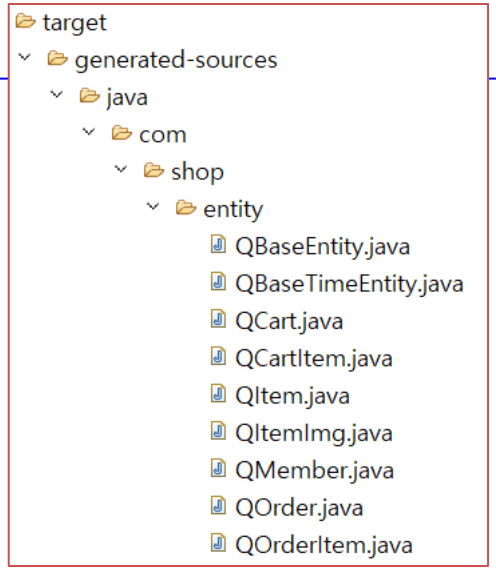




6.3 상품 관리하기

6.3.1. 상품 관리하기

- 상품 상세 페이지에 진입하기 위해서 등록된 상품번호를 직접 URL에 입력하여 상품 상세 페이지에 진입했다.
  그런데 상품번호를 모를 경우 상세 페이지로 진입할 수 없으므로 등록된 상품 리스트를 조회할 수 있는 화면을 만든다.
- 상품관리 화면에서는 상품을 조회하는 조건을 설정 후 페이징 기능을 통해 일정 개수의 상품만 불러오며, 선택한 상품 상세 페이지로 이동할 수 있는 기능까지 구현한다.
- 조회 조건
  상품 등록일 / 상품 판매 상태 / 상품명 또는 상품 등록자 아이디
- 조회 조건이 복잡한 화면은 Querydsl을 이용해 조건에 맞는 쿼리를 동적으로 쉽게 생성할 수 있다. Querydsl를 사용하면 유사한 쿼리를 재활용할 수 있다.
- Querydsl을 사용하기 위해선 QDomain을 생성해야 한다. 이미 2장에서 다뤘다.



6.3 상품 관리하기

ItemSearchDto.java

```
public class ItemSearchDto {
    private String searchDateType;
    private ItemSellStatus searchSellStatus;
    private String searchBy;
    private String searchQuery = "";
}
```

ItemController.java

```
@GetMapping(value = {"/admin/items", "/admin/items/{page}"})
    public String itemManage(ItemSearchDto itemSearchDto, @PathVariable("page")
Optional<Integer> page, Model model){

        Pageable pageable = PageRequest.of(page.isPresent() ? page.get() : 0, 3);
        Page<Item> items = itemService.getAdminItemPage(itemSearchDto, pageable);

        model.addAttribute("items", items);
        model.addAttribute("itemSearchDto", itemSearchDto);
        model.addAttribute("maxPage", 5);

        return "item/itemMng";
    }
```

ItemService.java

```
@Transactional(readOnly = true)
    public Page<Item> getAdminItemPage(ItemSearchDto itemSearchDto, Pageable pageable){
        return itemRepository.getAdminItemPage(itemSearchDto, pageable);
    }
```

ItemRepository.java

```
public interface ItemRepository extends JpaRepository<Item, Long>,
QuerydslPredicateExecutor<Item>, ItemRepositoryCustom {
    List<Item> findByItemNm(String itemNm);
}
```

ItemRepositoryCustom.java

```
public interface ItemRepositoryCustom {

    Page<Item> getAdminItemPage(ItemSearchDto itemSearchDto, Pageable pageable);
}
```

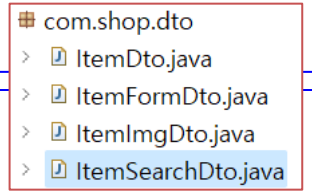
ItemRepositoryCustomImpl.java

```
public public class ItemRepositoryCustomImpl implements ItemRepositoryCustom{
        :
    private BooleanExpression searchSellStatusEq(ItemSellStatus searchSellStatus){
        return searchSellStatus == null ? null :
QItem.item.itemSellStatus.eq(searchSellStatus);
    }

    private BooleanExpression regDtsAfter(String searchDateType){

                    :
        return QItem.item.regTime.after(dateTime);
    }

 private BooleanExpression searchByLike(String searchBy, String searchQuery){

                    :
    }
@Override
    public Page<Item> getAdminItemPage(ItemSearchDto itemSearchDto, Pageable
pageable) {

        QueryResults<Item> results = queryFactory
                .selectFrom(QItem.item)
                .where(regDtsAfter(itemSearchDto.getSearchDateType()),
                        searchSellStatusEq(itemSearchDto.getSearchSellStatus()),
                        searchByLike(itemSearchDto.getSearchBy(),
                                itemSearchDto.getSearchQuery()))
                .orderBy(QItem.item.id.desc())
                .offset(pageable.getOffset())
                .limit(pageable.getPageSize())
                .fetchResults();

        List<Item> content = results.getResults();
        long total = results.getTotal();

        return new PageImpl<>(content, pageable, total);
}
```

6.3 상품 관리하기

6.3.2. 상품 관리하기

▪ 상품 데이터 조회 시 상품 조건을 가지고 있는 ItemSearchDto 클래스를 만든다.




```
package com.shop.dto;

import com.shop.constant.ItemSellStatus;
import lombok.Getter;
import lombok.Setter;

@Getter @Setter
public class ItemSearchDto {

    private String searchDateType; // 현재 시간과 상품 등록일을 비교해서 상품 데이터를 조회한다.

    private ItemSellStatus searchSellStatus; // 상품 판매상태를 기준으로 상품 데이터를 조회한다.

    private String searchBy;  // 상품을 조회할 때 어떤 유형으로 조회할지 선택한다. (itemNm: 상품명, createdBy: 상품등록자 아이디)

    private String searchQuery = "";  // 조회할 검색어 저장할 변수이다.

}
```

