JPA활용-REST API개발과 성능 최적화

백명숙(msbaek@hucloud.co.kr)

목 차

백명숙(msbaek@hucloud.co.kr)

 백명숙(msbaek@hucloud.co.kr)

백명숙(msbaek@hucloud.co.kr)

백명숙(msbaek@hucloud.co.kr)

제 1 장 프로젝트 환경 설정

1.1 Spring Boot 프로젝트 생성

1.1.1 스프링 부트 스타터

(https://start.spring.io/)
Project : Gradle Project
Language : Java
Spring Boot : 2.7.X
Project Metadata
   Group : jpastudy
   Artifact, Name : jpashop
   Packaging : Jar
Dependencies
   : web, jpa, maria, lombok, validation

백명숙(msbaek@hucloud.co.kr)

1.1.2 build.gradle – Gradle 전체 설정

```
plugins {
    id 'org.springframework.boot' version '2.7.6'
    id 'io.spring.dependency-management' version '1.0.15.RELEASE'
    id 'java'
}
group = 'jpastudy'
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-web'
  compileOnly 'org.projectlombok:lombok'
  runtimeOnly 'org.mariadb.jdbc:mariadb-java-client'
  annotationProcessor 'org.projectlombok:lombok'
  testImplementation 'org.springframework.boot:spring-boot-starter-test'
}
test {
  useJUnitPlatform()
}
```

[코드 1.1] build.gradle

백명숙(msbaek@hucloud.co.kr)

1.2 스프링 부트 라이브러리 살펴보기

- 핵심 라이브러리
  - 스프링 MVC
  - 스프링 ORM
  - 스프링 데이터 JPA
- 기타 라이브러리
  - H2 데이터베이스 클라이언트
  - Connection Pool : 부트 기본은 HikariCP
  - WEB(Thymeleaf)
  - 로깅 SLF4J & Logback
  - 테스트

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

1.3 JPA 와 DB 설정, 동작확인

1.3.1 application 설정파일 (application.yml)

```
spring: #띄어쓰기 없음
  datasource: #띄어쓰기 2칸
    url: jdbc:mariadb://127.0.0.1:3306/boot_db #4칸
    username: boot
    password: boot
    driver-class-name: org.mariadb.jdbc.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
```

[코드 1.2] main/resources/application.yml

- spring.jpa.hibernate.ddl-auto: create

이 옵션은 애플리케이션 실행 시점에 테이블을 drop 하고, 다시 생성한다.
참고: 모든 로그 출력은 로거를 통해 남겨야 한다.
> show_sql : 옵션은 System.out 에 하이버네이트 실행 SQL을 남긴다.
> org.hibernate.sql : 옵션은 logger를 통해 하이버네이트 실행 SQL을 남긴다.
> application.yml 파일은 띄어쓰기(스페이스) 2칸으로 계층을 만듭니다. 따라서 띄어쓰기 2칸을 필수로 적어 주어야 합니다.
> 예를 들어서 아래의 datasource 는 spring: 하위에 있고 앞에 띄어쓰기 2칸이 있으므로 spring.datasource 가 됩니다.

1.4 Query 파라미터 로그 남기기

1.4.1 외부 라이브러리 사용

로그에 다음을 추가하기 org.hibernate.type : SQL 실행 파라미터를 로그로 남긴다.

- 외부 라이브러리 사용
  https://github.com/gavlyukovskiy/spring-boot-data-source-decorator

- build.gradle에 추가

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

[코드 1.3] build.gradle

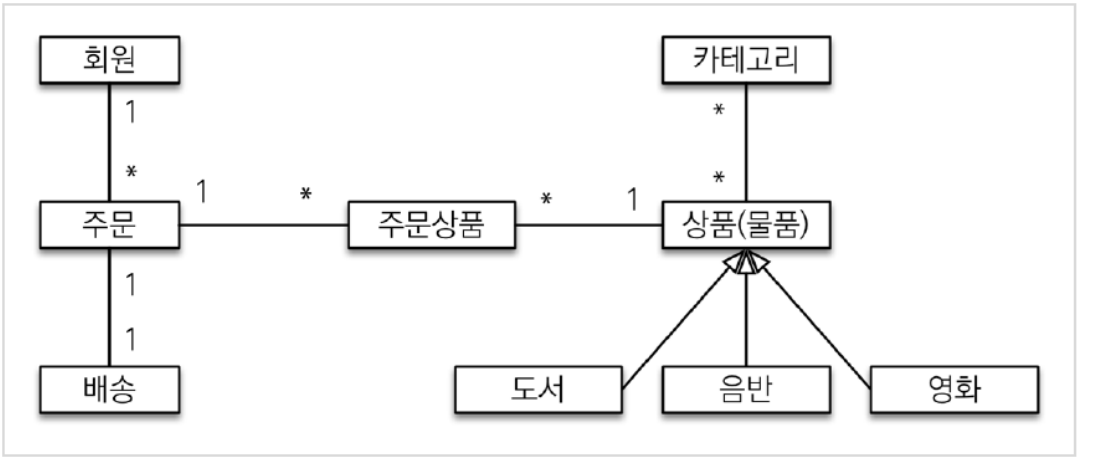
제 2 장 도메인 분석 설계

2.1 요구사항 분석

2.1.1 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.

2.2 도메인 모델과 테이블 설계



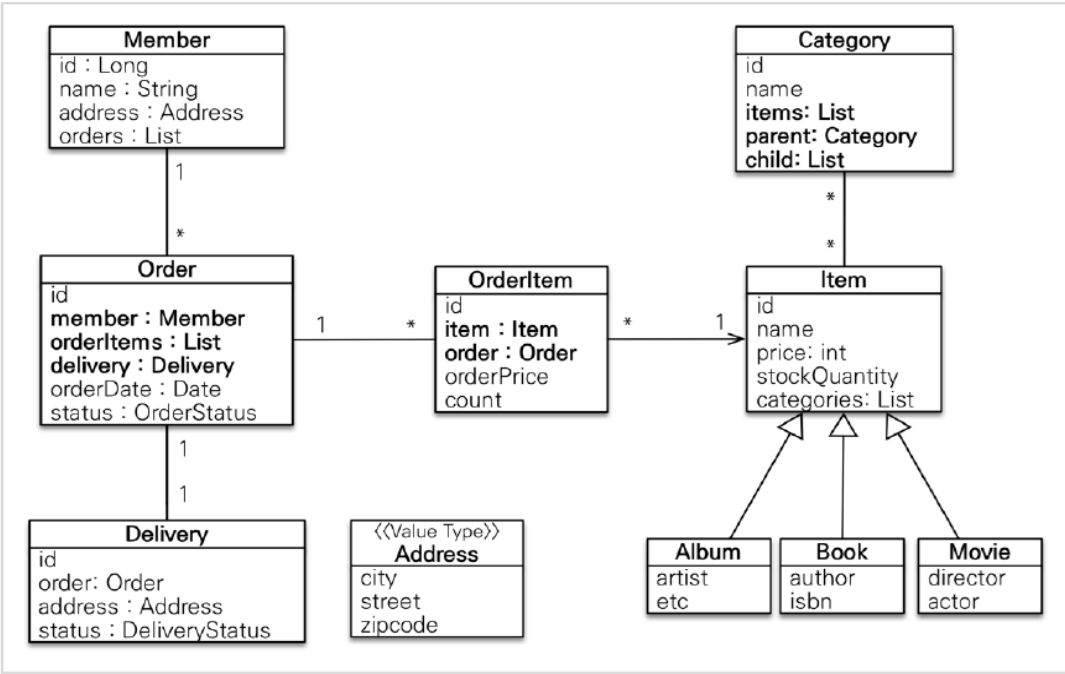
회원, 주문, 상품의 관계

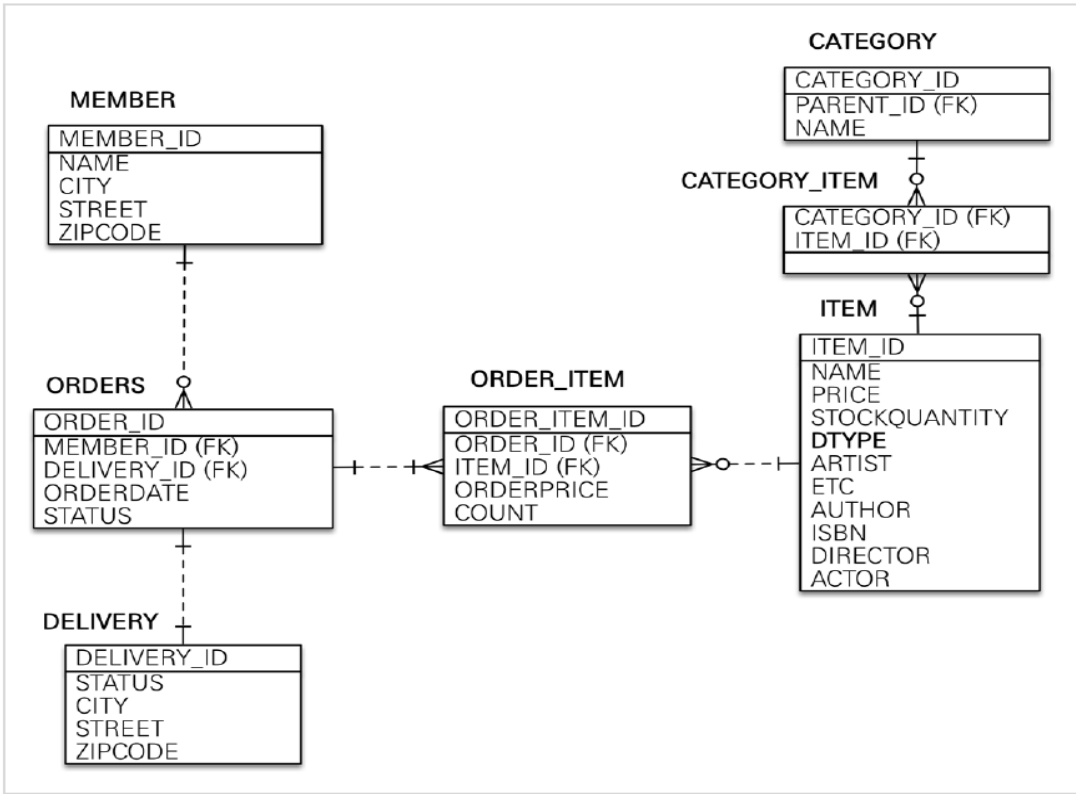
: 회원은 여러 상품을 주문할 수 있다.

