

07장 주문



● 학습 목표

- 주문, 주문 이력 조회, 주문 취소 기능 구현을 통해서 주문 프로세스를 학습한다.
- Spring Data JPA를 이용하여 주문 데이터 조회 시 조회를 최적화 하는 방법을 학습한다.

7장



7.1 주문 기능 구현하기

7.1.1. 주문 기능 구현하기

- 고객이 상품을 주문하면 현재 상품의 재고에서 주문 수량만큼 재고를 감소시켜야 한다. 고객이 주문했는데 실제 재고가 없다면 배송하지 못하고 결품처리 되기에 주문 수량만큼 상품의 재고를 감소시켜야 한다. 또한 주문 수량이 현재 재고 수보다 클 경우 주문되지 않도록 구현한다.
- 주문보다 재고가 적을 때 발생시킬 **RuntimeException**을 상속받는 **OutOfStockException**클래스를 생성한다.

```
package com.shop.exception;
```

```
public class OutOfStockException extends RuntimeException{
```

```
    public OutOfStockException(String message) {  
        super(message);  
    }
```

```
}
```

A screenshot of an IDE snippet showing the package `com.shop.exception` and the class `OutOfStockException.java`.



7.1 주문 기능 구현하기

7.1.2. 주문 기능 구현하기

- 상품을 주문할 경우 상품의 재고를 감소시키는 로직을 작성한다. 엔티티 클래스 안에서 비즈니스 로직을 메소드로 작성하면 코드의 재사용과 데이터 변경 포인트를 한 곳으로 모을 수 있다는 장점이 있다.

```
import com.shop.exception.OutOfStockException;
```

```
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();  
}
```

```
public void removeStock(int stockNumber) {  
    int restStock = this.stockNumber - stockNumber; // 상품재고 수량에서 주문 후 남은 재고수량을 구한다.  
    if(restStock < 0) {  
        throw new OutOfStockException("상품의 재고가 부족 합니다. (현재 재고 수량: " + this.stockNumber + ")");  
    }  
    // 상품재고가 주문 수량보다 적을 때 재고 부족 예외를 발생시킨다.  
    this.stockNumber = restStock; // 주문 후 남은 재고 수량을 상품의 현재 재고 값으로 할당한다.  
}
```

```
public static OrderItem createOrderItem(Item item, int count){  
    OrderItem orderItem = new OrderItem();  
    orderItem.setItem(item);  
    orderItem.setCount(count);  
    orderItem.setOrderPrice(item.getPrice());  
    item.removeStock(count);  
    return orderItem;  
}
```

```
com.shop.entity  
  > BaseEntity.java  
  > BaseTimeEntity.java  
  > Cart.java  
  > CartItem.java  
  > Item.java
```

```
OrderItem.java
```



7.1 주문 기능 구현하기

7.1.3. 주문 기능 구현하기

- 주문할 상품과 주문 수량을 통해 **OrderItem 객체**를 만드는 메소드를 작성한다.

```
private Order order;  
private int orderPrice; //주문가격  
private int count; //수량
```

```
public static OrderItem createOrderItem(Item item, int count) {  
    OrderItem orderItem = new OrderItem();  
    orderItem.setItem(item);    // 주문할 상품과 주문 수량을 세팅한다.  
    orderItem.setCount(count);  
    orderItem.setOrderPrice(item.getPrice()); // 현재시간 기준으로 상품가격을 주문가격으로 세팅  
    item.removeStock(count); // 주문 수량만큼 상품 재고 수량을 감소시킨다.  
    return orderItem;  
}
```

```
public int getTotalPrice() { // 주문 가격과 주문 수량을 곱해서 해당 상품을 주문한 총 가격을 계산하는 메소드  
    return orderPrice*count;  
}
```

```
}
```

```
com.shop.entity  
> BaseEntity.java  
> BaseTimeEntity.java  
> Cart.java  
> CartItem.java  
> Item.java  
> ItemImg.java  
> Member.java  
> Order.java  
> OrderItem.java
```

```
public void removeStock(int stockNumber){  
    int restStock = this.stockNumber - stockNumber;  
    if(restStock<0){  
        throw new OutOfStockException("상품의 재고가 부족 합니다. (현재 재고 수량: " + this.stockNumber + ")");  
    }  
    this.stockNumber = restStock;  
}
```

Item.java



7.1 주문 기능 구현하기

```
com.shop.entity
> BaseEntity.java
> BaseTimeEntity.java
> Cart.java
> CartItem.java
> Item.java
> ItemImg.java
> Member.java
> Order.java
```

7.1.4. 주문 기능 구현하기

- 생성한 주문 상품 객체를 이용하여 주문 객체를 만드는 메소드를 작성한다.
상품과 주문, 주문 상품 엔티티에 주문과 관련된 비즈니스 로직을 추가한다.

```
private List<OrderItem> orderItems = new ArrayList<>();
```

```
public void addOrderItem(OrderItem orderItem) { // ... 주문상품 정보들을 담아두는 메서드
    orderItems.add(orderItem); // orderItem 객체를 order객체의 orderItems에 추가한다.
    orderItem.setOrder(this); // Order 엔티티와 OrderItem 엔티티가 양방향 참조관계이므로, OrderItem객체에도 order객체를 세팅한다.
}
```

```
public static Order createOrder(Member member, List<OrderItem> orderItemList) { // ... 주문 엔티티 생성하는 메서드
    Order order = new Order();
    order.setMember(member); // 상품을 주문한 회원의 정보를 세팅한다.

    for(OrderItem orderItem : orderItemList) { // 상품 페이지에서는 1개의 상품을 주문하지만, 장바구니 페이지에선 한 번에 여러 개의 상품을
        // 주문할 수 있다. 따라서 여러 개의 주문 상품을 담을 수 있도록 리스트 형태로 파라미터 값을 받으며 주문 객체에 orderItem 객체를 추가한다.
        order.addOrderItem(orderItem);
    }
}
```

```
order.setOrderStatus(OrderStatus.ORDER); // 주문상태를 Order로 세팅
order.setOrderDate(LocalDateTime.now()); // 현재 시간을 주문시간으로 세팅
return order;
```

```
public int getTotalPrice() { // ... 총주문 금액을 구하는 메소드
    int totalPrice = 0;
    for(OrderItem orderItem : orderItems) {
        totalPrice += orderItem.getTotalPrice();
    }
    return totalPrice;
}
```

OrderService.java

```
orderItemList.add(orderItem);
Order order = Order.createOrder(member, orderItemList);
```

OrderItem.java

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

양방향

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

```
public void addOrderItem(OrderItem orderItem) {
    orderItems.add(orderItem);
    orderItem.setOrder(this);
}
```

Order.java



7.1 주문 기능 구현하기

7.1.5. 주문 기능 구현하기

- 상품 상세 페이지에서 주문할 상품의 아이디와 주문 수량을 전달받을 **OrderDto** 클래스를 생성한다.
주문 최소 수량은 1개, 주문 최대 수량은 999개로 제한한다.

```
package com.shop.dto;
```

```
import lombok.Getter;  
import lombok.Setter;
```

```
import javax.validation.constraints.Max;  
import javax.validation.constraints.Min;  
import javax.validation.constraints.NotNull;
```

```
@Getter @Setter
```

```
public class OrderDto {
```

```
    @NotNull(message = "상품 아이디는 필수 입력 값입니다.")  
    private Long itemId;
```

```
    @Min(value = 1, message = "최소 주문 수량은 1개 입니다.")  
    @Max(value = 999, message = "최대 주문 수량은 999개 입니다.")  
    private int count;
```

```
}
```

```
com.shop.dto  
> ItemDto.java  
> ItemFormDto.java  
> ItemImgDto.java  
> ItemSearchDto.java  
> MainItemDto.java  
> MemberFormDto.java  
> OrderDto.java
```



7.1 주문 기능 구현하기

7.1.6. 주문 기능 구현하기

- 주문 로직을 작성하기 위해서 **OrderService 클래스**를 생성한다.

```
package com.shop.service;
```

```
import com.shop.dto.OrderDto;
import com.shop.entity.*;
import com.shop.repository.ItemRepository;
import com.shop.repository.MemberRepository;
import com.shop.repository.OrderRepository;
import lombok.RequiredArgsConstructor;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
```

```
import javax.persistence.EntityNotFoundException;
import java.util.ArrayList;
import java.util.List;
```

```
@Service
@Transactional
@RequiredArgsConstructor
```

```
public class OrderService {
```

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

```
com.shop.service
> FileService.java
> ItemImgService.java
> ItemService.java
> MemberService.java
> OrderService.java
```



7.1 주문 기능 구현하기

7.1.6. 주문 기능 구현하기

- 주문 로직을 작성하기 위해서 **OrderService** 클래스를 생성한다.

```
public Long order(OrderDto orderDto, String email) {
```

```
    Item item = itemRepository.findById(orderDto.getItemId()) // 주문할 상품을 조회한다. ①  
    .orElseThrow(EntityNotFoundException::new);
```

```
    Member member = memberRepository.findByEmail(email); // 현재 로그인한 회원의 이메일 정보를 이용해서  
                                                         // 회원정보를 조회한다. ②
```

```
    List<OrderItem> orderItemList = new ArrayList<>();
```

```
    OrderItem orderItem = OrderItem.createOrderItem(item, orderDto.getCount()); // 주문할 상품 엔티티와 주문수량을  
                                                                                // 이용해서 주문상품 엔티티를 생성한다. ③
```

```
    orderItemList.add(orderItem);
```

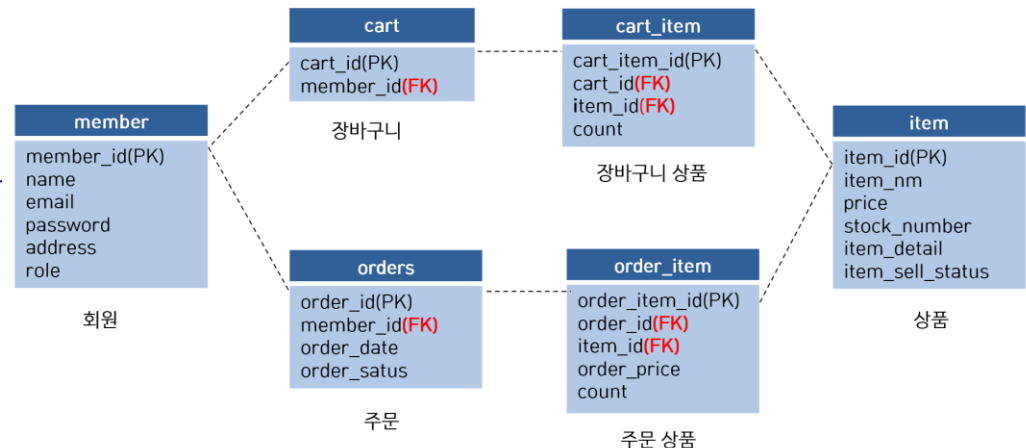
```
    Order order = Order.createOrder(member, orderItemList); // 회원정보와 주문할 상품 리스트 정보를 이용하여  
                                                           // 주문 엔티티를 생성한다. ④
```

```
    orderRepository.save(order); // 생성한 주문 엔티티를 저장한다. ⑤
```

```
    return order.getId();
```

```
}
```

```
}
```





7.1 주문 기능 구현하기

7.1.7. 주문 기능 구현하기

- 주문 관련 요청들을 처리하기 위해서 **OrderController** 클래스를 만든다.