조회시간 기준:

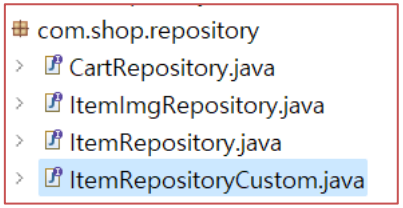
all: 상품등록일 전체
1d: 최근 하루 동안 등록된 상품
1w: 최근 일주일 동안 등록된 상품
1m: 최근 한 달 동안 등록된 상품
6m: 최근 6개월 동안 등록된 상품

6.3 상품 관리하기

6.3.3. 상품 관리하기

- Querydsl을 스프링JPA와 사용하기 위해선 사용자 정의 리포지토리를 만들어야 한다.
  1. 사용자 정의 인터페이스 작성 ItemRepositoryCustom
  2. 사용자 정의 인터페이스 구현 ItemRepositoryCustomImpl
  3. Spring Data Jpa Repository에서 사용자 정의 인터페이스 상속 ItemRepository extends …, …, ItemRepositoryCustom

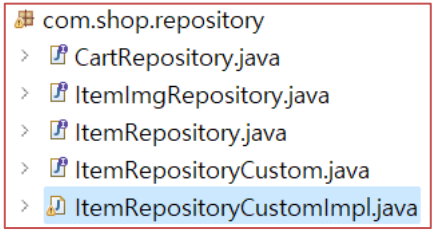
```
package com.shop.repository;

import com.shop.dto.ItemSearchDto;
import com.shop.entity.Item;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;

public interface ItemRepositoryCustom {

    Page<Item> getAdminItemPage(ItemSearchDto itemSearchDto, Pageable pageable);
    // 상품조회 조건을 담고 있는 itemSearchDto 객체와 페이징 정보를 담고 있는 pageable객체를 파라미터로 받는 getAdminItemPage
    // 메소드를 정의한다.
}
```



6.3 상품 관리하기

6.3.4. 상품 관리하기

- 사용자 정의 인터페이스 구현할 때 유의할 것은 클래스명 끝에 'Impl' 붙여줘야 정상적으로 작동한다.
- Querydsl에선 BooleanExpression이라는 where절에서 사용할 수 있는 값을 지원한다.
- BooleanExpression을 반환하는 메소드를 만들고 해당 조건들을 다른 쿼리를 생성할 때 사용할 수 있기에 중복코드를 줄 일 수 있다는 장점이 있다.




```
package com.shop.repository;

import com.querydsl.core.QueryResults;
import com.querydsl.core.types.dsl.BooleanExpression;
import com.querydsl.jpa.impl.JPAQueryFactory;
import com.shop.constant.ItemSellStatus;
import com.shop.dto.ItemSearchDto;
import com.shop.entity.Item;
import com.shop.entity.QItem;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.PageImpl;
import org.springframework.data.domain.Pageable;
import org.thymeleaf.util.StringUtils;

import javax.persistence.EntityManager;
import java.time.LocalDateTime;
import java.util.List;

public class ItemRepositoryCustomImpl implements ItemRepositoryCustom{  // 인터페이스를 상속하여 구현한다.

  private JPAQueryFactory queryFactory;  // 동적으로 쿼리를 생성하기 위해서 JPAQueryFactory 클래스를 사용한다.

  public ItemRepositoryCustomImpl(EntityManager em){
    this.queryFactory = new JPAQueryFactory(em);  //  JPAQueryFactory 생성자로 EntityManager 객체를 넣어준다.
  }
```

6.3 상품 관리하기

6.3.4. 상품 관리하기

- ItemRepositoryCustomImpl.java

```
private BooleanExpression searchSellStatusEq(ItemSellStatus searchSellStatus){
    return searchSellStatus == null ? null : QItem.item.itemSellStatus.eq(searchSellStatus);
} // 상품 판매 상태조건이 전체(null)일 경우는 null을 리턴, 결과값이 null이면 where절에서 해당조건은 무시. 상품판매 상태조건이 null이 아니라
  // 판매중 혹은 품절 상태라면 해당조건 상품만 조회한다.

private BooleanExpression regDtsAfter(String searchDateType){  // searchDateType의 값에 따라서 dateTime의 값을 이전 시간의 값으로
                    // 세팅 후 해당 시간 이후로 등록된 상품만 조회한다. 예를 들어, searchDateType의 값이 1m인 경우 dateTime의 시간을 한 달 전으로
                    // 세팅 후 최근 한 달 동안 등록된 상품만 조회하도록 조건값을 반환한다.
    LocalDateTime dateTime = LocalDateTime.now();

    if(StringUtils.equals("all", searchDateType) || searchDateType == null){
        return null;
    } else if(StringUtils.equals("1d", searchDateType)){
        dateTime = dateTime.minusDays(1);
    } else if(StringUtils.equals("1w", searchDateType)){
        dateTime = dateTime.minusWeeks(1);
    } else if(StringUtils.equals("1m", searchDateType)){
        dateTime = dateTime.minusMonths(1);
    } else if(StringUtils.equals("6m", searchDateType)){
        dateTime = dateTime.minusMonths(6);
    }
    return QItem.item.regTime.after(dateTime);
}

private BooleanExpression searchByLike(String searchBy, String searchQuery){ // searchBy의 값에 따라서 상품명에 검색어를 포함하고
                                        //  있는 상품 또는 상품 생성자의 아이디에 검색어를 포함하고 있는 상품을 조회하도록 조건값을 반환한다.
    if(StringUtils.equals("itemNm", searchBy)){
        return QItem.item.itemNm.like("%" + searchQuery + "%");
    } else if(StringUtils.equals("createdBy", searchBy)){
        return QItem.item.createdBy.like("%" + searchQuery + "%");
    }
    return null;
}
```

