실전! 스프링 부트와 JPA 활용1 - 웹 애플리케이션 개발v2.1

#인강/jpa활용편/활용편1

인프런 강의: 실전! 스프링 부트와 JPA 활용1 - 웹 애플리케이션 개발
인프런: https://www.inflearn.com

버전 수정 이력

v2.1 - 2020-12-27

스프링 부트 버전 업 2.3.x ➡ 2.4.x
junit4 적용 방법 변경

v2.0 - 2020-11-09

스프링 부트 최신 버전 사용 2.1.x ➡ 2.3.x
자바 버전 8 ➡ 11 사용

v1.13 - 2020-10-29

- OrderService에서 itemService ➡ itemRepository로 수정

v1.12 - 2020-10-15

- ${T(jpabook...) 부분 띄어쓰기 PDF 오류 수정

v1.11 - 2020-10-08

- junit5에서 실행시 테스트 없는 오류 보충 설명

v1.10 - 2020-10-01

- 회원 엔티티 분석 그림: Order ➡ Delivery 단방향을 양방향으로 수정

v1.9 - 2020-09-04

- yml 띄어쓰기 주의 내용 추가

v1.8 - 2020-08-12

- H2 데이터베이스 버전 기본을 1.4.199 ➜ **1.4.200으로 사용**

v1.7 - 2020-06-30

- H2 데이터베이스 버전 1.4.200 ➜ 1.4.199로 버전 지정 내용 추가

v1.6 - 2020-06-11

- `updateItemForm.html`에 isbn, submit 누락 수정
- 도움주신 분: OMG

v1.5 - 2020-05-22

- form-control 복사 붙이기 이슈 추가

v1.4 - 2020-05-08

- 스프링 부트 2.1.x 버전을 사용해주세요.

v1.3 - 2020-01-06

H2 데이터베이스 설치

- MVCC 옵션 제거

v1.2 - 2020-01-03

IntelliJ Gradle 대신에 자바 직접 실행

- 내용 추가

주문서비스개발 코드 보충

- delivery.setStatus(DeliveryStatus.READY); 코드 추가
- 도움주신 분: 강프로그래머

v1.1 - 2019-10-30

스프링 부트 버전 변경

2.1.6 ➜ 2.1.9

H2 데이터베이스 버전 명시

Version 1.4.199을 사용해주세요.

도메인 모델과 테이블 설계.연관관계 매핑 분석

도움주신 분: 열심

- 기존: **주문과 배송:** 일대일 단방향 관계다. `Order.delivery` 를 `ORDERS.DELIVERY_ID` 외래 키와 매핑한다.
- 변경: **주문과 배송:** 일대일 양방향 관계다. `Order.delivery` 를 `ORDERS.DELIVERY_ID` 외래 키와 매핑한다.

목차

- 웹 계층 개발
  - 홈 화면과 레이아웃
  - 회원 등록
  - 회원 목록 조회
  - 상품 등록
  - 상품 목록
  - 상품 수정
  - 변경 감지와 병합(merge)
  - 상품 주문
  - 주문 목록 검색, 취소

프로젝트 환경설정

- 프로젝트 생성
- 라이브러리 살펴보기
- View 환경 설정
- H2 데이터베이스 설치
- JPA와 DB 설정, 동작확인

프로젝트 생성

- 스프링 부트 스타터(https://start.spring.io/)
- 사용 기능: web, thymeleaf, jpa, h2, lombok, validation
  - groupId: jpabook
  - artifactId: jpashop

> **스프링 부트 스타터 설정 필독! 주의!**
> 스프링 부트 버전은 2.4.x 버전을 선택해주세요.
> 자바 버전은 11을 선택해주세요.
> Validation (JSR-303 validation with Hibernate validator) 모듈을 꼭! 추가해주세요.(영상에 없습니다.)

> **필독! 주의!**
> 잘 안되면 다음에 나오는 `build.gradle` 파일을 그대로 복사해서 사용해주세요. 강의 영상과 차이가 있습니다.
>
> - 스프링 부트 버전이 2.1.x ➡ 2.4.x로 업그레이드 되었습니다.

> - validation 모듈이 추가되었습니다. (최신 스프링 부트에서는 직접 추가해야 합니다.)
> - 자바 버전이 1.8 ➡ 11로 업그레이드 되었습니다.

`build.gradle` Gradle 전체 설정

```
plugins {
  id 'org.springframework.boot' version '2.4.1'
  id 'io.spring.dependency-management' version '1.0.10.RELEASE'
  id 'java'
}

group = 'jpabook'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = '11'

configurations {
  compileOnly {
      extendsFrom annotationProcessor
  }
}

repositories {
  mavenCentral()
}

dependencies {
  implementation 'org.springframework.boot:spring-boot-starter-data-jpa'
  implementation 'org.springframework.boot:spring-boot-starter-validation'
  implementation 'org.springframework.boot:spring-boot-starter-thymeleaf'
  implementation 'org.springframework.boot:spring-boot-starter-web'

  compileOnly 'org.projectlombok:lombok'
  runtimeOnly 'com.h2database:h2'

  annotationProcessor 'org.projectlombok:lombok'
  testImplementation 'org.springframework.boot:spring-boot-starter-test'
  //JUnit4 추가
  testImplementation("org.junit.vintage:junit-vintage-engine") {
```

```
        exclude group: "org.hamcrest", module: "hamcrest-core"
    }


}

test {
    useJUnitPlatform()
}
```

필독! 주의!

강의 영상이 JUnit4를 기준으로 하기 때문에 `build.gradle` 에 있는 다음 부분을 꼭 직접 추가해주세요. 해당 부분을 입력하지 않으면 JUnit5로 동작합니다. JUnit5를 잘 알고 선호하시면 입력하지 않아도 됩니다.

```
//JUnit4 추가
testImplementation("org.junit.vintage:junit-vintage-engine") {
    exclude group: "org.hamcrest", module: "hamcrest-core"
}
```

- 동작 확인
  - 기본 테스트 케이스 실행
  - 스프링 부트 메인 실행 후 에러페이지로 간단하게 동작 확인(`http://localhost:8080')

롬복 적용

1. Prefrences ➜ plugin ➜ lombok 검색 실행 (재시작)
2. Prefrences ➜ Annotation Processors 검색 ➜ Enable annotation processing 체크 (재시작)
3. 임의의 테스트 클래스를 만들고 @Getter, @Setter 확인

IntelliJ Gradle 대신에 자바 직접 실행

> 참고: 강의에 이후에 추가된 내용입니다.

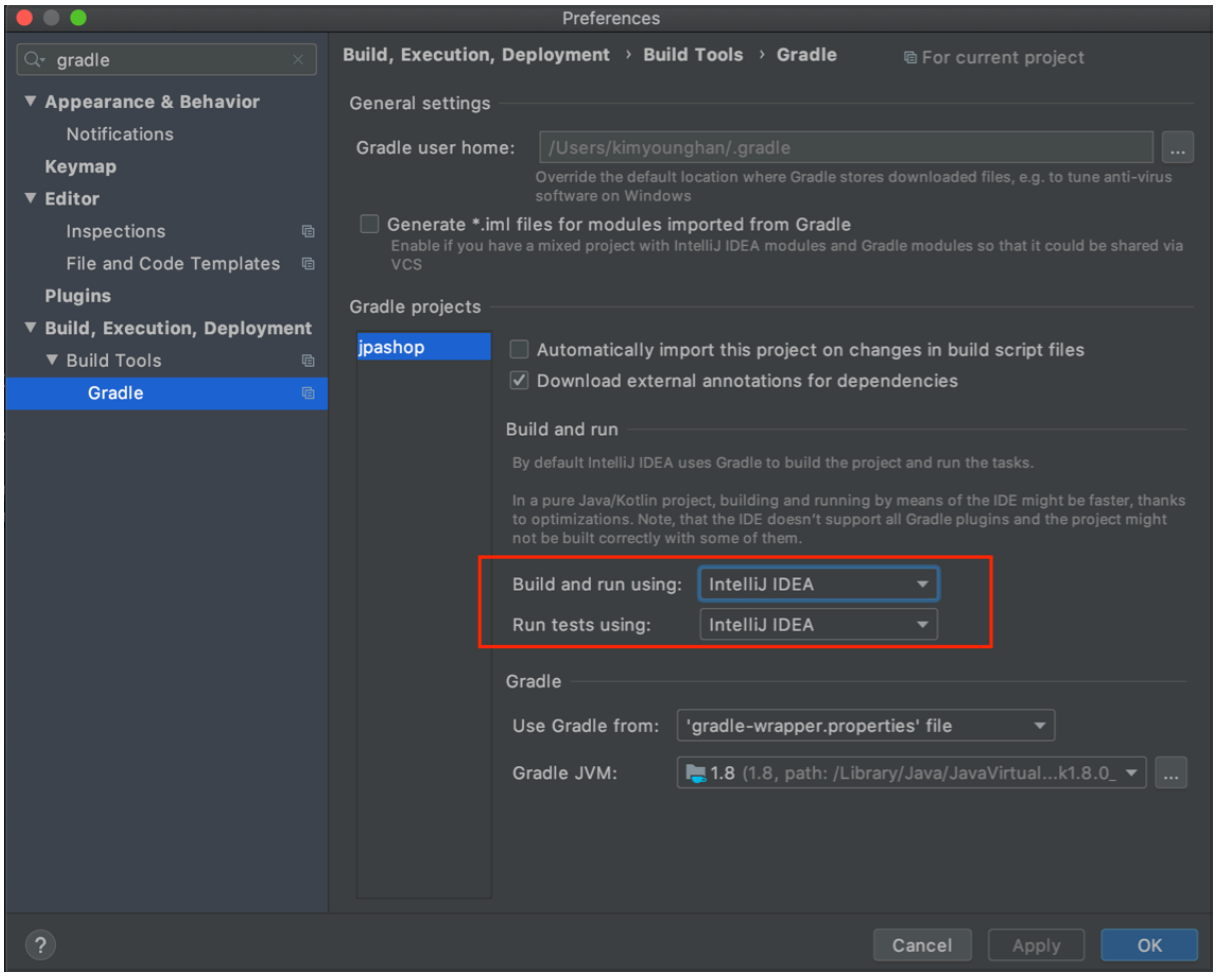
최근 IntelliJ 버전은 Gradle로 실행을 하는 것이 기본 설정이다. 이렇게 하면 실행속도가 느리다. 다음과 같이 변경하면 자바로 바로 실행해서 실행속도가 더 빠르다.

Preferences ➜ Build, Execution, Deployment ➜ Build Tools ➜ Gradle

Build and run using: Gradle ➜ IntelliJ IDEA

Run tests using: Gradle ➜ IntelliJ IDEA

설정 이미지



라이브러리 살펴보기

gradle 의존관계 보기

```
./gradlew dependencies –configuration compileClasspath
```

스프링 부트 라이브러리 살펴보기

- spring-boot-starter-web
  - spring-boot-starter-tomcat: 톰캣 (웹서버)
  - spring-webmvc: 스프링 웹 MVC
- spring-boot-starter-thymeleaf: 타임리프 템플릿 엔진(View)
- spring-boot-starter-data-jpa
  - spring-boot-starter-aop

    - spring-boot-starter-jdbc
        - HikariCP 커넥션 풀 (부트 2.0 기본)
    - hibernate + JPA: 하이버네이트 + JPA
    - spring-data-jpa: 스프링 데이터 JPA
- spring-boot-starter(공통): 스프링 부트 + 스프링 코어 + 로깅
    - spring-boot
        - spring-core
    - spring-boot-starter-logging
        - logback, slf4j

테스트 라이브러리

- spring-boot-starter-test
    - junit: 테스트 프레임워크
    - mockito: 목 라이브러리
    - assertj: 테스트 코드를 좀 더 편하게 작성하게 도와주는 라이브러리
    - spring-test: 스프링 통합 테스트 지원

- 핵심 라이브러리
    - 스프링 MVC
    - 스프링 ORM
    - JPA, 하이버네이트
    - 스프링 데이터 JPA
- 기타 라이브러리
    - H2 데이터베이스 클라이언트
    - 커넥션 풀: 부트 기본은 HikariCP
    - WEB(thymeleaf)
    - 로깅 SLF4J & LogBack
    - 테스트

참고: 스프링 데이터 JPA는 스프링과 JPA를 먼저 이해하고 사용해야 하는 응용기술이다.

View 환경 설정

thymeleaf 템플릿 엔진

- thymeleaf 공식 사이트: https://www.thymeleaf.org/
- 스프링 공식 튜토리얼: https://spring.io/guides/gs/serving-web-content/
- 스프링부트 메뉴얼: https://docs.spring.io/spring-boot/docs/2.1.6.RELEASE/reference/html/boot-features-developing-web-applications.html#boot-features-spring-mvc-template-

engines

- 스프링 부트 thymeleaf viewName 매핑
  - `resources:templates/` +{ViewName}+ `.html`

```
@Controller
public class HelloController {

    @GetMapping("hello")
    public String hello(Model model) {
        model.addAttribute("data", "hello!!");
        return "hello";
    }
}
```

thymeleaf 템플릿엔진 동작 확인(hello.html)

```
<!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <title>Hello</title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
</head>
<body>
<p th:text="'안녕하세요. ' + ${data}" >안녕하세요. 손님</p>
</body>
</html>
```

위치: `resources/templates/hello.html`

- index.html 하나 만들기
  - `static/index.html`

```
<!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org">
```

```
<head>
    <title>Hello</title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
</head>
<body>
Hello
<a href="/hello">hello</a>
</body>
</html>
```

> 참고: `spring-boot-devtools` 라이브러리를 추가하면, `html` 파일을 컴파일만 해주면 서버 재시작 없이 View 파일 변경이 가능하다.
> 인텔리J 컴파일 방법: 메뉴 build ➡ Recompile

H2 데이터베이스 설치

개발이나 테스트 용도로 가볍고 편리한 DB, 웹 화면 제공

> **주의! Version 1.4.200를 사용해주세요.**
> 1.4.200 버전 다운로드 링크

- 윈도우 설치 버전: https://h2database.com/h2-setup-2019-10-14.exe
- 윈도우, 맥, 리눅스 실행 버전: https://h2database.com/h2-2019-10-14.zip

- https://www.h2database.com
- 다운로드 및 설치
- 데이터베이스 파일 생성 방법
  - `jdbc:h2:~/jpashop` (최소 한번)
  - `~/jpashop.mv.db` 파일 생성 확인
  - 이후 부터는 `jdbc:h2:tcp://localhost/~/jpashop` 이렇게 접속

> 주의: H2 데이터베이스의 MVCC 옵션은 H2 1.4.198 버전부터 제거되었습니다. **1.4.200 버전에서는 MVCC 옵션을 사용하면 오류가 발생**합니다.