: 한 번 주문할 때 여러 상품을 선택할 수 있으므로 주문과 상품은 다대다 관계이므로 주문상품이라는 엔티티를 추가해서 다대다 관계를 일대다, 다대일 관계로 설정했다.

상품 분류

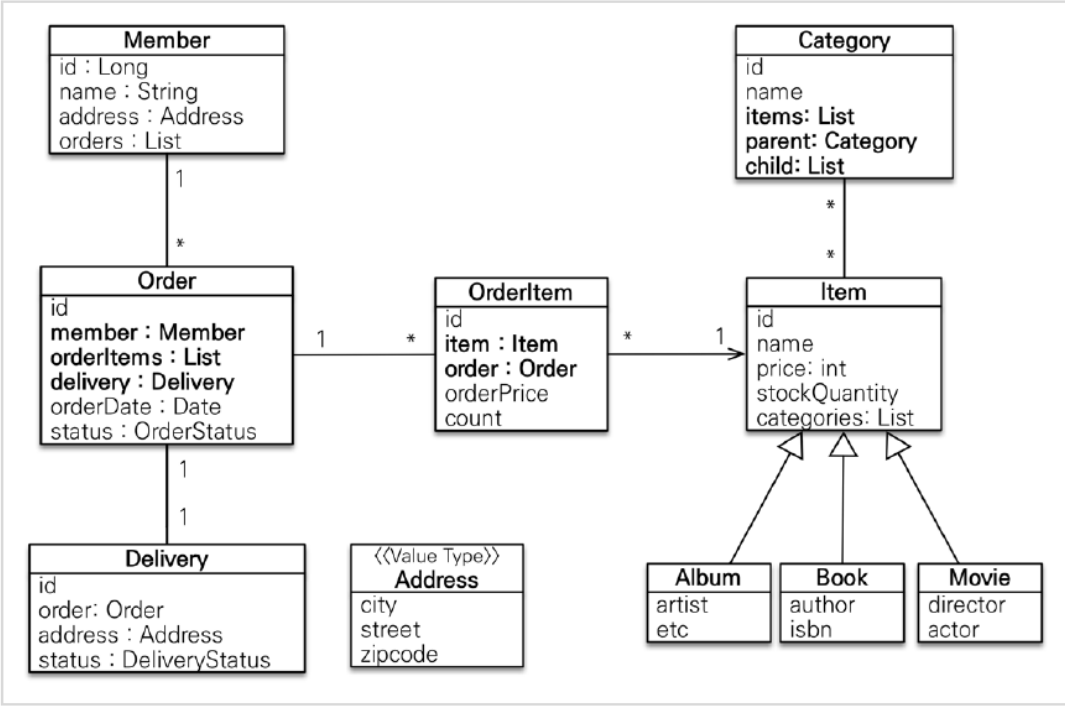
: 상품은 도서, 음반, 영화로 구분되는데 상품이라는 공통 속성을 사용하므로 상속 구조로 설정했다.





백명숙(msbaek@hucloud.co.kr)

2.2.1 엔티티 분석



회원(Member)

: 이름(name)과 임베디드 타입인 주소(address), 주문(orders) 리스트를 가진다.

주문(Order)

: 한 번 주문시 여러 상품을 주문할 수 있으므로 주문(Order)과 주문상품(OrderItem)은 일대다 관계이다.

: 주문은 상품을 주문한 회원(member)과 배송정보(delivery), 주문 날짜(orderDate), 주문 상태(status)를 가지고 있다.

: 주문 상태는 열거형을 사용하며 주문(ORDER), 취소(CANCEL) 상태를 표현할 수 있다.

주문상품(OrderItem)

: 주문한 상품(item)과 주문금액(orderPrice), 주문수량(count) 정보를 가지고 있다. (보통 OrderItem, LineItem 으로 많이 표현한다.)

상품(Item)

: 이름(name), 가격(price), 재고수량(stockQuantity)을 가지고 있다. 상품을 주문하면 재고 수량이 줄어든다.

: 상품의 종류로는 도서, 음반, 영화가 있는데 종류별로 사용 하는 속성이 조금씩 다르다.

배송(Delivery)

: 주문시 하나의 배송 정보를 생성한다. 주문과 배송은 일대일 관계다.

카테고리(Category)

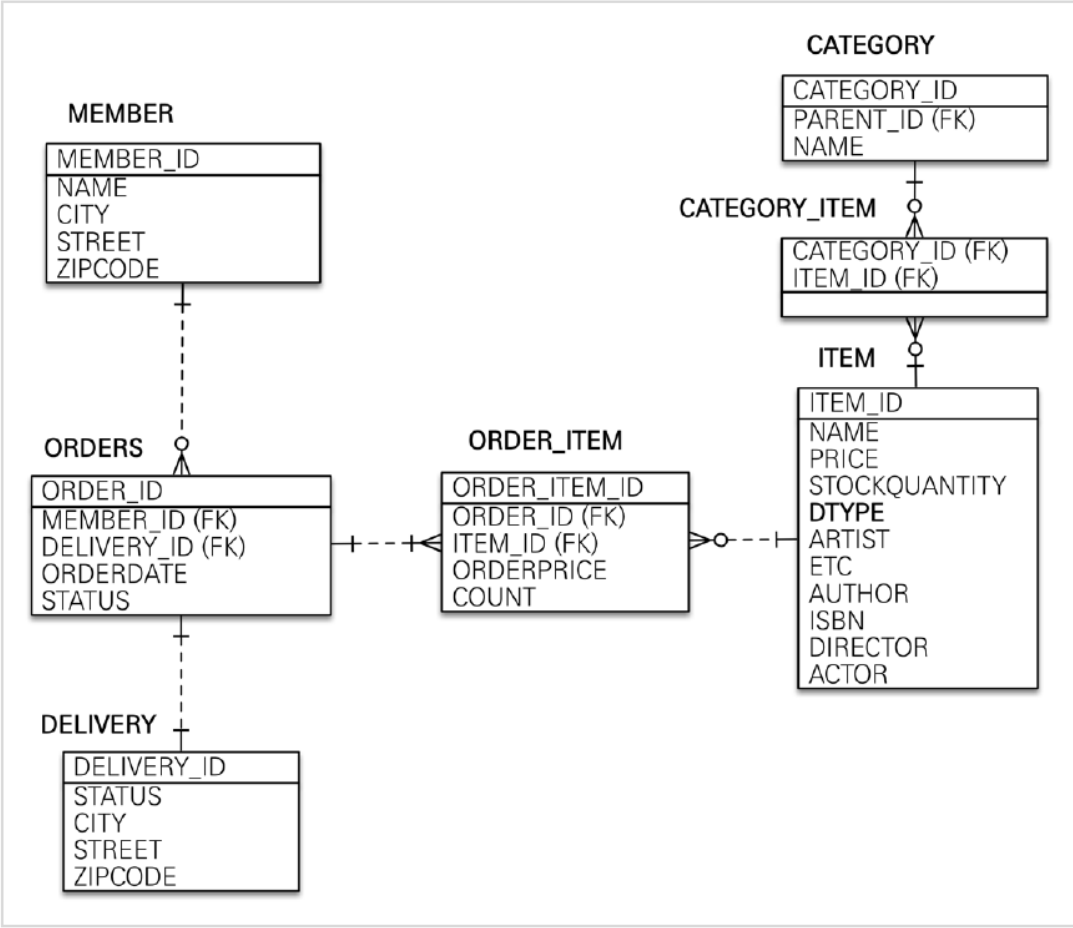
: 상품과 다대다 관계를 맺는다. parent, child 로 부모, 자식 카테고리를 연결한다.

주소(Address)

: 값 타입(임베디드 타입)이다. 회원(Member)과 배송(Delivery)에서 사용한다.

* 참고: 회원이 주문을 하기 때문에, 회원이 주문리스트를 가지는 것은 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.