ItemFormDto.java
ItemImgDto.java
ItemSearchDto.java

```
package com.shop.dto;

import com.shop.constant.ItemSellStatus;
import lombok.Getter;
import lombok.Setter;

@Getter @Setter
public class ItemSearchDto {

    private String searchDateType; // 현재 시간과 상품 등록일을 비교해서 상품 데이터를 조회한다.

    private ItemSellStatus searchSellStatus; // 상품 판매상태를 기준으로 상품 데이터를 조회한다.

    private String searchBy;  // 상품을 조회할 때 어떤 유형으로 조회할지 선택한다. (itemNm: 상품명, createdBy: 상품등록자 아이디)

    private String searchQuery = "";  // 조회할 검색어 저장할 변수이다.

}
```

조회시간 기준:

all: 상품등록일 전체
1d: 최근 하루 동안 등록된 상품
1w: 최근 일주일 동안 등록된 상품
1m: 최근 한 달 동안 등록된 상품
6m: 최근 6개월 동안 등록된 상품

6.3 상품 관리하기

6.3.4. 상품 관리하기

- ItemRepositoryCustomImpl.java

```
@Override
public Page<Item> getAdminItemPage(ItemSearchDto itemSearchDto, Pageable pageable) {

    QueryResults<Item> results = queryFactory // queryFactory를 이용해서 쿼리를 생성한다.
            .selectFrom(QItem.item)           // 상품 데이터를 조회하기 위해서 QItem의 item을 지정한다.
            .where(regDtsAfter(itemSearchDto.getSearchDateType()),  // BooleanExpression 반환하는 조건문들을 넣어준다.
                    searchSellStatusEq(itemSearchDto.getSearchSellStatus()),  // 컴마(,)단위로 넣어줄 경우 and조건으로 인식한다.
                    searchByLike(itemSearchDto.getSearchBy(),
                            itemSearchDto.getSearchQuery()))
            .orderBy(QItem.item.id.desc())
            .offset(pageable.getOffset())  // 데이터를 가지고 올 시작 인덱스를 지정한다.
            .limit(pageable.getPageSize()) // 한번에 가지고 올 최대 개수를 지정한다.
            .fetchResults(); // 조회한 리스트 및 전체 개수를 포함하는 QueryResult를 반환한다. 상품 데이터 리스트 조회 및 상품 데이터 전체 개수를
                             // 조회하는 2번의 쿼리문이 실행된다.

    List<Item> content = results.getResults();
    long total = results.getTotal();

    return new PageImpl<>(content, pageable, total); // 조회한 데이터를 Page 클래스의 구현체인 PageImpl 객체로 반환한다.
}

}
```

ItemController.java

```
Pageable pageable = PageRequest.of(page.isPresent() ? page.get() : 0, 3);
```

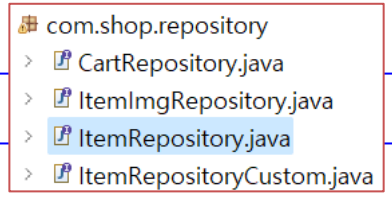
메소드	기능
QueryResults<T> fetchResults()	조회 대상 리스트 및 전체 개수를 포함하는 QueryResults 반환
List<T> fetch()	조회 대상 리스트 반환
T fetchOne()	조회 대상이 1건이면 해당 타입 반환 조회 대상이 1건 이상이면 에러 발생
T fetchFirst()	조회 대상이 1건 또는 1건 이상이면 1건만 반환
long fetchCount()	해당 데이터 전체 개수 반환, cont 쿼리 실행

6.3 상품 관리하기

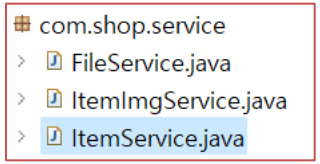
6.3.5. 상품 관리하기

- ItemRepository 에서 ItemRepositoryCustom 상속한다.
- 이제 ItemRepository에서 Querydsl로 구현한 상품 관리 페이지 목록을 불러오는 getAdminItemPage() 메소드를 사용할 수 있다.
- ItemService클래스에 상품조회조건과 페이지 정보를 파라미터로 받아서 상품 데이터를 조회하는 getAdminItemPage() 메소드를 추가한다. 데이터의 수정이 일어나지 않으므로 최적화를 위해 @Transational(readOnly=true) 설정한다.

```
public interface ItemRepository extends JpaRepository<Item, Long>, QuerydslPredicateExecutor<Item>, ItemRepositoryCustom {
    List<Item> findByItemNm(String itemNm);
}
```




```
import com.shop.dto.ItemSearchDto;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;

@Service
@Transactional
@RequiredArgsConstructor
public class ItemService {
    :

    @Transactional(readOnly = true)
    public Page<Item> getAdminItemPage(ItemSearchDto itemSearchDto, Pageable pageable) {
        return itemRepository.getAdminItemPage(itemSearchDto, pageable);
    }
```

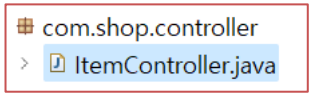


6.3 상품 관리하기

6.3.6. 상품 관리하기

- ItemController 클래스에 상품 관리 화면 이동 및 조회한 상품 데이터를 화면에 전달하는 로직을 구현한다.