상품주문에서 웹 페이지의 새로 고침 없이 서버에 주문을 요청하기 위해서 비동기 방식을 사용한다.

```
package com.shop.controller;
```

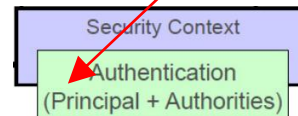
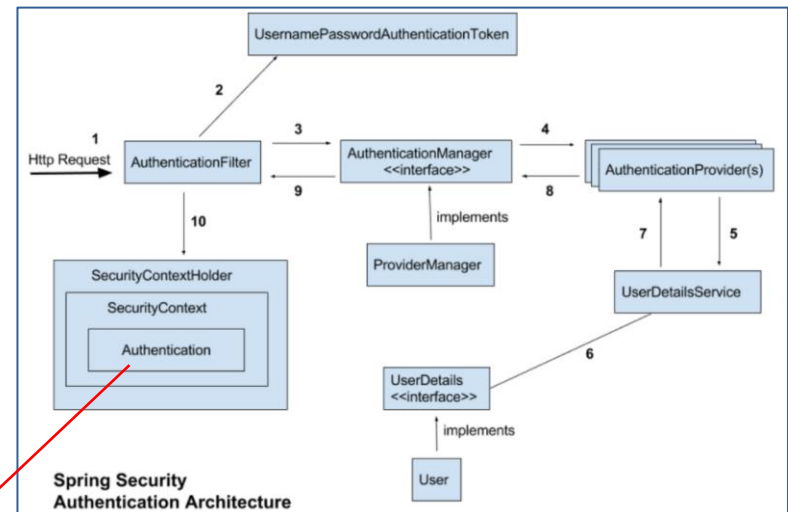
```
import com.shop.dto.OrderDto;
import com.shop.service.OrderService;
import lombok.RequiredArgsConstructor;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.stereotype.Controller;
import org.springframework.validation.BindingResult;
import org.springframework.validation.FieldError;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.ResponseBody;
```

```
import javax.validation.Valid;
import java.security.Principal;
import java.util.List;
```

```
@Controller
@RequiredArgsConstructor
public class OrderController {
```

```
    private final OrderService orderService;
```

```
com.shop.controller
> ItemController.java
> MainController.java
> MemberController.java
> OrderController.java
```



7.1 주문 기능 구현하기

새로 고침 없이 비동기 방식으로 처리하기 위해 `@ResponseBody`를 사용한다.

`bindingResult`에 에러가 있다면, String 값들을 모아줄 수 있는 `StringBuilder`를 선언하고 `@Valid`의 에러 정보가 담겨져 있는 `BindingResult`의 필드값들을 모두 가져와서 `StringBuilder`에 넣어준다. 그리고 `ResponseEntity`를 생성하여 반환해 준다.

`Principal`는 Security에서 제공하는 객체. `Principal` 말고도 `Authentication` 객체를 사용해도 무관하다. `Principal`은 현재 접속한 사용자의 기본 정보를 제공한다. 이때의 값은 `UserDetail` 값으로 전달받은 `username`값에 해당한다.

7.1.7. 주문 기능 구현하기

- 주문 관련 요청들을 처리하기 위해서 **OrderController** 클래스를 만든다.

```
@PostMapping(value = "/order")
public @ResponseBody ResponseEntity order(@RequestBody @Valid OrderDto orderDto
    , BindingResult bindingResult, Principal principal) { // 스프링에서 비동기 처리한다.
    // @RequestBody: http 요청 body 담긴 내용을 자바객체로 전달, @ResponseBody: 자바객체를 body로 전달
    if(bindingResult.hasErrors()) { // 주문정보를 받는 orderDto객체에 데이터바인딩 시 에러가 있는지 검사한다.
        StringBuilder sb = new StringBuilder();
        List<FieldError> fieldErrors = bindingResult.getFieldErrors();

        for (FieldError fieldError : fieldErrors) {
            sb.append(fieldError.getDefaultMessage());
        }

        return new ResponseEntity<String>(sb.toString(), HttpStatus.BAD_REQUEST);
    } // 에러정보를 ResponseEntity 객체에 담아 반환한다.
```

```
$.ajax({
    url : url,
    type : "POST",
    contentType : "application/json",
    data : param,
```

```
String email = principal.getName(); // 현재 로그인 유저의 정보를 얻기 위해서 @Controller가 선언된 클래스에서 메소드 인자로
// principal 객체를 넘겨 줄 경우, 해당 객체에 직접 접근할 수 있다. principal 객체에서 현재 로그인한 회원의 이메일 정보를 조회한다.
Long orderId;

try {
    orderId = orderService.order(orderDto, email); // 화면으로 넘어오는 주문정보와 회원의 이메일 정보를 이용하여 주문로직을 호출한다.
} catch (Exception e) {
    return new ResponseEntity<String>(e.getMessage(), HttpStatus.BAD_REQUEST);
}

return new ResponseEntity<Long>(orderId, HttpStatus.OK);
} // 결과값으로 생성된 주문번호와 요청이 성공했다는 http응답상태코드를 반환한다.
}
```



7.1 주문 기능 구현하기

7.1.8. 주문 기능 구현하기

- 정상적으로 동작하는지 테스트 코드를 생성한다.

```
package com.shop.service;
```

```
import com.shop.constant.ItemSellStatus;
import com.shop.dto.OrderDto;
import com.shop.entity.Item;
import com.shop.entity.Member;
import com.shop.entity.Order;
import com.shop.entity.OrderItem;
import com.shop.repository.ItemRepository;
import com.shop.repository.MemberRepository;
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.EntityNotFoundException;
import java.util.List;
import static org.junit.jupiter.api.Assertions.assertEquals;

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

```
com.shop.service
> ItemServiceTest.java
> MemberServiceTest.java
> OrderServiceTest.java
```



7.1 주문 기능 구현하기

7.1.8. 주문 기능 구현하기

- 정상적으로 동작하는지 테스트 코드를 생성한다.

```
@Autowired
private OrderService orderService;

@Autowired
private OrderRepository orderRepository;

@Autowired
ItemRepository itemRepository;

@Autowired
MemberRepository memberRepository;

public Item saveItem() { // 테스트를 위한 주문할 상품 정보를 저장하는 메소드
    Item item = new Item();
    item.setItemNm("테스트 상품");
    item.setPrice(10000);
    item.setItemDetail("테스트 상품 상세 설명");
    item.setItemSellStatus(ItemSellStatus.SELL);
    item.setStockNumber(100);
    return itemRepository.save(item);
}

public Member saveMember() { // 테스트를 위한 회원정보를 저장하는 메소드
    Member member = new Member();
    member.setEmail("test@test.com");
    return memberRepository.save(member);
}
```



7.1 주문 기능 구현하기

7.1.8. 주문 기능 구현하기

- 정상적으로 동작하는지 테스트 코드를 생성한다.

```
@Test
@DisplayName("주문 테스트")
public void order() {
    Item item = saveItem();
    Member member = saveMember();

    OrderDto orderDto = new OrderDto();
    orderDto.setCount(10); // 주문할 상품 수량과
    orderDto.setItemId(item.getId()); // 주문할 상품을 orderDto 객체에 세팅

    Long orderId = orderService.order(orderDto, member.getEmail()); // 주문로직 호출결과 생성된 주문번호를 orderId 변수에 저장한다.
    Order order = orderRepository.findById(orderId) // 주문번호를 이용하여 저장된 주문정보를 조회한다.
        .orElseThrow(EntityNotFoundException::new);

    List<OrderItem> orderItems = order.getOrderItems();

    int totalPrice = orderDto.getCount() * item.getPrice(); // 주문한 상품 총가격

    assertEquals(totalPrice, order.getTotalPrice()); // 주문한 상품 총가격과 DB에 저장된 상품의 가격을 비교하여 같으면 테스트가 성공적으로 종료
}
}
```

Runs: 1/1 Errors: 0 Failures: 0

OrderServiceTest [Runner: JUnit 5] (0.261 s)

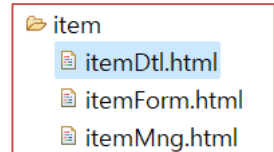
주문 테스트 (0.261 s)



7.1.9. 주문 기능 구현하기

- itemDtl.html에 주문기능인 **order()** 함수를 추가한다.
- form 태그의 submit 방식으로 서버에 요청하면 페이지가 새로 고쳐지는 단점이 있다. 따라서 Ajax를 이용하여 처리한다.

```
function order() {  
    var token = $("meta[name='_csrf']").attr("content");  
    var header = $("meta[name='_csrf_header']").attr("content");  
  
    var url = "/order";  
    var paramData = {  
        itemId : $("#itemId").val(),  
        count : $("#count").val()  
    };  
  
    var param = JSON.stringify(paramData);  
  
    $.ajax({  
        url : url,  
        type : "POST",  
        contentType : "application/json",  
        data : param,  
        beforeSend : function(xhr) {  
            /* 데이터를 전송하기 전에 헤더에 csrf값을 설정 */  
            xhr.setRequestHeader(header, token);  
        },  
        dataType : "json",  
        cache : false,  
        success : function(result, status) {  
            alert("주문이 완료 되었습니다.");  
            location.href='/';  
        },  
    });  
}
```



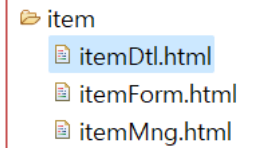


7.1 주문 기능 구현하기

7.1.9. 주문 기능 구현하기

- order()함수를 호출하도록 onclick 속성을 추가한다.

```
error : function(jqXHR, status, error) {  
    if(jqXHR.status == '401'){  
        alert('로그인 후 이용해주세요');  
        location.href='/members/login';  
    } else{  
        alert(jqXHR.responseText);  
    }  
}  
});  
}  
  
</script>  
</th:block>
```



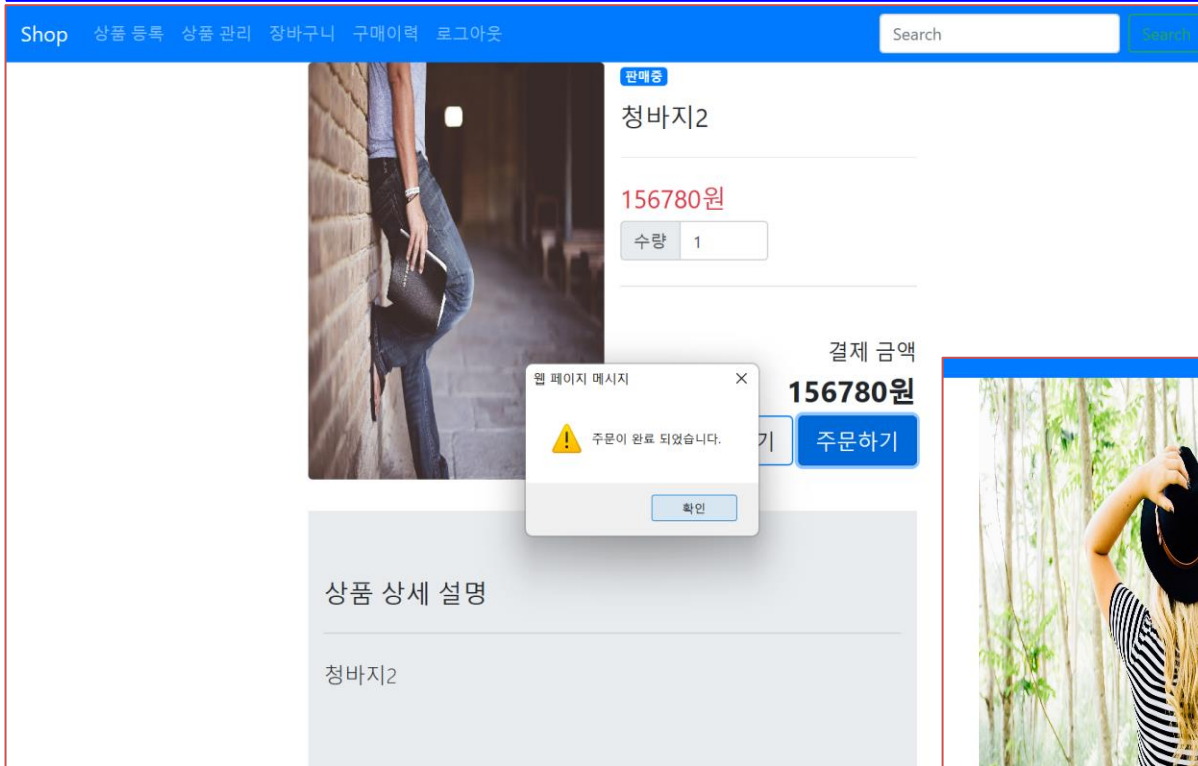
```
<button type="button" class="btn btn-primary btn-lg onclick="order()">주문하기</button>
```



7.1 주문 기능 구현하기

7.1.10. 주문 기능 구현하기

■ 실행하기





7.2 주문 이력 조회하기

7.2.1. 주문 이력 조회

- 주문 후 주문이력을 조회할 수 있는 화면을 만든다. 보통 주문 이력을 조회하는 페이지에선 주문부터 현재 배송상태까지 보여준다. 또한 반품, 교환, 주문 취소 등 기능들이 있다. **예제에선 주문 취소 기능만 구현해본다.**
- 조회한 주문 데이터를 화면에 보낼 때 사용할 즉 **주문 상품 정보**를 담은 **OrderItemDto** 클래스를 생성한다.