2.2.2 테이블 분석



MEMBER: 회원 엔티티의 Address 임베디드 타입 정보가 회원 테이블에 그대로 포함되었다. 이것은 DELIVERY 테이블도 마찬가지이다.

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

> 참고: 테이블명이 ORDER 가 아니라 ORDERS 인 것은 데이터베이스가 order by 를 예약어로 사용하므로 관례상 ORDERS 를 많이 사용한다.

> 참고: 실제 코드에서는 DB에 소문자 + _(언더스코어) 스타일을 사용합니다.

> 데이터베이스 테이블명, 컬럼명에 대한 관례는 회사마다 다르다. 보통은 대문자 + _(언더스코어)나 소문자 + _(언더스코어) 방식 중에 하나를 지정해서 일관성 있게 사용한다. **코드에서는 소문자 + _(언더스코어) 스타일을 사용합니다.**

2.2.3 연관관계 매핑 분석

회원과 주문: 일대다, 다대일의 양방향 관계이다.
외래 키가 있는 주문(Order)에 있으므로 주문이 연관관계의 주인이다.
그러므로 Order.member 를 ORDERS.MEMBER_ID 외래 키와 매핑한다.

주문상품과 주문: 다대일 양방향 관계이다.
외래 키가 주문상품(OrderItem)에 있으므로 주문상품이 연관관계의 주인이다.
그러므로 OrderItem.order 를 ORDER_ITEM.ORDER_ID 외래 키와 매핑한다.

주문상품과 상품: 다대일 단방향 관계이다.
외래키가 주문상품(OrderItem)에 있으므로 주문상품이 연관관계의 주인이다.
OrderItem.item 을 ORDER_ITEM.ITEM_ID 외래 키와 매핑한다.

주문과 배송: 일대일 양방향 관계이다.
외래키가 주문(Order)에 있으므로 주문이 연관관계의 주인이다.
Order.delivery 를 ORDERS.DELIVERY_ID 외래 키와 매핑한다.

카테고리와 상품: @ManyToMany 를 사용해서 매핑한다. (실무에서 @ManyToMany 는 사용하지 않는 것이 바람직하다. 여기서는 다대다 관계를 예제로 보여주기 위해 추가했을 뿐이다)

참고: 외래 키가 있는 곳을 연관관계의 주인으로 정해라.
> 연관관계의 주인은 단순히 외래 키를 누가 관리 하느냐의 문제입니다.
업무적으로 우위에 있다고 연관관계의 주인으로 정하면 안됩니다.

예를 들어서 자동차와 바퀴가 있으면, 일대다 관계에서 항상 다 쪽에 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 있습니다.

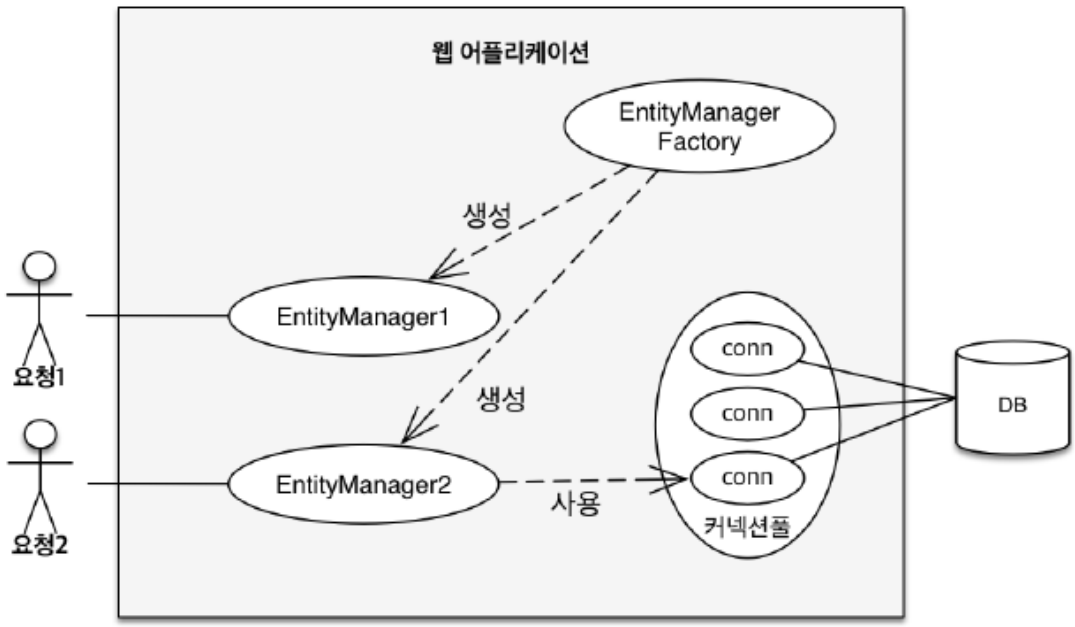
2.3 JPA 기본 개념들

2.3.1 영속성(Persistence) 관리

1) 엔티티 매니저 팩토리와 엔티티 매니저

: 웹 애플리케이션이 실행될 때 엔티티 매니저가 생성되고, 클라이언트의 요청이 올 때마다 엔티티 매니저 팩토리를 통해서 엔티티 매니저를 생성한다.

: 생성된 엔티티 매니저는 내부적으로 DB 커넥션을 사용해서 DB에 접근한다.



2) 영속성 컨텍스트

: "엔티티를 영구 저장하는 환경"이라는 뜻이며 엔티티 매니저를 통해서 영속성 컨텍스트에 접근한다.

: EntityManager.persist(entity)는 엔티티 객체를 DB가 아니라 JPA의 영속성 컨텍스트에 저장하는 것이다.

3) 영속성 컨텍스트와 데이터베이스 저장

: JPA는 트랜잭션을 커밋하는 순간 영속성 컨텍스트에 저장된 엔티티를 DB에 반영한다. 이것을 플러시(flush) 라고 한다.

4) 영속성 컨텍스트가 엔티티를 관리할 때 사용하는 전략

: 1차 캐시, 동일성(identity) 보장

: 변경 감지(Dirty Checking), 지연로딩(Lazy Loading)

2.3.2 엔티티의 생명주기 (Life Cycle)

- 비영속 (new/transient)

: 영속성 컨텍스트와 전혀 관계가 없는 새로운 상태

- 영속 (managed)

: 영속성 컨텍스트에 관리되는 상태

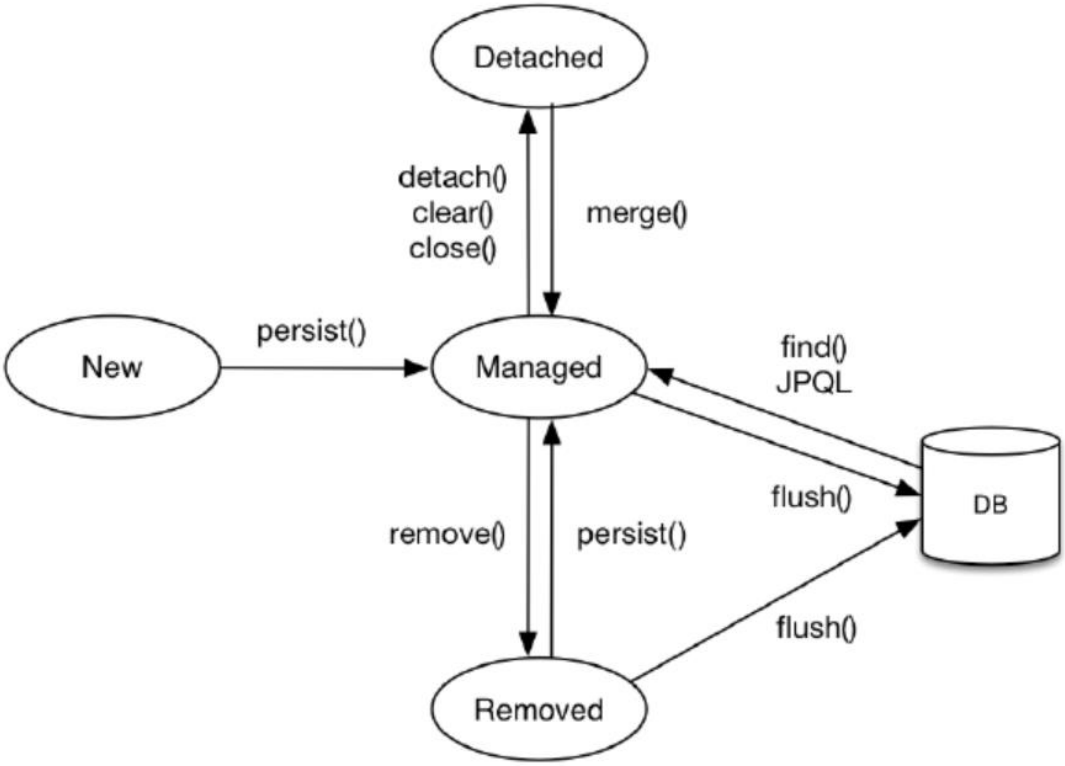
em.persist(entity)

- 준영속 (detached)

: 영속성 컨텍스트에 저장되었다가 분리된 상태

- 삭제 (removed)