  현재 상품 데이터가 많이 없기에 한 페이지당 총3개 상품만 보여주도록 한다. 페이지 번호는 0부터 시작하는 것에 유의.




```
import com.shop.entity.Item;
import com.shop.dto.ItemSearchDto;
import java.util.Optional;

import org.springframework.data.domain.Page;
import org.springframework.data.domain.PageRequest;
import org.springframework.data.domain.Pageable;



    @GetMapping(value = {"/admin/items", "/admin/items/{page}"})  // 페이지 번호가 없는 경우와 있는 경우 2가지 매핑
    public String itemManage(ItemSearchDto itemSearchDto, @PathVariable("page") Optional<Integer> page, Model model) {

        Pageable pageable = PageRequest.of(page.isPresent() ? page.get() : 0, 3);  // PageRequest.of 메서드를 통해 Pageable 객체
                                                                                   // 생성한다. 해당 페이지 조회, 페이지 번호가 없으면 0페이지에서 3개 조회한다.
        Page<Item> items = itemService.getAdminItemPage(itemSearchDto, pageable);
                                // 조회조건과 페이징 정보를 파라미터로 넘겨 Page<Item> 객체를 반환받는다.
        model.addAttribute("items", items); // 조회한 상품 데이터와 페이징 정보를 뷰에 전달한다.
        model.addAttribute("itemSearchDto", itemSearchDto); // 페이지 전환 시 기존 검색조건을 유지한 채 이동할 수 있도록 뷰에 다시 전달
        model.addAttribute("maxPage", 5);  // 상품 관리 하단에 보여줄 페이지 번호의 최대 개수이다.

        return "item/itemMng";
    }
```

6.3 상품 관리하기

6.3.7. 상품 관리하기

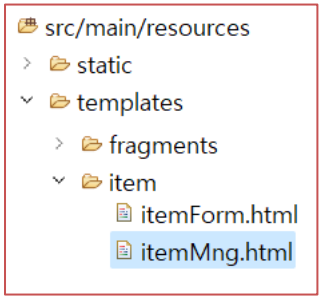
- 페이징을 체크하기 위해서 6개 이상 상품등록을 한 후 실행한다.
- 아래 결과를 보기위해서는 상품관리 itemMng.html을 생성해야 한다.

6.3 상품 관리하기

6.3.8. 상품 관리하기

- itemMng.html을 생성하기

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org"
      xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout"
      layout:decorate="~{layouts/layout1}">

<!-- 사용자 스크립트 추가 -->
<th:block layout:fragment="script">
    <script th:inline="javascript">

        $(document).ready(function(){
            $("#searchBtn").on("click",function(e) {
                e.preventDefault();
                page(0);
            });
        });

        function page(page){
            var searchDateType = $("#searchDateType").val();
            var searchSellStatus = $("#searchSellStatus").val();
            var searchBy = $("#searchBy").val();
            var searchQuery = $("#searchQuery").val();

            location.href="/admin/items/" + page + "?searchDateType=" + searchDateType
            + "&searchSellStatus=" + searchSellStatus
            + "&searchBy=" + searchBy
            + "&searchQuery=" + searchQuery;
        }

    </script>
</th:block>
```



6.3 상품 관리하기

6.3.8. 상품 관리하기

- itemMng.html을 생성하기

```
<!-- 사용자 CSS 추가 -->
<th:block layout:fragment="css">
  <style>
    select{
      margin-right:10px;
    }
  </style>
</th:block>

<div layout:fragment="content">

  <form th:action="@{'/admin/items/' + ${items.number}}" role="form" method="get" th:object="${items}">
    <table class="table">
      <thead>
      <tr>
        <td>상품아이디</td>
        <td>상품명</td>
        <td>상태</td>
        <td>등록자</td>
        <td>등록일</td>
      </tr>
      </thead>
      <tbody>
      <tr th:each="item, status: ${items.getContent()}">
        <td th:text="${item.id}"></td>
        <td>
          <a th:href="'/admin/item/'+${item.id}" th:text="${item.itemNm}"></a>
        </td>
        <td th:text="${item.itemSellStatus == T(com.shop.constant.ItemSellStatus).SELL} ? '판매중' : '품절'"></td>
        <td th:text="${item.createdBy}"></td>
        <td th:text="${item.regTime}"></td>
      </tr>
      </tbody>
    </table>
```

6.3 상품 관리하기

6.3.8. 상품 관리하기

- itemMng.html을 생성하기

```
    <div th:with="start=${(items.number/maxPage)*maxPage + 1}, end=(${(items.totalPages == 0) ? 1 : (start + (maxPage - 1) < items.totalPages ? start + (maxPage - 1) : items.totalPages)})" >
        <ul class="pagination justify-content-center">

            <li class="page-item" th:classappend="${items.first}?'disabled'">
                <a th:onclick="'javascript:page(' + ${items.number - 1} + ')'" aria-label='Previous' class="page-link">
                    <span aria-hidden='true'>Previous</span>
                </a>
            </li>

            <li class="page-item" th:each="page: ${#numbers.sequence(start, end)}" th:classappend="${items.number eq page-1}?'active':''">
                <a th:onclick="'javascript:page(' + ${page - 1} + ')'" th:inline="text" class="page-link">[[${page}]]</a>
            </li>

            <li class="page-item" th:classappend="${items.last}?'disabled'">
                <a th:onclick="'javascript:page(' + ${items.number + 1} + ')'" aria-label='Next' class="page-link">
                    <span aria-hidden='true'>Next</span>
                </a>
            </li>

        </ul>
    </div>
```

6.3 상품 관리하기

6.3.8. 상품 관리하기

▪ itemMng.html을 생성하기

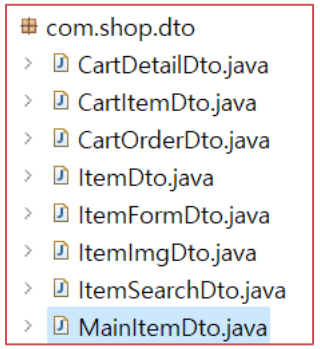
```
    <div class="form-inline justify-content-center" th:object="${itemSearchDto}">
        <select th:field="*{searchDateType}" class="form-control" style="width:auto;">
            <option value="all">전체기간</option>
            <option value="1d">1일</option>
            <option value="1w">1주</option>
            <option value="1m">1개월</option>
            <option value="6m">6개월</option>
        </select>
        <select th:field="*{searchSellStatus}" class="form-control" style="width:auto;">
            <option value="">판매상태(전체)</option>
            <option value="SELL">판매</option>
            <option value="SOLD_OUT">품절</option>
        </select>
        <select th:field="*{searchBy}" class="form-control" style="width:auto;">
            <option value="itemNm">상품명</option>
            <option value="createdBy">등록자</option>
        </select>
        <input th:field="*{searchQuery}" type="text" class="form-control" placeholder="검색어를 입력해주세요">
        <button id="searchBtn" type="submit" class="btn btn-primary">검색</button>
    </div>
</form>