JPA와 DB 설정, 동작확인

`main/resources/application.yml`

```yaml
spring:
  datasource:
    url: jdbc:h2:tcp://localhost/~/jpashop
    username: sa
    password:
    driver-class-name: org.h2.Driver

  jpa:
    hibernate:
      ddl-auto: create
    properties:
      hibernate:
#        show_sql: true
        format_sql: true

logging.level:
  org.hibernate.SQL: debug
#  org.hibernate.type: trace
```

- spring.jpa.hibernate.ddl-auto: create
  - 이 옵션은 애플리케이션 실행 시점에 테이블을 drop 하고, 다시 생성한다.

> 참고: 모든 로그 출력은 가급적 로거를 통해 남겨야 한다.
> `show_sql` : 옵션은 `System.out` 에 하이버네이트 실행 SQL을 남긴다.
> `org.hibernate.SQL` : 옵션은 logger를 통해 하이버네이트 실행 SQL을 남긴다.

> **주의!** `application.yml` 같은 `yml` 파일은 띄어쓰기(스페이스) 2칸으로 계층을 만듭니다. 따라서 띄어쓰기 2칸을 필수로 적어주어야 합니다.
> 예를 들어서 아래의 `datasource` 는 `spring:` 하위에 있고 앞에 띄어쓰기 2칸이 있으므로 `spring.datasource` 가 됩니다. 다음 코드에 주석으로 띄어쓰기를 적어두었습니다.

yml 띄어쓰기 주의

```yaml
spring: #띄어쓰기 없음
  datasource: #띄어쓰기 2칸
    url: jdbc:h2:tcp://localhost/~/jpashop #4칸
    username: sa
```

```
    password:
    driver-class-name: org.h2.Driver

  jpa: #띄어쓰기 2칸
    hibernate: #띄어쓰기 4칸
      ddl-auto: create #띄어쓰기 6칸
    properties: #띄어쓰기 4칸
      hibernate: #띄어쓰기 6칸
#         show_sql: true    #띄어쓰기 8칸
        format_sql: true  #띄어쓰기 8칸

logging.level: #띄어쓰기 없음
  org.hibernate.SQL: debug #띄어쓰기 2칸
#  org.hibernate.type: trace #띄어쓰기 2칸
```

실제 동작하는지 확인하기

회원 엔티티

```
@Entity
@Getter @Setter
public class Member {

    @Id @GeneratedValue
    private Long id;
    private String username;
    ...
}
```

회원 리포지토리

```
@Repository
public class MemberRepository {

    @PersistenceContext
```

```
    EntityManager em;

    public Long save(Member member) {
        em.persist(member);
        return member.getId();
    }

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

테스트

```
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.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit4.SpringRunner;
import org.springframework.transaction.annotation.Transactional;

import javax.persistence.EntityManager;

@RunWith(SpringRunner.class)
@SpringBootTest
public class MemberRepositoryTest {

    @Autowired MemberRepository memberRepository;

    @Test
    @Transactional
    @Rollback(false)
    public void testMember() {
        Member member = new Member();
        member.setUsername("memberA");
        Long savedId = memberRepository.save(member);
```

```
        Member findMember = memberRepository.find(savedId);

        Assertions.assertThat(findMember.getId()).isEqualTo(member.getId());

Assertions.assertThat(findMember.getUsername()).isEqualTo(member.getUsername())
;

        Assertions.assertThat(findMember).isEqualTo(member); //JPA 엔티티 동일성 보
장
    }
}
```

> **주의!** @Test는 JUnit4를 사용하면 org.junit.Test를 사용하셔야 합니다. 만약 JUnit5 버전을 사용하면 그것에 맞게 사용하시면 됩니다.

- Entity, Repository 동작 확인
- jar 빌드해서 동작 확인

> 오류: 테스트를 실행했는데 다음과 같이 테스트를 찾을 수 없는 오류가 발생하는 경우
> `No tests found for given includes: [jpabook.jpashop.MemberRepositoryTest](filter.includeTestsMatching)`
> 해결: 스프링 부트 2.1.x 버전을 사용하지 않고, 2.2.x 이상 버전을 사용하면 Junit5가 설치된다. 이때는 `build.gradle` 마지막에 다음 내용을 추가하면 테스트를 인식할 수 있다. Junit5 부터는 `build.gradle` 에 다음 내용을 추가해야 테스트가 인식된다.

`build.gradle` 마지막에 추가

```
test {
    useJUnitPlatform()
}
```

> 참고: 스프링 부트를 통해 복잡한 설정이 다 자동화 되었다. `persistence.xml` 도 없고, `LocalContainerEntityManagerFactoryBean` 도 없다. 스프링 부트를 통한 추가 설정은 스프링 부트 메뉴얼을 참고하고, 스프링 부트를 사용하지 않고 순수 스프링과 JPA 설정 방법은 자바 ORM 표준 JPA 프

> 로그래밍 책을 참고하자.

쿼리 파라미터 로그 남기기

- 로그에 다음을 추가하기 `org.hibernate.type` : SQL 실행 파라미터를 로그로 남긴다.
- 외부 라이브러리 사용
  - https://github.com/gavlyukovskiy/spring-boot-data-source-decorator

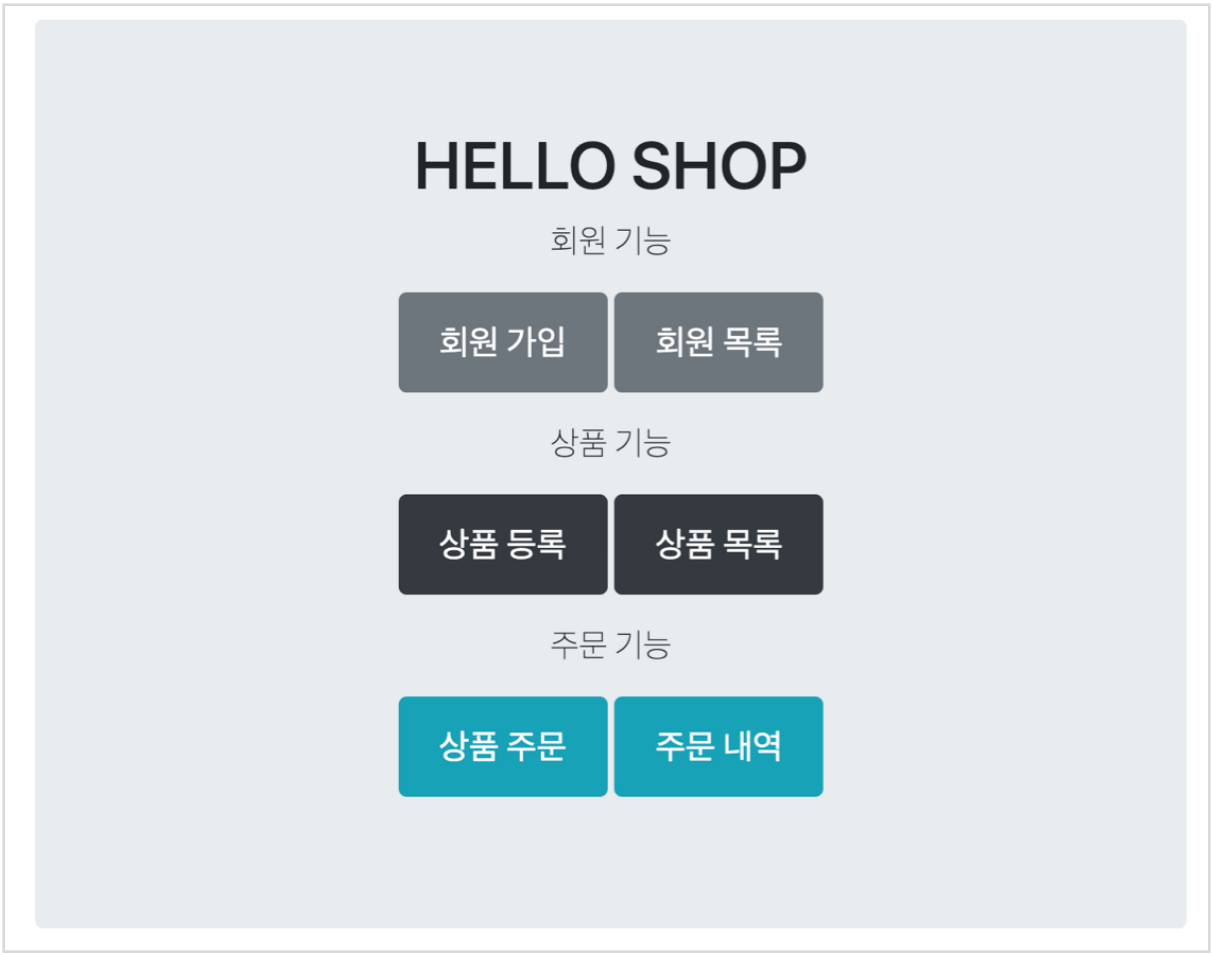
스프링 부트를 사용하면 이 라이브러리만 추가하면 된다.