: 삭제된 상태



백명숙(msbaek@hucloud.co.kr)

2.3.2.1 비영속 (new / transient)

- 객체를 생성한 후 아직 영속성 컨텍스트에 저장되지 않은 상태
- 단순히 자바 객체를 생성하고 초기화만 해준 상태
- JPA에 의해 관리되고 있지 않는 상태이다


```
//객체를 생성한 상태 (비영속)
Member member = new Member();
member.setId("member1");
member.setUsername("회원1");
```

2.3.2.2 영속(managed)

- 엔티티 매니저를 통해 영속성 컨텍스트에 접근한다.
- 엔티티 매니저를 통해 member 엔티티 객체를 영속성 컨텍스트에 저장한다.
- 영속성 컨텍스트 안에서 member 엔티티 객체가 관리되고 있다.
- 영속상태가 된다고 해서 DB에 저장되는 것이 아니다. DB에는 커밋을 해야 저장된다.


```
//객체를 생성한 상태 (비영속)
Member member = new Member();
member.setId("member1");
member.setUsername("회원1");
```

```
//JPA 영속성 컨텍스트로의 접근은 엔티티 매니저를 통해서 한다
EntityManger em = emf.createEntityManger();
//JPA의 모든 데이터 변경은 트랜잭션 안에서 일어난다
em.getTransaction.begin();
//객체를 저장한 상태 (영속)
em.persist(member);
```

2.3.2.3 준영속(detached)

- 영속 -> 준영속
- 영속 상태의 엔티티가 영속성 컨텍스트에서 분리 (detached)
- 영속성 컨텍스트가 제공하는 기능을 사용하지 못한다

```
//회원 엔티티를 영속성 컨텍스트에서 분리, 준영속 상태
em.detach(member);
```

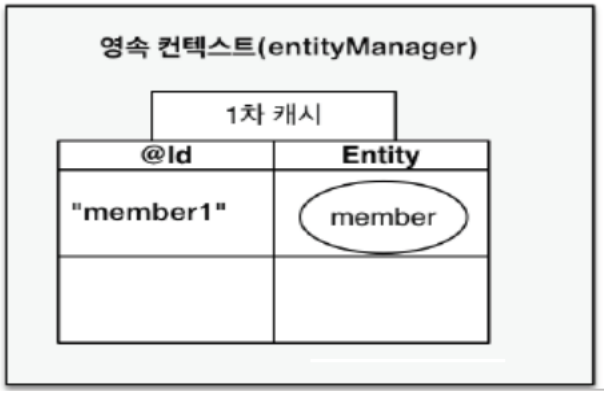
- 준영속 상태로 만드는 방법

em.detach(entity) : 특정 엔티티만 준영속 상태로 전환

em.clear() : 영속성 컨텍스트를 완전히 초기화

2.3.2.4 엔티티 조회

- 영속성 컨텍스트는 내부에 캐시를 가지고 있다. 이 캐시를 1차 캐시라 하며 영속 상태의 엔티티는 모두 이곳에 저장된다.
- 영속성 컨텍스트 내부에 Map이 존재하며, 키는 @Id, 값은 엔티티 인스턴스이다.




```
// 엔티티 생성(비영속)
Memeber member = new Member();
member.setId("member1");
```

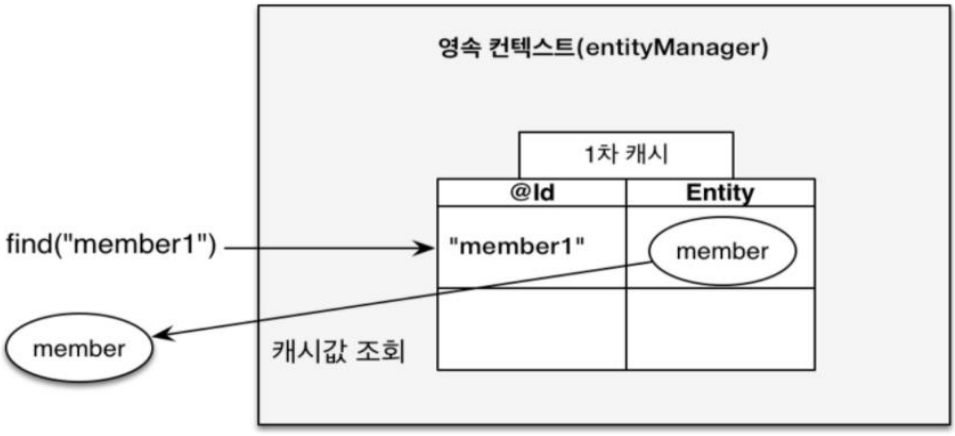
```
member.setUsername("회원1");
// 엔티티 영속
em.persis(member);
```

1) 엔티티 1차 캐시에서 조회

```
Member member = em.find(Member.class, "member1");
```

em.find(member1) 호출시, 우선 1차 캐시에서 엔티티를 찾음

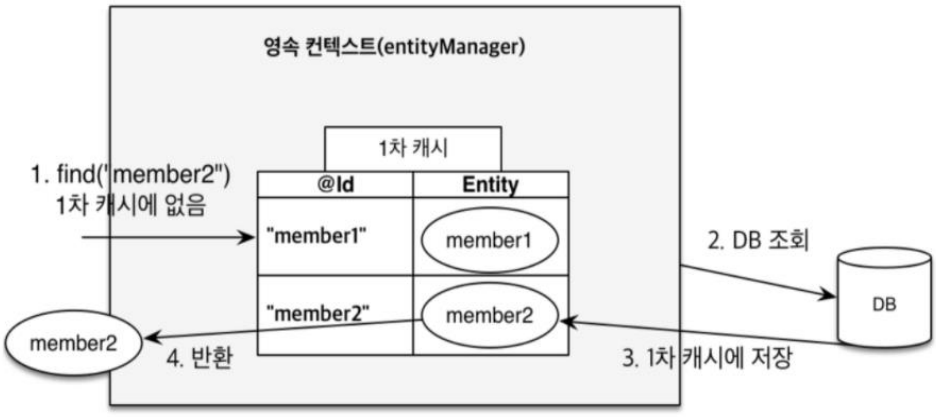
찾는 엔티티가 있으면 데이터베이스에서 조회하지 않고, 메모리에 있는 1차 캐시에서 엔티티를 조회한다.



2) 데이터베이스에서 조회

```
Member member2 = em.find(Member.class, "member2");
```

em.find(member2) 호출시, 1차 캐시에 존재하지 않는다면 엔티티 매니저는 데이터베이스를 조회해서 엔티티를 생성한다. 그리고 1차 캐시에 저장한 후에 영속 상태의 엔티티를 반환하게 된다.

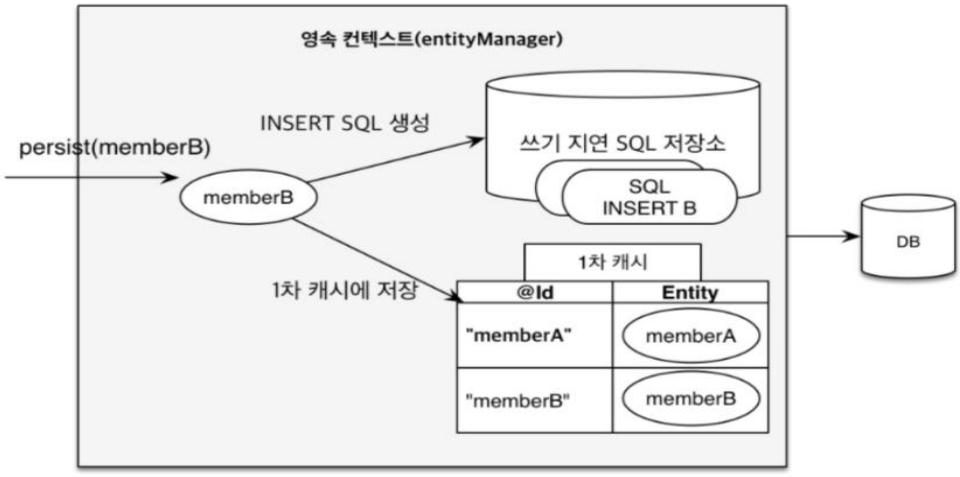


백명숙(msbaek@hucloud.co.kr)