</div>

</html>
```

6.4 메인 화면

6.4.1. 메인 페이지 구현하기

- 등록한 상품을 메인 페이지에서 고객이 볼 수 있도록 구현한다.
- 메인 페이지도 상품관리 메뉴와 비슷하며, 동일하게 Querydsl을 사용하여 페이징 처리 및 네비게이션바에 있는 Search버튼을 이용하여 상품명으로 검색이 가능하도록 구현한다.
- 앞서 Querydsl을 이용하여 상품 조회 시 결과값을 받을 때 Item객체로 반환값을 받았다면 이번엔 @QueryProjection을 이용해 상품 조회 시 DTO객체로 결과값을 받는 방법을 알아본다.
- @QueryProjection이용하면 Item객체로 받은 후 DTO클래스로 변환하는 과정 없이 바로 DTO객체를 뽑아낼 수 있다.

```
package com.shop.dto;

import com.querydsl.core.annotations.QueryProjection;
import lombok.Getter;
import lombok.Setter;

@Getter @Setter
public class MainItemDto {

    private Long id;
    private String itemNm;
    private String itemDetail;
    private String imgUrl;
    private Integer price;

    @QueryProjection  // 생성자에 @QueryProjection 선언하여 Querydsl로 결과 조회 시 MainItemDto 객체로 바로 받아오도록 활용한다.
    public MainItemDto(Long id, String itemNm, String itemDetail, String imgUrl, Integer price){
        this.id = id;
        this.itemNm = itemNm;
        this.itemDetail = itemDetail;
        this.imgUrl = imgUrl;
        this.price = price;
    }
}
```

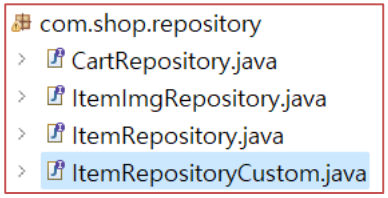


MainItemDto 클래스 생성후 프로젝트 shop 에 마우스 올린 후 [F5] 클릭

6.4 메인 화면

6.4.2. 메인 페이지 구현하기

- ItemRepositoryCustom 인터페이스에 메인 페이지에 보여줄 상품 리스트를 가져오는 메소드를 생성한다.

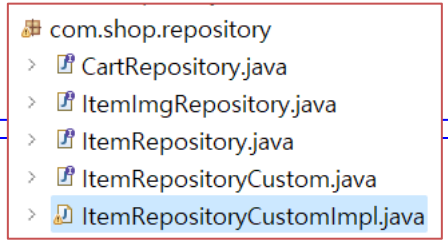



```
package com.shop.repository;

import com.shop.dto.ItemSearchDto;
import com.shop.entity.Item;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import com.shop.dto.MainItemDto;

public interface ItemRepositoryCustom {

    Page<Item> getAdminItemPage(ItemSearchDto itemSearchDto, Pageable pageable);

    Page<MainItemDto> getMainItemPage(ItemSearchDto itemSearchDto, Pageable pageable);

}
```

6.4 메인 화면

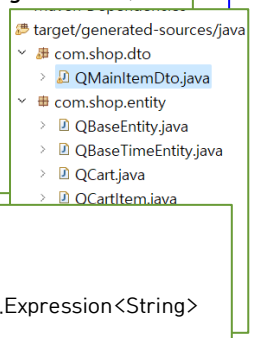
6.4.3. 메인 페이지 구현하기

- ItemRepositoryCustomImpl 클래스에 메인 페이지에 getMainItemPage() 메소드를 구현한다.




```
import com.shop.dto.MainItemDto;
import com.shop.dto.QMainItemDto;
import com.shop.entity.QItemImg;

  private BooleanExpression itemNmLike(String searchQuery){ // 검색어가 null이 아니면 상품명에 해당 검색어가 포함되는 상품을 조회하는 조건을 반환한다.
    return StringUtils.isEmpty(searchQuery) ? null : QItem.item.itemNm.like("%" + searchQuery + "%");
  }

  @Override
  public Page<MainItemDto> getMainItemPage(ItemSearchDto itemSearchDto, Pageable pageable) {
    QItem item = QItem.item;
    QItemImg itemImg = QItemImg.itemImg;

    QueryResults<MainItemDto> results = queryFactory
        .select(
            new QMainItemDto(  // QMainItemDto의 생성자에 반환할 값들을 넣어준다. @QueryProjection 을 사용하면 DTO로 바로 조회가 가능하다.
                item.id,              // 엔티티 조회 후 DTO로 변환하는 과정을 줄일 수 있다.
                item.itemNm,
                item.itemDetail,
                itemImg.imgUrl,
                item.price)
        )
        .from(itemImg)
        .join(itemImg.item, item)  // itemImg와 item을 내부 조인한다.
        .where(itemImg.repimgYn.eq("Y"))   // 상품 이미지의 경우 대표 상품 이미지만 불러온다.
        .where(itemNmLike(itemSearchDto.getSearchQuery()))
        .orderBy(item.id.desc())
        .offset(pageable.getOffset())
        .limit(pageable.getPageSize())
        .fetchResults();

