- **애플리케이션 구현**
  - **- 개발 방법**
  - **- 회원 기능**
    - – 회원 기능 코드
    - – @PersistenceContext
    - – @PersistenceUnit
    - – 회원 기능 테스트
  - **- 상품 기능**
    - – 상품 기능 코드
  - **- 주문 기능**
    - – 주문 기능 코드
    - – 주문 검색 기능
    - – 주문 기능 테스트
  - **- 웹 계층 구현**
    - – 상품 등록
    - – 상품 목록
    - – 상품 수정
    - – 상품 주문

애플리케이션 구현

요구사항을 분석해서 필요한 엔티티와 테이블 설계를 완료했다. 지금부터 실제 애플리케이션이 동작하도록 구현해보자.

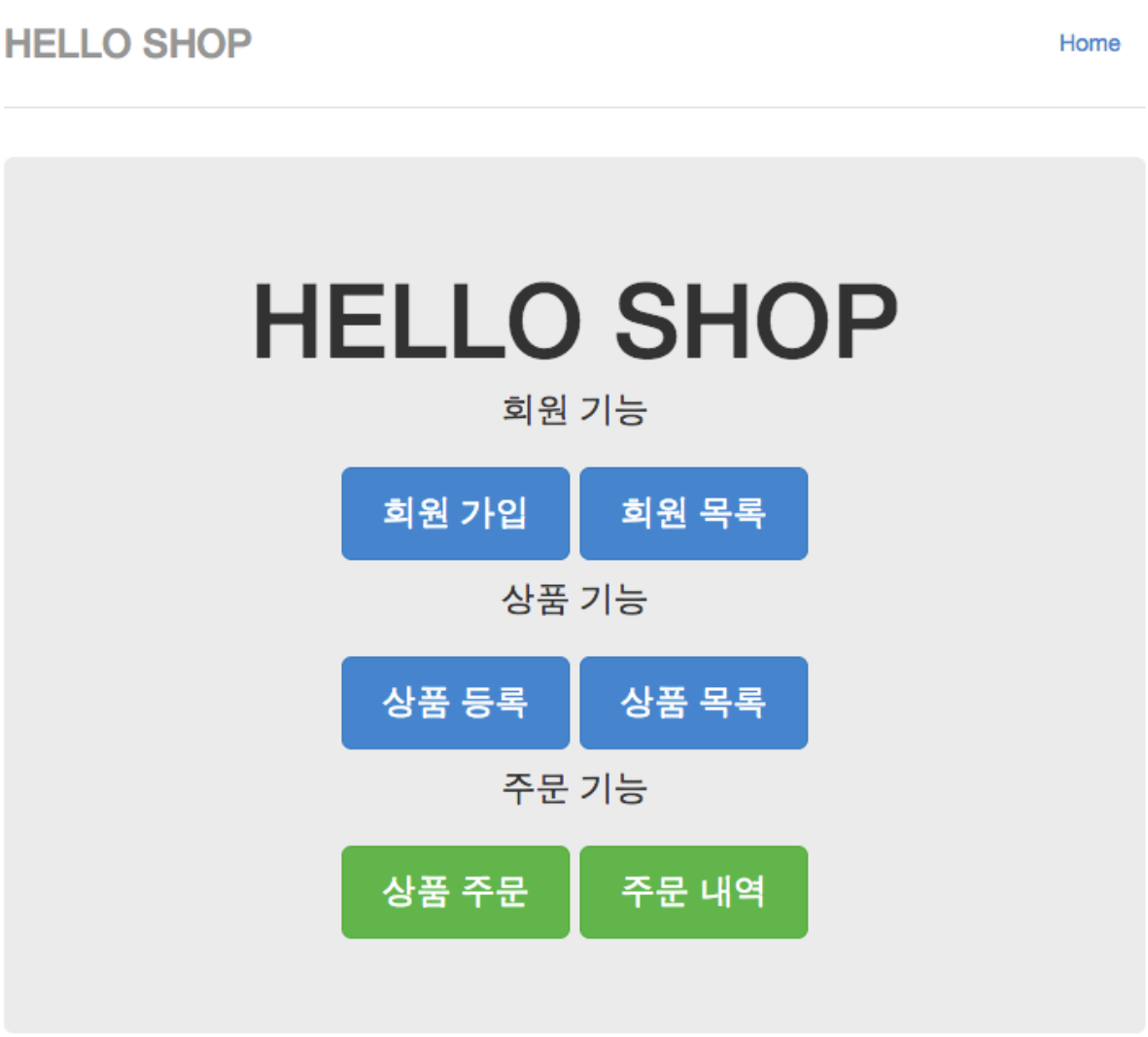

그림 - 메인 화면

구현 순서는 다음과 같다.

- 회원 기능
  - 회원 등록
  - 회원 목록 조회

- 상품 기능
  - 상품 등록
  - 상품 목록 조회
  - 상품 수정

- 주문 기능
  - 상품 주문
  - 주문 내역 조회
  - 주문 취소

이 예제는 스프링과 JPA로 웹 애플리케이션을 개발하는 방법을 설명하는 것이 목적이다. 따라서 비즈니스 로직은 최대한 단순화해서 회원, 상품, 주문의 핵심 기능만 구현하겠다. 예제를 단순화하기 위해 다음 기능은 구현하지 않는다.

- 로그인과 권한 관리는 하지 않는다.
- 파라미터 검증과 예외 처리는 하지 않는다.
- 상품은 도서만 사용한다.
- 카테고리는 사용하지 않는다.
- 배송 정보는 사용하지 않는다.

- 개발 방법

예제는 일반적으로 많이 사용하는 계층형 구조를 사용한다.

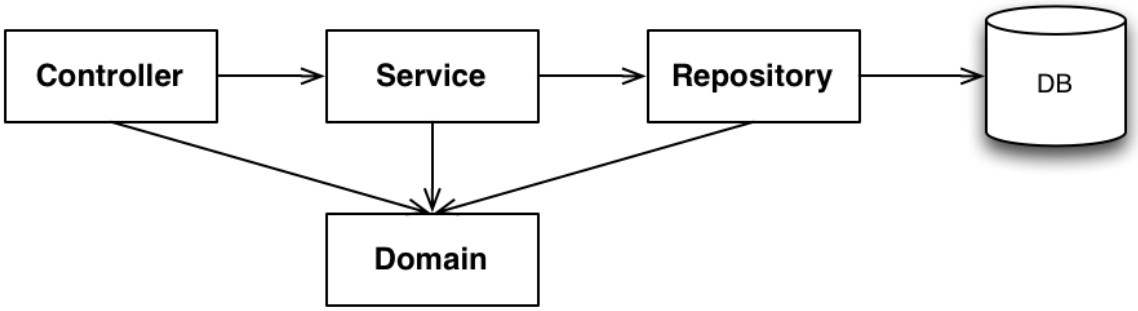

그림 - 계층 의존관계

[그림 - 계층 의존관계]를 보자.