2.3.2.5 엔티티 등록

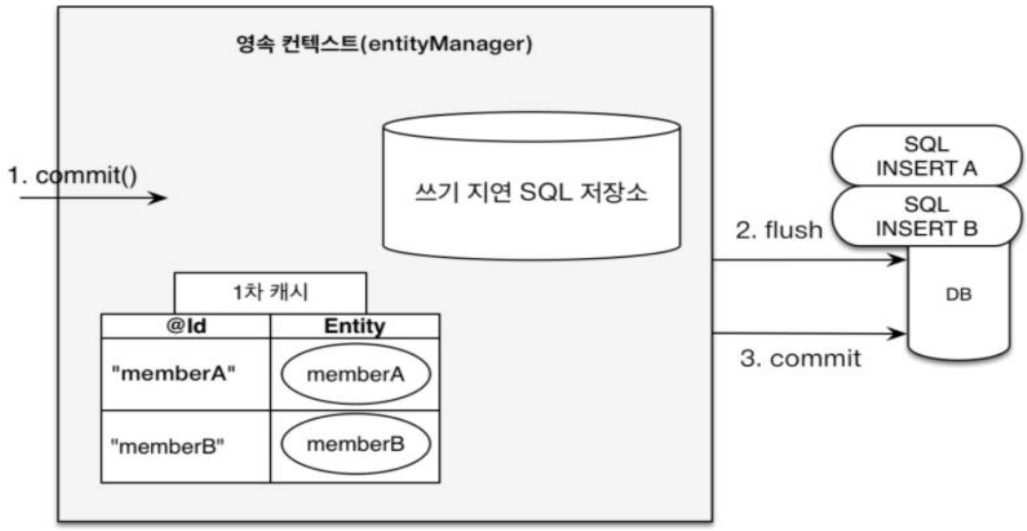
```
EntityManager em = emf.createEntityManager();
EntityTransaction transaction = em.getTransaction();
// 엔티티 매니저는 데이터 변경 시 트랜잭션을 시작해야 한다
transaction.begin(); // [트랜잭션] 시작
em.persist(memberA);
em.persist(memberB);
// 영속 상태이며, DB INSERT SQL을 보내지 않음
transaction.commit(); // [트랜잭션] 커밋
```

엔티티 매니저는 트랜잭션을 커밋하기 전까지 내부 쿼리 저장소에 INSERT SQL를 모아둔 후, 트랜잭션 커밋 시점에 모아둔 쿼리를 데이터베이스에 보낸다.
이것을 트랜잭션을 지원하는 쓰기 지연(transactional write-behind)라고 한다.



INSERT SQL을 쓰기 지연 SQL 저장소에 저장해 둔 다음, 트랜잭션 커밋시점에 아래와 같이 영속성 컨텍스트를 플러시(flush)한다. (영속성 컨텍스트의 변경 내용을 데이터베이스에 동기화하는 작업, 동기화 후 실제 데이터베이스 트랜잭션을 커밋한다.)

백명숙(msbaek@hucloud.co.kr)

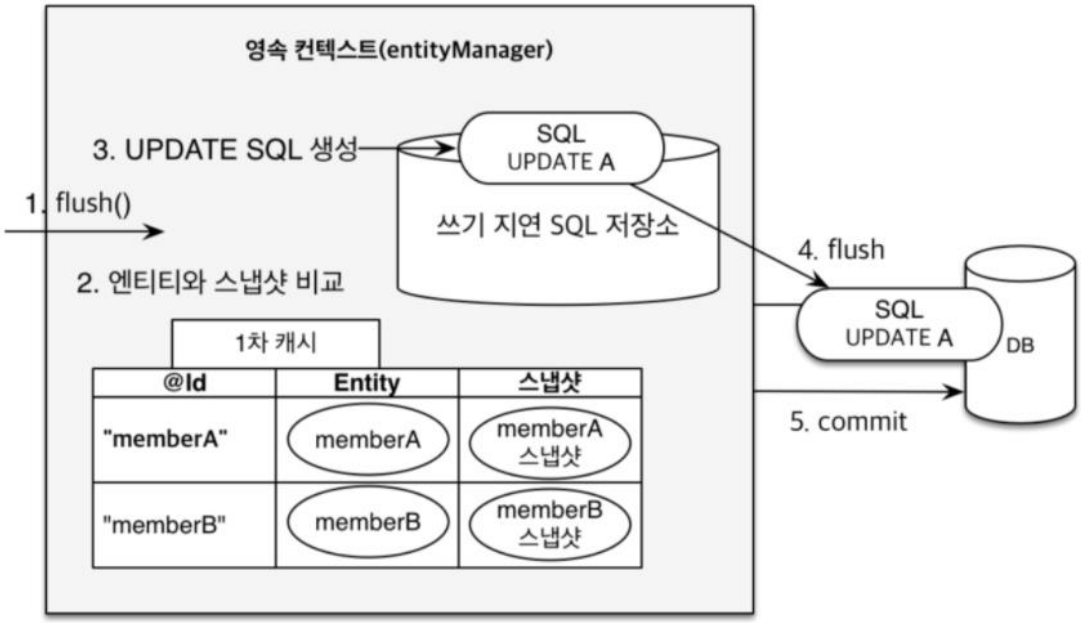


2.3.2.6 엔티티 수정

1) 변경감지

```
EntityManager em = emf.createEntityManager();
EntityTransaction transaction = em.getTransaction();
transaction.begin() // [트랜잭션] 시작
// 영속 엔티티 조회
Member memberA = em.find(Member.class, "memberA");
// 영속 엔티티 데이터 수정
memberA.setUsername("hi");
memberA.setAge(10);
transaction.commit(); // [트랜잭션] 커밋
```

JPA로 엔티티를 수정할 때는 엔티티를 조회한 후 데이터만 변경하면 된다.
즉, JPA는 em.update()와 같은 update 메소드가 존재하지 않는다.
memberA를 setter로만 변경했을 뿐인데 데이터베이스에 반영이 된 이유는 JPA의 변경감지(Dirty Checking) 기능 때문이다.



JPA는 엔티티를 영속성 컨텍스트에 보관할 때, 최초 상태를 복사해서 저장해 두는데 이것을 스냅샷이라고 한다.

1. JPA는 커밋하는 시점에 내부적으로 flush()가 호출된다
2. 영속성 컨텍스트 flush()가 호출되면, JPA는 1차캐시에 저장된 엔티티와 스냅샷을 비교한다
3. 스냅샷은 최초로 1차 캐시에 들어온 상태를 저장해둔 것이다
4. 만약 스냅샷과 다른 부분이 있다면 JPA는 UPDATE 쿼리를 쓰기지연 SQL저장소에 저장한다
5. 마지막으로 해당 쿼리를 DB에 반영하고 (flush)
6. 커밋을 하고 마친다

변경 감지(Dirty Checking)는 영속성 컨텍스트가 관리하는 영속 상태의 엔티티에만 적용된다. (비영속, 준영속 상태는 영속성 컨텍스트의 관리를 받지 못하는 엔티티이기 때문에 값을 변경해도 데이터베이스에 반영되지 않는다.)

2.3.2.7 엔티티 삭제

```
Member memberA = em.find(Member.class, "memberA");
em.remove(memberA);
```

em.remove()에 삭제할 엔티티를 넘겨주면 엔티티를 삭제한다. 물론 엔티티를 즉시 삭제하는 것이 아니라, 엔티티 등록과 비슷하게 삭제 쿼리를 쓰기 지연 SQL 저장소에 등록한다. 이 후 트랜잭션을 커밋하게 되면 플러시가 호출되어 실제 데이터베이스에 삭제 쿼리를 전달하게 된다.
em.remove(memberA)를 호출하는 순간 memberA는 영속성 컨텍스트에서 제거된다.

2.3.2.8 영속성 컨텍스트를 flush 하는 방법

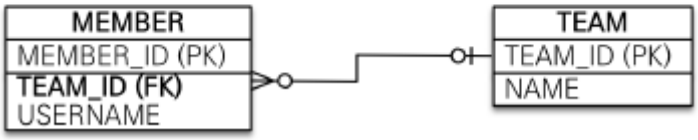
- em.flush : 직접 호출
- 트랜잭션 커밋 : flush 자동 호출
- JPQL 쿼리 실행 : flush 자동 호출

2.3.3 연관관계 기본

2.3.3.1 객체를 테이블에 맞추어 모델링 (연관 관계가 없는 객체)

[객체 연관관계]

[테이블 연관관계]



1) 외래키 식별자를 직접 다룬다.

```java
@Entity
public class Member {

  @Id @GeneratedValue
  private Long id;

  @Column(name = "USERNAME")
  private String name;


  @Column(name = "TEAM_ID")
  private Long teamId;
  …
}
@Entity
public class Team {

  @Id @GeneratedValue
  private Long id;
  private String name;
  …
}
```

```java
//팀 저장
Team team = new Team();
team.setName("TeamA");
em.persist(team);

//회원 저장
Member member = new Member();
member.setName("member1");
member.setTeamId(team.getId());
em.persist(member);
```

2) 식별자로 다시 조회, 객체 지향적인 방법이 아니다.

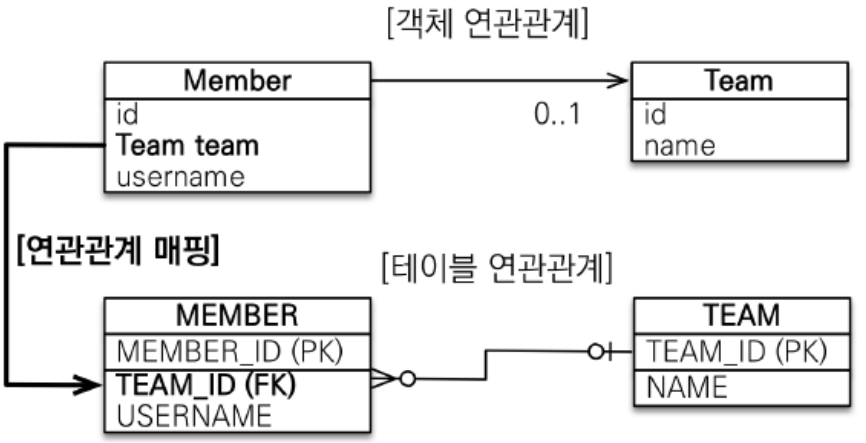
```java
//조회
Member findMember = em.find(Member.class, member.getId());

//연관관계가 없음
Team findTeam = em.find(Team.class, team.getId());
```


백명숙(msbaek@hucloud.co.kr)

3) 객체를 테이블에 맞추어 데이터 중심으로 모델링하면 협력관계를 만들 수 없다.
4) 테이블은 외래 키로 조인을 사용해서 연관된 테이블을 찾는다.
5) 객체는 참조를 사용해서 연관된 객체를 찾는다.