    List<MainItemDto> content = results.getResults();
    long total = results.getTotal();
    return new PageImpl<>(content, pageable, total);
  }
```

```
@QueryProjection
public MainItemDto(Long id, String itemNm, String itemDetail,
String imgUrl, Integer price){
    this.id = id;
    this.itemNm = itemNm;
    this.itemDetail = itemDetail;
    this.imgUrl = imgUrl;
    this.price = price;
}
```



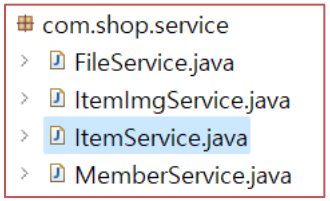

```
public class QMainItemDto extends ConstructorExpression<MainItemDto> {
                :
    public QMainItemDto(com.querydsl.core.types.Expression<Long> id,
com.querydsl.core.types.Expression<String> itemNm, com.querydsl.core.types.Expression<String>
itemDetail, com.querydsl.core.types.Expression<String> imgUrl,
com.querydsl.core.types.Expression<Integer> price) {
        super(MainItemDto.class, new Class<?>[]{long.class, String.class, String.class, String.class,
int.class}, id, itemNm, itemDetail, imgUrl, price);
    }

}
```

6.4 메인 화면

6.4.4. 메인 페이지 구현하기

- ItemService 클래스에 메인 페이지에 보여줄 상품 데이터를 조회하는 메소드를 추가한다.

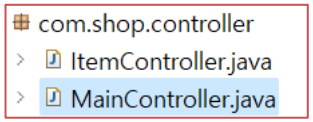
```
import com.shop.dto.MainItemDto;

@Service
@Transactional
@RequiredArgsConstructor
public class ItemService {
        :

   @Transactional(readOnly = true)
    public Page<MainItemDto> getMainItemPage(ItemSearchDto itemSearchDto, Pageable pageable) {
       return itemRepository.getMainItemPage(itemSearchDto, pageable);
    }
```



6.4 메인 화면

6.4.5. 메인 페이지 구현하기

- 메인 페이지에 상품 데이터를 보여주기 위해서 MainController 클래스를 수정한다.

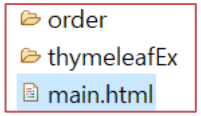



```
package com.shop.controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;

import com.shop.dto.ItemSearchDto;
import com.shop.dto.MainItemDto;
import com.shop.service.ItemService;
import lombok.RequiredArgsConstructor;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.PageRequest;
import org.springframework.data.domain.Pageable;
import org.springframework.ui.Model;
import java.util.Optional;

@Controller
@RequiredArgsConstructor
public class MainController {

    private final ItemService itemService;

    @GetMapping(value = "/")
    public String main(ItemSearchDto itemSearchDto, Optional<Integer> page, Model model){

        Pageable pageable = PageRequest.of(page.isPresent() ? page.get() : 0, 6);
        Page<MainItemDto> items = itemService.getMainItemPage(itemSearchDto, pageable);

        model.addAttribute("items", items);
        model.addAttribute("itemSearchDto", itemSearchDto);
        model.addAttribute("maxPage", 5);

        return "main";
    }

}
```

6.4 메인 화면

6.4.6. 메인 페이지 구현하기

- 메인 페이지에 상품 데이터를 보여주기 위해서 main.html을 수정한다.




```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org"
    xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout"
    layout:decorate="~{layouts/layout1}">

<!-- 사용자 CSS 추가 -->
<th:block layout:fragment="css">
  <style>
    .carousel-inner > .item {
      height: 350px;
    }
    .margin{
      margin-bottom:30px;
    }
    .banner{
      height: 300px;
      position: absolute; top:0; left: 0;
      width: 100%;
      height: 100%;
    }
    .card-text{
      text-overflow: ellipsis;
      white-space: nowrap;
      overflow: hidden;
    }
    a:hover{
      text-decoration:none;
    }
    .center{
      text-align:center;
    }
  </style>
</th:block>
```

6.4 메인 화면

6.4.6. 메인 페이지 구현하기

▪ 메인 페이지에 상품 데이터를 보여주기 위해서 main.html을 수정한다.

```
<div layout:fragment="content">

    <div id="carouselControls" class="carousel slide margin" data-ride="carousel">
        <div class="carousel-inner">
            <div class="carousel-item active item">
                <img class="d-block w-100 banner" src="https://user-images.githubusercontent.com/13268420/112147492-1ab76200-8c20-11eb-8649-
3d2f282c3c02.png" alt="First slide">
            </div>
        </div>
    </div>

    <input type="hidden" name="searchQuery" th:value="${itemSearchDto.searchQuery}">
    <div th:if="${not #strings.isEmpty(itemSearchDto.searchQuery)}" class="center">
        <p class="h3 font-weight-bold" th:text="${itemSearchDto.searchQuery} + '검색 결과'"></p>
    </div>