```
package com.shop.dto;
```

```
import com.shop.entity.OrderItem;  
import lombok.Getter;  
import lombok.Setter;
```

```
@Getter @Setter  
public class OrderItemDto {
```

```
    public OrderItemDto(OrderItem orderItem, String imgUrl) { // orderItem객체와 이미지 경로를 파라미터로 받아 멤버변수 값을 세팅  
        this.itemNm = orderItem.getItem().getItemNm();  
        this.count = orderItem.getCount();  
        this.orderPrice = orderItem.getOrderPrice();  
        this.imgUrl = imgUrl;  
    }
```

```
    private String itemNm; //상품명  
    private int count; //주문 수량  
    private int orderPrice; //주문 금액  
    private String imgUrl; //상품 이미지 경로
```

```
}
```

```
com.shop.dto  
├─ ItemDto.java  
├─ ItemFormDto.java  
├─ ItemImgDto.java  
├─ ItemSearchDto.java  
├─ MainItemDto.java  
├─ MemberFormDto.java  
├─ OrderDto.java  
└─ OrderItemDto.java
```

구매 이력

2022-03-30 23:35 주문 [주문취소](#)



청바지2
156780원 1개



7.2 주문 이력 조회하기

7.2.2. 주문 이력 조회

- **주문 정보**를 담은 **OrderHistDto** 클래스를 생성한다. (구매 이력 Dto)

```
package com.shop.dto;

import com.shop.constant.OrderStatus;
import com.shop.entity.Order;
import lombok.Getter;
import lombok.Setter;

import java.time.format.DateTimeFormatter;
import java.util.ArrayList;
import java.util.List;

@Getter @Setter
public class OrderHistDto {

    public OrderHistDto(Order order) {
        this.orderId = order.getId();
        this.orderDate = order.getOrderDate().format(DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm")); // 주문날짜 형태 수정하기
        this.orderStatus = order.getOrderStatus();
    }

    private Long orderId; //주문아이디
    private String orderDate; //주문날짜
    private OrderStatus orderStatus; //주문 상태

    private List<OrderItemDto> orderItemDtoList = new ArrayList<>();

    //주문 상품리스트
    public void addOrderItemDto(OrderItemDto orderItemDto) {
        orderItemDtoList.add(orderItemDto);
    }
}
```

```
com.shop.dto
> ItemDto.java
> ItemFormDto.java
> ItemImgDto.java
> ItemSearchDto.java
> MainItemDto.java
> MemberFormDto.java
> OrderDto.java
> OrderHistDto.java
```





7.2 주문 이력 조회하기

7.2.3. 주문 이력 조회

- **OderRepository** 인터페이스에 쿼리문을 작성한다. 조건이 복잡하지 않으면 Querydsl을 사용하기보단 @Query를 사용하는 것이 낫다. 문자열을 연결할 때 명령어 사이 띄어쓰기를 하는 것에 조심하자.

```
package com.shop.repository;

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

import org.springframework.data.domain.Pageable;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.query.Param;

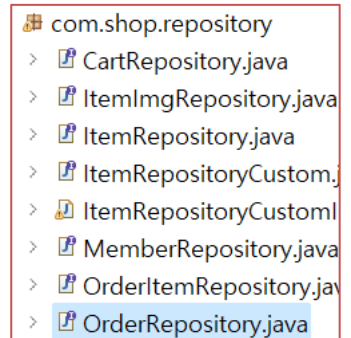
import java.util.List;
```

```
public interface OrderRepository extends JpaRepository<Order, Long> {
```

```
    @Query("select o from Order o " +
        "where o.member.email = :email " +
        "order by o.orderDate desc"
    )
    List<Order> findOrders(@Param("email") String email, Pageable pageable); // 현재 로그인한 사용자 주문 데이터 페이징 조건에 맞춰 조회

    @Query("select count(o) from Order o " +
        "where o.member.email = :email"
    )
    Long countOrder(@Param("email") String email); // 현재 로그인한 회원 주문 개수가 몇 개인지 조회한다.

}
```





7.2 주문 이력 조회하기

7.2.4. 주문 이력 조회

- **ItemImgRepository 인터페이스**에는 상품의 대표 이미지를 찾는 쿼리 메소드를 추가한다.
구매 이력 페이지에서 **주문 상품의 대표 이미지**를 보여주기 위해서 추가한다.

```
package com.shop.repository;

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

public interface ItemImgRepository extends JpaRepository<ItemImg, Long> {

    List<ItemImg> findByItemIdOrderByIdAsc(Long itemId);

    ItemImg findByIdAndRepimgYn(Long itemId, String repimgYn);

}
```

```
com.shop.repository
> CartRepository.java
> ItemImgRepository.java
```



7.2.5. 주문 이력 조회

- **OrderService** 클래스에 주문 목록을 조회하는 로직을 구현한다.

```
package com.shop.service;
```

```
import com.shop.dto.OrderHistDto;  
import com.shop.dto.OrderItemDto;  
import com.shop.repository.ItemImgRepository;  
import org.springframework.data.domain.Page;  
import org.springframework.data.domain.PageImpl;  
import org.springframework.data.domain.Pageable;
```

```
@Service  
@Transactional  
@RequiredArgsConstructor  
public class OrderService {
```

```
    private final ItemRepository itemRepository;  
    private final MemberRepository memberRepository;  
    private final OrderRepository orderRepository;  
    private final ItemImgRepository itemImgRepository;
```

```
com.shop.service  
├─ CartService.java  
├─ FileService.java  
├─ ItemImgService.java  
├─ ItemService.java  
├─ MemberService.java  
└─ OrderService.java
```



7.2 주문 이력 조회하기

7.2.5. 주문 이력 조회

- **OrderService** 클래스에 주문 목록을 조회하는 로직을 구현한다.

```
@Transactional(readOnly = true)
public Page<OrderHistDto> getOrderList(String email, Pageable pageable) { //

    List<Order> orders = orderRepository.findOrders(email, pageable); // 유저의 아이디와 페이징 조건을 이용 주문목록 조회
    Long totalCount = orderRepository.countOrder(email); // 유저의 주문 총개수 구한다.

    List<OrderHistDto> orderHistDtos = new ArrayList<>();

    for (Order order : orders) { // 주문 리스트를 순회하면서 구매 이력 페이지에 전달할 DTO 객체생성한다.
        OrderHistDto orderHistDto = new OrderHistDto(order);
        List<OrderItem> orderItems = order.getOrderItems();
        for (OrderItem orderItem : orderItems) {
            ItemImg itemImg = itemImgRepository.findByItemIdAndRepimgYn
                (orderItem.getItem().getId(), "Y"); // 주문한 상품의 대표 이미지를 조회
            OrderItemDto orderItemDto =
                new OrderItemDto(orderItem, itemImg.getItemUrl());
            orderHistDto.addOrderItemDto(orderItemDto);
        }

        orderHistDtos.add(orderHistDto);
    }

    return new PageImpl<OrderHistDto>(orderHistDtos, pageable, totalCount); // 페이지 구현 객체를 생성하여 반환
}
```

ItemImg

```
@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
```



7.2 주문 이력 조회하기

7.2.6. 주문 이력 조회

- **OrderController** 클래스에 구매 이력을 조회할 수 있도록 로직을 구현한다.

```
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import com.shop.dto.OrderHistDto;
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;
```

```
com.shop.controller
> ItemController.java
> MainController.java
> MemberController.java
> OrderController.java
```

```
@GetMapping(value = {"/orders", "/orders/{page}"})
public String orderHist(@PathVariable("page") Optional<Integer> page, Principal principal, Model model) {

    Pageable pageable = PageRequest.of(page.isPresent() ? page.get() : 0, 4); // 한번에 갖고 올 주문 개수 4개
    Page<OrderHistDto> ordersHistDtoList = orderService.getOrderList(principal.getName(), pageable);
    // 현재 로그인한 회원은 이메일과 페이징 객체를 파라미터로 전달하여 화면에 전달한 주문 목록 데이터를 리턴 값으로 받음
    model.addAttribute("orders", ordersHistDtoList);
    model.addAttribute("page", pageable.getPageNumber());
    model.addAttribute("maxPage", 5);

    return "order/orderHist";
}
```



7.2 주문 이력 조회하기

7.2.7. 주문 이력 조회

- 구매이력 페이지 **OrderHist.html**를 만든다.

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

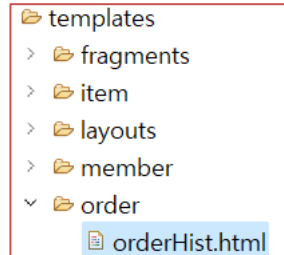
<head>
  <meta name="_csrf" th:content="${_csrf.token}"/>
  <meta name="_csrf_header" th:content="${_csrf.headerName}"/>
</head>

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

  <script th:inline="javascript">
    function cancelOrder(orderId) {
      var token = $("meta[name='_csrf']").attr("content");
      var header = $("meta[name='_csrf_header']").attr("content");

      var url = "/order/" + orderId + "/cancel";
      var paramData = {
        orderId : orderId,
      };

      var param = JSON.stringify(paramData);
```





7.2 주문 이력 조회하기

7.2.7. 주문 이력 조회

- 구매이력 페이지 **OrderHist.html**를 만든다.

```
$.ajax({
  url    : url,
  type   : "POST",
  contentType : "application/json",
  data    : param,
  beforeSend : function(xhr) {
    /* 데이터를 전송하기 전에 헤더에 csrf값을 설정 */
    xhr.setRequestHeader(header, token);
  },
  dataType : "json",
  cache    : false,
  success  : function(result, status) {
    alert("주문이 취소 되었습니다.");
    location.href='/orders/' + [${page}];
  },
  error : function(jqXHR, status, error) {
    if(jqXHR.status == '401') {
      alert('로그인 후 이용해주세요');
      location.href='/members/login';
    } else {
      alert(jqXHR.responseText);
    }
  }
});
</script>
</th:block>
```



7.2 주문 이력 조회하기

7.2.7. 주문 이력 조회

- 구매이력 페이지 **OrderHist.html**를 만든다.

```
<!-- 사용자 CSS 추가 -->
<th:block layout:fragment="css">
  <style>
    .content-mg{
      margin-left:30%;
      margin-right:30%;
      margin-top:2%;
      margin-bottom:100px;
    }
    .replmgDiv{
      margin-right:15px;
      margin-left:15px;
      height:auto;
    }
    .replmg{
      height:100px;
      width:100px;
    }
    .card{
      width:750px;
      height:100%;
      padding:30px;
      margin-bottom:20px;
    }
    .fs18{
      font-size:18px
    }
    .fs24{
      font-size:24px
    }
  </style>
</th:block>
```



7.2 주문 이력 조회하기

7.2.7. 주문 이력 조회

- 구매이력 페이지 **OrderHist.html**를 만든다.

```
<div layout:fragment="content" class="content-mg">

    <h2 class="mb-4">
        구매 이력
    </h2>

    <div th:each="order : ${orders.getContent()}">

        <div class="d-flex mb-3 align-self-center">
            <h4 th:text="${order.orderDate} + ' 주문'"></h4>
            <div class="ml-3">
                <th:block th:if="${order.orderStatus == T(com.shop.constant.OrderStatus).ORDER}">
                    <button type="button" class="btn btn-outline-secondary" th:value="${order.orderId}" onclick="cancelOrder(this.value)">주문취소</button>
                </th:block>
                <th:block th:unless="${order.orderStatus == T(com.shop.constant.OrderStatus).ORDER}">
                    <h4>(취소 완료)</h4>
                </th:block>
            </div>
        </div>

        <div class="card d-flex">
            <div th:each="orderItem : ${order.orderItemDtoList}" class="d-flex mb-3">
                <div class="replmgDiv">
                    
                </div>
                <div class="align-self-center w-75">
                    <span th:text="${orderItem.itemNm}" class="fs24 font-weight-bold"></span>
                    <div class="fs18 font-weight-light">
                        <span th:text="${orderItem.orderPrice} + '원'"></span>
                        <span th:text="${orderItem.count} + '개'"></span>
                    </div>
                </div>
            </div>
        </div>
    </div>
</div>
```



7.2 주문 이력 조회하기

7.2.7. 주문 이력 조회

- 구매이력 페이지 **OrderHist.html**를 만든다.