- `Controller` : MVC의 컨트롤러가 모여 있는 곳이다. 컨트롤러는 서비스 계층을 호출하고 결과를 뷰(JSP)에 전달한다.
- `Service` : 서비스 계층에는 비즈니스 로직이 있고 트랜잭션을 시작한다. 서비스 계층은 데이터 접근 계층인 리파지토리를 호출한다.
- `Repository` : JPA를 직접 사용하는 곳은 리파지토리 계층이다. 여기서 엔티티 매니저를 사용해서 엔티티를 저장하고 조회한다.
- `Domain` : 엔티티가 모여있는 계층이다. 모든 계층에서 사용한다.

개발 순서는 비즈니스 로직을 수행하는 서비스와 리파지토리 계층을 먼저 개발하고 테스트 케이스를 작성해서 검증하겠다. 그리고 검증을 완료하면 컨트롤러와 뷰를 개발하는 순서로 진행하겠다. 회원 기능부터 하나씩 개발해보자.

- 회원 기능

구현해야 할 회원 기능 목록은 다음과 같다.

- 회원 등록
- 회원 목록 조회

– 회원 기능 코드

도메인 모델에서 설명한 회원 엔티티 코드를 다시 보자.

====== 회원 엔티티 코드 ======

```
package jpabook.jpashop.domain;

@Entity
public class Member {

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

    private String name;

    @Embedded
    private Address address;

    @OneToMany(mappedBy = "member")
    private List<Order> orders = new ArrayList<Order>();
    ...
}
```

이 회원 엔티티를 저장하고 관리할 리파지토리 코드를 보자.

회원 리파지토리 분석

====== 회원 리파지토리 코드 ======

```
package jpabook.jpashop.repository;

import jpabook.jpashop.domain.Member;
import org.springframework.stereotype.Repository;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
```

```
import java.util.List;

@Repository
public class MemberRepository {

    @PersistenceContext
    EntityManager em;

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

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

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

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

회원 리파지토리 코드를 순서대로 분석해보자.

```
@Repository
public class MemberRepository {...}
```

`@Repository` 어노테이션이 붙어 있으면 `<context:component-scan>` 에 의해 스프링 빈으로 자동 등록된다. 그리고 이 어노테이션에는 한 가지 기능이 더 있는데 JPA 전용 예외가 발생하면 스프링이 추상화한 예외로 변환해준다.

예를 들어 리파지토리 계층에서 JPA 예외인 `javax.persistence.NoResultException` 가 발생하면 스프링이 추상화한 예외인 `org.springframework.dao.EmptyResultDataAccessException` 로 변환해서 서비스 계층에 반환한다. 따라서 서비스 계층은 JPA에 의존적인 예외를 처리하지 않아도 된다. 예외 변환에 대한 자세한 내용은 고급 주제에서 다룬다.

– @PersistenceContext

```
@PersistenceContext //엔티티 매니저 주입
EntityManager em;
```

순수 자바 환경에서는 엔티티 매니저 팩토리에서 엔티티 매니저를 직접 생성해서 사용했지만 스프링이나 J2EE 컨테이너를 사용하면 컨테이너가 엔티티 매니저를 관리하고 제공해준다. 따라서 엔티티 매니저 팩토리에서 엔티티 매니저를 생성하는 것이 아니라 컨테이너가 제공하는 엔티티 매니저를 사용해야 한다. `@PersistenceContext` 는 컨테이너가 관리하는 엔티티 매니저를 주입하라는 어노테이션이다. 이렇게 엔티티 매니저를 컨테이너로부터 주입 받아서 사용해야 컨테이너가 제공하는 트랜잭션 기능과 연계해서 컨테이너의 다양한 기능들을 사용할 수 있다. 자세한 내용은 웹 애플리케이션 심화장에서 알아보겠다.

– @PersistenceUnit

`@PersistenceContext`를 사용해서 컨테이너가 관리하는 엔티티 매니저를 주입 받을 수 있어서 엔티티 매니저 팩토리를 직접 사용할 일은 거의 없겠지만, 엔티티 매니저 팩토리를 주입받으려면 다음처럼 `@PersistenceUnit` 어노테이션을 사용하면 된다.

```
@PersistenceUnit
EntityManagerFactory emf;
```

예제 코드를 이어가자.

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

회원 엔티티를 저장한다. (더 정확히는 영속화한다.)

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

회원 식별자로 회원 엔티티를 조회한다.

```
public List<Member> findByName(String name) {
```

```
        return em.createQuery("select m from Member m where m.name = :name", Member.
                .setParameter("name", name)
                .getResultList();
    }
```

JPQL을 사용해서 이름(`name`)으로 회원 엔티티 리스트를 조회한다.

회원 서비스 분석

====== 회원 서비스 코드 ======

```
package jpabook.jpashop.service;

import jpabook.jpashop.domain.Member;
import jpabook.jpashop.repository.MemberRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
import java.util.List;

@Service
@Transactional
public class MemberService {

    @Autowired
    MemberRepository memberRepository;

    /**
     * 회원가입
     */
    public Long join(Member member) {

        validateDuplicateMember(member); //중복 회원 검증
        memberRepository.save(member);
        return member.getId();
    }

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

    /**
     * 전체 회원 조회
     */
    public List<Member> findMembers() {
```

```
        return memberRepository.findAll();
    }

    public Member fineOne(Long memberId) {
        return memberRepository.fineOne(memberId);
    }
}
```

비즈니스 로직과 트랜잭션을 담당하는 회원 서비스 코드를 순서대로 분석해보자.

```
@Service
@Transactional
public class MemberService {...}
```

- `@Service` : 이 어노테이션이 붙어 있는 클래스는 `<context:component-scan>` 에 의해 스프링 빈으로 등록된다.
- `@Transactional` : 스프링 프레임워크는 이 어노테이션이 붙어 있는 클래스나 메서드에 트랜잭션을 적용한다. 외부에서 이 클래스의 메서드를 호출할 때 트랜잭션을 시작하고 메서드를 종료할 때 트랜잭션을 커밋한다. 만약 예외가 발생하면 트랜잭션을 롤백한다.(더 자세한 내용은 스프링 프레임워크 문서를 참고하자.)

> **주의:** `@Transactional` 은 `RuntimeException` 과 그 자식들인 언체크(Unchecked) 예외만 롤백한다. 만약 체크 예외가 발생해도 롤백하고 싶다면 `@Transactional(rollbackFor = Exception.class)` 처럼 롤백할 예외를 지정해야 한다.

```
@Autowired
MemberRepository memberRepository;
```

`@Autowired` 를 사용하면 스프링 컨테이너가 적절한 스프링 빈을 주입해준다. 여기서는 회원 리파지토리를 주입한다.

회원가입

```
public Long join(Member member) {

    validateDuplicateMember(member); //중복 회원 검증
```

```
        memberRepository.save(member);
        return member.getId();
    }

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

회원가입은 `join()` 메서드를 사용한다. 이 메서드는 먼저 `validateDuplicateMember()` 로 같은 이름을 가진 회원이 있는지 검증하고 검증을 완료하면 회원 리파지토리에 회원 저장을 요청한다. 만약 같은 이름을 가진 회원이 존재해서 검증에 실패하면 “이미 존재하는 회원입니다.” 라는 메시지를 가지는 예외가 발생한다. 회원가입에 성공하면 생성된 회원 식별자를 반환한다.