```
implementation 'com.github.gavlyukovskiy:p6spy-spring-boot-starter:1.5.6'
```

> 참고: 쿼리 파라미터를 로그로 남기는 외부 라이브러리는 시스템 자원을 사용하므로, 개발 단계에서는 편하게 사용해도 된다. 하지만 운영시스템에 적용하려면 꼭 성능테스트를 하고 사용하는 것이 좋다.

도메인 분석 설계

요구사항 분석



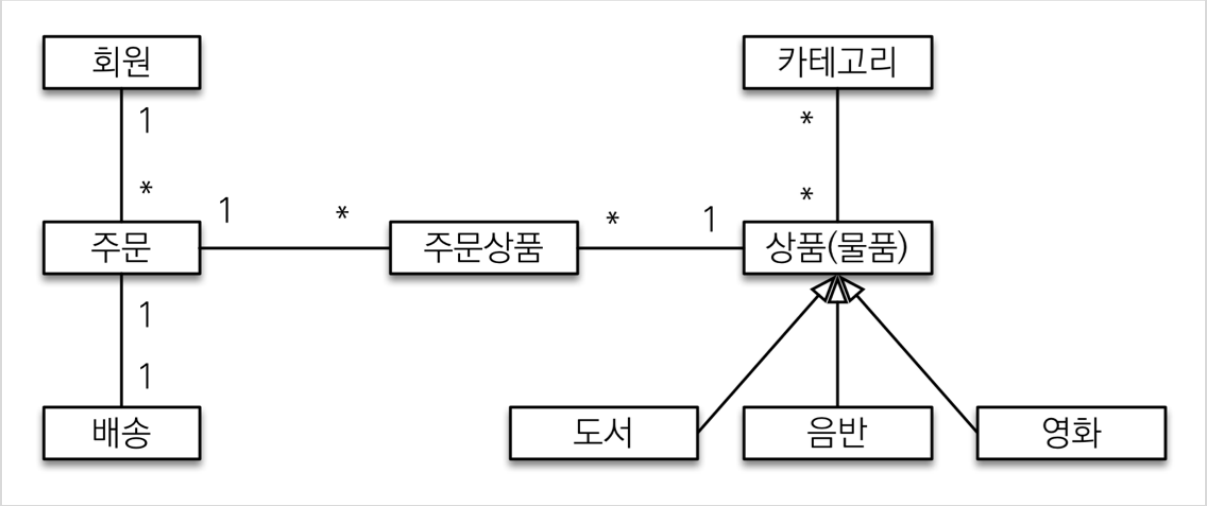
실제 동작하는 화면을 먼저 확인한다.

기능 목록

- 회원 기능
  - 회원 등록
  - 회원 조회
- 상품 기능
  - 상품 등록
  - 상품 수정
  - 상품 조회
- 주문 기능
  - 상품 주문
  - 주문 내역 조회
  - 주문 취소
- 기타 요구사항
  - 상품은 재고 관리가 필요하다.
  - 상품의 종류는 도서, 음반, 영화가 있다.
  - 상품을 카테고리로 구분할 수 있다.

- 상품 주문시 배송 정보를 입력할 수 있다.

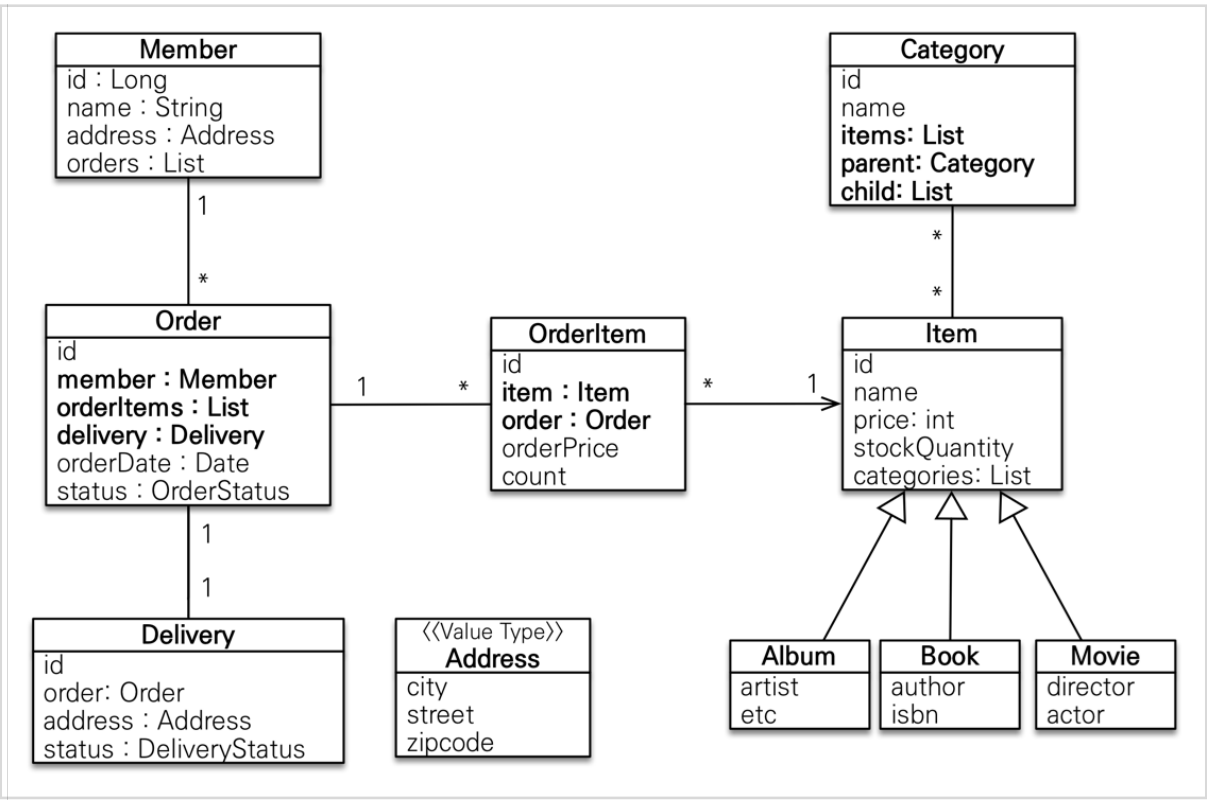
도메인 모델과 테이블 설계



회원, 주문, 상품의 관계: 회원은 여러 상품을 주문할 수 있다. 그리고 한 번 주문할 때 여러 상품을 선택할 수 있으므로 주문과 상품은 다대다 관계다. 하지만 이런 다대다 관계는 관계형 데이터베이스는 물론이고 엔티티에서도 거의 사용하지 않는다. 따라서 그림처럼 주문상품이라는 엔티티를 추가해서 다대다 관계를 일대다, 다대일 관계로 풀어냈다.

상품 분류: 상품은 도서, 음반, 영화로 구분되는데 상품이라는 공통 속성을 사용하므로 상속 구조로 표현했다.

회원 엔티티 분석



회원(Member): 이름과 임베디드 타입인 주소(`Address`), 그리고 주문(`orders`) 리스트를 가진다.

주문(Order): 한 번 주문시 여러 상품을 주문할 수 있으므로 주문과 주문상품(`OrderItem`)은 일대다 관계다. 주문은 상품을 주문한 회원과 배송 정보, 주문 날짜, 주문 상태(`status`)를 가지고 있다. 주문 상태는 열거형을 사용했는데 주문(`ORDER`), 취소(`CANCEL`)을 표현할 수 있다.

주문상품(OrderItem): 주문한 상품 정보와 주문 금액(`orderPrice`), 주문 수량(`count`) 정보를 가지고 있다. (보통 `OrderLine` , `LineItem` 으로 많이 표현한다.)

상품(Item): 이름, 가격, 재고수량(`stockQuantity`)을 가지고 있다. 상품을 주문하면 재고수량이 줄어든다. 상품의 종류로는 도서, 음반, 영화가 있는데 각각은 사용하는 속성이 조금씩 다르다.

배송(Delivery): 주문시 하나의 배송 정보를 생성한다. 주문과 배송은 일대일 관계다.

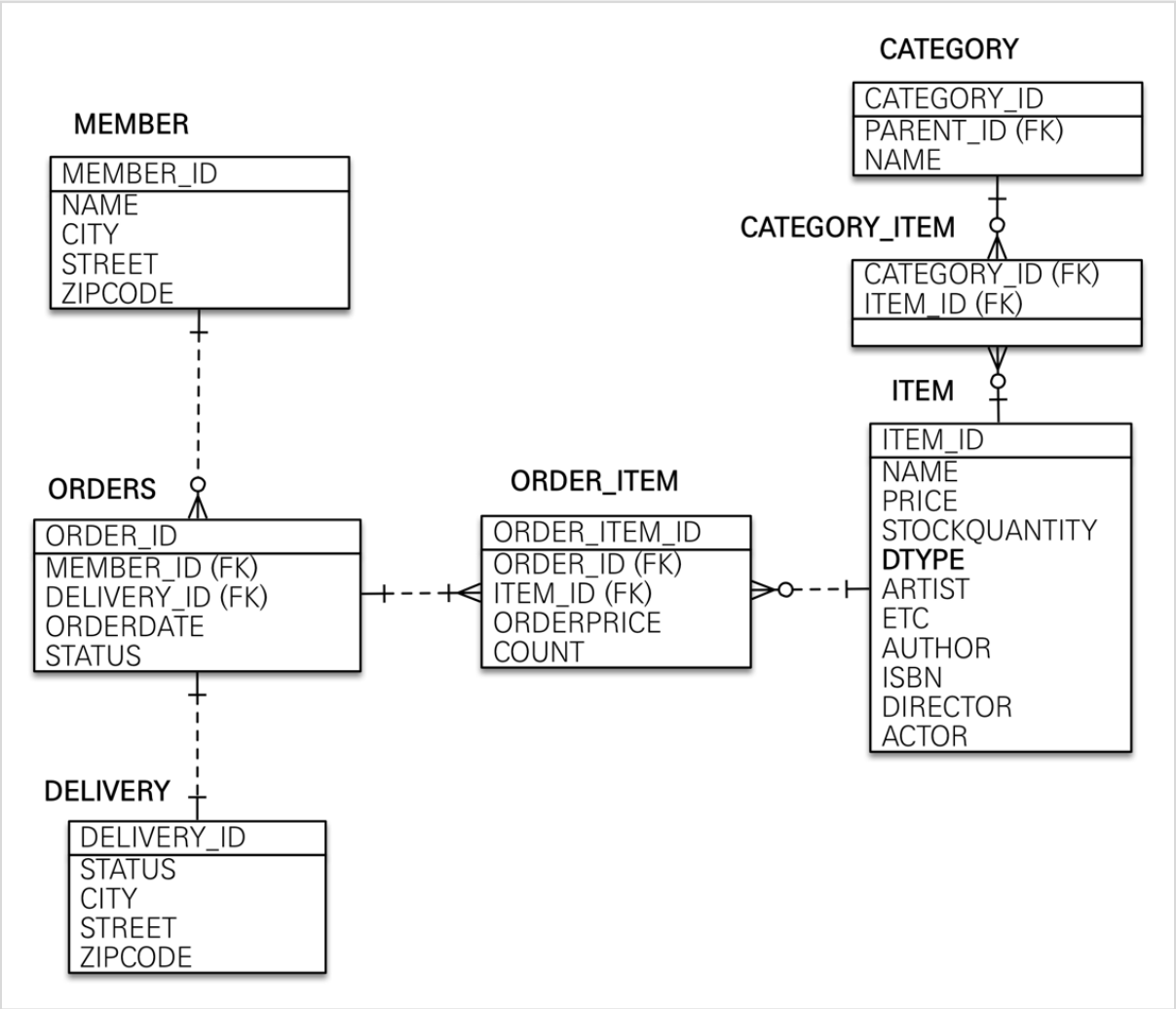
카테고리(Category): 상품과 다대다 관계를 맺는다. `parent` , `child` 로 부모, 자식 카테고리를 연결한다.

주소(Address): 값 타입(임베디드 타입)이다. 회원과 배송(Delivery)에서 사용한다.

> 참고: 회원 엔티티 분석 그림에서 Order와 Delivery가 단방향 관계로 잘못 그려져 있다. 양방향 관계가 맞다.

> 참고: 회원이 주문을 하기 때문에, 회원이 주문리스트를 가지는 것은 얼핏 보면 잘 설계한 것 같지만, 객체 세상은 실제 세계와는 다르다. 실무에서는 회원이 주문을 참조하지 않고, 주문이 회원을 참조하는 것으로 충분하다. 여기서는 일대다, 다대일의 양방향 연관관계를 설명하기 위해서 추가했다.

회원 테이블 분석



MEMBER: 회원 엔티티의 `Address` 임베디드 타입 정보가 회원 테이블에 그대로 들어갔다. 이것은 `DELIVERY` 테이블도 마찬가지다.

ITEM: 앨범, 도서, 영화 타입을 통합해서 하나의 테이블로 만들었다. `DTYPE` 컬럼으로 타입을 구분한다.

> 참고: 테이블명이 `ORDER`가 아니라 `ORDERS`인 것은 데이터베이스가 `order by` 때문에 예약어로 잡고 있는 경우가 많다. 그래서 관례상 `ORDERS`를 많이 사용한다.

> **참고: 실제 코드에서는 DB에 소문자 + _(언더스코어) 스타일을 사용하겠다.**
> 데이터베이스 테이블명, 컬럼명에 대한 관례는 회사마다 다르다. 보통은 대문자 + _(언더스코어)나 소문자 + _(언더스코어) 방식 중에 하나를 지정해서 일관성 있게 사용한다. 강의에서 설명할 때는 객체와 차이를 나

> 타내기 위해 데이터베이스 테이블, 컬럼명은 대문자를 사용했지만, **실제 코드에서는 소문자 + _(언더스코어) 스타일을 사용하겠다.**

연관관계 매핑 분석

회원과 주문: 일대다 , 다대일의 양방향 관계다. 따라서 연관관계의 주인을 정해야 하는데, 외래 키가 있는 주문을 연관관계의 주인으로 정하는 것이 좋다. 그러므로 `Order.member` 를 `ORDERS.MEMBER_ID` 외래 키와 매핑한다.

주문상품과 주문: 다대일 양방향 관계다. 외래 키가 주문상품에 있으므로 주문상품이 연관관계의 주인이다. 그러므로 `OrderItem.order` 를 `ORDER_ITEM.ORDER_ID` 외래 키와 매핑한다.

주문상품과 상품: 다대일 단방향 관계다. `OrderItem.item` 을 `ORDER_ITEM.ITEM_ID` 외래 키와 매핑한다.

주문과 배송: 일대일 양방향 관계다. `Order.delivery` 를 `ORDERS.DELIVERY_ID` 외래 키와 매핑한다.

카테고리와 상품: `@ManyToMany` 를 사용해서 매핑한다.(실무에서 @ManyToMany는 사용하지 말자. 여기서는 다대다 관계를 예제로 보여주기 위해 추가했을 뿐이다)

> **참고: 외래 키가 있는 곳을 연관관계의 주인으로 정해라.**
> 연관관계의 주인은 단순히 외래 키를 누가 관리하냐의 문제이지 비즈니스상 우위에 있다고 주인으로 정하면 안된다.. 예를 들어서 자동차와 바퀴가 있으면, 일대다 관계에서 항상 다쪽에 외래 키가 있으므로 외래 키가 있는 바퀴를 연관관계의 주인으로 정하면 된다. 물론 자동차를 연관관계의 주인으로 정하는 것이 불가능 한 것은 아니지만, 자동차를 연관관계의 주인으로 정하면 자동차가 관리하지 않는 바퀴 테이블의 외래 키 값이 업데이트 되므로 관리와 유지보수가 어렵고, 추가적으로 별도의 업데이트 쿼리가 발생하는 성능 문제도 있다. 자세한 내용은 JPA 기본편을 참고하자.

엔티티 클래스 개발

- 예제에서는 설명을 쉽게하기 위해 엔티티 클래스에 Getter, Setter를 모두 열고, 최대한 단순하게 설계
- 실무에서는 가급적 Getter는 열어두고, Setter는 꼭 필요한 경우에만 사용하는 것을 추천

> 참고: 이론적으로 Getter, Setter 모두 제공하지 않고, 꼭 필요한 별도의 메서드를 제공하는게 가장 이상적이다. 하지만 실무에서 엔티티의 데이터는 조회할 일이 너무 많으므로, Getter의 경우 모두 열어두는 것이 편리하다. Getter는 아무리 호출해도 호출 하는 것 만으로 어떤 일이 발생하지는 않는다. 하지만 Setter는 문제가 다르다. Setter를 호출하면 데이터가 변한다. Setter를 막 열어두면 가까운 미래에 엔티티에가 도대체 왜 변경되는지 추적하기 점점 힘들어진다. 그래서 엔티티를 변경할 때는 Setter 대신에 변경 지점이 명확

> 하도록 변경을 위한 비즈니스 메서드를 별도로 제공해야 한다.

회원 엔티티

```
package jpabook.jpashop.domain;

import javax.persistence.*;
import java.util.ArrayList;
import java.util.List;

@Entity
@Getter @Setter
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<>();