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

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

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

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

  </ul>
</div>

</div>

</html>
```



7.2 주문 이력 조회하기


7.2.8. 주문 이력 조회

- 구매이력 실행해보기

[Shop](#) [상품 등록](#) [상품 관리](#) [장바구니](#) [구매이력](#) [로그아웃](#)


구매 이력

2022-03-30 23:35 주문



청바지2
156780원 1개

2022-03-30 23:27 주문



청바지2
156780원 1개



7.2 주문 이력 조회하기

7.2.9. 주문 이력 조회

- 성능상 향상시킬 수 있는 부분이 있다. OrderService 클래스에 구현한 getOrderList() 메소드에서 for문을 순회하면서 order.getOrderItems()를 호출할 때마다 조회 쿼리문이 추가로 실행되는 것을 볼 수 있다.

```
@Transactional(readOnly = true)
public Page<OrderHistDto> getOrderList(String email, Pageable pageable) {

    List<Order> orders = orderRepository.findOrders(email, pageable);
    Long totalCount = orderRepository.countOrder(email);

    List<OrderHistDto> orderHistDtos = new ArrayList<>();

    for (Order order : orders) {
        OrderHistDto orderHistDto = new OrderHistDto(order);
        List<OrderItem> orderItems = order.getOrderItems();
        for (OrderItem orderItem : orderItems) {
            ItemImg itemImg = itemImgRepository.findByItemIdAndRepimgYn
                (orderItem.getItem().getId(), "Y");
            OrderItemDto orderItemDto =
                new OrderItemDto(orderItem, itemImg.getImgUrl());
            orderHistDto.addOrderItemDto(orderItemDto);
        }
        orderHistDtos.add(orderHistDto);
    }

    return new PageImpl<OrderHistDto>(orderHistDtos, pageable, totalCount);
}
```



7.2 주문 이력 조회하기

7.2.9. 주문 이력 조회

- orders 리스트의 사이즈 만큼 쿼리문이 실행된다. 만약 orders 사이즈가 100이었다면 100번의 쿼리문이 더 실행되는 것이다. 현재는 order_id에 하나의 주문번호가 조건으로 설정되는 것을 볼 수 있다.

```
Hibernate:
  select
    orderitems0_.order_id as order_id9_5_0_,
    orderitems0_.order_item_id as order_it1_5_0_,
    orderitems0_.order_item_id as order_it1_5_1_,
    orderitems0_.reg_time as reg_time2_5_1_,
    orderitems0_.update_time as update_t3_5_1_,
    orderitems0_.created_by as created_4_5_1_,
    orderitems0_.modified_by as modified5_5_1_,
    orderitems0_.count as count6_5_1_,
    orderitems0_.item_id as item_id8_5_1_,
    orderitems0_.order_id as order_id9_5_1_,
    orderitems0_.order_price as order_pr7_5_1_
  from
    order item orderitems0
  where
    orderitems0_.order_id=?
2022-03-31 10:42:01.500 INFO: 2700 -- [io-8090-exec-10]
```



7.2 주문 이력 조회하기

7.2.9. 주문 이력 조회

- 만약 orders의 주문 아이디를 “where order_id in (209, 210, 211, 212)” 이런 식으로 in 쿼리로 한 번에 조회할 수 있다면 100개가 실행될 쿼리를 하나의 쿼리로 조회할 수 있다. 이때 성능 향상을 위해서 “default_batch_fetch_size”라는 옵션을 사용할 수 있다. 조회 쿼리 한번으로 지정한 사이즈 만큼 한 번에 조회할 수 있다.
- application.properties** 설정 추가하기
`spring.jpa.properties.hibernate.default_batch_fetch_size=1000`
- 해당 옵션을 추가한 후 다시 구매 이력을 조회할 때, 반복문에서 `order.getOrderItems()` 처음 실행할 때처럼 아래와 같이 **조건절에 in 쿼리문이 실행**되는 것을 볼 수 있다. JPA를 사용하다보면 N+1 문제를 만난다. 이때 성능상 이슈가 생길 수 있기에 조심해야 한다.

```
main.html
application.properties
application-test.properties
```

```
Hibernate:
select
  orderitems0_.order_id as order_id9_5_1_,
  orderitems0_.order_item_id as order_it1_5_1_,
  orderitems0_.order_item_id as order_it1_5_0_,
  orderitems0_.reg_time as reg_time2_5_0_,
  orderitems0_.update_time as update_t3_5_0_,
  orderitems0_.created_by as created_4_5_0_,
  orderitems0_.modified_by as modified5_5_0_,
  orderitems0_.count as count6_5_0_,
  orderitems0_.item_id as item_id8_5_0_,
  orderitems0_.order_id as order_id9_5_0_,
  orderitems0_.order_price as order_pr7_5_0_
from
  order_item orderitems0_
where
  orderitems0_.order_id in (
    ?, ?, ?, ?
  )
```

```
2022-03-31 10:52:12.348 TRACE 15512 --- [nio-8090-exec-4]
```




7.3 주문 취소하기

7.3.1. 주문 취소 기능 구현

- 주문 취소 경우 주문 상태를 취소 상태로 만들어 주고 주문할 때 상품의 재고를 감소시켰던 만큼 다시 더해주면 된다.
- 상품의 재고를 더해주기 위해서 Item 클래스에 addStock 메소드를 생성한다.
- 주문을 취소할 경우 주문 수량만큼 상품의 재고를 증가시키는 메소드를 OrderItem에 구현한다.

```
public void addStock(int stockNumber) {  
    this.stockNumber += stockNumber; // 상품 재고를 증가시키는 메소드  
}
```

```
com.shop.entity  
> BaseEntity.java  
> BaseTimeEntity.java  
> Cart.java  
> CartItem.java  
> Item.java
```

```
public void cancel() {  
    this.getItem().addStock(count); // 주문 취소 시 주문 수량만큼 상품의 재고를 더한다.  
}
```

```
com.shop.entity  
> BaseEntity.java  
> BaseTimeEntity.java  
> Cart.java  
> CartItem.java  
> Item.java  
> ItemImg.java  
> Member.java  
> Order.java  
> OrderItem.java
```



7.3 주문 취소하기

7.3.2. 주문 취소 기능 구현

- Item 클래스에 주문 취소 시 주문 수량을 상품의 재고에 더해주는 로직과 주문 상태를 취소 상태로 바꿔주는 메소드를 **Order 클래스**에 구현한다.

```
public void cancelOrder() {  
    this.orderStatus = OrderStatus.CANCEL;  
    for (OrderItem orderItem : orderItems) {  
        orderItem.cancel();  
    }  
}
```

```
com.shop.entity  
> BaseEntity.java  
> BaseTimeEntity.java  
> Cart.java  
> CartItem.java  
> Item.java  
> ItemImg.java  
> Member.java  
> Order.java
```



7.3 주문 취소하기

7.3.3. 주문 취소 기능 구현

- **OrderService** 클래스에 주문을 취소하는 로직을 구현한다.

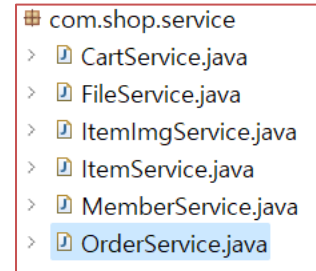
```
import org.thymeleaf.util.StringUtils;
```

```
@Transactional(readOnly = true)
public boolean validateOrder(Long orderId, String email) {
    // 현재 로그인한 사용자와 주문 데이터를 생성한 사용자가 같은지 검사를 한다. 같을 때는 true를 반환하고 같지 않을 경우 false 반환
    Member curMember = memberRepository.findByEmail(email);
    Order order = orderRepository.findById(orderId)
        .orElseThrow(EntityNotFoundException::new);
    Member savedMember = order.getMember();

    if(!StringUtils.equals(curMember.getEmail(), savedMember.getEmail())) {
        return false;
    }

    return true;
}

public void cancelOrder(Long orderId) {
    Order order = orderRepository.findById(orderId)
        .orElseThrow(EntityNotFoundException::new);
    order.cancelOrder(); // 주문 취소 상태로 변경하면 변경 감지 기능에 의해서 트랜잭션이 끝날 때 update 쿼리가 실행
}
```





7.3 주문 취소하기

7.3.4. 주문 취소 기능 구현

- **OrderController** 클래스에 주문번호(orderId)를 받아서 주문 취소 로직을 호출하는 메소드를 만든다. 상품을 장바구니에 담을 때처럼 비동기 요청을 받아서 처리한다.

```
@PostMapping("/order/{orderId}/cancel")
public @ResponseBody ResponseEntity cancelOrder(@PathVariable("orderId") Long orderId, Principal principal) {

    if(!orderService.validateOrder(orderId, principal.getName())) {
        return new ResponseEntity<String>("주문 취소 권한이 없습니다.", HttpStatus.FORBIDDEN);
    } // 자바스크립트에서 취소할 주문 번호는 조작이 가능하므로 다른 사람의 주문을 취소하지 못하도록 주문 취소 권한을 검사한다.

    orderService.cancelOrder(orderId); // 주문 취소 로직을 호출
    return new ResponseEntity<Long>(orderId, HttpStatus.OK);
}
```

```
com.shop.controller
> ItemController.java
> MainController.java
> MemberController.java
> OrderController.java
```



7.3 주문 취소하기

7.3.5. 주문 취소 기능 구현

- **OrderServiceTest** 클래스에 주문번호(orderId)를 받아서 주문 취소 로직을 호출하는 메소드를 만든다. 상품을 장바구니에 담을 때처럼 비동기 요청을 받아서 처리한다.