⇨ 테이블과 객체 사이에는 이런 차이점이 있다.

2.3.3.2 단방향 연관관계



1) 객체의 참조와 테이블의 외래 키를 매핑한다.

```java
@Entity
public class Member {

  @Id @GeneratedValue
  private Long id;

  @Column(name = "USERNAME")
  private String name;
  private int age;

//    @Column(name = "TEAM_ID")
//    private Long teamId;

   @ManyToOne
   @JoinColumn(name = "TEAM_ID")
   private Team team;
```

```java
//팀 저장
Team team = new Team();
team.setName("TeamA");
em.persist(team);

//회원 저장
Member member = new Member();
member.setName("member1");
member.setTeam(team); //단
em.persist(member);
```

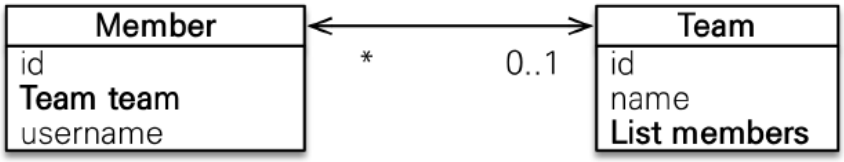
2) 참조로 연관관계를 조회한다 – 객체 그래프 탐색.

```java
//조회
Member findMember = em.find(Member.class, member.getId());

//참조를 사용해서 연관관계 조회
Team findTeam = findMember.getTeam();
```

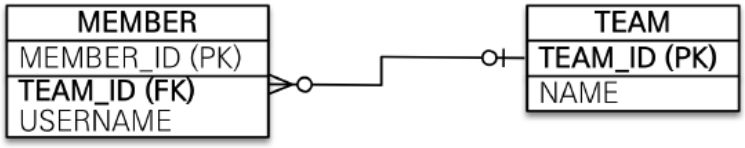

백명숙(msbaek@hucloud.co.kr)

2.3.3.3 양방향 연관관계

[양방향 객체 연관관계]



[테이블 연관관계]



1) Member 엔티티는 단방향과 동일하다.

```
@Entity
public class Member {

  @Id @GeneratedValue
  private Long id;

  @Column(name = "USERNAME")
  private String name;
  private int age;

   @ManyToOne
   @JoinColumn(name = "TEAM_ID")
   private Team team;
```

2) Team 엔티티는 컬렉션 추가

```
@Entity
public class Team {

  @Id @GeneratedValue
  private Long id;

  private String name;

   @OneToMany(mappedBy = "team")
   List<Member> members = new ArrayList<Member>();
  …
}
```


백명숙(msbaek@hucloud.co.kr)

3) 반대 방향으로 객체 그래프 탐색

```
//조회
Team findTeam = em.find(Team.class, team.getId());

int memberSize = findTeam.getMembers().size(); //역방향 조회
```

4) 객체와 테이블이 관계를 맺는 차이

■ 객체 연관관계 = 2개

: 회원 -> 팀 연관관계 1개(단방향) @ManyToOne

: 팀 -> 회원 연관관계 1개(단방향) @OneToMany

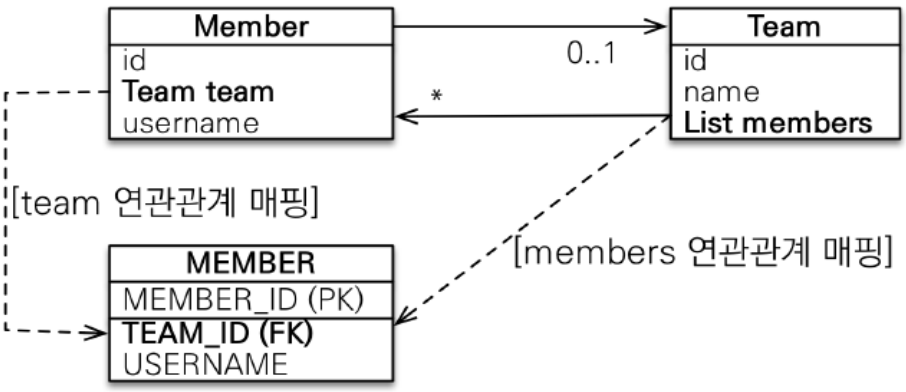
객체의 양방향 관계는 서로 다른 단뱡향 관계가 2개가 있는 것이다.

■ 테이블 연관관계 = 1개

: 회원 <-> 팀의 연관관계 1개(양방향)

테이블은 외래 키 하나로 두 테이블의 연관관계를 관리할 수 있다.

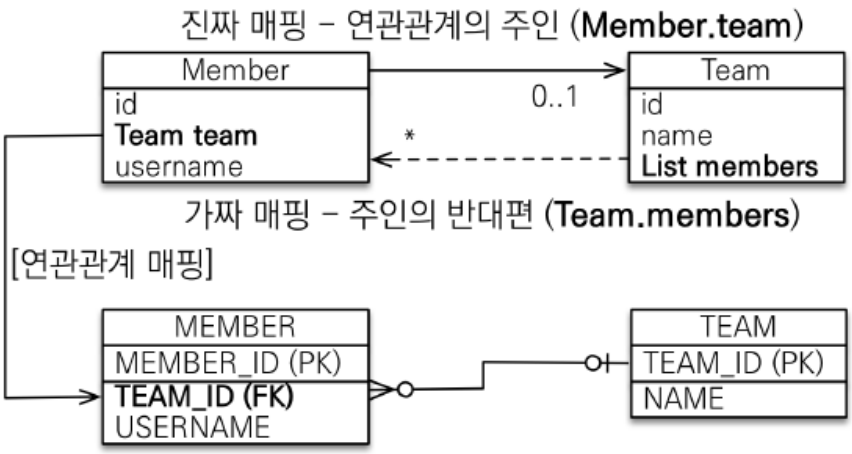
5) 둘 중 하나로 외래 키를 관리해야 한다.



백명숙(msbaek@hucloud.co.kr)

6) 양방향 연관관계의 주인(Owner)

- 객체의 두 관계 중 하나를 연관관계의 주인으로 지정해야 한다.
- 연관관계의 주인만이 외래 키를 관리(등록, 수정)
- 주인이 아닌쪽은 읽기만 가능하고, mappedBy 속성으로 주인을 지정한다.
- 외래 키(Foreign Key)가 있는 있는 곳을 주인으로 정해라
- 여기서는 Member.team이 연관관계의 주인이다.



7) 양방향 매핑시 가장 많이 하는 실수

(연관관계의 주인에 값을 입력하지 않음)

```
Team team = new Team();
team.setName("TeamA");
em.persist(team);

Member member = new Member();
member.setName("member1");

//역방향(주인이 아닌 방향)만 연관관계 설정
team.getMembers().add(member);

em.persist(member);
```

ID	USERNAME	TEAM_ID
1	member1	null

백명숙(msbaek@hucloud.co.kr)

8) 양방향 매핑시 연관관계의 주인에 값을 입력해야 한다.

```
Team team = new Team();
team.setName("TeamA");
em.persist(team);

Member member = new Member();
member.setName("member1");

team.getMembers().add(member);
//연관관계의 주인에 값 설정
member.setTeam(team); //**

em.persist(member);
```

ID	USERNAME	TEAM_ID
1	member1	2

9) 양방향 매핑 정리.

- 단방향 매핑만으로도 이미 연관관계 매핑은 완료
- 양방향 매핑은 반대 방향으로 조회(객체 그래프 탐색) 기능이 추가된 것이다.
- 단방향 매핑을 먼저 해놓고, 양방향 매핑은 필요할 때 추가해도 됩니다.
  (테이블에는 영향을 주지 않음)
- 비즈니스 로직을 기준으로 연관관계의 주인을 선택하면 않됨.
- 연관관계의 주인은 외래 키의 위치를 기준으로 정해야 한다.

백명숙(msbaek@hucloud.co.kr)

2.3.4 즉시로딩(EAGER)과 지연로딩(LAZY)

즉시로딩(FetchType.EAGER) : 엔티티를 조회할 때 연관된 엔티티도 함께 조회한다.
지연로딩(FetchType.LAZY) : 엔티티를 조회할 때 연관된 엔티티를 실제 사용할 때 조회한다. (즉, 연관관계에 있는 엔티티를 같이 가져오기 않고, getter 로 접근할 때 가져온다.)