}
```

> 참고: 엔티티의 식별자는 `id`를 사용하고 PK 컬럼명은 `member_id`를 사용했다. 엔티티는 타입(여기서는 `Member`)이 있으므로 `id` 필드만으로 쉽게 구분할 수 있다. 테이블은 타입이 없으므로 구분이 어렵다. 그리고 테이블은 관례상 `테이블명 + id`를 많이 사용한다. 참고로 객체에서 `id` 대신에 `memberId`를 사용해도 된다. 중요한 것은 일관성이다.

주문 엔티티

```
package jpabook.jpashop.domain;

import lombok.Getter;
```

```java
import lombok.Setter;

import javax.persistence.*;
import java.time.LocalDateTime;
import java.util.ArrayList;
import java.util.List;

@Entity
@Table(name = "orders")
@Getter @Setter
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<>();

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

    private LocalDateTime orderDate; //주문시간

    @Enumerated(EnumType.STRING)
    private OrderStatus status; //주문상태 [ORDER, CANCEL]

    //==연관관계 메서드==//
    public void setMember(Member member) {
        this.member = member;
        member.getOrders().add(this);
    }

    public void addOrderItem(OrderItem orderItem) {
```

```
        orderItems.add(orderItem);
        orderItem.setOrder(this);
    }

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

}
```

주문상태

```
package jpabook.jpashop.domain;

public enum OrderStatus {
    ORDER, CANCEL
}
```

주문상품 엔티티

```
package jpabook.jpashop.domain;

import lombok.Getter;
import lombok.Setter;

import jpabook.jpashop.domain.item.Item;
import javax.persistence.*;

@Entity
@Table(name = "order_item")
@Getter @Setter
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

}
```

상품 엔티티

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

import lombok.Getter;
import lombok.Setter;

import jpabook.jpashop.domain.Category;
import javax.persistence.*;
import java.util.ArrayList;
import java.util.List;

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

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

    private String name;
```

```
    private int price;
    private int stockQuantity;

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

}
```

상품 - 도서 엔티티

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

import lombok.Getter;
import lombok.Setter;

import javax.persistence.DiscriminatorValue;
import javax.persistence.Entity;

@Entity
@DiscriminatorValue("B")
@Getter @Setter
public class Book extends Item {

    private String author;
    private String isbn;

}
```

상품 - 음반 엔티티

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

import lombok.Getter;
import lombok.Setter;

import javax.persistence.DiscriminatorValue;
import javax.persistence.Entity;
```

```
@Entity
@DiscriminatorValue("A")
@Getter @Setter
public class Album extends Item {

    private String artist;
    private String etc;

}
```

상품 - 영화 엔티티

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

import lombok.Getter;
import lombok.Setter;

import javax.persistence.DiscriminatorValue;
import javax.persistence.Entity;

@Entity
@DiscriminatorValue("M")
@Getter @Setter
public class Movie extends Item {

    private String director;
    private String actor;

}
```

배송 엔티티

```
package jpabook.jpashop.domain;

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

```java
import javax.persistence.*;

@Entity
@Getter @Setter
public class Delivery {

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

    @OneToOne(mappedBy = "delivery", fetch = FetchType.LAZY)
    private Order order;

    @Embedded
    private Address address;

    @Enumerated(EnumType.STRING)
    private DeliveryStatus status; //ENUM [READY(준비), COMP(배송)]

}
```

배송 상태

```java
package jpabook.jpashop.domain;

public enum DeliveryStatus {
    READY, COMP
}
```

카테고리 엔티티

```java
package jpabook.jpashop.domain;

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

```java
import jpabook.jpashop.domain.item.Item;

import javax.persistence.*;
import java.util.ArrayList;
import java.util.List;

@Entity
@Getter @Setter
public class Category {

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

    private String name;

    @ManyToMany
    @JoinTable(name = "category_item",
            joinColumns = @JoinColumn(name = "category_id"),
            inverseJoinColumns = @JoinColumn(name = "item_id"))
    private List<Item> items = new ArrayList<>();

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "parent_id")
    private Category parent;

    @OneToMany(mappedBy = "parent")
    private List<Category> child = new ArrayList<>();

    //==연관관계 메서드==//
    public void addChildCategory(Category child) {
        this.child.add(child);
        child.setParent(this);
    }

}
```

> 참고: 실무에서는 `@ManyToMany` 를 사용하지 말자

`@ManyToMany`는 편리한 것 같지만, 중간 테이블(`CATEGORY_ITEM`)에 컬럼을 추가할 수 없고, 세밀하게 쿼리를 실행하기 어렵기 때문에 실무에서 사용하기에는 한계가 있다. 중간 엔티티(`CategoryItem`를 만들고 `@ManyToOne`, `@OneToMany`로 매핑해서 사용하자. 정리하면 대다대 매핑을 일대다, 다대일 매핑으로 풀어내서 사용하자.

주소 값 타입

```
package jpabook.jpashop.domain;

import lombok.Getter;
import lombok.Setter;

import javax.persistence.Embeddable;

@Embeddable
@Getter
public class Address {

    private String city;
    private String street;
    private String zipcode;

    protected Address() {
    }

    public Address(String city, String street, String zipcode) {
        this.city = city;
        this.street = street;
        this.zipcode = zipcode;
    }
}
```

참고: 값 타입은 변경 불가능하게 설계해야 한다.
`@Setter`를 제거하고, 생성자에서 값을 모두 초기화해서 변경 불가능한 클래스를 만들자. JPA 스펙상 엔티티나 임베디드 타입(`@Embeddable`)은 자바 기본 생성자(default constructor)를 `public` 또는 `protected`로 설정해야 한다. `public`으로 두는 것 보다는 `protected`로 설정하는 것이 그나마 더 안전하다.
JPA가 이런 제약을 두는 이유는 JPA 구현 라이브러리가 객체를 생성할 때 리플랙션 같은 기술을 사용할 수

있도록 지원해야 하기 때문이다.

엔티티 설계시 주의점

엔티티에는 가급적 Setter를 사용하지 말자

Setter가 모두 열려있다. 변경 포인트가 너무 많아서, 유지보수가 어렵다. 나중에 리펙토링으로 Setter 제거

모든 연관관계는 지연로딩으로 설정!

- 즉시로딩(`EAGER`)은 예측이 어렵고, 어떤 SQL이 실행될지 추적하기 어렵다. 특히 JPQL을 실행할 때 N+1 문제가 자주 발생한다.
- 실무에서 모든 연관관계는 지연로딩(`LAZY`)으로 설정해야 한다.
- 연관된 엔티티를 함께 DB에서 조회해야 하면, fetch join 또는 엔티티 그래프 기능을 사용한다.
- @XToOne(OneToOne, ManyToOne) 관계는 기본이 즉시로딩이므로 직접 지연로딩으로 설정해야 한다.

컬렉션은 필드에서 초기화 하자.

컬렉션은 필드에서 바로 초기화 하는 것이 안전하다.

- `null` 문제에서 안전하다.
- 하이버네이트는 엔티티를 영속화 할 때, 컬랙션을 감싸서 하이버네이트가 제공하는 내장 컬렉션으로 변경한다. 만약 `getOrders()` 처럼 임의의 메서드에서 컬력션을 잘못 생성하면 하이버네이트 내부 메커니즘에 문제가 발생할 수 있다. 따라서 필드레벨에서 생성하는 것이 가장 안전하고, 코드도 간결하다.

```
Member member = new Member();
System.out.println(member.getOrders().getClass());
em.persist(team);
System.out.println(member.getOrders().getClass());

//출력 결과
class java.util.ArrayList
class org.hibernate.collection.internal.PersistentBag
```

테이블, 컬럼명 생성 전략

스프링 부트에서 하이버네이트 기본 매핑 전략을 변경해서 실제 테이블 필드명은 다름

- https://docs.spring.io/spring-boot/docs/2.1.3.RELEASE/reference/htmlsingle/#howto-configure-hibernate-naming-strategy
- http://docs.jboss.org/hibernate/orm/5.4/userguide/html_single/Hibernate_User_Guide.html#naming

하이버네이트 기존 구현: 엔티티의 필드명을 그대로 테이블의 컬럼명으로 사용
(`SpringPhysicalNamingStrategy`)

스프링 부트 신규 설정 (엔티티(필드) ➜ 테이블(컬럼))

1. 카멜 케이스 ➜ 언더스코어(memberPoint ➜ member_point)
2. .(점) ➜ _(언더스코어)
3. 대문자 ➜ 소문자

적용 2 단계

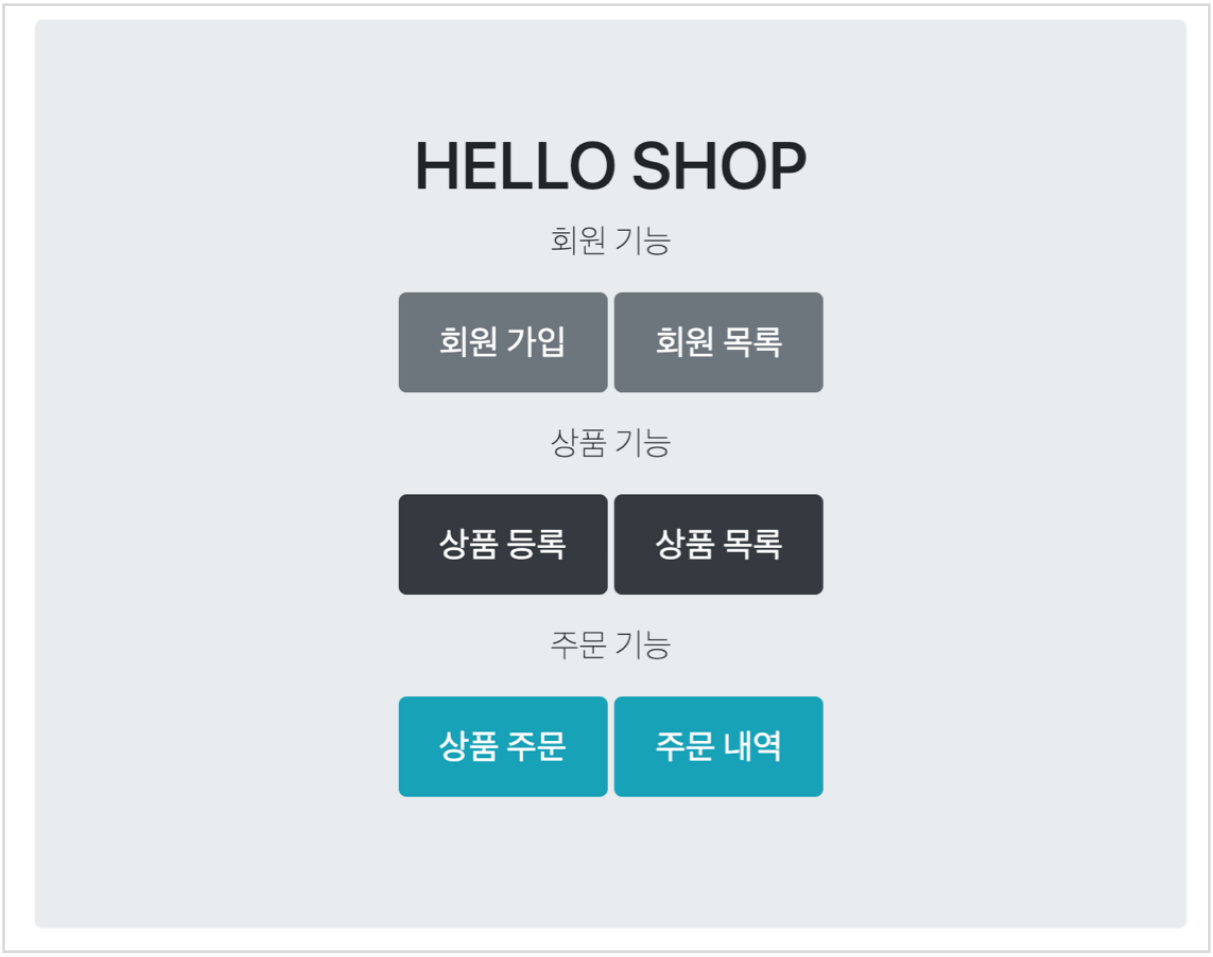
1. 논리명 생성: 명시적으로 컬럼, 테이블명을 직접 적지 않으면 ImplicitNamingStrategy 사용
   `spring.jpa.hibernate.naming.implicit-strategy` : 테이블이나, 컬럼명을 명시하지 않을 때 논리명 적용,
2. 물리명 적용:
   `spring.jpa.hibernate.naming.physical-strategy` : 모든 논리명에 적용됨, 실제 테이블에 적용 (username ➜ usernm 등으로 회사 룰로 바꿀 수 있음)

스프링 부트 기본 설정