```
import com.shop.constant.OrderStatus;
```

```
@Test
```

```
@DisplayName("주문 취소 테스트")
```

```
public void cancelOrder() {
```

```
    Item item = saveItem();
```

```
    Member member = saveMember(); // 상품과 회원 데이터를 생성한다. 생성한 상품 재고는 100개다.
```

```
    OrderDto orderDto = new OrderDto();
```

```
    orderDto.setCount(10);
```

```
    orderDto.setItemId(item.getId());
```

```
    Long orderId = orderService.order(orderDto, member.getEmail()); // 주문 데이터를 생성, 주문개수는 총10개
```

```
    Order order = orderRepository.findById(orderId)
```

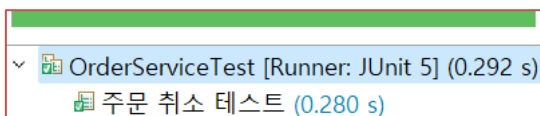
```
        .orElseThrow(EntityNotFoundException::new); // 생성한 주문 엔티티를 조회
```

```
    orderService.cancelOrder(orderId); // 해당 주문을 취소
```

```
    assertEquals(OrderStatus.CANCEL, order.getOrderStatus()); // 주문 상태가 취소 상태라면 테스트 통과
```

```
    assertEquals(100, item.getStockNumber()); // 취소 후 상품 재고가 처음 책 개수인 100와 동일하면 테스트 통과
```

```
}
```





7.3 주문 취소하기

7.3.6. 주문 취소 기능 구현

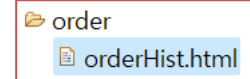
- **orderHist.html**에 주문취소 기능을 호출하는 함수를 만든다.

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

<script th:inline="javascript">
    function cancelOrder(orderId) {
        var token = $("meta[name='_csrf']").attr("content");
        var header = $("meta[name='_csrf_header']").attr("content");

        var url = "/order/" + orderId + "/cancel";
        var paramData = {
            orderId : orderId,
        };

        var param = JSON.stringify(paramData);
        $.ajax({
            url : url,
            type : "POST",
            contentType : "application/json",
            data : param,
            beforeSend : function(xhr) {
                /* 데이터를 전송하기 전에 헤더에 csrf값을 설정 */
                xhr.setRequestHeader(header, token);
            },
            dataType : "json",
            cache : false,
            success : function(result, status) {
                alert("주문이 취소 되었습니다.");
                location.href = '/orders/' + [{${page}}];
            },
        });
    }
</script>
</th:block>
```





7.3 주문 취소하기

7.3.6. 주문 취소 기능 구현

- 주문취소를 클릭했을 때 함수가 호출되도록 한다.

```
error : function(jqXHR, status, error){  
    if(jqXHR.status == '401'){  
        alert('로그인 후 이용해주세요');  
        location.href='/members/login';  
    } else{  
        alert(jqXHR.responseText);  
    }  
}  
});  
</script>  
</th:block>
```

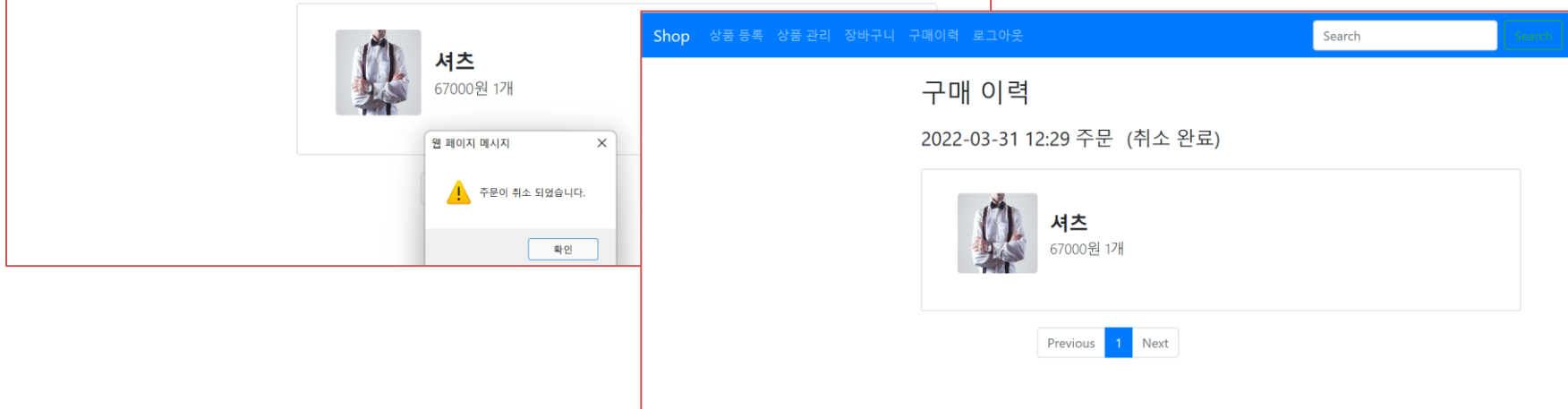
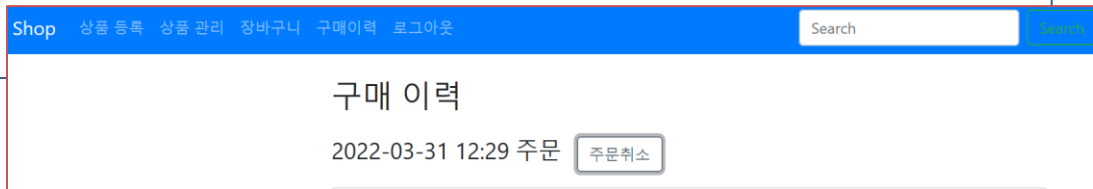
```
<button type="button" class="btn btn-outline-secondary" th:value="${order.orderId}" onclick="cancelOrder(this.value)">주문취소  
</button>
```



7.3 주문 취소하기

7.3.7. 주문 취소 기능 구현

- **OrderServiceTest 클래스**에 주문번호(orderId)를 받아서 주문 취소 로직을 호출하는 메소드를 만든다. 상품을 장바구니에 담을 때처럼 비동기 요청을 받아서 처리한다.



08장 장바구니



- 학습 목표

- 장바구니 기능으로 상품을 장바구니에 담거나 주문, 삭제하는 프로세스를 학습한다.

8장



8.1 장바구니 담기

8.1.1. 장바구니 담기 구현하기

- 상품 상세 페이지에서 장바구니에 담을 수량을 선택하고 장바구니 담기 버튼을 클릭할 때 상품이 장바구니에 담기는 기능을 먼저 구현한다.
- 상품 상세 페이지에서 장바구니에 담을 상품의 아이디와 수량을 전달받을 CartItemDto를 생성한다.
장바구니에 담을 상품의 최소 수량은 1개 이상으로 제한한다.

```
package com.shop.dto;
```

```
import lombok.Getter;  
import lombok.Setter;
```

```
import javax.validation.constraints.Min;  
import javax.validation.constraints.NotNull;
```

```
@Getter @Setter  
public class CartItemDto {
```

```
    @NotNull(message = "상품 아이디는 필수 입력 값 입니다.")  
    private Long itemId;
```

```
    @Min(value = 1, message = "최소 1개 이상 담아주세요")  
    private int count;
```

```
}
```

```
com.shop.dto  
> CartItemDto.java
```



8.1 장바구니 담기

8.1.2. 장바구니 담기 구현하기

- 회원 1명당 1개의 장바구니를 갖기에 처음 장바구니에 상품을 담을 때는 해당 회원의 바구니를 생성해줘야 한다.
- Cart 클래스에 회원 엔티티를 파라미터로 받아서 장바구니 엔티티를 생성하는 로직을 추가한다.

```
public static Cart createCart(Member member) {  
    Cart cart = new Cart();  
    cart.setMember(member);  
    return cart;  
}
```

```
com.shop.entity  
> BaseEntity.java  
> BaseTimeEntity.java  
> Cart.java
```



8.1 장바구니 담기

8.1.3. 장바구니 담기 구현하기

- 장바구니에 담을 상품 엔티티를 생성하는 메소드와 장바구니에 담을 수량을 증가시켜주는 메소드를 **CartItem** 클래스에 추가한다.

```
com.shop.entity
> BaseEntity.java
> BaseTimeEntity.java
> Cart.java
> CartItem.java
```

```
public static CartItem createCartItem(Cart cart, Item item, int count) {
    CartItem cartItem = new CartItem();
    cartItem.setCart(cart);
    cartItem.setItem(item);
    cartItem.setCount(count);
    return cartItem;
}
```

```
public void addCount(int count) { // 장바구니에 기존에 담겨 있는 상품인데, 해당 상품을 추가로 장바구니에 담을 때
    // 기존 수량에 현재 담을 수량을 더해줄 때 사용할 메소드
    this.count += count;
}
```



8.1 장바구니 담기

8.1.4. 장바구니 담기 구현하기

- 현재 로그인한 회원의 Cart 엔티티를 찾기 위해서 CartRepository에 쿼리 메소드를 추가한다.
- 장바구니에 들어갈 상품을 저장하거나 조회하기 위해서 CartItemRepository 인터페이스를 생성한다.

```
package com.shop.repository;

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

public interface CartRepository extends JpaRepository<Cart, Long> {

    Cart findByMemberId(Long memberId);

}
```

```
com.shop.repository
> CartRepository.java
```

```
package com.shop.repository;

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

public interface CartItemRepository extends JpaRepository<CartItem, Long> {

    CartItem findByCartIdAndItemId(Long cartId, Long itemId);

}
```

```
com.shop.repository
> CartItemRepository.java
> CartRepository.java
```



8.1 장바구니 담기

8.1.5. 장바구니 담기 구현하기

- 장바구니에 상품을 담는 로직을 작성하기 위하여 **CartService** 클래스를 생성한다.

```
package com.shop.service;
```

```
import com.shop.dto.CartItemDto;
import com.shop.entity.Cart;
import com.shop.entity.CartItem;
import com.shop.entity.Item;
import com.shop.entity.Member;
import com.shop.repository.CartItemRepository;
import com.shop.repository.CartRepository;
import com.shop.repository.ItemRepository;
import com.shop.repository.MemberRepository;
import lombok.RequiredArgsConstructor;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
import javax.persistence.EntityNotFoundException;
```

```
@Service
```

```
@RequiredArgsConstructor
```

```
@Transactional
```

```
public class CartService {
```

```
    private final ItemRepository itemRepository;
    private final MemberRepository memberRepository;
    private final CartRepository cartRepository;
    private final CartItemRepository cartItemRepository;
```

```
com.shop.service
> CartService.java
```



8.1 장바구니 담기

8.1.6. 장바구니 담기 구현하기

- 장바구니에 상품을 담는 로직을 작성하기 위하여 **CartService 클래스**를 생성한다.

```
public Long addCart(CartItemDto cartItemDto, String email) {  
    Item item = itemRepository.findById(cartItemDto.getItemId()) // 장바구니에 담을 상품 엔티티를 조회한다.  
        .orElseThrow(EntityNotFoundException::new);  
    Member member = memberRepository.findByEmail(email); // 현재 로그인한 회원 엔티티를 조회한다.  
  
    Cart cart = cartRepository.findByMemberId(member.getId()); // 현재 로그인한 회원의 장바구니 엔티티를 조회한다.  
    if(cart == null) { // 상품을 처음으로 장바구니에 담을 경우 해당 회원의 장바구니 엔티티를 생성한다.  
        cart = Cart.createCart(member);  
        cartRepository.save(cart);  
    }  
  
    CartItem savedCartItem = cartItemRepository.findByCartIdAndItemId(cart.getId(), item.getId());  
        // 현재 상품이 장바구니에 이미 들어가 있는지 조회한다.  
    if(savedCartItem != null) {  
        savedCartItem.addCount(cartItemDto.getCount()); // 장바구니에 이미 있던 상품일 경우 기존 수량에 현재 바구니에 담을 수량만큼을 더해준다.  
        return savedCartItem.getId();  
    } else {  
        CartItem cartItem = CartItem.createCartItem(cart, item, cartItemDto.getCount());  
        // 장바구니 엔티티, 상품 엔티티, 장바구니에 담을 수량을 이용하여 CartItem 엔티티를 생성한다.  
        cartItemRepository.save(cartItem); // 장바구니에 들어갈 상품을 저장한다.  
        return cartItem.getId();  
    }  
}
```



8.1 장바구니 담기

8.1.7. 장바구니 담기 구현하기

- 장바구니와 관련된 요청들을 처리하기 위해 **CartController** 클래스를 생성한다.

```
package com.shop.controller;

import com.shop.dto.CartItemDto;
import com.shop.service.CartService;
import lombok.RequiredArgsConstructor;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.stereotype.Controller;
import org.springframework.validation.BindingResult;
import org.springframework.validation.FieldError;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.ResponseBody;
import javax.validation.Valid;
import java.security.Principal;
import java.util.List;

@Controller
@RequiredArgsConstructor
```

```
com.shop.controller
> CartController.java
```




8.1 장바구니 담기

8.1.7. 장바구니 담기 구현하기

- 장바구니와 관련된 요청들을 처리하기 위해 **CartController** 클래스를 생성한다.

```
public class CartController {  
    private final CartService cartService;  
  
    @PostMapping(value = "/cart")  
    public @ResponseBody ResponseEntity order(@RequestBody @Valid CartItemDto cartItemDto, BindingResult bindingResult, Principal  
principal) {  
        if (bindingResult.hasErrors()) {  
            StringBuilder sb = new StringBuilder();  
            List<FieldError> fieldErrors = bindingResult.getFieldErrors();  
  
            for (FieldError fieldError : fieldErrors) {  
                sb.append(fieldError.getDefaultMessage());  
            }  
  
            return new ResponseEntity<String>(sb.toString(), HttpStatus.BAD_REQUEST);  
        }  
  
        String email = principal.getName();  
        Long cartItemId;  
  
        try {  
            cartItemId = cartService.addCart(cartItemDto, email);  
        } catch (Exception e) {  
            return new ResponseEntity<String>(e.getMessage(), HttpStatus.BAD_REQUEST);  
        }  
  
        return new ResponseEntity<Long>(cartItemId, HttpStatus.OK);  
    }  
}
```



8.1 장바구니 담기

8.1.8. 장바구니 담기 구현하기

- 상품을 장바구니에 담는 로직 구현을 끝내고, 해당 로직을 테스트하는 **CartServiceTest** 클래스를 생성한다.

```
package com.shop.service;

import com.shop.constant.ItemSellStatus;
import com.shop.dto.CartItemDto;
import com.shop.entity.CartItem;
import com.shop.entity.Item;
import com.shop.entity.Member;
import com.shop.repository.CartItemRepository;
import com.shop.repository.ItemRepository;
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.test.context.TestPropertySource;
import org.springframework.transaction.annotation.Transactional;

import javax.persistence.EntityNotFoundException;

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

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

```
com.shop.service
> CartServiceTest.java
```



8.1 장바구니 담기

8.1.8. 장바구니 담기 구현하기

- 상품을 장바구니에 담는 로직 구현을 끝내고, 해당 로직을 테스트하는 **CartServiceTest** 클래스를 생성한다.