회원 서비스와 리파지토리 개발을 완료했다. 작성한 코드가 잘 동작하는지 테스트 코드를 작성해서 테스트해보자.

> **참고:** 검증 로직이 있어도 멀티 쓰레드 상황을 고려해서 회원 테이블의 회원명 컬럼에 유니크 제약 조건을 추가하는 것이 안전하다.

– 회원 기능 테스트

개발한 회원 비즈니스 로직이 정상 동작하는지 JUnit으로 테스트를 작성해서 검증해보자. 회원 기능에서 검증해야 할 핵심 비즈니스 로직은 다음과 같다.

- 회원가입을 성공해야 한다.
- 회원가입 할 때 같은 이름이 있으면 예외가 발생해야 한다.

먼저 회원가입 기능이 정상 동작하는지 테스트해보자.

회원가입 테스트

====== 회원가입 테스트 코드 ======

```
package jpabook.jpashop.service;

import jpabook.jpashop.domain.Member;
import jpabook.jpashop.repository.MemberRepository;
import org.junit.Test;
```

```
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
import org.springframework.transaction.annotation.Transactional;
import static org.junit.Assert.*;

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = "classpath:appConfig.xml")
@Transactional
public class MemberServiceTest {

    @Autowired MemberService memberService;
    @Autowired MemberRepository memberRepository;

    @Test
    public void 회원가입() throws Exception {

        //Given
        Member member = new Member();
        member.setName("kim");

        //When
        Long saveId = memberService.join(member);

        //Then
        assertEquals(member, memberRepository.fineOne(saveId));
    }

    @Test(expected = IllegalStateException.class)
    public void 중복_회원_예외() throws Exception {
        ...
    }

}
```

> **참고: Given, When, Then[1]**
> 이 주석들이 특별한 기능을 하는 것은 아니지만 테스트를 이해하기 쉽게 도와준다. Given 절에서 테스트할 상황을 설정하고 When 절에서 테스트 대상을 실행하고 Then 절에서 결과를 검증한다.

먼저 테스트를 스프링 프레임워크와 함께 실행하기 위해 스프링 프레임워크와 JUnit을 통합해야 한다.

스프링 프레임워크와 테스트 통합

```
@RunWith(SpringJUnit4ClassRunner.class)
```

JUnit으로 작성한 테스트 케이스를 스프링 프레임워크와 통합하려면 `@org.junit.runner.RunWith`에 `org.springframework.test.context.junit4.SpringJUnit4ClassRunner.class`를 지정하면 된다. 이렇게 하면 테스트가 스프링 컨테이너에서 실행되므로 스프링 프레임워크가 제공하는 `@Autowired` 같은 기능들을 사용할 수 있다.

```
@ContextConfiguration(locations = "classpath:appConfig.xml")
```

테스트 케이스를 실행할 때 사용할 스프링 설정 정보를 지정한다. 여기서는 설정 정보로 `appConfig.xml`를 지정했다. 참고로 웹과 관련된 정보는 필요하지 않으므로 `webAppConfig.xml`은 지정하지 않았다.

```
@Transactional
```

테스트는 반복해서 실행할 수 있어야 한다. 문제는 회원가입 테스트를 실행하면 데이터베이스에 회원 데이터가 저장된다. 그리고 다시 테스트를 실행하면 이미 저장된 데이터 때문에 테스트가 실패할 수 있다.

`@Transactional` 어노테이션은 보통 비즈니스 로직이 있는 서비스 계층에서 사용한다. 그런데 이 어노테이션을 테스트에서 사용하면 동작 방식이 달라진다. 이때는 각각의 테스트를 실행할 때마다 트랜잭션을 시작하고 **테스트가 끝나면 트랜잭션을 강제로 롤백**한다. 따라서 테스트를 진행하면서 데이터베이스에 저장한 데이터가 테스트가 끝나면 롤백되므로 반복해서 테스트를 진행할 수 있다.

회원가입() 테스트 케이스

회원가입 테스트 코드를 보면 먼저 회원 엔티티를 하나 생성하고 `join()` 메서드로 회원가입을 시도한다. 그리고 실제 회원이 저장되었는지 검증하기 위해 리파지토리에서 회원 id로 회원을 찾아서 저장한 회원과 같은지 `assertEquals`로 검증한다.

중복 회원 예외처리 테스트

이름이 같은 회원은 중복으로 저장되면 안 되고 예외가 발생해야 한다. 이번에는 예외 상황을 검증해보자.

====== 중복 회원 예외처리 테스트 코드 ======

```
@Test(expected = IllegalStateException.class)
public void 중복_회원_예외() throws Exception {

    //Given
    Member member1 = new Member();
    member1.setName("kim");

    Member member2 = new Member();
    member2.setName("kim");

    //When
    memberService.join(member1);
    memberService.join(member2); //예외가 발생해야 한다.

    //Then
    fail("예외가 발생해야 한다.");
}
```

코드를 분석해보자.

```
@Test(expected = IllegalStateException.class)
```

`@Test.expected` 속성에 예외 클래스를 지정하면 테스트의 결과로 지정한 예외가 발생해야 테스트가 성공한다.

이번 테스트는 이름이 중복된 회원이 가입했을 때 예외가 발생하길 기대하는 테스트다. 따라서 테스트의 결과로 `IllegalStateException`이 발생해야 테스트가 성공한다. 테스트 코드를 보면 이름이 kim인 같은 회원이 두 명 가입했다. 이 로직은 두 번째 회원가입(`join()`)시에 회원가입 검증 로직에서 `IllegalStateException`이 발생한다. 따라서 마지막의 `fail()`은 호출되지 않는다. 만약 `fail()`을 호출하거나 `IllegalStateException` 예외가 발생하지 않으면 테스트는 실패한다.

- 상품 기능

구현해야 할 상품 기능 목록은 다음과 같다.

- 상품 등록
- 상품 목록 조회
- 상품 수정

– 상품 기능 코드

상품 엔티티 분석

상품 엔티티를 먼저 살펴보자. 상품 엔티티에는 단순히 접근자(Getter)와 수정자(Setter) 메서드만 있는 것이 아니라 재고 관련 비즈니스 로직을 처리하는 메서드도 있다.

====== 상품 엔티티 코드 ======

```
package jpabook.jpashop.domain.item;

@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "DTYPE")
public abstract class Item {

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

    private String name;        //이름
    private int price;          //가격
    private int stockQuantity;  //재고수량

    @ManyToMany(mappedBy = "items")
    private List<Category> categories = new ArrayList<Category>();

    //==비즈니스 로직==//
    public void addStock(int quantity) {
        this.stockQuantity += quantity;
    }

    public void removeStock(int quantity) {
        int restStock = this.stockQuantity - quantity;
        if (restStock < 0) {
            throw new NotEnoughStockException("need more stock");
        }
        this.stockQuantity = restStock;
    }
    ...
}
```

있는 재고 관리용 비즈니스 로직을 살펴보자.

- `addStock()` 메서드는 파라미터로 넘어온 수만큼 재고를 늘린다. 이 메서드는 재고가 증가하거나 상품 주문을 취소해서 재고를 다시 늘려야 할 때 사용한다.
- `removeStock()` 메서드는 파라미터로 넘어온 수만큼 재고를 줄인다. 만약 재고가 부족하면 예외

가 발생한다. 주로 상품을 주문할 때 사용한다.