```
spring.jpa.hibernate.naming.implicit-strategy:
org.springframework.boot.orm.jpa.hibernate.SpringImplicitNamingStrategy
spring.jpa.hibernate.naming.physical-strategy:
org.springframework.boot.orm.jpa.hibernate.SpringPhysicalNamingStrategy
```

애플리케이션 구현 준비

구현 요구사항

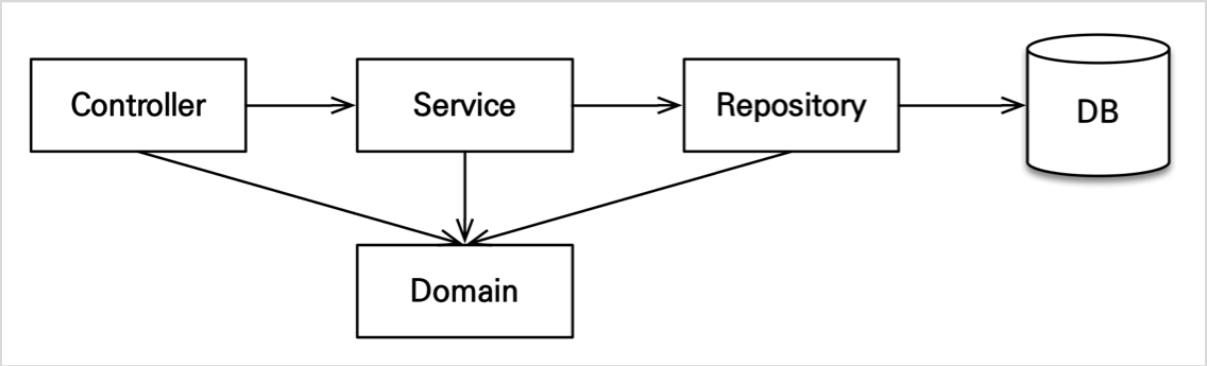


- 회원 기능
  - 회원 등록
  - 회원 조회
- 상품 기능
  - 상품 등록
  - 상품 수정
  - 상품 조회
- 주문 기능
  - 상품 주문
  - 주문 내역 조회
  - 주문 취소

예제를 단순화 하기 위해 다음 기능은 구현X

- 로그인과 권한 관리X
- 파라미터 검증과 예외 처리X
- 상품은 도서만 사용
- 카테고리는 사용X
- 배송 정보는 사용X

애플리케이션 아키텍처



계층형 구조 사용

- controller, web: 웹 계층
- service: 비즈니스 로직, 트랜잭션 처리
- repository: JPA를 직접 사용하는 계층, 엔티티 매니저 사용
- domain: 엔티티가 모여 있는 계층, 모든 계층에서 사용

패키지 구조

- jpabook.jpashop
  - domain
  - exception
  - repository
  - service
  - web

개발 순서: 서비스, 리포지토리 계층을 개발하고, 테스트 케이스를 작성해서 검증, 마지막에 웹 계층 적용

회원 도메인 개발

구현 기능

- 회원 등록
- 회원 목록 조회

순서

- 회원 엔티티 코드 다시 보기
- 회원 리포지토리 개발
- 회원 서비스 개발
- 회원 기능 테스트

회원 리포지토리 개발

회원 리포지토리 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
    private EntityManager em;

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

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

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

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

기술 설명

- `@Repository` : 스프링 빈으로 등록, JPA 예외를 스프링 기반 예외로 예외 변환
- `@PersistenceContext` : 엔티티 메니저(`EntityManager`) 주입
- `@PersistenceUnit` : 엔티티 메니터 팩토리(`EntityManagerFactory`) 주입

기능 설명

- `save()`
- `findOne()`
- `findAll()`
- `findByName()`

회원 서비스 개발

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(readOnly = true)
public class MemberService {

    @Autowired
    MemberRepository memberRepository;

    /**
     * 회원가입
     */
    @Transactional //변경
    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 findOne(Long memberId) {
        return memberRepository.findOne(memberId);
    }
}
```

기술 설명

- `@Service`
- `@Transactional` : 트랜잭션, 영속성 컨텍스트
  - `readOnly=true` : 데이터의 변경이 없는 읽기 전용 메서드에 사용, 영속성 컨텍스트를 플러시 하지 않으므로 약간의 성능 향상(읽기 전용에는 다 적용)
  - 데이터베이스 드라이버가 지원하면 DB에서 성능 향상
- `@Autowired`
  - 생성자 Injection 많이 사용, 생성자가 하나면 생략 가능

기능 설명

- `join()`
- `findMembers()`
- `findOne()`

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

> 참고: 스프링 필드 주입 대신에 생성자 주입을 사용하자.

필드 주입

```
public class MemberService {

    @Autowired
    MemberRepository memberRepository;
    ...
}
```

생성자 주입

```
public class MemberService {

    private final MemberRepository memberRepository;

    public MemberService(MemberRepository memberRepository) {
        this.memberRepository = memberRepository;
    }
    ...
}
```

- 생성자 주입 방식을 권장
- 변경 불가능한 안전한 객체 생성 가능
- 생성자가 하나면, `@Autowired`를 생략할 수 있다.
- `final` 키워드를 추가하면 컴파일 시점에 `memberRepository`를 설정하지 않는 오류를 체크할 수 있다. (보통 기본 생성자를 추가할 때 발견)

lombok

```
@RequiredArgsConstructor
public class MemberService {
```

```
    private final MemberRepository memberRepository;
    ...
}
```

> 참고: 스프링 데이터 JPA를 사용하면 `EntityManager` 도 주입 가능

```
@Repository
@RequiredArgsConstructor
public class MemberRepository {

    private final EntityManager em;
    ...
}
```

`MemberService` 최종 코드

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

    private final MemberRepository memberRepository;

    /**
     * 회원가입
     */
    @Transactional //변경
    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 findOne(Long memberId) {
        return memberRepository.findOne(memberId);
    }
}
```

회원 기능 테스트

테스트 요구사항

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

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.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit4.SpringRunner;
import org.springframework.transaction.annotation.Transactional;
```

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

@RunWith(SpringRunner.class)
@SpringBootTest
@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.findOne(saveId));
    }

    @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.");
        }
    }
```

기술 설명

- `@RunWith(SpringRunner.class)` : 스프링과 테스트 통합
- `@SpringBootTest` : 스프링 부트 띄우고 테스트(이게 없으면 `@Autowired` 다 실패)
- `@Transactional` : 반복 가능한 테스트 지원, 각각의 테스트를 실행할 때마다 트랜잭션을 시작하고 **테스트가 끝나면 트랜잭션을 강제로 롤백** (이 어노테이션이 테스트 케이스에서 사용될 때만 롤백)

기능 설명

- 회원가입 테스트
- 중복 회원 예외처리 테스트

> 참고: 테스트 케이스 작성 고수 되는 마법: Given, When, Then
> (http://martinfowler.com/bliki/GivenWhenThen.html)
> 이 방법이 필수는 아니지만 이 방법을 기본으로 해서 다양하게 응용하는 것을 권장한다.

테스트 케이스를 위한 설정

테스트는 케이스 격리된 환경에서 실행하고, 끝나면 데이터를 초기화하는 것이 좋다. 그런 면에서 메모리 DB를 사용하는 것이 가장 이상적이다.
추가로 테스트 케이스를 위한 스프링 환경과, 일반적으로 애플리케이션을 실행하는 환경은 보통 다르므로 설정 파일을 다르게 사용하자.
다음과 같이 간단하게 테스트용 설정 파일을 추가하면 된다.

- `test/resources/application.yml`

```
spring:
#  datasource:
#    url: jdbc:h2:mem:testdb
#    username: sa
#    password:
#    driver-class-name: org.h2.Driver

#  jpa:
#    hibernate:
```

```
#      ddl-auto: create
#    properties:
#      hibernate:
         #        show_sql: true
#        format_sql: true
#    open-in-view: false

logging.level:
  org.hibernate.SQL: debug
#  org.hibernate.type: trace
```

이제 테스트에서 스프링을 실행하면 이 위치에 있는 설정 파일을 읽는다.
(만약 이 위치에 없으면 `src/resources/application.yml` 을 읽는다.)

스프링 부트는 datasource 설정이 없으면, 기본적을 메모리 DB를 사용하고, driver-class도 현재 등록된 라이브러리를 보고 찾아준다. 추가로 `ddl-auto` 도 `create-drop` 모드로 동작한다. 따라서 데이터소스나, JPA 관련된 별도의 추가 설정을 하지 않아도 된다.

상품 도메인 개발

구현 기능

- 상품 등록
- 상품 목록 조회
- 상품 수정

순서

- 상품 엔티티 개발(비즈니스 로직 추가)
- 상품 리포지토리 개발
- 상품 서비스 개발
- 상품 기능 테스트

상품 엔티티 개발(비즈니스 로직 추가)

상품 엔티티 코드

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

```java
import jpabook.jpashop.exception.NotEnoughStockException;
import lombok.Getter;
import lombok.Setter;

import jpabook.jpashop.domain.Category;
import javax.persistence.*;
import java.util.ArrayList;
import java.util.List;

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

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

    private String name;
    private int price;
    private int stockQuantity;

    @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;
    }
```

```
}
```

예외 추가

```
package jpabook.jpashop.exception;

public class NotEnoughStockException extends RuntimeException {

    public NotEnoughStockException() {
    }

    public NotEnoughStockException(String message) {
        super(message);
    }

    public NotEnoughStockException(String message, Throwable cause) {
        super(message, cause);
    }

    public NotEnoughStockException(Throwable cause) {
        super(cause);
    }

}
```

비즈니스 로직 분석

- `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 예외가 발생한다. 주로 상품을 주문할 때 사용한다.

상품 리포지토리 개발

상품 리포지토리 코드

```
package jpabook.jpashop.repository;

import jpabook.jpashop.domain.item.Item;
import lombok.RequiredArgsConstructor;
import org.springframework.stereotype.Repository;

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

@Repository
@RequiredArgsConstructor
public class ItemRepository {

    private final EntityManager em;

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

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

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

기능 설명

- `save()`
  - `id`가 없으면 신규로 보고 `persist()` 실행
  - `id`가 있으면 이미 데이터베이스에 저장된 엔티티를 수정한다고 보고, `merge()`를 실행, 자세한 내용

은 뒤에 웹에서 설명(그냥 지금은 저장한다 정도로 생각하자)

상품 서비스 개발

상품 서비스 코드

```
package jpabook.jpashop.service;

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

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

    private final ItemRepository itemRepository;

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

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

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

상품 서비스는 상품 리포지토리에 단순히 위임만 하는 클래스

상품 기능 테스트

상품 테스트는 회원 테스트와 비슷하므로 생략

주문 도메인 개발

구현 기능

- 상품 주문
- 주문 내역 조회
- 주문 취소

순서

- 주문 엔티티, 주문상품 엔티티 개발
- 주문 리포지토리 개발
- 주문 서비스 개발
- 주문 검색 기능 개발
- 주문 기능 테스트

주문, 주문상품 엔티티 개발

주문 엔티티 개발

주문 엔티티 코드

```
package jpabook.jpashop.domain;

import lombok.Getter;
import lombok.Setter;

import javax.persistence.*;
import java.time.LocalDateTime;
import java.util.ArrayList;
import java.util.List;

@Entity
@Table(name = "orders")
@Getter @Setter
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<>();

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

private LocalDateTime orderDate; //주문시간

@Enumerated(EnumType.STRING)
private OrderStatus status; //주문상태 [ORDER, CANCEL]

//==연관관계 메서드==//
public void setMember(Member member) {
    this.member = member;
    member.getOrders().add(this);
}

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

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

//==생성 메서드==//
public static Order createOrder(Member member, Delivery delivery,
```

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

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

        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;
    }

}
```

기능 설명

- **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. (실무에서는 주로 주문에 전체 주문 가격 필드를 두고 역정규화 한다.)

주문상품 엔티티 개발

주문상품 엔티티 코드

```
package jpabook.jpashop.domain;

import lombok.Getter;
import lombok.Setter;

import jpabook.jpashop.domain.item.Item;
import javax.persistence.*;

@Entity
@Table(name = "order_item")
@Getter @Setter
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();
    }

}
```

기능 설명

- **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 리포지토리 개발

주문 리포지토리 코드

```
package jpabook.jpashop.repository;

import jpabook.jpashop.domain.Order;
import lombok.RequiredArgsConstructor;
import org.springframework.stereotype.Repository;

import javax.persistence.EntityManager;

@Repository
@RequiredArgsConstructor
public class OrderRepository {

    private final EntityManager em;

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

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

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

주문 리포지토리에는 주문 엔티티를 저장하고 검색하는 기능이 있다. 마지막의 `findAll(OrderSearch orderSearch)` 메서드는 조금 뒤에 있는 주문 검색 기능에서 자세히 알아보자.

주문 서비스 개발

주문 서비스 코드

```java
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.repository.ItemRepository;
import jpabook.jpashop.repository.MemberRepository;
import jpabook.jpashop.repository.OrderRepository;
import lombok.RequiredArgsConstructor;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

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

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

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

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

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

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

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

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

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

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

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

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

참고: 예제를 단순화하려고 한 번에 하나의 상품만 주문할 수 있다.

- **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 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 객체 지향의 특성을 적극 활용하는 것을 도메인 모델 패턴(http://martinfowler.com/eaaCatalog/domainModel.html)이라 한다. 반대로 엔티티에는 비즈니스 로직이 거의 없고 서비스 계층에서 대부분의 비즈니스 로직을 처리하는 것을 트랜잭션 스크립트 패턴(http://martinfowler.com/eaaCatalog/transactionScript.html)이라 한다.

주문 기능 테스트

테스트 요구사항

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

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.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit4.SpringRunner;
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(SpringRunner.class)
@SpringBootTest
@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(),
orderCount);

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

        assertEquals("상품 주문시 상태는 ORDER",OrderStatus.ORDER,
getOrder.getStatus());
        assertEquals("주문한 상품 종류 수가 정확해야 한다.",1,
getOrder.getOrderItems().size());
        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.