```
@Autowired
ItemRepository itemRepository;

@Autowired
MemberRepository memberRepository;

@Autowired
CartService cartService;

@Autowired
CartItemRepository cartItemRepository;

public Item saveItem() {
    Item item = new Item();
    item.setItemNm("테스트 상품");
    item.setPrice(10000);
    item.setItemDetail("테스트 상품 상세 설명");
    item.setItemSellStatus(ItemSellStatus.SELL);
    item.setStockNumber(100);
    return itemRepository.save(item);
}

public Member saveMember() {
    Member member = new Member();
    member.setEmail("test@test.com");
    return memberRepository.save(member);
}
```



8.1 장바구니 담기

8.1.8. 장바구니 담기 구현하기

- 상품을 장바구니에 담는 로직 구현을 끝내고, 해당 로직을 테스트하는 **CartServiceTest** 클래스를 생성한다.

```
@Test
@DisplayName("장바구니 담기 테스트")
public void addCart() {
    Item item = saveItem();
    Member member = saveMember();

    CartItemDto cartItemDto = new CartItemDto();
    cartItemDto.setCount(5);
    cartItemDto.setItemId(item.getId());

    Long cartItemId = cartService.addCart(cartItemDto, member.getEmail());
    CartItem cartItem = cartItemRepository.findById(cartItemId)
        .orElseThrow(EntityNotFoundException::new);

    assertEquals(item.getId(), cartItem.getItem().getId());
    assertEquals(cartItemDto.getCount(), cartItem.getCount());
}
}
```

Runs: 1/1 Errors: 0 Failures: 0

✓ CartServiceTest [Runner: JUnit 5] (0.272 s)
 ✓ 장바구니 담기 테스트 (0.271 s)

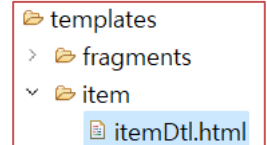


8.1 장바구니 담기

8.1.9. 장바구니 담기 구현하기

- 상품 상세 페이지에서 구현한 장바구니 담기 로직을 호출하는 스크립트 영역에 추가한다. 기존의 order() 함수와 유사하다.

```
function addCart() {  
    var token = $("meta[name='_csrf']").attr("content");  
    var header = $("meta[name='_csrf_header']").attr("content");  
  
    var url = "/cart";  
    var paramData = {  
        itemId : $("#itemId").val(),  
        count : $("#count").val()  
    };  
  
    var param = JSON.stringify(paramData);  
  
    $.ajax({  
        url : url,  
        type : "POST",  
        contentType : "application/json",  
        data : param,  
        beforeSend : function(xhr) {  
            /* 데이터를 전송하기 전에 헤더에 csrf값을 설정 */  
            xhr.setRequestHeader(header, token);  
        },  
        dataType : "json",  
        cache : false,  
        success : function(result, status) {  
            alert("상품을 장바구니에 담았습니다.");  
            location.href = '/';  
        },  
    });  
}
```





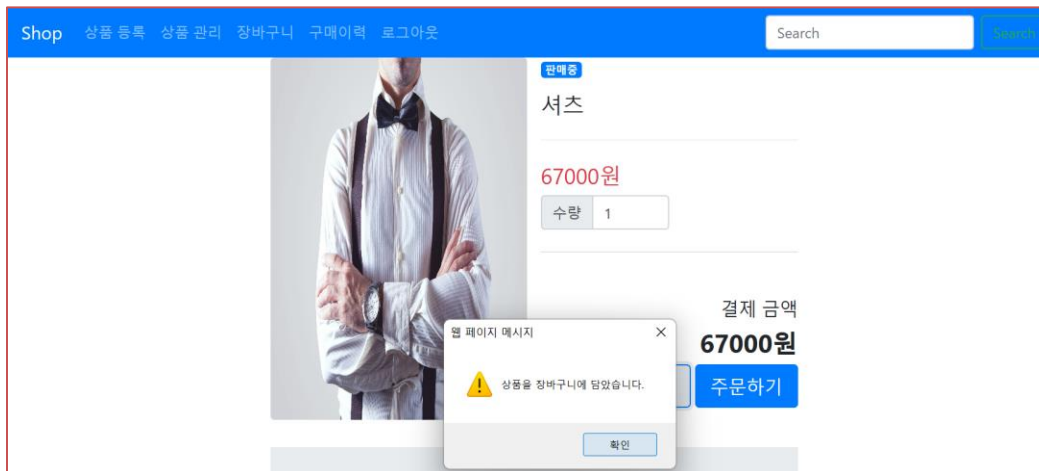
8.1 장바구니 담기

8.1.9. 장바구니 담기 구현하기

- 상품 상세 페이지에서 구현한 장바구니 담기 로직을 호출하는 스크립트 영역에 추가한다. 기존의 order() 함수와 유사하다.

```
error : function(jqXHR, status, error) {  
    if(jqXHR.status == '401'){  
        alert('로그인 후 이용해주세요');  
        location.href='/members/login';  
    } else{  
        alert(jqXHR.responseText);  
    }  
}  
});
```

<button type="button" class="btn btn-light border border-primary btn-lg" onclick="addCart()">장바구니 담기</button>





8.2 장바구니 조회하기

8.2.1. 장바구니 조회하기

- 장바구니 조회 페이지에 전달할 DTO 클래스를 생성한다. JPQL로 쿼리 작성 시 생성자를 이용하여 DTO로 반환한다.

```
package com.shop.dto;

import lombok.Getter;
import lombok.Setter;

@Getter @Setter
public class CartDetailDto {

    private Long cartItemId; //장바구니 상품 아이디

    private String itemNm; //상품명

    private int price; //상품 금액

    private int count; //수량

    private String imgUrl; //상품 이미지 경로

    public CartDetailDto(Long cartItemId, String itemNm, int price, int count, String imgUrl) {
        // 장바구니 페이지에 전달할 데이터를 생성자의 파라미터로 만들어준다.
        this.cartItemId = cartItemId;
        this.itemNm = itemNm;
        this.price = price;
        this.count = count;
        this.imgUrl = imgUrl;
    }
}
```

```
# com.shop.dto
> CartDetailDto.java
```



8.2 장바구니 조회하기

8.2.2. 장바구니 조회하기

- 장바구니 조회 페이지에 전달할 CartDetailDto 리스트를 쿼리 하나로 조회하는 JPQL문을 작성한다. 연관 관계 매핑을 지연 로딩으로 설정할 경우 엔티티에 매핑된 다른 엔티티를 조회할 때 추가적으로 쿼리문이 실행된다. 따라서 성능 최적화가 필요할 경우 **DTO 생성자**를 이용하여 반환값으로 DTO객체를 생성할 수 있다.

```
package com.shop.repository;
```

```
import com.shop.dto.CartDetailDto;
import com.shop.entity.CartItem;
import java.util.List;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
```

```
public interface CartItemRepository extends JpaRepository<CartItem, Long> {
```

```
    CartItem findByCartIdAndItemId(Long cartId, Long itemId);
```

```
    @Query("select new com.shop.dto.CartDetailDto(ci.id, i.itemNm, i.price, ci.count, im.imgUrl) " +
        "from CartItem ci, ItemImg im " +
        "join ci.item i " +
        "where ci.cart.id = :cartId " +
        "and im.item.id = ci.item.id " +
        "and im.repimgYn = 'Y' " +
        "order by ci.regTime desc"
    )
```

```
    List<CartDetailDto> findCartDetailDtoList(Long cartId);
```

```
}
```

```
com.shop.repository
> CartItemRepository.java
```

생성자의 파라미터 순서는 DTO 클래스에 명시한 순서로 넣어줘야 한다.

장바구니에 담겨 있는 상품의 대표 이미지만 가지고 오도록 조건문을 작성한다.



8.2 장바구니 조회하기

8.2.3. 장바구니 조회하기

- 현재 로그인한 회원정보를 이용하여 장바구니에 들어있는 상품을 조회하는 로직을 작성한다.

```
import com.shop.dto.CartDetailDto;
import java.util.ArrayList;
import java.util.List;
```

```
com.shop.service
> CartService.java
```

```
@Transactional(readOnly = true)
public List<CartDetailDto> getCartList(String email) {

    List<CartDetailDto> cartDetailDtoList = new ArrayList<>();

    Member member = memberRepository.findByEmail(email);
    Cart cart = cartRepository.findByMemberId(member.getId()); // 현재 로그인한 회원의 장바구니 엔티티를 조회한다.
    if(cart == null) { // 장바구니에 한 번도 상품을 안 담았을 경우 장바구니 엔티티가 없으므로 빈 리스트를 반환한다.
        return cartDetailDtoList;
    }

    cartDetailDtoList = cartItemRepository.findCartDetailDtoList(cart.getId()); // 장바구니에 담겨 있는 상품 정보를 조회한다.
    return cartDetailDtoList;
}
```



8.2 장바구니 조회하기

8.2.4. 장바구니 조회하기

- 장바구니 페이지로 이동할 수 있도록 **CartController** 클래스에 메소드를 추가한다.

```
import com.shop.dto.CartDetailDto;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;
```

```
com.shop.controller
> CartController.java
```

```
@GetMapping(value = "/cart")
public String orderHist(Principal principal, Model model) {
    List<CartDetailDto> cartDetailList = cartService.getCartList(principal.getName());
    // 현재 로그인한 사용자의 이메일 정보를 이용하여 장바구니에 담겨 있는 상품 정보를 조회한다.
    model.addAttribute("cartItems", cartDetailList);
    // 조회한 장바구니 상품 정보를 뷰로 전달한다.
    return "cart/cartList";
}
```



8.2 장바구니 조회하기

8.2.5. 장바구니 조회하기

- 조회한 장바구니 상품 정보를 이용해 장바구니 목록을 보여주는 페이지 **cartList.html**을 구현한다.

화면에서 구현해야 하는 이벤트가 많아서 자바스크립트 함수가 많다

- . 장바구니 상품 선택 시 총 주문 금액 계산
- . 버튼 클릭 시 장바구니에 담긴 상품 삭제
- . 장바구니 상품 수량 변경 시 상품 금액 계산
- . 장바구니 상품 수량 변경 시 장바구니에 담긴 상품 수량 업데이트
- . 장바구니 상품 주문하기
- . 장바구니 상품 전체 선택

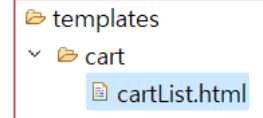
```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org"
      xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout"
      layout:decorate="~{layouts/layout1}">

<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() {
      $("input[name=cartChkBox]").change( function() { // 주문할 상품을 체크하거나 해제할 경우 총 주문 금액을 구하는 함수를 호출한다.
        getOrderTotalPrice();
      });
    });
  </script>
</th:block>
```





8.2 장바구니 조회하기

8.2.5. 장바구니 조회하기

- 조회한 장바구니 상품 정보를 이용해 장바구니 목록을 보여주는 페이지 **cartList.html**을 구현한다.