상품 리파지토리 분석

상품 리파지토리 코드를 분석해보자.

====== 상품 리파지토리 코드 ======

```
package jpabook.jpashop.repository;

import jpabook.jpashop.domain.item.Item;
import org.springframework.stereotype.Repository;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import java.util.List;

@Repository
public class ItemRepository {

    @PersistenceContext
    EntityManager em;

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

    public Item fineOne(Long id) {
        return em.find(Item.class, id);
    }

    public List<Item> findAll() {
        return em.createQuery("select i from Item i",Item.class).getResultList()
    }
}
```

상품 리파지토리에선 `save()` 메서드를 유심히 봐야 하는데 이 메서드 하나로 저장과 수정(병합)을 다 처리한다. 코드를 보면 식별자 값이 없으면 새로운 엔티티로 판단해서 `persist()`로 영속화하고 만약 식별자 값이 있으면 이미 한번 영속화 되었던 엔티티로 판단해서 `merge()`로 수정(병합)한다. 결국 여기서의 저장(save)이라는 의미는 신규 데이터를 저장하는 것뿐만 아니라 변경된 데이터의 저장이라는 의미도 포함한다. 이렇게 함으로써 이 메서드를 사용하는 클라이언트는 저장과 수정을 구분하지 않아도 되므로 클라이언트의 로직이 단순해진다.

참고로 여기서 사용하는 수정(병합)은 준영속 상태의 엔티티를 수정할 때 사용한다. 영속 상태의 엔티티는 변경 감지(dirty checking)기능이 동작해서 트랜잭션을 커밋할 때 자동으로 수정되므로 별도의 수정 메서드를 호출할 필요가 없고 그런 메서드도 없다. 조금 뒤에 나오는 웹 계층 구현에서 상품을 수정할 때 `save()` 메서드의 `merge()`를 사용하는 예제에서 자세히 설명하겠다.

상품 서비스 분석

상품 서비스는 상품 리파지토리에 위임만 하는 단순한 클래스다.

====== 상품 서비스 코드 ======

```
package jpabook.jpashop.service;

import jpabook.jpashop.domain.item.Item;
import jpabook.jpashop.repository.ItemRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
import java.util.List;

@Service
@Transactional
public class ItemService {

    @Autowired
    ItemRepository itemRepository;

    public void saveItem(Item item) {
        itemRepository.save(item);
    }

    public List<Item> findItems() {
        return itemRepository.findAll();
    }

    public Item fineOne(Long itemId) {
        return itemRepository.fineOne(itemId);
    }
}
```

상품 테스트

상품 테스트는 회원 테스트와 비슷하므로 생략하겠다.

- 주문 기능

구현해야 할 주문 기능 목록은 다음과 같다.

- 상품 주문
- 주문 내역 조회
- 주문 취소

지금까지 살펴본 회원과 상품 기능은 단순한 CRUD가 대부분이지만 주문에는 의미 있는 비즈니스 로직이 있다. 코드를 분석해보자.

– 주문 기능 코드

주문 기능을 분석하려면 주문(`Order`) 엔티티와 주문상품(`OrderItem`) 엔티티를 함께 살펴봐야 한다.

주문 엔티티

====== 주문 엔티티 코드 ======

```
package jpabook.jpashop.domain;

@Entity
@Table(name = "ORDERS")
public class Order {

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

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "MEMBER_ID")
    private Member member;       //주문 회원

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

    @OneToOne(cascade = CascadeType.ALL, fetch = FetchType.LAZY)
    @JoinColumn(name = "DELIVERY_ID")
    private Delivery delivery;  //배송정보

    private Date orderDate;     //주문시간

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

    //==생성 메서드==//
    public static Order createOrder(Member member, Delivery delivery, OrderItem.
        Order order = new Order();
```

```
            order.setMember(member);
            order.setDelivery(delivery);
            for (OrderItem orderItem : orderItems) {
                order.addOrderItem(orderItem);
            }
            order.setStatus(OrderStatus.ORDER);
            order.setOrderDate(new Date());
            return order;
        }

        //==비즈니스 로직==//
        /** 주문 취소 */
        public void cancel() {
            if (delivery.getStatus() == DeliveryStatus.COMP) {
                throw new RuntimeException("이미 배송완료된 상품은 취소가 불가능합니다.");
            }

            this.setStatus(OrderStatus.CANCEL);
            for (OrderItem orderItem : orderItems) {
                orderItem.cancel();
            }
        }

        //==조회 로직==//
        /** 전체 주문 가격 조회 */
        public int getTotalPrice() {
            int totalPrice = 0;
            for (OrderItem orderItem : orderItems) {
                totalPrice += orderItem.getTotalPrice();
            }
            return totalPrice;
        }

        ...
    }
```

주문 엔티티에는 주문을 생성하는 생성 메서드(`createOrder()`), 주문 취소 비즈니스 로직 (`cancel()`), 그리고 전체 주문 가격을 조회하는 조회 로직(`getTotalPrice()`)이 있다.

- **생성 메서드(`createOrder()`)**: 주문 엔티티를 생성할 때 사용한다. 주문 회원, 배송정보, 주문상품의 정보를 받아서 실제 주문 엔티티를 생성한다.
- **주문 취소(`cancel()`)**: 주문 취소시 사용한다. 주문 상태를 취소로 변경하고 주문상품에 주문 취소를 알린다. 만약 이미 배송을 완료한 상품이면 주문을 취소하지 못하도록 예외를 발생시킨다.
- **전체 주문 가격 조회:** 주문 시 사용한 전체 주문 가격을 조회한다. 전체 주문 가격을 알려면 각각의 주문상품 가격을 알아야 한다. 로직을 보면 연관된 주문상품들의 가격을 조회해서 더한 값을 반환한다.

주문상품 엔티티

====== 주문상품 엔티티 코드 ======

```
package jpabook.jpashop.domain;

@Entity
@Table(name = "ORDER_ITEM")
public class OrderItem {

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

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "ITEM_ID")
    private Item item;      //주문 상품

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "ORDER_ID")
    private Order order;    //주문

    private int orderPrice; //주문 가격
    private int count;      //주문 수량

    //==생성 메서드==//
    public static OrderItem createOrderItem(Item item, int orderPrice, int count
        OrderItem orderItem = new OrderItem();
        orderItem.setItem(item);
        orderItem.setOrderPrice(orderPrice);
        orderItem.setCount(count);

        item.removeStock(count);
        return orderItem;
    }

    //==비즈니스 로직==//
    /** 주문 취소 */
    public void cancel() {
        getItem().addStock(count);
    }

    //==조회 로직==//
    /** 주문상품 전체 가격 조회 */
    public int getTotalPrice() {
        return getOrderPrice() * getCount();
    }
    ...
}
```

- **생성 메서드(`createOrderItem()`)**: 주문 상품, 가격, 수량 정보를 사용해서 주문상품 엔티티를 생성한다. 그리고 `item.removeStock(count)` 를 호출해서 주문한 수량만큼 상품의 재고를 줄인다.
- **주문 취소(`cancel()`)**: `getItem().addStock(count)` 를 호출해서 취소한 주문 수량만큼 상품의 재고를 증가시킨다.
- **주문 가격 조회(`getTotalPrice()`)**: 주문 가격에 수량을 곱한 값을 반환한다.