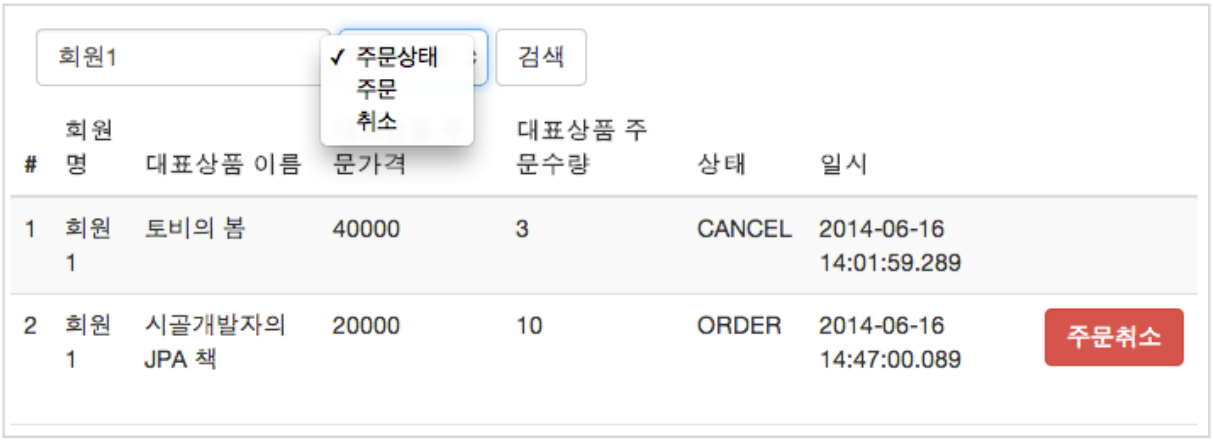
주문 취소 테스트 코드

```
@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.findOne(orderId);

    assertEquals("주문 취소시 상태는 CANCEL 이다.",OrderStatus.CANCEL,
getOrder.getStatus());
    assertEquals("주문이 취소된 상품은 그만큼 재고가 증가해야 한다.", 10,
item.getStockQuantity());
}
```

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.

주문 검색 기능 개발

JPA에서 **동적 쿼리**를 어떻게 해결해야 하는가?



*검색 조건 파라미터 `OrderSearch` *

```
package jpabook.jpashop.domain;

public class OrderSearch {

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

    //Getter, Setter
}
```

검색을 추가한 주문 리포지토리 코드

```
package jpabook.jpashop.repository;

@Repository
public class OrderRepository {

    @PersistenceContext
    EntityManager em;

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

    public Order findOne(Long id) {
```

```
        return em.find(Order.class, id);
    }

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

`findAll(OrderSearch orderSearch)` 메서드는 검색 조건에 동적으로 쿼리를 생성해서 주문 엔티티를 조회한다.

JPQL로 처리

```
public List<Order> findAllByString(OrderSearch orderSearch) {
    //language=JPAQL
    String jpql = "select o From Order o join o.member m";
    boolean isFirstCondition = true;

    //주문 상태 검색
    if (orderSearch.getOrderStatus() != null) {
        if (isFirstCondition) {
            jpql += " where";
            isFirstCondition = false;
        } else {
            jpql += " and";
        }
        jpql += " o.status = :status";
    }
    //회원 이름 검색
    if (StringUtils.hasText(orderSearch.getMemberName())) {
        if (isFirstCondition) {
            jpql += " where";
            isFirstCondition = false;
        } else {
            jpql += " and";
        }
        jpql += " m.name like :name";
    }
```

```
    TypedQuery<Order> query = em.createQuery(jpql, Order.class)
            .setMaxResults(1000); //최대 1000건

    if (orderSearch.getOrderStatus() != null) {
        query = query.setParameter("status", orderSearch.getOrderStatus());
    }
    if (StringUtils.hasText(orderSearch.getMemberName())) {
        query = query.setParameter("name", orderSearch.getMemberName());
    }

    return query.getResultList();
}
```

JPQL 쿼리를 문자로 생성하기는 번거롭고, 실수로 인한 버그가 충분히 발생할 수 있다.

JPA Criteria로 처리

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

    CriteriaBuilder cb = em.getCriteriaBuilder();
    CriteriaQuery<Order> cq = cb.createQuery(Order.class);
    Root<Order> o = cq.from(Order.class);
    Join<Order, Member> m = o.join("member", JoinType.INNER); //회원과 조인

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

    //주문 상태 검색
    if (orderSearch.getOrderStatus() != null) {
        Predicate status = cb.equal(o.get("status"),
orderSearch.getOrderStatus());
        criteria.add(status);
    }
    //회원 이름 검색
    if (StringUtils.hasText(orderSearch.getMemberName())) {
        Predicate name =
                cb.like(m.<String>get("name"), "%" +
orderSearch.getMemberName() + "%");
        criteria.add(name);
```

```
    }

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

JPA Criteria는 JPA 표준 스펙이지만 실무에서 사용하기에 너무 복잡하다. 결국 다른 대안이 필요하다. 많은 개발자가 비슷한 고민을 했지만, 가장 멋진 해결책은 Querydsl이 제시했다. Querydsl 소개장에서 간단히 언급하겠다. 지금은 이대로 진행하자.

> 참고: JPA Criteria에 대한 자세한 내용은 자바 ORM 표준 JPA 프로그래밍 책을 참고하자

웹 계층 개발

- 홈 화면
- 회원 기능
  - 회원 등록
  - 회원 조회
- 상품 기능
  - 상품 등록
  - 상품 수정
  - 상품 조회
- 주문 기능
  - 상품 주문
  - 주문 내역 조회
  - 주문 취소

상품 등록
상품 목록
상품 수정
변경 감지와 병합
상품 주문

홈 화면과 레이아웃

홈 컨트롤러 등록

```
package jpabook.jpashop.web;

import lombok.extern.slf4j.Slf4j;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
@Slf4j
public class HomeController {

    @RequestMapping("/")
    public String home() {
        log.info("home controller");
        return "home";
    }
}
```

스프링 부트 타임리프 기본 설정

```
spring:
  thymeleaf:
    prefix: classpath:/templates/
    suffix: .html
```

- 스프링 부트 타임리프 viewName 매핑
  - `resources:templates/` +{ViewName}+ `.html`
  - `resources:templates/home.html`

반환한 문자(`home`)과 스프링부트 설정 `prefix` , `suffix` 정보를 사용해서 렌더링할 뷰(`html`)를 찾는다.

참고: https://docs.spring.io/spring-boot/docs/2.1.7.RELEASE/reference/html/common-application-properties.html

타임리프 템플릿 등록

home.html

```
<!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org">
<head th:replace="fragments/header :: header">
    <title>Hello</title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
</head>

<body>

<div class="container">

    <div th:replace="fragments/bodyHeader :: bodyHeader" />

    <div class="jumbotron">
        <h1>HELLO SHOP</h1>
        <p class="lead">회원 기능</p>
        <p>
            <a class="btn btn-lg btn-secondary" href="/members/new">회원 가입</a>
            <a class="btn btn-lg btn-secondary" href="/members">회원 목록</a>
        </p>
        <p class="lead">상품 기능</p>
        <p>
            <a class="btn btn-lg btn-dark" href="/items/new">상품 등록</a>
            <a class="btn btn-lg btn-dark" href="/items">상품 목록</a>
        </p>
        <p class="lead">주문 기능</p>
        <p>
            <a class="btn btn-lg btn-info" href="/order">상품 주문</a>
            <a class="btn btn-lg btn-info" href="/orders">주문 내역</a>
        </p>
    </div>

    <div th:replace="fragments/footer :: footer" />

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

```
</body>
</html>
```

fragments/header

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head th:fragment="header">
    <!-- Required meta tags -->
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">

    <!-- Bootstrap CSS -->
    <link rel="stylesheet" href="/css/bootstrap.min.css" integrity="sha384-ggOyR0iXCbMQv3Xipma34MD+dH/1fQ784/j6cY/iJTQUOhcWr7x9JvoRxT2MZw1T" crossorigin="anonymous">
    <!-- Custom styles for this template -->
    <link href="/css/jumbotron-narrow.css" rel="stylesheet">

    <title>Hello, world!</title>
</head>
```

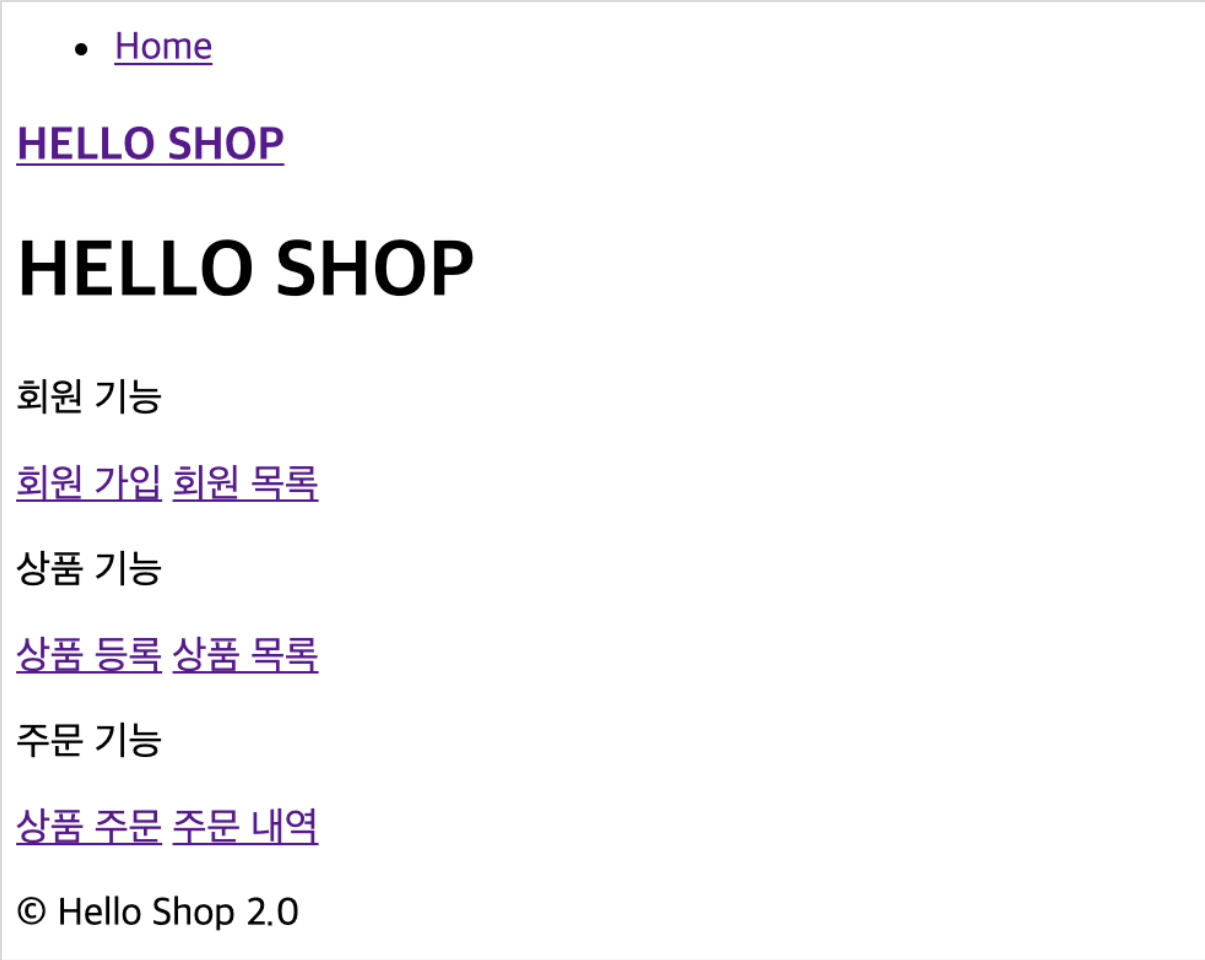
fragments/bodyHeader

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<div class="header" th:fragment="bodyHeader">
    <ul class="nav nav-pills pull-right">
        <li><a href="/">Home</a></li>
    </ul>
    <a href="/"><h3 class="text-muted">HELLO SHOP</h3></a>
</div>
```

fragments/footer

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<div class="footer" th:fragment="footer">
    <p>&copy; Hello Shop V2</p>
</div>
```

결과 화면



> **참고: Hierarchical-style layouts**
> 예제에서는 뷰 템플릿을 최대한 간단하게 설명하려고, `header`, `footer` 같은 템플릿 파일을 반복해서 포함한다. 다음 링크의 Hierarchical-style layouts을 참고하면 이런 부분도 중복을 제거할 수 있다.
> https://www.thymeleaf.org/doc/articles/layouts.html

> **참고: 뷰 템플릿 변경사항을 서버 재시작 없이 즉시 반영하기**
> 1. spring-boot-devtools 추가
> 2. html 파일 build-> Recompile

view 리소스 등록

이쁜 디자인을 위해 부트스트랩을 사용하겠다. (v4.3.1) (https://getbootstrap.com/)

- `resources/static` 하위에 `css`, `js` 추가
- `resources/static/css/jumbotron-narrow.css` 추가

jumbotron-narrow.css 파일

```
/* Space out content a bit */
body {
  padding-top: 20px;
  padding-bottom: 20px;
}

/* Everything but the jumbotron gets side spacing for mobile first views */
.header,
.marketing,
.footer {
  padding-left: 15px;
  padding-right: 15px;
}

/* Custom page header */
.header {
  border-bottom: 1px solid #e5e5e5;
}
/* Make the masthead heading the same height as the navigation */
.header h3 {
  margin-top: 0;
  margin-bottom: 0;
  line-height: 40px;
  padding-bottom: 19px;
}

/* Custom page footer */
.footer {
  padding-top: 19px;
  color: #777;
```

```
  border-top: 1px solid #e5e5e5;
}

/* Customize container */
@media (min-width: 768px) {
  .container {
    max-width: 730px;
  }
}
.container-narrow > hr {
  margin: 30px 0;
}

/* Main marketing message and sign up button */
.jumbotron {
  text-align: center;
  border-bottom: 1px solid #e5e5e5;
}
.jumbotron .btn {
  font-size: 21px;
  padding: 14px 24px;
}

/* Supporting marketing content */
.marketing {
  margin: 40px 0;
}
.marketing p + h4 {
  margin-top: 28px;
}

/* Responsive: Portrait tablets and up */
@media screen and (min-width: 768px) {
  /* Remove the padding we set earlier */
  .header,
  .marketing,
  .footer {
    padding-left: 0;
    padding-right: 0;
```