```
function getOrderTotalPrice() { // 총 주문 금액을 구하는 함수
    var orderTotalPrice = 0;
    $("input[name=cartChkBox]:checked").each(function() { // 현재 체크된 장바구니 상품들의 가격과 수량을 곱해서 총 주문 금액을 계산한다.
        var cartItemId = $(this).val();
        var price = $("#price_" + cartItemId).attr("data-price");
        var count = $("#count_" + cartItemId).val();
        orderTotalPrice += price*count;
    });

    $("#orderTotalPrice").html(orderTotalPrice+'원');
}

function changeCount(obj) { // 장바구니에 들어있는 상품의 수량 변경 시 상품의 가격과 상품의 수량을 곱해서 상품 금액을 변경해준다. 변경된 총 주문 금액을 계산
    // 하기 위해서 마지막에 getOrderTotalPrice() 함수를 호출한다.
    var count = obj.value;
    var cartItemId = obj.id.split('_')[1];
    var price = $("#price_" + cartItemId).data("price");
    var totalPrice = count*price;
    $("#totalPrice_" + cartItemId).html(totalPrice+"원");
    getOrderTotalPrice();
    updateCartItemCount(cartItemId, count);
}

function checkAll() { // 장바구니에 들어있는 전체 상품을 체크하거나 체크 해제하는 함수이다. 변경된 총 주문 금액을 계산하기 위해서 마지막에
    if($("#checkall").prop("checked")) { // getOrderTotalPrice() 함수를 호출한다.
        $("input[name=cartChkBox]").prop("checked",true);
    } else {
        $("input[name=cartChkBox]").prop("checked",false);
    }
    getOrderTotalPrice();
}
```



8.2 장바구니 조회하기

8.2.5. 장바구니 조회하기

- 조회한 장바구니 상품 정보를 이용해 장바구니 목록을 보여주는 페이지 [cartList.html](#)을 구현한다.

```
<!-- 사용자 CSS 추가 -->
<th:block layout:fragment="css">
  <style>
    .content-mg{
      margin-left:25%;
      margin-right:25%;
      margin-top:2%;
      margin-bottom:100px;
    }
    .replmgDiv{
      margin-right:15px;
      margin-left:15px;
      height:auto;
    }
    .replmg{
      height:100px;
      width:100px;
    }
    .fs18{
      font-size:18px
    }
    .fs24{
      font-size:24px
    }
  </style>
</th:block>

<div layout:fragment="content" class="content-mg">

  <h2 class="mb-4">
    장바구니 목록
  </h2>
```



8.2 장바구니 조회하기

8.2.5. 장바구니 조회하기

- 조회한 장바구니 상품 정보를 이용해 장바구니 목록을 보여주는 페이지 [cartList.html](#)을 구현한다.

```
<div>
  <table class="table">
    <colgroup>
      <col width="15%" />
      <col width="70%" />
      <col width="15%" />
    </colgroup>
    <thead>
      <tr class="text-center">
        <td>
          <input type="checkbox" id="checkall" onclick="checkAll()" /> 전체선택
        </td>
        <td>상품정보</td>
        <td>상품금액</td>
      </tr>
    </thead>

    <tbody>
      <tr th:each="cartItem : ${cartItems}">
        <td class="text-center align-middle">
          <input type="checkbox" name="cartChkBox" th:value="${cartItem.cartItemId}" />
        </td>
        <td class="d-flex">
          <div class="replmgDiv align-self-center">
            
          </div>
          <div class="align-self-center">
            <span th:text="${cartItem.itemNm}" class="fs24 font-weight-bold"></span>
            <div class="fs18 font-weight-light">
              <span class="input-group mt-2">
                <span th:id="price_" + ${cartItem.cartItemId} th:data-price="${cartItem.price}" th:text="${cartItem.price} + '원'" class="align-self-center mr-2">

```



8.2 장바구니 조회하기

8.2.5. 장바구니 조회하기

- 조회한 장바구니 상품 정보를 이용해 장바구니 목록을 보여주는 페이지 **cartList.html**을 구현한다.

```
<input type="number" name="count" th:id="count_" + ${cartItem.cartItemId}"
    th:value="${cartItem.count}" min="1"
    onchange="changeCount(this)" class="form-control mr-2" >
<button type="button" class="close" aria-label="Close">
    <span aria-hidden="true" th:data-id="${cartItem.cartItemId}" >&times;</span>
</button>
</span>
</div>
</div>
</td>
<td class="text-center align-middle">
    <span th:id="totalPrice_" + ${cartItem.cartItemId}"
        name="totalPrice" th:text="${cartItem.price * cartItem.count} + '원'">
    </span>
</td>
</tr>
</tbody>
</table>

<h2 class="text-center">
    총 주문 금액 : <span id="orderTotalPrice" class="text-danger">0원</span>
</h2>

<div class="text-center mt-3">
    <button type="button" class="btn btn-primary btn-lg">주문하기</button>
</div>
</div>
</html>
```



8.2 장바구니 조회하기

8.2.6. 장바구니 조회하기

- 장바구니에서 상품의 수량을 변경할 경우 실시간으로 해당 회원의 장바구니 상품의 수량도 변경하도록 로직을 추가한다.
- CartService 클래스에 장바구니 상품의 수량을 업데이트하는 로직을 추가한다. 자바스크립트에서 업데이트할 장바구니 상품번호는 조작이 가능하므로 현재 로그인한 회원과 해당 장바구니 상품을 저장한 회원이 같은지 검사하는 로직도 작성한다.

```
public void updateCount(int count) {  
    this.count = count;  
}
```

```
com.shop.entity  
> BaseEntity.java  
> BaseTimeEntity.java  
> Cart.java  
> CartItem.java
```

```
import org.thymeleaf.util.StringUtils;
```

```
@Transactional(readOnly = true)  
public boolean validateCartItem(Long cartItemId, String email) {  
    Member curMember = memberRepository.findByEmail(email); // 현재 로그인 상품을 저장한 회원을 조회한다.  
    CartItem cartItem = cartItemRepository.findById(cartItemId)  
        .orElseThrow(EntityNotFoundException::new);  
    Member savedMember = cartItem.getCart().getMember(); // 장바구니 상품을 저장한 회원을 조회한다.  
  
    if(!StringUtils.equals(curMember.getEmail(), savedMember.getEmail())){ // 현재 로그인한 회원과 장바구니 상품을 저장한 회원이 다를 경우  
        return false; // false를 같으면 true를 반환한다.  
    }  
  
    return true;  
}  
  
public void updateCartItemCount(Long cartItemId, int count) { // 장바구니 상품의 수량을 업데이트하는 메소드  
    CartItem cartItem = cartItemRepository.findById(cartItemId)  
        .orElseThrow(EntityNotFoundException::new);  
  
    cartItem.updateCount(count);  
}
```

```
com.shop.service  
> CartService.java
```




8.2 장바구니 조회하기

8.2.7. 장바구니 조회하기

- **CartController** 클래스에 장바구니 상품의 수량을 업데이트하는 요청을 처리할 수 있도록 로직을 추가한다.

```
import org.springframework.web.bind.annotation.PatchMapping;  
import org.springframework.web.bind.annotation.PathVariable;
```

```
com.shop.controller  
> CartController.java
```

```
@PatchMapping(value = "/cartItem/{cartItemId}") // http 메소드에서 요청된 자원 일부를 업데이트할 땐 @PatchMapping을 사용한다.  
public @ResponseBody ResponseEntity updateCartItem(@PathVariable("cartItemId") Long cartItemId, int count, Principal principal) {  
  
    if(count <= 0) { // 장바구니에 담겨있는 상품의 개수를 0개 이하로 업데이트 요청할 때 에러 메시지를 담아서 반환한다.  
        return new ResponseEntity<String>("최소 1개 이상 담아주세요", HttpStatus.BAD_REQUEST);  
    } else if(!cartService.validateCartItem(cartItemId, principal.getName())) { // 수정 권한을 체크한다.  
        return new ResponseEntity<String>("수정 권한이 없습니다.", HttpStatus.FORBIDDEN);  
    }  
  
    cartService.updateCartItemCount(cartItemId, count); // 장바구니 상품의 개수를 업데이트한다.  
    return new ResponseEntity<Long>(cartItemId, HttpStatus.OK);  
}
```



8.2 장바구니 조회하기

8.2.8. 장바구니 조회하기

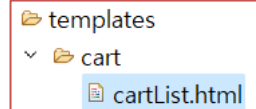
- **cartList.html** 파일에서 장바구니 상품의 수량을 수정할 경우 업데이트 요청을 하도록 자바스크립트 함수를 추가한다.

```
function updateCartItemCount(cartItemId, count) {
    var token = $("meta[name='_csrf']").attr("content");
    var header = $("meta[name='_csrf_header']").attr("content");

    var url = "/cartItem/" + cartItemId + "?count=" + count;

    $.ajax({
        url : url,
        type : "PATCH", // 부분 업데이트이므로 PATCH TYPE으로 설정한다.
        beforeSend : function(xhr) {
            /* 데이터를 전송하기 전에 헤더에 csrf값을 설정 */
            xhr.setRequestHeader(header, token);
        },
        dataType : "json",
        cache : false,
        success : function(result, status) {
            console.log("cartItem count update success");
        },
        error : function(jqXHR, status, error) {

            if(jqXHR.status == '401'){
                alert('로그인 후 이용해주세요');
                location.href='/members/login';
            } else{
                alert(jqXHR.responseJSON.message);
            }
        }
    });
}
```





8.2 장바구니 조회하기

8.2.9. 장바구니 조회하기

■ 실행하기

Shop

상품 목록

상품 관리

장바구니


구매이력

로그아웃

Search

장바구니 목록

☐



셔츠5


80000원

2

x

160000원

☐



스웨터


67800원

1

x

67800원

☐



셔츠2

56000원

2

x

112000원

☐



셔츠

12300원

1

x

12300원

총 주문 금액: 0원

주문하기

Shop

상품 목록

상품 관리

장바구니

구매이력

로그아웃

Search

장바구니 목록

☒



셔츠5


80000원

2

x

160000원

☒



스웨터

67800원

1

x

67800원

☒



셔츠2

56000원

2

x

112000원

☒



셔츠

12300원

1

x

12300원

총 주문 금액: 352100원

주문하기



8.2 장바구니 조회하기

8.2.10. 장바구니 삭제하기

- **CartService**에 장바구니 상품 번호를 파라미터로 받아서 삭제하는 로직을 추가한다.
- 상품정보에 [x] 삭제 버튼을 클릭할 때 장바구니에 넣은 상품을 삭제한다.
- **CartController**에 장바구니 상품을 삭제하는 요청을 처리할 수 있도록 로직을 추가한다.

```
public void deleteCartItem(Long cartItemId) {  
    CartItem cartItem = cartItemRepository.findById(cartItemId)  
        .orElseThrow(EntityNotFoundException::new);  
    cartItemRepository.delete(cartItem);  
}
```

```
com.shop.service  
> CartService.java
```

```
import org.springframework.web.bind.annotation.DeleteMapping;
```

```
@DeleteMapping(value = "/cartItem/{cartItemId}") // http메소드에서 DELETE 경우 자원을 삭제하는 어노테이션  
public @ResponseBody ResponseEntity deleteCartItem(@PathVariable("cartItemId") Long cartItemId, Principal principal) {  
  
    if(!cartService.validateCartItem(cartItemId, principal.getName())){ // 수정 권한 체크  
        return new ResponseEntity<String>("수정 권한이 없습니다.", HttpStatus.FORBIDDEN);  
    }  
  
    cartService.deleteCartItem(cartItemId); // 해당 장바구니 상품을 삭제한다.  
  
    return new ResponseEntity<Long>(cartItemId, HttpStatus.OK);  
}
```

```
com.shop.controller  
> CartController.java
```

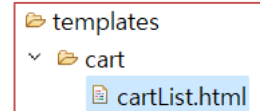


8.2 장바구니 조회하기

8.2.11. 장바구니 삭제하기

- **cartList.html** 파일에서 [x] 버튼을 클릭하면 해당 상품을 삭제하는 로직을 작성한다.