주문 리파지토리

주문 리파지토리에는 주문 엔티티를 저장하고 검색하는 기능이 있다. 마지막의 `findAll(OrderSearch orderSearch)` 메서드는 주문 내역 검색에서 자세히 알아보자.

====== 주문 리파지토리 코드 ======

```
package jpabook.jpashop.repository;

import jpabook.jpashop.domain.Member;
import jpabook.jpashop.domain.Order;
import jpabook.jpashop.domain.OrderSearch;
import org.springframework.stereotype.Repository;
import org.springframework.util.StringUtils;

import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.persistence.TypedQuery;
import javax.persistence.criteria.*;
import java.util.ArrayList;
import java.util.List;

@Repository
public class OrderRepository {

    @PersistenceContext
    EntityManager em;

    public void save(Order order) {
        em.persist(order);
    }

    public Order fineOne(Long id) {
        return em.find(Order.class, id);
    }

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

주문 서비스

주문 서비스는 주문 엔티티와 주문 상품 엔티티의 비즈니스 로직을 활용해서 주문, 주문 취소, 주문 내역 검색 기능을 제공한다.

====== 주문 서비스 코드 ======

```
package jpabook.jpashop.service;

import jpabook.jpashop.domain.Delivery;
import jpabook.jpashop.domain.Member;
import jpabook.jpashop.domain.Order;
import jpabook.jpashop.domain.OrderItem;
import jpabook.jpashop.domain.item.Item;
import jpabook.jpashop.domain.OrderSearch;
import jpabook.jpashop.repository.MemberRepository;
import jpabook.jpashop.repository.OrderRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
import java.util.List;

@Service
@Transactional
public class OrderService {

    @Autowired MemberRepository memberRepository;
    @Autowired OrderRepository orderRepository;
    @Autowired ItemService itemService;

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

        //엔티티 조회
        Member member = memberRepository.fineOne(memberId);
        Item item = itemService.fineOne(itemId);

        //배송정보 생성
        Delivery delivery = new Delivery(member.getAddress());
        //주문상품 생성
        OrderItem orderItem = OrderItem.createOrderItem(item, item.getPrice(), c
        //주문 생성
        Order order = Order.createOrder(member, delivery, orderItem);

        //주문 저장
        orderRepository.save(order);
        return order.getId();
```

```
    }

    /** 주문 취소 */
    public void cancelOrder(Long orderId) {

        //주문 엔티티 조회
        Order order = orderRepository.fineOne(orderId);
        //주문 취소
        order.cancel();
    }

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

예제를 단순화하려고 한 번에 하나의 상품만 주문할 수 있도록 했다.

- **주문(`order()`)**: 주문하는 회원 식별자, 상품 식별자, 주문 수량 정보를 받아서 실제 주문 엔티티를 생성한 후 저장한다.
- **주문 취소(`cancelOrder()`)**: 주문 식별자를 받아서 주문 엔티티를 조회한 후 주문 엔티티에 주문 취소를 요청한다.
- **주문 검색(`findOrders()`)**: `OrderSearch` 라는 검색 조건을 가진 객체로 주문 엔티티를 검색한다. 자세한 내용은 바로 다음에 나오는 주문 검색 기능에서 알아보자.

> **참고:** 주문 서비스의 주문과 주문 취소 메서드를 보면 비즈니스 로직 대부분이 엔티티에 있다. 서비스 계층은 단순히 엔티티에 필요한 요청을 위임하는 역할을 한다. 이처럼 엔티티가 비즈니스 로직을 가지고 객체지향의 특성을 적극 활용하는 것을 도메인 모델 패턴[2]이라 한다. 반대로 엔티티에는 비즈니스 로직이 거의 없고 서비스 계층에서 대부분의 비즈니스 로직을 처리하는 것을 트랜잭션 스크립트 패턴[3]이라 한다.

– 주문 검색 기능

회원 이름과 주문 상태를 검색 조건으로 주문 내역을 검색하는 기능을 살펴보자.

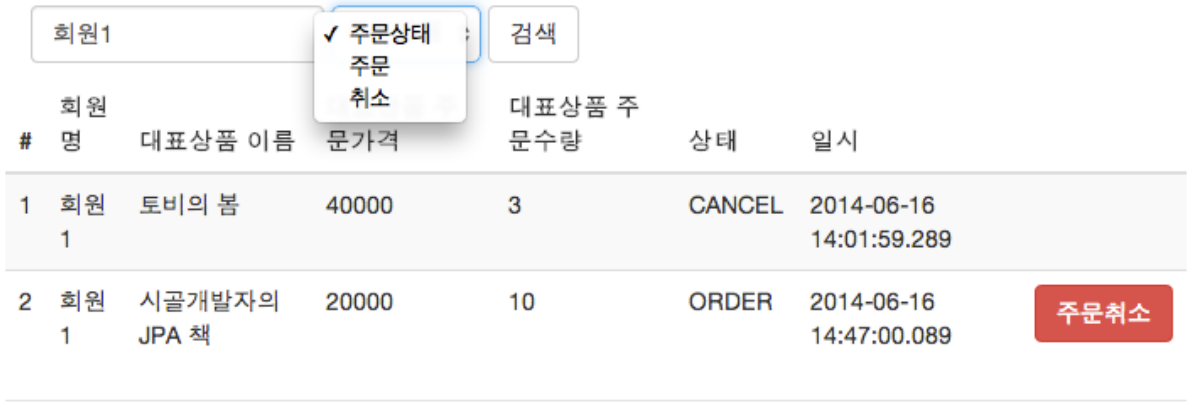

그림 - 주문 내역 검색

그림은 이름이 “회원1”인 회원을 검색하는 화면이다. 결과를 보면 “회원1”만 검색된 것을 확인할 수 있다. 이름 검색 옆에 주문상태(주문,취소)를 선택하면 검색 범위를 더 줄일 수 있다.

이 기능을 구현하기 위해 먼저 검색 조건을 가지는 `OrderSearch` 클래스를 보자.

```
package jpabook.jpashop.domain;

public class OrderSearch {

    private String memberName;       //회원 이름
    private OrderStatus orderStatus;//주문 상태[ORDER, CANCEL]

    //Getter, Setter
    public String getMemberName() {return memberName;}
    public void setMemberName(String memberName) {this.memberName = memberName;}
    public OrderStatus getOrderStatus() {return orderStatus;}
    public void setOrderStatus(OrderStatus orderStatus) {this.orderStatus = orde

}
```

화면에서 회원 이름과 주문 상태를 선택하고 검색 버튼을 클릭하면 `OrderSearch` 객체를 통해 검색 조건이 전달된다. 주문 리파지토리의 `findAll()` 메서드는 이 검색 조건을 활용해서 주문 상품을 검색한다.