```
  }
  /* Space out the masthead */
  .header {
    margin-bottom: 30px;
  }
  /* Remove the bottom border on the jumbotron for visual effect */
  .jumbotron {
    border-bottom: 0;
  }
}
```

회원 등록

- 폼 객체를 사용해서 화면 계층과 서비스 계층을 명확하게 분리한다.

회원 등록 폼 객체

```
package jpabook.jpashop.web;

import lombok.Getter;
import lombok.Setter;

import javax.validation.constraints.NotEmpty;

@Getter @Setter
public class MemberForm {

    @NotEmpty(message = "회원 이름은 필수 입니다")
    private String name;

    private String city;
    private String street;
    private String zipcode;
}
```

회원 등록 컨트롤러

```
package jpabook.jpashop.web;

import jpabook.jpashop.domain.Address;
import jpabook.jpashop.domain.Member;
import jpabook.jpashop.service.MemberService;
import lombok.RequiredArgsConstructor;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.validation.BindingResult;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

import javax.validation.Valid;
import java.util.List;

@Controller
@RequiredArgsConstructor
public class MemberController {

    private final MemberService memberService;

    @GetMapping(value = "/members/new")
    public String createForm(Model model) {
        model.addAttribute("memberForm", new MemberForm());
        return "members/createMemberForm";
    }

    @PostMapping(value = "/members/new")
    public String create(@Valid MemberForm form, BindingResult result) {

        if (result.hasErrors()) {
            return "members/createMemberForm";
        }

        Address address = new Address(form.getCity(), form.getStreet(),
form.getZipcode());
        Member member = new Member();
        member.setName(form.getName());
```

```
        member.setAddress(address);

        memberService.join(member);

        return "redirect:/";
    }

}
```

회원 등록 폼 화면(`templates/members/createMemberForm.html`)

```
<!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org">
<head th:replace="fragments/header :: header" />
<style>
    .fieldError {
        border-color: #bd2130;
    }
</style>
<body>

<div class="container">
    <div th:replace="fragments/bodyHeader :: bodyHeader"/>

    <form role="form" action="/members/new" th:object="${memberForm}"
method="post">
        <div class="form-group">
            <label th:for="name">이름</label>
            <input type="text" th:field="*{name}" class="form-control"
placeholder="이름을 입력하세요"
                   th:class="${#fields.hasErrors('name')}? 'form-control
fieldError' : 'form-control'">
            <p th:if="${#fields.hasErrors('name')}"
th:errors="*{name}">Incorrect date</p>
        </div>
        <div class="form-group">
            <label th:for="city">도시</label>
            <input type="text" th:field="*{city}" class="form-control"
```

```
placeholder="도시를 입력하세요">
        </div>
        <div class="form-group">
            <label th:for="street">거리</label>
            <input type="text" th:field="*{street}" class="form-control"
placeholder="거리를 입력하세요">
        </div>
        <div class="form-group">
            <label th:for="zipcode">우편번호</label>
            <input type="text" th:field="*{zipcode}" class="form-control"
placeholder="우편번호를 입력하세요">
        </div>
        <button type="submit" class="btn btn-primary">Submit</button>
    </form>
    <br/>
    <div th:replace="fragments/footer :: footer" />
</div> <!-- /container -->

</body>
</html>
```

회원 목록 조회

회원 목록 컨트롤러 추가

```
package jpabook.jpashop.web;

@Controller
@RequiredArgsConstructor
public class MemberController {

    //추가
    @GetMapping(value = "/members")
    public String list(Model model) {
        List<Member> members = memberService.findMembers();
        model.addAttribute("members", members);
```

```
        return "members/memberList";
    }

}
```

- 조회한 상품을 뷰에 전달하기 위해 스프링 MVC가 제공하는 모델(`Model`) 객체에 보관
- 실행할 뷰 이름을 반환

회원 목록 뷰(`templates/members/memberList.html`)

```
<!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org">
<head th:replace="fragments/header :: header" />
<body>

<div class="container">
    <div th:replace="fragments/bodyHeader :: bodyHeader" />
    <div>
        <table class="table table-striped">
            <thead>
            <tr>
                <th>#</th>
                <th>이름</th>
                <th>도시</th>
                <th>주소</th>
                <th>우편번호</th>
            </tr>
            </thead>
            <tbody>
            <tr th:each="member : ${members}">
                <td th:text="${member.id}"></td>
                <td th:text="${member.name}"></td>
                <td th:text="${member.address?.city}"></td>
                <td th:text="${member.address?.street}"></td>
                <td th:text="${member.address?.zipcode}"></td>
            </tr>
            </tbody>
```

```
        </table>
    </div>

    <div th:replace="fragments/footer :: footer" />

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

</body>
</html>
```

> 참고: 타임리프에서 ?를 사용하면 `null`을 무시한다.

> 참고: 폼 객체 vs 엔티티 직접 사용
> 참고: 요구사항이 정말 단순할 때는 폼 객체(`MemberForm`) 없이 엔티티(`Member`)를 직접 등록과 수정 화면에서 사용해도 된다. 하지만 화면 요구사항이 복잡해지기 시작하면, 엔티티에 화면을 처리하기 위한 기능이 점점 증가한다. 결과적으로 엔티티는 점점 화면에 종속적으로 변하고, 이렇게 화면 기능 때문에 지저분해진 엔티티는 결국 유지보수하기 어려워진다.
> 실무에서 **엔티티는 핵심 비즈니스 로직만 가지고 있고, 화면을 위한 로직은 없어야 한다**. 화면이나 API에 맞는 폼 객체나 DTO를 사용하자. 그래서 화면이나 API 요구사항을 이것들로 처리하고, 엔티티는 최대한 순수하게 유지하자.

상품 등록

상품 등록 폼

```
package jpabook.jpashop.web;

import lombok.Getter;
import lombok.Setter;

@Getter @Setter
public class BookForm {

    private Long id;

    private String name;
    private int price;
    private int stockQuantity;
```

```
    private String author;
    private String isbn;

}
```

상품 등록 컨트롤러

```
package jpabook.jpashop.web;

import jpabook.jpashop.domain.item.Book;
import jpabook.jpashop.domain.item.Item;
import jpabook.jpashop.service.ItemService;
import lombok.RequiredArgsConstructor;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@Controller
@RequiredArgsConstructor
public class ItemController {

    private final ItemService itemService;

    @GetMapping(value = "/items/new")
    public String createForm(Model model) {

        model.addAttribute("form", new BookForm());
        return "items/createItemForm";
    }

    @PostMapping(value = "/items/new")
    public String create(BookForm form) {

        Book book = new Book();
        book.setName(form.getName());
```

```
        book.setPrice(form.getPrice());
        book.setStockQuantity(form.getStockQuantity());
        book.setAuthor(form.getAuthor());
        book.setIsbn(form.getIsbn());

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

}
```

상품 등록 뷰(`items/createItemForm.html`)

```
<!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org">
<head th:replace="fragments/header :: header" />
<body>

<div class="container">
    <div th:replace="fragments/bodyHeader :: bodyHeader"/>

    <form th:action="@{/items/new}" th:object="${form}" method="post">
        <div class="form-group">
            <label th:for="name">상품명</label>
            <input type="text" th:field="*{name}" class="form-control"
placeholder="이름을 입력하세요">
        </div>
        <div class="form-group">
            <label th:for="price">가격</label>
            <input type="number" th:field="*{price}" class="form-control"
placeholder="가격을 입력하세요">
        </div>
        <div class="form-group">
            <label th:for="stockQuantity">수량</label>
            <input type="number" th:field="*{stockQuantity}" class="form-
control" placeholder="수량을 입력하세요">
        </div>
        <div class="form-group">
```

```
            <label th:for="author">저자</label>
            <input type="text" th:field="*{author}" class="form-control"
placeholder="저자를 입력하세요">
        </div>
        <div class="form-group">
            <label th:for="isbn">ISBN</label>
            <input type="text" th:field="*{isbn}" class="form-control"
placeholder="ISBN을 입력하세요">
        </div>
        <button type="submit" class="btn btn-primary">Submit</button>
    </form>
    <br/>
    <div th:replace="fragments/footer :: footer" />

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



</body>
</html>
```

> 주의: PDF에서 html 파일을 복사할 때 줄바꿈 때문에, `form-control`이 `formcontrol`이라고 붙는 경우가 있다. 이 경우 `form-control`로 수정해야 한다.

상품 등록

- 상품 등록 폼에서 데이터를 입력하고 Submit 버튼을 클릭하면 `/items/new`를 POST 방식으로 요청
- 상품 저장이 끝나면 상품 목록 화면(`redirect:/items`)으로 리다이렉트

상품 목록

상품 목록 컨트롤러

```
package jpabook.jpashop.web;


@Controller
@RequiredArgsConstructor
public class ItemController {
```

```
    private final ItemService itemService;

    /**
     * 상품 목록
     */
    @GetMapping(value = "/items")
    public String list(Model model) {

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

}
```

상품 목록 뷰(`items/itemList.html`)

```
<!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org">
<head th:replace="fragments/header :: header" />
<body>

<div class="container">
    <div th:replace="fragments/bodyHeader :: bodyHeader"/>

    <div>
        <table class="table table-striped">
            <thead>
            <tr>
                <th>#</th>
                <th>상품명</th>
                <th>가격</th>
                <th>재고수량</th>
                <th></th>
            </tr>
            </thead>
            <tbody>
```

```
            <tr th:each="item : ${items}">
                <td th:text="${item.id}"></td>
                <td th:text="${item.name}"></td>
                <td th:text="${item.price}"></td>
                <td th:text="${item.stockQuantity}"></td>
                <td>
                    <a href="#" th:href="@{/items/{id}/edit (id=${item.id})}"
class="btn btn-primary" role="button">수정</a>
                </td>
            </tr>
            </tbody>
        </table>
    </div>

    <div th:replace="fragments/footer :: footer"/>

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

</body>
</html>
```

`model`에 담아둔 상품 목록인 `items`를 꺼내서 상품 정보를 출력

상품 수정

상품 수정과 관련된 컨트롤러 코드

```
package jpabook.jpashop.web;

import jpabook.jpashop.domain.item.Book;
import jpabook.jpashop.domain.item.Item;
import jpabook.jpashop.service.ItemService;
import lombok.RequiredArgsConstructor;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
```

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

import java.util.List;

@Controller
@RequiredArgsConstructor
public class ItemController {

    /**
     * 상품 수정 폼
     */
    @GetMapping(value = "/items/{itemId}/edit")
    public String updateItemForm(@PathVariable("itemId") Long itemId, Model
model) {

        Book item = (Book) itemService.findOne(itemId);

        BookForm form = new BookForm();
        form.setId(item.getId());
        form.setName(item.getName());
        form.setPrice(item.getPrice());
        form.setStockQuantity(item.getStockQuantity());
        form.setAuthor(item.getAuthor());
        form.setIsbn(item.getIsbn());

        model.addAttribute("form", form);
        return "items/updateItemForm";
    }

    /**
     * 상품 수정
     */
    @PostMapping(value = "/items/{itemId}/edit")
    public String updateItem(@ModelAttribute("form") BookForm form) {

        Book book = new Book();
        book.setId(form.getId());
        book.setName(form.getName());
        book.setPrice(form.getPrice());
```

```
        book.setStockQuantity(form.getStockQuantity());
        book.setAuthor(form.getAuthor());
        book.setIsbn(form.getIsbn());

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


}
```

상품 수정 폼 화면(`items/updateItemForm`)

```
<!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org">
<head th:replace="fragments/header :: header" />
<body>

<div class="container">
    <div th:replace="fragments/bodyHeader :: bodyHeader"/>

    <form th:object="${form}" method="post">
        <!-- id -->
        <input type="hidden" th:field="*{id}" />

        <div class="form-group">
            <label th:for="name">상품명</label>
            <input type="text" th:field="*{name}" class="form-control"
placeholder="이름을 입력하세요" />
        </div>
        <div class="form-group">
            <label th:for="price">가격</label>
            <input type="number" th:field="*{price}" class="form-control"
placeholder="가격을 입력하세요" />
        </div>
        <div class="form-group">
            <label th:for="stockQuantity">수량</label>
            <input type="number" th:field="*{stockQuantity}" class="form-
```

```
control" placeholder="수량을 입력하세요" />
        </div>
        <div class="form-group">
            <label th:for="author">저자</label>
            <input type="text" th:field="*{author}" class="form-control"
placeholder="저자를 입력하세요" />
        </div>
        <div class="form-group">
            <label th:for="isbn">ISBN</label>
            <input type="text" th:field="*{isbn}" class="form-control"
placeholder="ISBN을 입력하세요" />
        </div>
        <button type="submit" class="btn btn-primary">Submit</button>
    </form>

    <div th:replace="fragments/footer :: footer" />

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

</body>
</html>
```

상품 수정 폼 이동

1. 수정 버튼을 선택하면 `/items/{itemId}/edit` URL을 GET 방식으로 요청
2. 그 결과로 `updateItemForm()` 메서드를 실행하는데 이 메서드는 `itemService.findOne(itemId)`를 호출해서 수정할 상품을 조회
3. 조회 결과를 모델 객체에 담아서 뷰(`items/updateItemForm`)에 전달

상품 수정 실행

상품 수정 폼 HTML에는 상품의 id(hidden), 상품명, 가격, 수량 정보 있음

1. 상품 수정 폼에서 정보를 수정하고 Submit 버튼을 선택
2. `/items/{itemId}/edit` URL을 POST 방식으로 요청하고 `updateItem()` 메서드를 실행
3. 이때 컨트롤러에 파라미터로 넘어온 `item` 엔티티 인스턴스는 현재 준영속 상태다. 따라서 영속성 컨텍스트의 지원을 받을 수 없고 데이터를 수정해도 변경 감지 기능은 동작X

변경 감지와 병합(merge)

> 참고: 정말 중요한 내용이니 꼭! 완벽하게 이해하셔야 합니다.