<JPA 기본 Fetch 전략>
- @ManyToOne, @OneToOne: 즉시로딩(FetchType.EAGER)
- @OneToMany, @ManyToMany : 지연로딩(FetchType.LAZY)

JPA의 기본 Fetch 전략은 연관된 엔티티가 하나이면 즉시로딩(Eager Loading)을 사용하고, 컬렉션이면 지연로딩(Lazy Loading)을 사용한다. 그 이유는 컬렉션을 즉시로딩하면 비용이 많이 들고 많은 데이터를 로딩할 수도 있기 때문이다.
추천하는 방법은 모든 연관관계를 지연로딩으로 구현하고 애플리케이션 개발이 어느정도 완료단계에 왔을 때 사용처를 보고 필요한 곳에만 즉시로딩을 사용하도록 최적화 하는 것이 좋습니다.

<지연로딩>

Member와 Team 사이가 다대일 @ManyToOne 관계로 매핑되어 있는 상황에서 지연로딩(Lazy Loading)으로 설정되어 있다면
Member 엔티티를 조회할 때 Team 엔티티를 같이 조회하지 않는다.
대신 조회한 Member 엔티티의 member변수에 프록시 객체를 넣어둔다.
이 프록시 객체는 실제 사용 될 때까지 데이터베이스를 조회하지 않고 데이터 로딩을 미뤄 지연로딩을 하게 된다.
데이터가 필요한 순간이 되어서야 데이터베이스를 조회해서 프록시 객체를 초기화 한다.

백명숙(msbaek@hucloud.co.kr)

- 로딩 되는 시점에 Lazy 로딩 설정이 되어 있는 Team 엔티티는 프록시 객체로 가져온다.
- 후에 실제 객체를 사용하는 시점에(Team 을 사용하는 시점에) 초기화가 되며 DB 에 쿼리가 나간다.
  - ✓ getTeam()으로 Team 을 조회하면 프록시 객체가 조회가 된다.
  - ✓ getTeam().getXXX()으로 팀의 필드에 접근 할 때, 쿼리가 나간다.

<즉시로딩>

- fetch 타입을 EAGER 로 설정하면 된다.
- 실제 조회할 때 한번의 쿼리로 모두 조회해온다. (실제 Team 을 사용할 때 쿼리 안나가도 된다.)
- 실행 결과를 보면 Team 객체도 프록시 객체가 아니라 실제 객체이다.

<즉시로딩 주의할 점>

가급적 지연 로딩만 사용합니다. 즉시 로딩은 가급적이면 사용하지 않아야 합니다.
왜냐하면 즉시 로딩은 JPQL에서 N+1 문제를 일으킨다.

- N+1 문제는 쿼리를 1개 날렸는데, 추가 쿼리가 N개 나간다는 의미이다.
- em.find()는 PK를 정해 놓고 DB에서 가져오기 때문에 JPA 내부에서 최적화 한다.
- JPQL에서는 입력 받은 query string이 그대로 SQL로 변환 된다.

"select m from Member m" 이 문장으로 당연히 Member만 SELECT 하게 된다.
Member를 가져왔을 때 Member 엔티티의 Team의 fetchType이 EAGER 이면 Member를 가져오고 나서, 그 Member와 연관된 Team을 다시 가져온다.

- JPQL의 fetch join 을 통해서 해당 시점에 한번의 쿼리로 가져와서 쓸 수 있다.

2.3.5 JPA Entity 관련 어노테이션

1) @Entity

: @Entity 어노테이션이 붙은 클래스는 JPA가 관리하는 클래스로, 해당 클래스를 엔티티라고 부른다. JPA를 사용하여 테이블과 매핑해야 할 클래스는 반드시 @Entity를 선언해야 한다.

2) @Id

: 기본키(Primary Key)를 지정한다.

3) @GeneratedValue

: 기본키를 생성하는 방법을 지정한다. @Id 어노테이션과 같이 선언된다.

기본키 생성 전략 - strategy(GenerationType, Optional)

- GenerationType.AUTO: (기본값)

방언(dialect)에 따라 나머지 세 가지 전략을 자동으로 지정한다.

- 데이터베이스에 따라서 IDENTITY, SEQUENCE, TABLE 방법 중 하나를 자동으로 선택 해주는 방법입니다.

- GenerationType.IDENTITY

기본 키 생성을 데이터베이스에 위임한다.

id 값을 null로 하면 DB가 알아서 AUTO_INCREMENT를 수행한다.

- MySQL, PostgresSQL, SQL Server, DB2에서 사용

- GenerationType.SEQUENCE

데이터베이스 Sequence Object를 사용한다. DB Sequence는 유일한 값을 순서대로 생성하는 특별한 데이터베이스 오브젝트 DB가 자동으로 숫자를 만들어준다. @SequenceGenerator가 필요하다.

- Oracle일 경우 SEQUENCE를 자동으로 선택해서 사용

- GenerationType.TABLE

키 생성 전용 테이블 하나를 만들어 데이터베이스 Sequence 를 흉내내는 전략 @TableGenerator가 필요하다

- 테이블을 사용하므로, 데이터베이스 벤더에 상관없이 모든 데이터베이스에 적용이 가능

4) @Embeddable과 @Embedded

: 엔티티 내 참조 클래스의 필드를 그대로 자신의 컬럼으로 만들고 싶다면 @Embedded와 @Embeddable 어노테이션을 사용한다.

: 참조 클래스에는 @Embeddable 어노테이션을 선언하고, 엔티티 내에서

참조되는 필드는 @Embedded 어노테이션을 선언한다.

5) @CascadeType

: 개발도중 PersistentObjectException: detached entity passed to persist 라는 에러메세지가 발생하는 경우가 있다.
: JPA 사용시 자동으로 생성되는 값을 가진 필드에 직접 값을 할당해서 저장하고자 할 때 발생되는 에러이다.

CascadeType의 종류에는 다음과 같은 것들이 있다.

- CascadeType.PERSIST

– 엔티티를 생성하고, 연관 엔티티를 추가하였을 때 persist() 를 수행하면 연관 엔티티도 함께 persist()가 수행된다. 만약 연관 엔티티가 DB에 등록된 키값을 가지고 있다면 detached entity passed to persist Exception이 발생한다.

- CascadeType.MERGE

– 트랜잭션이 종료되고 detach 상태에서 연관 엔티티를 추가하거나 변경된 이후에 부모 엔티티가 merge()를 수행하게 되면 변경사항이 적용된다.(연관 엔티티의 추가 및 수정 모두 반영됨)

- CascadeType.REMOVE

– 삭제 시 연관된 엔티티도 같이 삭제됨

- CascadeType.DETACH

– 부모 엔티티가 detach()를 수행하게 되면, 연관된 엔티티도 detach() 상태가 되어 변경사항이 반영되지 않는다.

- CascadeType.ALL

– 모든 Cascade 적용

6) @Enumerated

: Java enum 타입을 엔티티 클래스의 속성으로 사용할 수 있다.
: @Enumerated 애노테이션에는 두 가지 EnumType이 존재한다.

EnumType.ORDINAL : enum 순서 값을 DB에 저장
EnumType.STRING : enum 이름을 DB에 저장 (권장함)

7) @InheritanceType

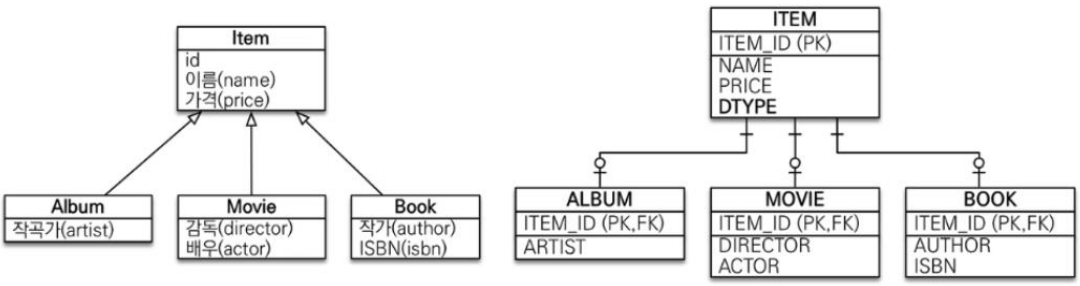
객체는 상속을 지원하므로 모델링과 구현이 똑같지만, DB는 상속을 지원하지 않으므로 논리 모델을 물리 모델로 구현할 방법이 필요하다.

@Inheritance(strategy=InheritanceType.XXX)의 stategy를 설정해주면 된다.

default 전략은 SINGLE_TABLE(단일 테이블 전략)이다.

- InheritanceType 종류
  JOINED
  SINGLE_TABLE
  TABLE_PER_CLASS
- @DiscriminatorColumn(name="DTYPE")
  : 부모 클래스에 선언한다. 하위 클래스를 구분하는 용도의 컬럼이다.
  관례는 default = DTYPE
- @DiscriminatorValue("XXX")
  : 하위 클래스에 선언한다. 엔티티를 저장할 때 슈퍼타입의 구분 컬럼에 저장할 값을 지정한다.
  : 이 어노테이션을 선언하지 않으며 기본값으로 클래스 이름이 들어간다.

⇨ **각각의 테이블로 변환하는 조인 전략**(JOINED)



* JOINED 장점

테이블이 정규화가 되어있고, 외래 키 참조 무결성 제약조건 활용 가능

ITEM의 PK가 ALBUM, MOVIE, BOOK의 PK이며 FK이다.