====== 주문 리파지토리 코드 ======

```
package jpabook.jpashop.repository;

@Repository
public class OrderRepository {
```

```
    @PersistenceContext
    EntityManager em;

    public void save(Order order) {
        em.persist(order);
    }

    public Order fineOne(Long id) {
        return em.find(Order.class, id);
    }

    public List<Order> findAll(OrderSearch orderSearch) {

        CriteriaBuilder cb = em.getCriteriaBuilder();
        CriteriaQuery<Order> cq = cb.createQuery(Order.class);
        Root<Order> o = cq.from(Order.class);

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

        //주문 상태 검색
        if (orderSearch.getOrderStatus() != null) {
            Predicate status = cb.equal(o.get("status"), orderSearch.getOrderSta
            criteria.add(status);
        }
        //회원 이름 검색
        if (StringUtils.hasText(orderSearch.getMemberName())) {
            Join<Order, Member> m = o.join("member", JoinType.INNER); //회원과 조인
            Predicate name =
              cb.like(m.<String>get("name"), "%" + orderSearch.getMemberName() +
            criteria.add(name);
        }

        cq.where(cb.and(criteria.toArray(new Predicate[criteria.size()])));
        TypedQuery<Order> query = em.createQuery(cq).setMaxResults(1000); //최대
        return query.getResultList();
    }
}
```

주문 내역을 검색하는 `findAll(OrderSearch orderSearch)` 메서드는 검색 조건에 따라 Criteria를 동적으로 생성해서 주문 엔티티를 조회한다.

– 주문 기능 테스트

개발한 주문 기능이 정상 동작하는지 테스트 해보자. 검증해야 할 핵심 로직은 다음과 같다.

- 상품 주문이 성공해야 한다.
- 상품을 주문할 때 재고 수량을 초과하면 안 된다.

- 주문 취소가 성공해야 한다.

상품 주문 테스트

상품 주문 테스트를 작성해보자.

====== 상품 주문 테스트 코드 ======

```
package jpabook.jpashop.service;

import jpabook.jpashop.domain.Address;
import jpabook.jpashop.domain.Member;
import jpabook.jpashop.domain.Order;
import jpabook.jpashop.domain.OrderStatus;
import jpabook.jpashop.domain.item.Book;
import jpabook.jpashop.domain.item.Item;
import jpabook.jpashop.exception.NotEnoughStockException;
import jpabook.jpashop.repository.OrderRepository;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
import org.springframework.transaction.annotation.Transactional;

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

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.fail;

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = "classpath:appConfig.xml")
@Transactional
public class OrderServiceTest {

    @PersistenceContext
    EntityManager em;

    @Autowired OrderService orderService;
    @Autowired OrderRepository orderRepository;

    @Test
    public void 상품주문() throws Exception {

        //Given
        Member member = createMember();
        Item item = createBook("시골 JPA", 10000, 10); //이름, 가격, 재고
        int orderCount = 2;
```

```
        //When
        Long orderId = orderService.order(member.getId(), item.getId(), orderCou

        //Then
        Order getOrder = orderRepository.fineOne(orderId);

        assertEquals("상품 주문시 상태는 ORDER",OrderStatus.ORDER, getOrder.getStatu
        assertEquals("주문한 상품 종류 수가 정확해야 한다.",1, getOrder.getOrderItems().
        assertEquals("주문 가격은 가격 * 수량이다.", 10000 * 2, getOrder.getTotalPrice
        assertEquals("주문 수량만큼 재고가 줄어야 한다.",8, item.getStockQuantity());
    }

    @Test(expected = NotEnoughStockException.class)
    public void 상품주문_재고수량초과() throws Exception {
        //...
    }

    @Test
    public void 주문취소() {
        //...
    }

    private Member createMember() {
        Member member = new Member();
        member.setName("회원1");
        member.setAddress(new Address("서울", "강가", "123-123"));
        em.persist(member);
        return member;
    }

    private Book createBook(String name, int price, int stockQuantity) {
        Book book = new Book();
        book.setName(name);
        book.setStockQuantity(stockQuantity);
        book.setPrice(price);
        em.persist(book);
        return book;
    }
}
```

상품주문이 정상 동작하는지 확인하는 테스트다. Given 절에서 테스트를 위한 회원과 상품을 만들고 When 절에서 실제 상품을 주문하고 Then 절에서 주문 가격이 올바른지, 주문 후 재고 수량이 정확히 줄었는지 검증한다.

재고 수량 초과 테스트

재고 수량을 초과해서 상품을 주문해보자. 이때는 `NotEnoughStockException` 예외가 발생해야 한다.

====== 재고 수량 초과 테스트 코드 ======

```
@Test(expected = NotEnoughStockException.class)
public void 상품주문_재고수량초과() throws Exception {

    //Given
    Member member = createMember();
    Item item = createBook("시골 JPA", 10000, 10); //이름, 가격, 재고

    int orderCount = 11; //재고보다 많은 수량

    //When
    orderService.order(member.getId(), item.getId(), orderCount);

    //Then
    fail("재고 수량 부족 예외가 발생해야 한다.");
}
```

코드를 보면 재고는 10권인데 `orderCount = 11`로 재고보다 1권 더 많은 수량을 주문했다. 주문 초과로 다음 로직에서 예외가 발생한다.

```
public abstract class Item {

    //...

    public void removeStock(int orderQuantity) {
        int restStock = this.stockQuantity - orderQuantity;
        if (restStock < 0) {
            throw new NotEnoughStockException("need more stock");
        }
        this.stockQuantity = restStock;
    }
}
```

주문 취소 테스트

주문 취소 테스트 코드를 작성하자. 주문을 취소하면 그만큼 재고가 증가해야 한다.

====== 주문 취소 테스트 코드 ======

```
@Test
public void 주문취소() {
```

```
        //Given
        Member member = createMember();
        Item item = createBook("시골 JPA", 10000, 10); //이름, 가격, 재고
        int orderCount = 2;

        Long orderId = orderService.order(member.getId(), item.getId(), orderCount);

        //When
        orderService.cancelOrder(orderId);

        //Then
        Order getOrder = orderRepository.fineOne(orderId);

        assertEquals("주문 취소시 상태는 CANCEL 이다.",OrderStatus.CANCEL, getOrder.getSt
        assertEquals("주문이 취소된 상품은 그만큼 재고가 증가해야 한다.", 10, item.getStockQuan
    }
```

주문을 취소하려면 먼저 주문을 해야 한다. Given 절에서 주문하고 When 절에서 해당 주문을 취소했다. Then 절에서 주문상태가 주문 취소 상태인지(`CANCEL`), 취소한 만큼 재고가 증가했는지 검증한다.

정리

이것으로 필요한 비즈니스 로직을 모두 구현하고 테스트를 작성해서 검증도 했다. 이제 이 비즈니스 로직을 활용하는 웹 계층을 구현해보자.

- 웹 계층 구현

웹 계층 구현은 간추려서 상품과 주문 위주로 분석하겠다.