    <div class="row">
        <th:block th:each="item, status: ${items.getContent()}">
            <div class="col-md-4 margin">
                <div class="card">
                    <a th:href="'/item/' +${item.id}" class="text-dark">
                        <img th:src="${item.imgUrl}" class="card-img-top" th:alt="${item.itemNm}" height="400">
                        <div class="card-body">
                            <h4 class="card-title">[[${item.itemNm}]]</h4>
                            <p class="card-text">[[${item.itemDetail}]]</p>
                            <h3 class="card-title text-danger">[[${item.price}]]원</h3>
                        </div>
                    </a>
                </div>
            </div>
        </th:block>
    </div>
```

부트스트랩의 슬라이드를 보여주는 Carousel 컴포넌트를 이용하여 배너를 만든다. 쇼핑몰 경우 보통 행사 중인 상품을 광고하는 데 사용한다.

쇼핑몰 오른쪽 상단의 Search 기능을 이용해서 상품을 검색할 때 페이징 처리 시 해당 검색어를 유지하기 위해서 hidden 값으로 검색어를 유지한다.

조회한 메인 상품 데이터를 보여준다. 부트트랩의 Card 컴포넌트를 이용해서 사용자가 카드 형태로 상품의 이름, 내용, 가격을 볼 수 있다.

6.4 메인 화면

6.4.6. 메인 페이지 구현하기

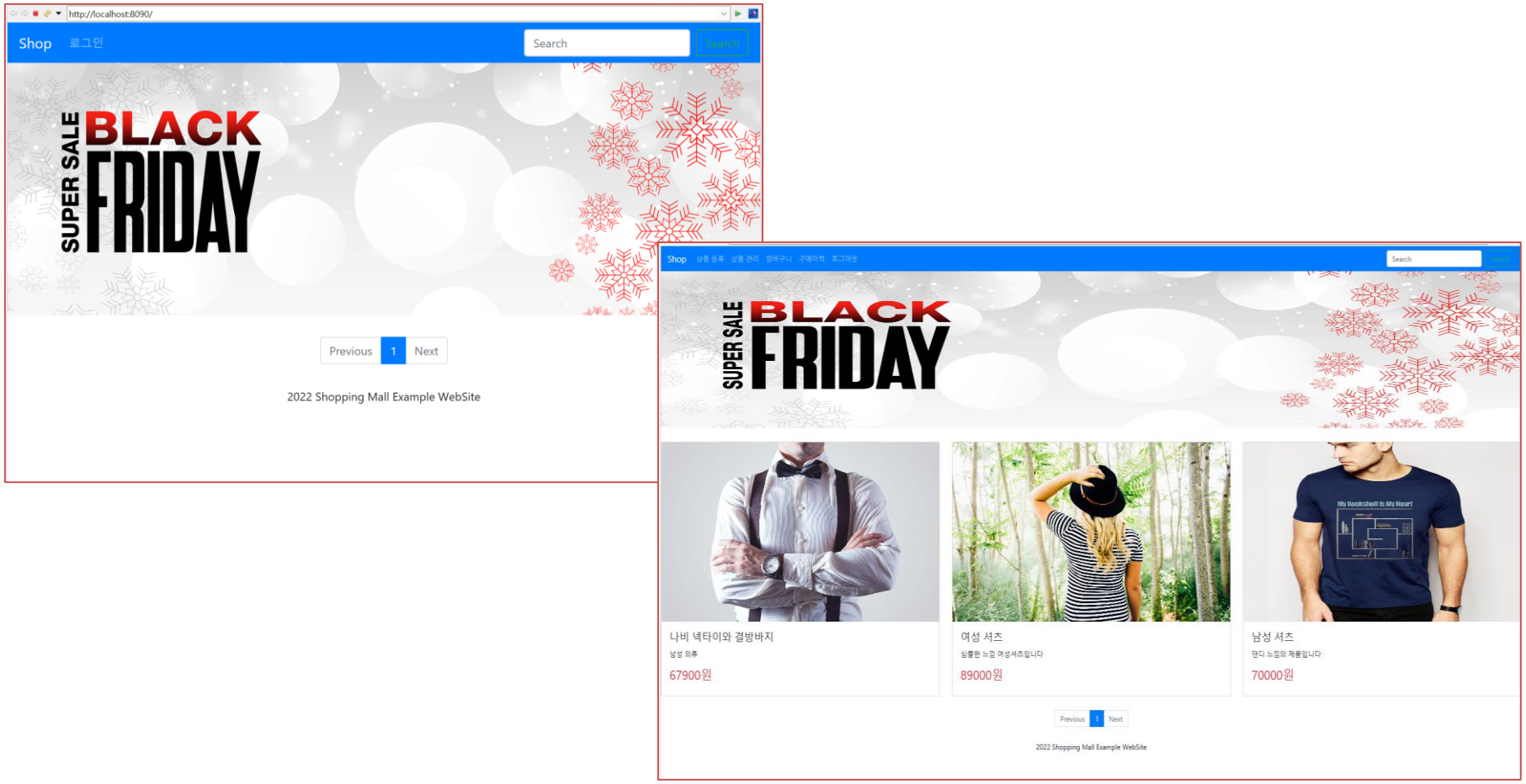
- 메인 페이지에 상품 데이터를 보여주기 위해서 main.html을 수정한다.

```
<div th:with="start=${(items.number/maxPage)*maxPage + 1}, end=(${(items.totalPages == 0) ? 1 : (start + (maxPage - 1) < items.totalPages ? start + (maxPage - 1) : items.totalPages)})" >
    <ul class="pagination justify-content-center">

        <li class="page-item" th:classappend="${items.number eq 0}?'disabled':''">
            <a th:href="@{'/' + '?searchQuery=' + ${itemSearchDto.searchQuery} + '&page=' + ${items.number-1}}" aria-label='Previous' class="page-link">
                <span aria-hidden='true'>Previous</span>
            </a>
        </li>

        <li class="page-item" th:each="page: ${#numbers.sequence(start, end)}" th:classappend="${items.number eq page-1}?'active':''">
            <a th:href="@{'/' +'?searchQuery=' + ${itemSearchDto.searchQuery} + '&page=' + ${page-1}}" th:inline="text" class="page-link">[[${page}]]</a>
        </li>

        <li class="page-item" th:classappend="${items.number+1 ge items.totalPages}?'disabled':''">
            <a th:href="@{'/' +'?searchQuery=' + ${itemSearchDto.searchQuery} + '&page=' + ${items.number+1}}" aria-label='Next' class="page-link">
                <span aria-hidden='true'>Next</span>
            </a>
        </li>

    </ul>
</div>

</div>
```

6.4 메인 화면

6.4.7. 메인 페이지 구현하기

- 메인 페이지 실행하기

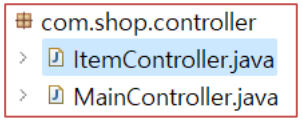


6.5 상품 상세 페이지

6.5.1. 상품 상세 페이지 구현하기

- 메인 페이지에서 상품 이미지나 상품 정보를 클릭 시 상품의 상세 정보를 보여주는 페이지를 구현하기
- 상품 상세 페이지로 이동할 수 있도록 ItemController 클래스에 코드를 추가한다. 기존 상품 수정 페이지 구현에서 미리 만들어 둔 상품을 가져오는 로직을 그대로 사용한다.