준영속 엔티티?

영속성 컨텍스트가 더는 관리하지 않는 엔티티를 말한다.

(여기서는 `itemService.saveItem(book)` 에서 수정을 시도하는 `Book` 객체다. `Book` 객체는 이미 DB에 한번 저장되어서 식별자가 존재한다. 이렇게 임의로 만들어낸 엔티티도 기존 식별자를 가지고 있으면 준영속 엔티티로 볼 수 있다.)

준영속 엔티티를 수정하는 2가지 방법

- 변경 감지 기능 사용
- 병합(`merge`) 사용

변경 감지 기능 사용

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

영속성 컨텍스트에서 엔티티를 다시 조회한 후에 데이터를 수정하는 방법

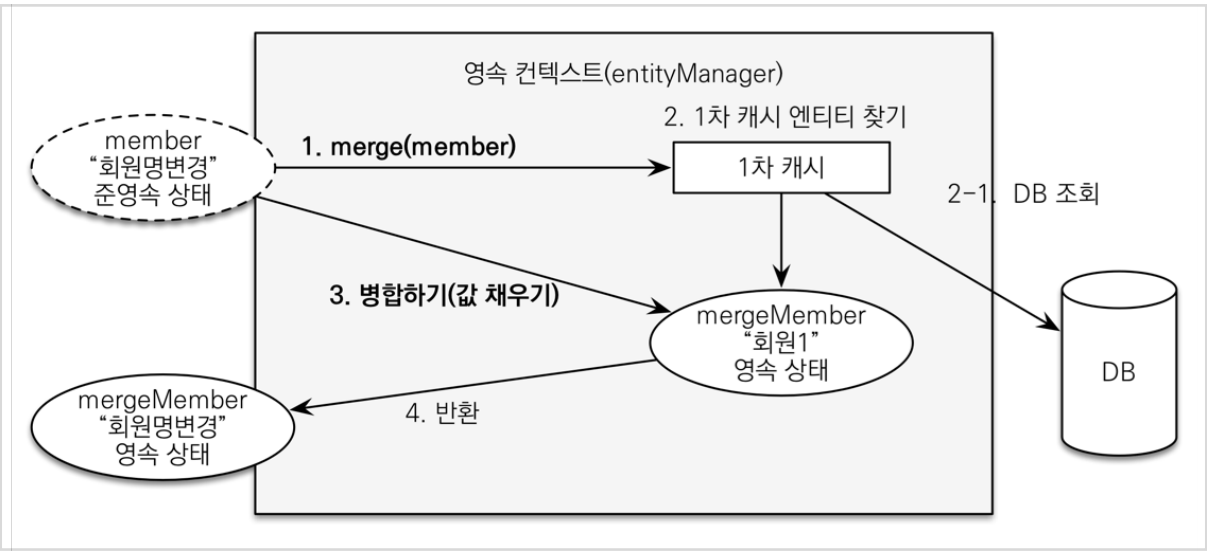
트랜잭션 안에서 엔티티를 다시 조회, 변경할 값 선택 ➜ 트랜잭션 커밋 시점에 변경 감지(Dirty Checking)이 동작해서 데이터베이스에 UPDATE SQL 실행

병합 사용

병합은 준영속 상태의 엔티티를 영속 상태로 변경할 때 사용하는 기능이다.

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

병합: 기존에 있는 엔티티



병합 동작 방식

1. `merge()`를 실행한다.
2. 파라미터로 넘어온 준영속 엔티티의 식별자 값으로 1차 캐시에서 엔티티를 조회한다.
   2-1. 만약 1차 캐시에 엔티티가 없으면 데이터베이스에서 엔티티를 조회하고, 1차 캐시에 저장한다.
3. 조회한 영속 엔티티(`mergeMember`)에 `member` 엔티티의 값을 채워 넣는다. (member 엔티티의 모든 값을 mergeMember에 밀어 넣는다. 이때 mergeMember의 "회원1"이라는 이름이 "회원명변경"으로 바뀐다.)
4. 영속 상태인 mergeMember를 반환한다.

> 참고: 책 자바 ORM 표준 JPA 프로그래밍 3.6.5

병합시 동작 방식을 간단히 정리

1. 준영속 엔티티의 식별자 값으로 영속 엔티티를 조회한다.
2. 영속 엔티티의 값을 준영속 엔티티의 값으로 모두 교체한다.(병합한다.)
3. 트랜잭션 커밋 시점에 변경 감지 기능이 동작해서 데이터베이스에 UPDATE SQL이 실행

> 주의: 변경 감지 기능을 사용하면 원하는 속성만 선택해서 변경할 수 있지만, 병합을 사용하면 모든 속성이 변경된다. 병합시 값이 없으면 `null`로 업데이트 할 위험도 있다. (병합은 모든 필드를 교체한다.)

상품 리포지토리의 저장 메서드 분석 `ItemRepository`

```
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);
        }
    }
    //...
}
```

- `save()` 메서드는 식별자 값이 없으면(`null`) 새로운 엔티티로 판단해서 영속화(persist)하고 식별자가 있으면 병합(merge)
- 지금처럼 준영속 상태인 상품 엔티티를 수정할 때는 `id` 값이 있으므로 병합 수행

새로운 엔티티 저장과 준영속 엔티티 병합을 편리하게 한번에 처리

상품 리포지토리에선 `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.

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.

> 참고: `save()` 메서드는 식별자를 자동 생성해야 정상 동작한다. 여기서 사용한 `Item` 엔티티의 식별자는 자동으로 생성되도록 `@GeneratedValue` 를 선언했다. 따라서 식별자 없이 `save()` 메서드를 호출하면 `persist()` 가 호출되면서 식별자 값이 자동으로 할당된다. 반면에 식별자를 직접 할당하도록 `@Id` 만 선언했다고 가정하자. 이 경우 식별자를 직접 할당하지 않고, `save()` 메서드를 호출하면 식별자가 없는 상태로 `persist()` 를 호출한다. 그러면 식별자가 없다는 예외가 발생한다.

> 참고: 실무에서는 보통 업데이트 기능이 매우 재한적이다. 그런데 병합은 모든 필드를 변경해버리고, 데이터가 없으면 `null` 로 업데이트 해버린다. 병합을 사용하면서 이 문제를 해결하려면, 변경 폼 화면에서 모든 데이터를 항상 유지해야 한다. 실무에서는 보통 변경가능한 데이터만 노출하기 때문에, 병합을 사용하는 것이 오히려 번거롭다.

가장 좋은 해결 방법

엔티티를 변경할 때는 항상 변경 감지를 사용하세요

- 컨트롤러에서 어설프게 엔티티를 생성하지 마세요.
- 트랜잭션이 있는 서비스 계층에 식별자(`id`)와 변경할 데이터를 명확하게 전달하세요.(파라미터 or dto)
- 트랜잭션이 있는 서비스 계층에서 영속 상태의 엔티티를 조회하고, 엔티티의 데이터를 직접 변경하세요.
- 트랜잭션 커밋 시점에 변경 감지가 실행됩니다.

```
@Controller
@RequiredArgsConstructor
public class ItemController {

    private final ItemService itemService;

    /**
     * 상품 수정, 권장 코드
     */
    @PostMapping(value = "/items/{itemId}/edit")
    public String updateItem(@ModelAttribute("form") BookForm form) {
        itemService.updateItem(form.getId(), form.getName(), form.getPrice());
        return "redirect:/items";
    }
}
```

```
package jpabook.jpashop.service;

@Service
@RequiredArgsConstructor
```

```
public class ItemService {

    private final ItemRepository itemRepository;

    /**
     * 영속성 컨텍스트가 자동 변경
     */
    @Transactional
    public void updateItem(Long id, String name, int price) {
        Item item = itemRepository.findOne(id);
        item.setName(name);
        item.setPrice(price);
    }
}
```

상품 주문

상품 주문 컨트롤러

```
package jpabook.jpashop.web;

import jpabook.jpashop.domain.Member;
import jpabook.jpashop.domain.Order;
import jpabook.jpashop.domain.OrderSearch;
import jpabook.jpashop.domain.item.Item;
import jpabook.jpashop.service.ItemService;
import jpabook.jpashop.service.MemberService;
import jpabook.jpashop.service.OrderService;
import lombok.RequiredArgsConstructor;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@Controller
```

```java
@RequiredArgsConstructor
public class OrderController {

    private final OrderService orderService;
    private final MemberService memberService;
    private final ItemService itemService;

    @GetMapping(value = "/order")
    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";
    }

    @PostMapping(value = "/order")
    public String order(@RequestParam("memberId") Long memberId,
@RequestParam("itemId") Long itemId, @RequestParam("count") int count) {

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

}
```

주문 폼 이동

- 메인 화면에서 상품 주문을 선택하면 `/order` 를 GET 방식으로 호출
- `OrderController` 의 `createForm()` 메서드
- 주문 화면에는 주문할 고객정보와 상품 정보가 필요하므로 `model` 객체에 담아서 뷰에 넘겨줌

주문 실행

- 주문할 회원과 상품 그리고 수량을 선택해서 Submit 버튼을 누르면 `/order` URL을 POST 방식으로 호출
- 컨트롤러의 `order()` 메서드를 실행

- 이 메서드는 고객 식별자(`memberId`), 주문할 상품 식별자(`itemId`), 수량(`count`) 정보를 받아서 주문 서비스에 주문을 요청
- 주문이 끝나면 상품 주문 내역이 있는 `/orders` URL로 리다이렉트

상품 주문 폼(`order/orderForm`)

```
<!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org">
<head th:replace="fragments/header :: header" />
<body>

<div class="container">
    <div th:replace="fragments/bodyHeader :: bodyHeader"/>

    <form role="form" action="/order" method="post">

        <div class="form-group">
            <label for="member">주문회원</label>
            <select name="memberId" id="member" class="form-control">
                <option value="">회원선택</option>
                <option th:each="member : ${members}"
                        th:value="${member.id}"
                        th:text="${member.name}" />
            </select>
        </div>

        <div class="form-group">
            <label for="item">상품명</label>
            <select name="itemId" id="item" class="form-control">
                <option value="">상품선택</option>
                <option th:each="item : ${items}"
                        th:value="${item.id}"
                        th:text="${item.name}" />
            </select>
        </div>

        <div class="form-group">
            <label for="count">주문수량</label>
            <input type="number" name="count" class="form-control" id="count"
```

```
placeholder="주문 수량을 입력하세요">
        </div>

        <button type="submit" class="btn btn-primary">Submit</button>
    </form>
    <br/>
    <div th:replace="fragments/footer :: footer" />

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

</body>
</html>
```

주문 목록 검색, 취소

주문 목록 검색 컨트롤러

```
package jpabook.jpashop.web;

@Controller
@RequiredArgsConstructor
public class OrderController {

    @GetMapping(value = "/orders")
    public String orderList(@ModelAttribute("orderSearch") OrderSearch
orderSearch, Model model) {

        List<Order> orders = orderService.findOrders(orderSearch);
        model.addAttribute("orders", orders);

        return "order/orderList";
    }

}
```

주문 목록 검색 화면(`order/orderList`)

```
<!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org">
<head th:replace="fragments/header :: header"/>
<body>

<div class="container">

    <div th:replace="fragments/bodyHeader :: bodyHeader"/>

    <div>
        <div>
            <form th:object="${orderSearch}" class="form-inline">
                <div class="form-group mb-2">
                    <input type="text" th:field="*{memberName}" class="form-
control" placeholder="회원명"/>
                </div>
                <div class="form-group mx-sm-1 mb-2">
                    <select th:field="*{orderStatus}" class="form-control">
                        <option value="">주문상태</option>
                        <option th:each=
                "status : ${T(jpabook.jpashop.domain.OrderStatus).values()}"
                                th:value="${status}"
                                th:text="${status}">option
                        </option>
                    </select>
                </div>
                <button type="submit" class="btn btn-primary mb-2">검색</button>
            </form>
        </div>

        <table class="table table-striped">
            <thead>
            <tr>
                <th>#</th>
                <th>회원명</th>
                <th>대표상품 이름</th>
                <th>대표상품 주문가격</th>
```

```
                <th>대표상품 주문수량</th>
                <th>상태</th>
                <th>일시</th>
                <th></th>
            </tr>
            </thead>
            <tbody>
            <tr th:each="item : ${orders}">
                <td th:text="${item.id}"></td>
                <td th:text="${item.member.name}"></td>
                <td th:text="${item.orderItems[0].item.name}"></td>
                <td th:text="${item.orderItems[0].orderPrice}"></td>
                <td th:text="${item.orderItems[0].count}"></td>
                <td th:text="${item.status}"></td>
                <td th:text="${item.orderDate}"></td>
                <td>
                    <a th:if="${item.status.name() == 'ORDER'}" href="#"
th:href="'javascript:cancel('+${item.id}+')'"
                       class="btn btn-danger">CANCEL</a>
                </td>
            </tr>

            </tbody>
        </table>
    </div>

    <div th:replace="fragments/footer :: footer"/>

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

</body>
<script>
    function cancel(id) {
        var form = document.createElement("form");
        form.setAttribute("method", "post");
        form.setAttribute("action", "/orders/" + id + "/cancel");
        document.body.appendChild(form);
        form.submit();
    }
```

```
</script>
</html>
```

주문 취소

```
package jpabook.jpashop.web;

@Controller
@RequiredArgsConstructor
public class OrderController {

    @PostMapping(value = "/orders/{orderId}/cancel")
    public String cancelOrder(@PathVariable("orderId") Long orderId) {

        orderService.cancelOrder(orderId);

        return "redirect:/orders";
    }
}
```