– 상품 등록

웹 화면에서 상품을 어떻게 등록하는지 알아보자. 다음 코드는 상품 컨트롤러 중 상품 등록 시나리오에서 사용하는 부분이다.

```
package jpabook.jpashop.web;

import jpabook.jpashop.domain.item.Book;
import jpabook.jpashop.domain.item.Item;
import jpabook.jpashop.service.ItemService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.PathVariable;
```

```
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import java.util.List;

@Controller
public class ItemController {

    @Autowired ItemService itemService;

    @RequestMapping(value = "/items/new", method = RequestMethod.GET)
    public String createForm() {
        return "items/createItemForm";
    }

    @RequestMapping(value = "/items/new", method = RequestMethod.POST)
    public String create(Book item) {

        itemService.saveItem(item);
        return "redirect:/items";
    }
    ...

}
```

상품 등록 폼

첫 화면에서 상품 등록을 선택하면 `/items/new` URL을 HTTP GET 방식으로 요청한다. 스프링 MVC는 HTTP 요청 정보와 `@RequestMapping` 의 속성 값을 비교해서 실행할 메서드를 찾는다. 따라서 요청 정보와 매핑되는 `createForm()` 메서드를 실행한다.

이 메서드는 단순히 `items/createItemForm` 문자를 반환한다. 스프링MVC의 뷰 리졸버는 이 정보를 바탕으로 실행할 뷰를 찾는다.

====== `webAppConfig.xml` 에 등록된 뷰 리졸버 ======

```
<bean id="viewResolver" class="org.springframework.web.servlet.view.InternalResc
    <property name="viewClass" value="org.springframework.web.servlet.view.JstlV
    <property name="prefix" value="/WEB-INF/jsp/"/>
    <property name="suffix" value=".jsp"/>
</bean>
```

방금 반환한 문자(`items/createItemForm`)와 뷰 리졸버에 등록한 `setPrefix()` , `setSuffix()` 정보를 사용해서 렌더링할 뷰(JSP)를 찾는다.

- 변환 전: `items/createItemForm` -> `{prefix}items/createItemForm{subffix}`
- 변환 후: `items/createItemForm` -> `/WEB-INF/jsp/items/createItemForm.jsp`

마지막으로 해당 위치의 JSP를 실행한 결과 HTML을 클라이언트에 응답한다. 다음 [그림 - 상품 등록 폼]이 그 결과다.

그림 - 상품 등록 폼

상품 등록

상품 등록 폼에서 데이터를 입력하고 Submit 버튼을 클릭하면 `/items/new`를 POST 방식으로 요청한다. 그러면 요청 정보와 매핑되는 상품 컨트롤러의 `create(Book item)` 메서드를 실행한다. 파라미터로 전달한 `item`에는 화면에서 입력한 데이터가 모두 바인딩 되어 있다.(`HttpServletRequest`의 파라미터와 객체의 프로퍼티 이름을 비교해서 같으면 스프링 프레임워크가 값을 바인딩 해준다.)

이 메서드는 상품 서비스에 상품 저장을 요청(`itemService.saveItem(item)`)하고 저장이 끝나면 상품 목록 화면(`redirect:/items`)으로 리다이렉트 한다.

– 상품 목록

화면에서 상품 목록의 URL은 `/items`다.

그림 - 상품 목록 화면

상품 목록을 클릭하면 `ItemController`에 있는 `list()` 메서드를 실행한다.

```
package jpabook.jpashop.web;

@Controller
public class ItemController {

    @Autowired ItemService itemService;

    @RequestMapping(value = "/items", method = RequestMethod.GET)
    public String list(Model model) {

        List<Item> items = itemService.findItems();
        model.addAttribute("items", items);
        return "items/itemList";
    }
    //...
}
```

이 메서드는 `itemService.findItems()`를 호출해서 서비스 계층에서 상품 목록을 조회한다. 그리고 조회한 상품을 뷰에 전달하기 위해 스프링 MVC가 제공하는 모델(`Model`) 객체에 담아둔다. 그리고 마지막으로 실행할 뷰 이름을 반환한다.

====== src/main/webapp/WEB-INF/jsp/items/itemList.jsp ======

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<!DOCTYPE html>
<html>
<jsp:include page="../fragments/head.jsp"/>
<body>
```

```
<div class="container">
    <jsp:include page="../fragments/bodyHeader.jsp" />

    <div>
        <table class="table table-striped">
            <thead>
            <tr>
                <th>#</th>
                <th>상품명</th>
                <th>가격</th>
                <th>재고수량</th>
                <th></th>
            </tr>
            </thead>
            <tbody>
            <c:forEach items="${items}" var="item">
                <tr>
                    <td>${item.id}</td>
                    <td>${item.name}</td>
                    <td>${item.price}</td>
                    <td>${item.stockQuantity}</td>
                    <td>
                        <a href="/items/${item.id}/edit" class="btn btn-primary'
                    </td>
                </tr>
            </c:forEach>
            </tbody>
        </table>
    </div>

    <jsp:include page="../fragments/footer.jsp" />

</div> <!-- /container -->

</body>
</html>
```

`itemList.jsp`를 보면 `model`에 담아둔 상품 목록인 `items`를 꺼내서 상품 정보를 출력한다.

– 상품 수정

이번에는 상품을 수정해보자. 방금 보았던 [그림 - 상품 목록 화면]에서 수정 버튼을 선택하면 상품 수정 화면으로 이동한다.

다음은 상품 컨트롤러 중 상품 수정 부분만 뽑았다.

====== 상품 수정과 관련된 컨트롤러 코드 ======

```
package jpabook.jpashop.web;

@Controller
public class ItemController {

    @Autowired ItemService itemService;

    /** 상품 수정 폼 */
    @RequestMapping(value = "/items/{itemId}/edit", method = RequestMethod.GET)
    public String updateItemForm(@PathVariable("itemId") Long itemId, Model mode

        Item item = itemService.fineOne(itemId);
        model.addAttribute("item", item);
        return "items/updateItemForm";
    }

    /** 상품 수정 */
    @RequestMapping(value = "/items/{itemId}/edit", method = RequestMethod.POST)
    public String updateItem(@ModelAttribute("item") Book item) {

        itemService.saveItem(item);
        return "redirect:/items";
    }
    ...
}
```

수정 버튼을 선택하면 `/items/{itemId}/edit` URL을 GET 방식으로 요청한다. 그 결과로 `updateItemForm()` 메서드를 실행하는데 이 메서드는 `itemService.fineOne(itemId)`를 호출해서 수정할 상품을 조회한 다음 조회 결과를 모델 객체에 담아서 뷰(`items/updateItemForm`)에 전달한다.

그림 - 상품 수정폼

상품 수정 폼 HTML에는 상품의 id(hidden), 상품명, 가격, 수량 정보가 있다.

상품 수정 폼에서 정보를 수정하고 Submit 버튼을 선택하면 `/items/{itemId}/edit` URL을 POST 방식으로 요청하고 `updateItem()` 메서드를 실행한다. 이때 컨트롤러에 파라미터로 넘어온 `item` 엔티티 인스턴스는 현재 준영속 상태다. 따라서 영속성 컨텍스트의 지원을 받을 수 없고 데이터를 수정해도 변경 감지 기능은 동작하지 않는다. 이런 준영속 엔티티를 수정하는 방법은 2가지가 있다.

- 변경 감지 기능 사용
- 병합 사용

변경 감지 기능 사용

변경 감지 기능을 사용하는 방법은 영속성 컨텍스트에서 엔티티를 다시 조회한 후에 데이터를 수정하는 방법이다. 예를 들어 다음과 같은 코드가 있다고 가정하자.