```
@GetMapping(value = "/item/{itemId}")
public String itemDtl(Model model, @PathVariable("itemId") Long itemId) {
    ItemFormDto itemFormDto = itemService.getItemDtl(itemId);
    model.addAttribute("item", itemFormDto);
    return "item/itemDtl";
}
```



6.5 상품 상세 페이지

6.5.2. 상품 상세 페이지 구현하기

- 상품 상세 페이지 itemDtl.html 만들기

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org"
      xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout"
      layout:decorate="~{layouts/layout1}">

<head>
    <meta name="_csrf" th:content="${_csrf.token}"/>
    <meta name="_csrf_header" th:content="${_csrf.headerName}"/>
</head>

<!-- 사용자 스크립트 추가 -->
<th:block layout:fragment="script">
    <script th:inline="javascript">
        $(document).ready(function(){

            calculateToalPrice();

            $("#count").change( function(){
                calculateToalPrice();
            });
        });

        function calculateToalPrice(){
            var count = $("#count").val();
            var price = $("#price").val();
            var totalPrice = price*count;
            $("#totalPrice").html(totalPrice + '원');
        }

    </script>
</th:block>
```

item
itemDtl.html

현재 주문할 수량과 상품 한 개 당 가격을 곱해서 결제 금액을 구해주는 함수

6.5 상품 상세 페이지

6.5.2. 상품 상세 페이지 구현하기

- 상품 상세 페이지 itemDtl.html 만들기

```
<!-- 사용자 CSS 추가 -->
<th:block layout:fragment="css">
    <style>
        .mgb-15{
            margin-bottom:15px;
        }
        .mgt-30{
            margin-top:30px;
        }
        .mgt-50{
            margin-top:50px;
        }
        .repImgDiv{
            margin-right:15px;
            height:auto;
            width:50%;
        }
        .repImg{
            width:100%;
            height:400px;
        }
        .wd50{
            height:auto;
            width:50%;
        }
    </style>
</th:block>
```

6.5 상품 상세 페이지

6.5.2. 상품 상세 페이지 구현하기

▪ 상품 상세 페이지 itemDtl.html 만들기

```
<div layout:fragment="content" style="margin-left:25%;margin-right:25%">

    <input type="hidden" id="itemId" th:value="${item.id}">

    <div class="d-flex">
        <div class="repImgDiv">
            <img th:src="${item.itemImgDtoList[0].imgUrl}" class = "rounded repImg" th:alt="${item.itemNm}">
        </div>
        <div class="wd50">
            <span th:if="${item.itemSellStatus == T(com.shop.constant.ItemSellStatus).SELL}" class="badge badge-primary mgb-15">
                판매중
            </span>
            <span th:unless="${item.itemSellStatus == T(com.shop.constant.ItemSellStatus).SELL}" class="badge btn-danger mgb-15" >
                품절
            </span>
            <div class="h4" th:text="${item.itemNm}"></div>
            <hr class="my-4">

            <div class="text-right">
                <div class="h4 text-danger text-left">
                    <input type="hidden" th:value="${item.price}" id="price" name="price">
                    <span th:text="${item.price}"></span>원
                </div>
                <div class="input-group w-50">
                    <div class="input-group-prepend">
                        <span class="input-group-text">수량</span>
                    </div>
                    <input type="number" name="count" id="count" class="form-control" value="1" min="1">
                </div>
            </div>
            <hr class="my-4">
```

6.5 상품 상세 페이지

6.5.2. 상품 상세 페이지 구현하기

▪ 상품 상세 페이지 itemDtl.html 만들기

```
        <div class="text-right mgt-50">
            <h5>결제 금액</h5>
            <h3 name="totalPrice" id="totalPrice" class="font-weight-bold"></h3>
        </div>
        <div th:if="${item.itemSellStatus == T(com.shop.constant.ItemSellStatus).SELL}" class="text-right">
            <button type="button" class="btn btn-light border border-primary btn-lg">장바구니 담기</button>
            <button type="button" class="btn btn-primary btn-lg">주문하기</button>
        </div>
        <div th:unless="${item.itemSellStatus == T(com.shop.constant.ItemSellStatus).SELL}" class="text-right">
            <button type="button" class="btn btn-danger btn-lg">품절</button>
        </div>
    </div>
</div>

<div class="jumbotron jumbotron-fluid mgt-30">
    <div class="container">
        <h4 class="display-5">상품 상세 설명</h4>
        <hr class="my-4">
        <p class="lead" th:text="${item.itemDetail}"></p>
    </div>
</div>

<div th:each="itemImg : ${item.itemImgDtoList}" class="text-center">
    <img th:if="${not #strings.isEmpty(itemImg.imgUrl)}" th:src="${itemImg.imgUrl}" class="rounded mgb-15" width="800">
</div>

</div>

</html>
```

등록된 상품 이미지를 반복 구문을 통해 보여주고 있다. 실제 쇼핑몰에선 상품에 대한 정보를 예쁘게 이미지로 만들어서 보여준다.

6.5 상품 상세 페이지

6.5.3. 상품 상세 페이지 구현하기

- 실행하기 (장바구니와 주문은 7장에서 구현한다.)