```
function deleteCartItem(obj) {  
    var cartItemId = obj.dataset.id;  
    var token = $("meta[name='_csrf']").attr("content");  
    var header = $("meta[name='_csrf_header']").attr("content");  
  
    var url = "/cartItem/" + cartItemId;  
  
    $.ajax({  
        url : url,  
        type : "DELETE", // 장바구니 상품을 삭제하는 경우이므로 DELETE TYPE을 사용한다.  
        beforeSend : function(xhr) {  
            /* 데이터를 전송하기 전에 헤더에 csrf값을 설정 */  
            xhr.setRequestHeader(header, token);  
        },  
        dataType : "json",  
        cache : false,  
        success : function(result, status) {  
            location.href = '/cart'; // 삭제 요청이 정상적으로 처리되면 장바구니 페이지를 새로고침한다.  
        },  
        error : function(jqXHR, status, error) {  
  
            if(jqXHR.status == '401'){  
                alert('로그인 후 이용해주세요');  
                location.href = '/members/login';  
            } else {  
                alert(jqXHR.responseJSON.message);  
            }  
  
        }  
    });  
}
```





8.2 장바구니 조회하기




8.2.12. 장바구니 삭제하기

- **cartList.html** 파일에서 [x] 버튼을 클릭하면 해당 상품을 삭제하는 로직을 작성한다.

```
<button type="button" class="close" aria-label="Close">
  <span aria-hidden="true" th:data-id="{cartItem.cartItemId}" onclick="deleteCartItem(this)">&times;</span>
</button>
```

[Shop](#) [상품 등록](#) [상품 관리](#) [장바구니](#) [구매이력](#) [로그아웃](#)

장바구니 목록

<input checked="" type="checkbox"/> 전체선택	상품정보	상품금액
<input checked="" type="checkbox"/>	<div> 스웨터 67800원 <input type="text" value="1"/> x</div>	67800원
<input checked="" type="checkbox"/>	<div> 셔츠2 56000원 <input type="text" value="2"/> x</div>	112000원
<input checked="" type="checkbox"/>	<div> 셔츠 12300원 <input type="text" value="1"/> x</div>	12300원

총 주문 금액 : **192100원**

주문하기



8.3 장바구니 상품 주문하기

8.3.1. 장바구니 상품 주문하기

- 장바구니 목록 중 체크박스가 선택된 상품을 주문하는 로직을 작성한다.

장바구니에 주문을 하면 기존 주문 로직과 차이점은 여러 개의 상품을 하나의 주문에 담을 수 있다는 점과 주문한 상품은 장바구니에서 삭제해야 한다는 점이다.

- 장바구니 페이지에서 주문할 상품 데이터를 전달할 DTO를 생성한다.

```
package com.shop.dto;
```

```
import lombok.Getter;  
import lombok.Setter;  
import java.util.List;
```

```
@Getter
```

```
@Setter
```

```
public class CartOrderDto {
```

```
    private Long cartItemId;
```

```
    private List<CartOrderDto> cartOrderDtoList;
```

```
    // 장바구니에서 여러 개의 상품을 주문하므로 CartOrderDto 클래스가 자기 자신을 List로 가지고 있도록 만든다.
```

```
}
```

```
com.shop.dto  
> CartDetailDto.java  
> CartItemDto.java  
> CartOrderDto.java
```



8.3 장바구니 상품 주문하기

8.3.2. 장바구니 상품 주문하기

- **OrderService** 클래스에 장바구니에서 주문할 상품 데이터를 전달받아서 주문을 생성하는 로직을 만든다.

```
public Long orders(List<OrderDto> orderDtoList, String email) {
```

```
    Member member = memberRepository.findByEmail(email);  
    List<OrderItem> orderItemList = new ArrayList<>();
```

```
    for (OrderDto orderDto : orderDtoList) { // 주문할 상품 리스트를 만들어 준다.  
        Item item = itemRepository.findById(orderDto.getItemId())  
            .orElseThrow(EntityNotFoundException::new);
```

```
        OrderItem orderItem = OrderItem.createOrderItem(item, orderDto.getCount());  
        orderItemList.add(orderItem);  
    }
```

```
    Order order = Order.createOrder(member, orderItemList); // 현재 로그인한 회원과 주문 상품 목록을 이용하여 주문 엔티티를 만든다.  
    orderRepository.save(order); // 주문 데이터를 저장한다.
```

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

```
com.shop.service  
> CartService.java  
> FileService.java  
> ItemImgService.java  
> ItemService.java  
> MemberService.java  
> OrderService.java
```




8.3 장바구니 상품 주문하기

8.3.3. 장바구니 상품 주문하기

- **CartService 클래스**에서는 주문 로직으로 전달할 orderDto 리스트 생성 및 주문 로직 호출, 주문한 상품은 장바구니에 제거하는 로직을 구현한다.

```
import com.shop.dto.CartOrderDto;  
import com.shop.dto.OrderDto;
```

```
com.shop.service  
> CartService.java
```

```
private final OrderService orderService;
```

```
public Long orderCartItem(List<CartOrderDto> cartOrderDtoList, String email) {  
    List<OrderDto> orderDtoList = new ArrayList<>();
```

```
    for (CartOrderDto cartOrderDto : cartOrderDtoList) { // 장바구니 페이지에서 전달받은 주문 상품 번호를 이용하여 주문로직으로 전달할 orderDto객체를 만든다.  
        CartItem cartItem = cartItemRepository  
            .findById(cartOrderDto.getCartItemid())  
            .orElseThrow(EntityNotFoundException::new);
```

```
        OrderDto orderDto = new OrderDto();  
        orderDto.setItemId(cartItem.getItem().getId());  
        orderDto.setCount(cartItem.getCount());  
        orderDtoList.add(orderDto);  
    }
```

```
    Long orderId = orderService.orders(orderDtoList, email); // 장바구니에 담은 상품을 주문하도록 주문 로직을 호출한다.
```

```
    for (CartOrderDto cartOrderDto : cartOrderDtoList) { // 주문한 상품들을 장바구니에서 제거한다.
```

```
        CartItem cartItem = cartItemRepository  
            .findById(cartOrderDto.getCartItemid())  
            .orElseThrow(EntityNotFoundException::new);  
        cartItemRepository.delete(cartItem);  
    }
```

```
    return orderId;
```

```
}
```



8.3 장바구니 상품 주문하기

8.3.4. 장바구니 상품 주문하기

- **CartController 클래스**에서는 주문 로직으로 전달할 orderDto 리스트 생성 및 주문 로직 호출, 주문한 상품은 장바구니에 제거하는 로직을 구현한다.

```
import com.shop.dto.CartOrderDto;
```

```
com.shop.controller  
> CartController.java
```

```
@PostMapping(value = "/cart/orders")  
public @ResponseBody ResponseEntity orderCartItem(@RequestBody CartOrderDto cartOrderDto, Principal principal) {  
  
    List<CartOrderDto> cartOrderDtoList = cartOrderDto.getCartOrderDtoList();  
  
    if(cartOrderDtoList == null || cartOrderDtoList.size() == 0) { // 주문할 상품을 선택하지 않았는지 체크한다.  
        return new ResponseEntity<String>("주문할 상품을 선택해주세요", HttpStatus.FORBIDDEN);  
    }  
  
    for (CartOrderDto cartOrder : cartOrderDtoList) { // 주문 권한을 체크한다.  
        if(!cartService.validateCartItem(cartOrder.getCartItemid(), principal.getName())) { // 주문 로직 호출 결과 생성된 주문번호를 반환 받는다.  
            return new ResponseEntity<String>("주문 권한이 없습니다.", HttpStatus.FORBIDDEN);  
        }  
    }  
  
    Long orderId = cartService.orderCartItem(cartOrderDtoList, principal.getName());  
    return new ResponseEntity<Long>(orderId, HttpStatus.OK); // 생성된 주문 번호와 요청이 성공했다는 http 응답 상태 코드를 반환한다.  
}
```

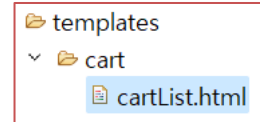


8.3 장바구니 상품 주문하기

8.3.5. 장바구니 상품 주문하기

- **cartList.html** 파일에 장바구니에서 선택한 상품 주문을 요청하도록 자바스크립트 함수를 추가한다.

```
function orders() {  
    var token = $("meta[name='_csrf']").attr("content");  
    var header = $("meta[name='_csrf_header']").attr("content");  
  
    var url = "/cart/orders";  
  
    var dataList = new Array();  
    var paramData = new Object();  
  
    $("input[name=cartChkBox]:checked").each(function() { // 체크된 장바구니 상품 아이디를 전달하기 위해서  
                                                            // dataList 배열에 장바구니 상품 아이디를 객체로 만들어서 저장한다.  
        var cartItemId = $(this).val();  
        var data = new Object();  
        data["cartItemId"] = cartItemId;  
        dataList.push(data);  
    });  
  
    paramData['cartOrderDtoList'] = dataList; // 장바구니 상품 아이디를 저장하고 있는 dataList 배열을 paramData 객체에 추가한다.  
  
    var param = JSON.stringify(paramData);
```





8.3 장바구니 상품 주문하기

8.3.5. 장바구니 상품 주문하기

- **cartList.html** 파일에 장바구니에서 선택한 상품 주문을 요청하도록 자바스크립트 함수를 추가한다.

```
$.ajax({
  url    : url,
  type   : "POST",
  contentType : "application/json",
  data   : param,
  beforeSend : function(xhr) {
    /* 데이터를 전송하기 전에 헤더에 csrf값을 설정 */
    xhr.setRequestHeader(header, token);
  },
  dataType : "json",
  cache    : false,
  success  : function(result, status) {
    alert("주문이 완료 되었습니다.");
    location.href='/orders'; // 주문 요청 결과 성공하였다면 구매이력 페이지로 이동한다.
  },
  error    : function(jqXHR, status, error) {

    if(jqXHR.status == '401'){
      alert('로그인 후 이용해주세요');
      location.href='/members/login';
    } else{
      alert(jqXHR.responseJSON.message);
    }
  }
});
```

<button type="button" class="btn btn-primary btn-lg onclick="orders()">주문하기</button>






8.3 장바구니 상품 주문하기

8.3.6. 장바구니 상품 주문하기

■ 실행하기

Shop 상품 목록 상품 관리 장바구니 구매이력 로그인

장바구니 목록

전체선택	상품정보	상품금액
<input checked="" type="checkbox"/>	 스웨터 67800원 1 x	67800원
<input checked="" type="checkbox"/>	 셔츠2 56000원 2 x	112000원
<input checked="" type="checkbox"/>	 셔츠 12300원 1 x	12300원

총 주문 금액 : 192100원

주문하기

localhost:8090 내용:
주문이 완료 되었습니다.


확인


장바구니 목록


Shop 상품 목록 상품 관리 장바구니 구매이력 로그인

구매 이력


2022-04-02 00:18 주문 주문취소

 스웨터 67800원 1개

 셔츠2 56000원 2개

 셔츠 12300원 1개

2022-04-01 21:52 주문 주문취소


 청바지 70000원 1개


localhost:8090 내용:
주문이 취소 되었습니다.


확인

구매 이력

2022-04-02 00:18 주문 주문취소

 스웨터 67800원 1개

 셔츠2 56000원 2개

 셔츠 12300원 1개

2022-04-01 21:52 주문 주문취소

구매 이력

2022-04-02 00:18 주문 주문취소

 스웨터 67800원 1개

 셔츠2 56000원 2개

 셔츠 12300원 1개

2022-04-01 21:52 주문 (취소 완료)

 청바지 70000원 1개