```
@Transactional
void update(Item itemParam) { //itemParam: 파리미터로 넘어온 준영속 상태의 엔티티
    Item findItem = em.find(Item.class, itemParam.getId()); //같은 엔티티를 조회한다.
    findItem.setPrice(itemParam.getPrice()); //데이터를 수정한다.
}
```

이 코드처럼 트랜잭션 안에서 준영속 엔티티의 식별자로 엔티티를 다시 조회하면 영속 상태의 엔티티를 얻을 수 있다. 이렇게 영속 상태인 엔티티의 값을 파라미터로 넘어온 준영속 상태의 엔티티 값으로 변경하면 된다. 이렇게 하면 이후 트랜잭션이 커밋될 때 변경 감지 기능이 동작해서 데이터베이스에 수정사항이 반영된다.

병합 사용

```
@Transactional
void update(Item itemParam) { //itemParam: 파리미터로 넘어온 준영속 상태의 엔티티
    Item mergeItem = em.merge(item);
}
```

병합은 방금 설명한 방식과 거의 비슷하게 동작한다. 파라미터로 넘긴 준영속 엔티티의 식별자 값으로 영속 엔티티를 조회한 다음 영속 엔티티의 값을 준영속 엔티티의 값으로 채워 넣는다. (병합에 대한 자세한 내용은 3장 영속성 관리에서 설명했다.)

변경 감지 기능을 사용하면 원하는 속성만 선택해서 변경할 수 있지만 병합을 사용하면 모든 속성이 변경된다.

다시 컨트롤러의 상품 수정 메서드인 `updateItem()`으로 돌아가보자. 이 메서드는 `itemService.saveItem(item)` 호출해서 준영속 상태인 `item` 엔티티를 상품 서비스에 전달한다. 상품 서비스는 트랜잭션을 시작하고 상품 리파지토리에 저장을 요청한다.

```
package jpabook.jpashop.service;

@Service
@Transactional
public class ItemService {

    @Autowired
    ItemRepository itemRepository;

    public void saveItem(Item item) {
        itemRepository.save(item);
    }
    //...
}
```

다음은 상품 리파지토리의 저장 메서드다. 이 메서드는 식별자가 없으면 새로운 엔티티로 판단해서 영속화(persist)하고 식별자가 있으면 병합(merge)을 수행한다. 지금처럼 준영속 상태인 상품 엔티티를 수정할 때는 `id` 값이 있으므로 병합을 수행한다.

```
package jpabook.jpashop.repository;

@Repository
public class ItemRepository {
```

```
    @PersistenceContext
    EntityManager em;

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

– 상품 주문

마지막으로 상품 주문를 살펴보자.

====== 상품 주문과 관련된 컨트롤러 코드 ======

```
package jpabook.jpashop.web;

import jpabook.jpashop.domain.Member;
import jpabook.jpashop.domain.Order;
import jpabook.jpashop.domain.item.Item;
import jpabook.jpashop.domain.OrderSearch;
import jpabook.jpashop.service.ItemService;
import jpabook.jpashop.service.MemberService;
import jpabook.jpashop.service.OrderService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.*;
import java.util.List;

@Controller
public class OrderController {

    @Autowired OrderService orderService;
    @Autowired MemberService memberService;
    @Autowired ItemService itemService;

    @RequestMapping(value = "/order", method = RequestMethod.GET)
    public String createForm(Model model) {

        List<Member> members = memberService.findMembers();
        List<Item> items = itemService.findItems();
```

```
        model.addAttribute("members", members);
        model.addAttribute("items", items);

        return "order/orderForm";
    }

    @RequestMapping(value = "/order", method = RequestMethod.POST)
    public String order(@RequestParam("memberId") Long memberId, @RequestParam('

        orderService.order(memberId, itemId, count);
        return "redirect:/orders";
    }
    ...
}
```

메인 화면에서 상품 주문를 선택하면 `/order` 를 GET 방식으로 호출해서 컨트롤러의 `createForm()` 메서드를 실행한다. 주문 화면에는 주문할 고객정보와 상품 정보가 필요하므로 `model` 객체에 담아서 뷰에 넘겨준다.

그림 - 상품 주문 화면

상품 주문 화면에서 주문할 회원과 상품 그리고 수량을 선택해서 Submit 버튼을 누르면 `/order` URL을 POST 방식으로 호출해서 컨트롤러의 `order()` 메서드를 실행한다. 이 메서드는 고객 식별자(`memberId`), 주문할 상품 식별자(`itemId`), 수량(`count`) 정보를 받아서 주문 서비스에 주문을 요청한다. 주문이 끝나면 상품 주문 내역이 있는 `/orders` URL로 리다이렉트 한다.

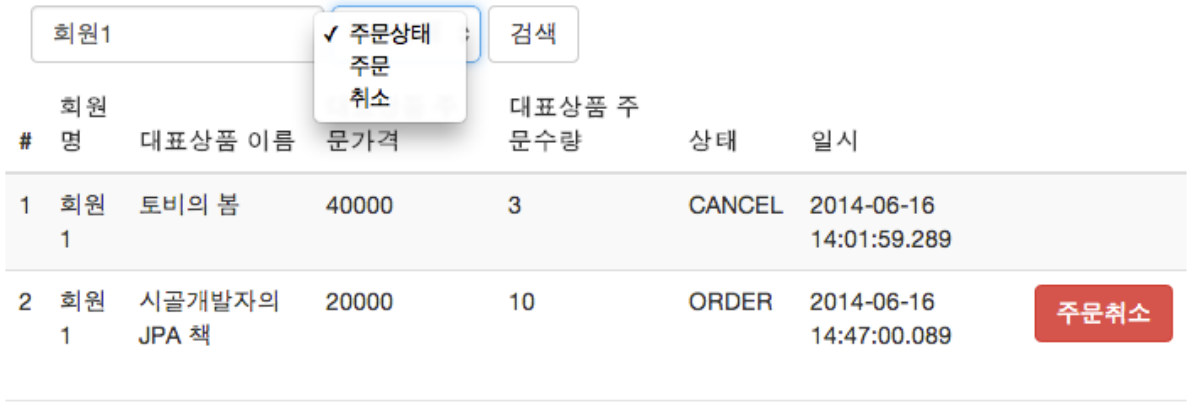

그림 - 주문 내역 화면

정리

지금까지 스프링 프레임워크와 JPA를 사용해서 실제 웹 애플리케이션을 개발해보았다. 부족한 감이 있지만, 스프링과 JPA를 어떻게 활용해야 하는지 감을 잡기에는 충분할 것이다. 그리고 설명하지는 않았지만 뷰(JSP)에서 지연 로딩을 지원하기 위해 OSIV(Open Session In View)를 사용했다. OSIV는 웹 애플리케이션 심화편에서 다룬다.

---

1. http://martinfowler.com/bliki/GivenWhenThen.html ↩
2. http://martinfowler.com/eaaCatalog/domainModel.html ↩
3. http://martinfowler.com/eaaCatalog/transactionScript.html ↩