테이블 정규화로 저장공간이 딱 필요한 만큼 소비되어 저장공간 효율화

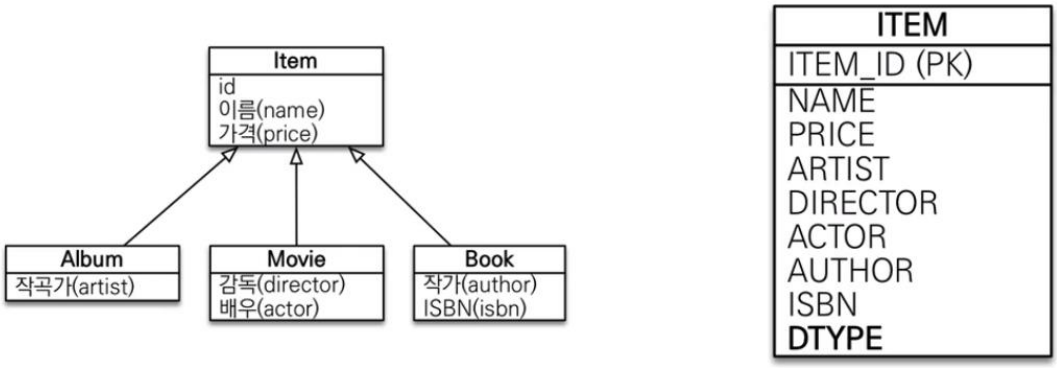
* JOINED 단점

조회시 조인을 많이 사용하므로 쿼리가 복잡하며 단일 테이블 전략에 비하면

성능이 좋지 않다.

데이터 저장시에 INSERT 쿼리가 상위, 하위 테이블 두번 발생한다.

⇨ **통합 테이블로 변환하는 단일 테이블 전략(SINGLE_TABLE)**



* SINGLE_TABLE 장점

조인이 필요 없으므로 일반적인 조회 성능이 빠르다.

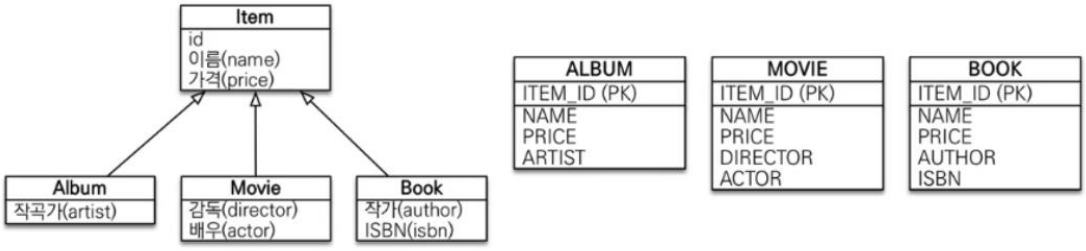
조회 쿼리가 단순하다.

* SINGLE_TABLE 단점

자식 엔티티가 매핑한 컬럼은 모두 NULL을 허용해야 한다.

단일 테이블에 모든 것을 저장하므로 테이블이 커질 수 있다.

⇨ **서브타입 테이블로 변환하는 구현 클래스마다 테이블을 생성하는 전략 (TABLE_PER_CLASS)**



* TABLE_PER_CLASS 장점

서브 타입을 명확하게 구분해서 처리할 때 효과적이다

NOT NULL 제약조건을 사용할 수 있다.

* TABLE_PER_CLASS 단점

여러 자식 테이블을 함께 조회할 때 성능이 느리다(UNION SQL)

자식 테이블을 통합해서 쿼리하기 어렵다.

2.4 엔티티 클래스 개발

2.4.1 엔티티 클래스 작성시 주의사항

- ✓ 실전에서는 가급적 Getter는 열어두고, Setter는 꼭 필요한 경우에만 사용하는 것을 추천합니다.
- ✓ 하지만 과정의 예제에서는 설명을 쉽게 하기 위해 엔티티 클래스에 Getter, Setter를 모두 열고, 최대한 단순하게 설계하였습니다.

2.4.2 회원 엔티티

```
package jpastudy.jpashop.domain;

import lombok.Getter;
import lombok.Setter;
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<>();
}
```

[코드 2.1] Member.java

엔티티의 식별자는 id 를 사용하고 PK 컬럼명은 member_id 로 설정합니다.
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. 중요한 것은 일관성이다.

2.4.3 주문 엔티티

```
package jpastudy.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);
     }
}
```

[코드 2.2] Order.java

2.4.4 주문 상태 엔티티

```
package jpastudy.jpashop.domain;
public enum OrderStatus {
   ORDER, CANCEL
}
```

[코드 2.3] OrderStatus.java

2.4.5 주문 상품 엔티티

```
package jpastudy.jpashop.domain;

import lombok.Getter;
import lombok.Setter;
import jpastudy.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
}
```

[코드 2.4] OrderItem.java

2.4.6 상품 엔티티

```
package jpastudy.jpashop.domain.item;

import lombok.Getter;
import lombok.Setter;

import jpastudy.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>();
}
```

[코드 2.5] Item.java

백명숙(msbaek@hucloud.co.kr)

2.4.7 상품 – 도서 엔티티

```
package jpastudy.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;
}
```

[코드 2.6] Book.java

백명숙(msbaek@hucloud.co.kr)

2.4.8 상품 – 음반 엔티티

```
package jpastudy.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;
}
```

[코드 2.7] Album.java

2.4.9 상품 – 영화 엔티티

```
package jpastudy.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;
}
```

[코드 2.8] Movie.java

2.4.10 배송 엔티티

```
package jpastudy.jpashop.domain;

import lombok.Getter;
import lombok.Setter;
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)]

}
```

[코드 2.9] Delivery.java

백명숙(msbaek@hucloud.co.kr)

2.4.11 배송상태 엔티티

```
package jpastudy.jpashop.domain;


public enum DeliveryStatus {
    READY, COMP
}
```

[코드 2.10] DeliveryStatus.java

2.4.12 카테고리 엔티티

```
package jpastudy.jpashop.domain;

import lombok.Getter;
import lombok.Setter;
import jpastudy.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);
    }

}
```

[코드 2.11] Category.java

2.4.13 주소 값 타입 엔티티

```
package jpastudy.jpashop.domain;

import lombok.Getter;
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;
    }
}
```

[코드 2.12] Address.java

참고: 값 타입은 변경 불가능하게 설계해야 한다.

> @Setter를 제거하고, 생성자에서 값을 모두 초기화 해서 변경 불가능한 클래스를 만들자. JPA 스펙상 엔티티나 임베디드 타입(@Embeddable)은 자바 기본 생성자(default constructor)를 public 또는 protected 로 설정해야 한다. public 보다는 protected 로 설정하는 것이 더 안전하다.

> JPA가 이런 제약을 두는 이유는 JPA 구현 라이브러리가 객체를 생성할 때 리플랙션 같은 기술을 사용할 수 있도록 지원해야 하기 때문이다.

2.5 엔티티 설계시 주의점

2.5.1 지연로딩 (Lazy Loading)

- 모든 연관 관계는 지연로딩(Lazy Loading)으로 설정해야 한다.

1. 즉시로딩(EAGER Loading)은 예측이 어렵고, 어떤 SQL이 실행될지 추적하기 어렵다. 특히 JPQL을 실행할 때 N+1 문제가 자주 발생한다.

2. 모든 연관관계는 지연로딩(LAZY Loading)으로 설정해야 한다.
연관된 엔티티를 함께 DB에서 조회해야 한다면, fetch join 또는 엔티티 그래프 기능을 사용한다.

3. @ToOne(OneToOne, ManyToOne) 관계는 기본이 즉시로딩(EAGER Loading) 이므로 직접 지연로딩(LAZY Loading)으로 설정해야 한다.

2.5.2 컬렉션은 필드에서 초기화 하자.

- 컬렉션은 필드에서 바로 초기화 하는 것이 안전하다.

1. Null 문제에서 안전하다.
2. 하이버네이트는 엔티티를 영속화 할 때, 컬렉션을 감싸서 하이버네이트가 제공하는 내장 컬렉션으로 변경한다.
3. 컬렉션은 필드 레벨에서 생성하는 것이 가장 안전하고, 코드도 간결하다.

```
Member member = new Member();
System.out.println(member.getOrders().getClass());
// class java.util.ArrayList
em.persist(member);
System.out.println(member.getOrders().getClass());
//출력 결과
// class org.hibernate.collection.internal.PersistentBag
```

2.5.3 테이블, 컬럼명 생성 전략

기본적으로 스프링 부트는 물리적 이름 지정 전략을 사용하여 구성합니다

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

SpringPhysicalNamingStrategy. 이 구현은 Hibernate4와 같은 테이블 구조를 제공합니다. 모든 도트(.)는 밑줄(_)로 대체되고 Camel의 대문자는 밑줄로 대체된다. 기본적으로 모든 테이블 이름은 소문자로 생성되지만 스키마에 필요하면 해당 플래그를 무시할 수 있습니다.

적용 2 단계

1. 논리명 생성: 명시적으로 컬럼, 테이블명을 직접 기술하지 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

제 3 장 어플리케이션 구현 준비

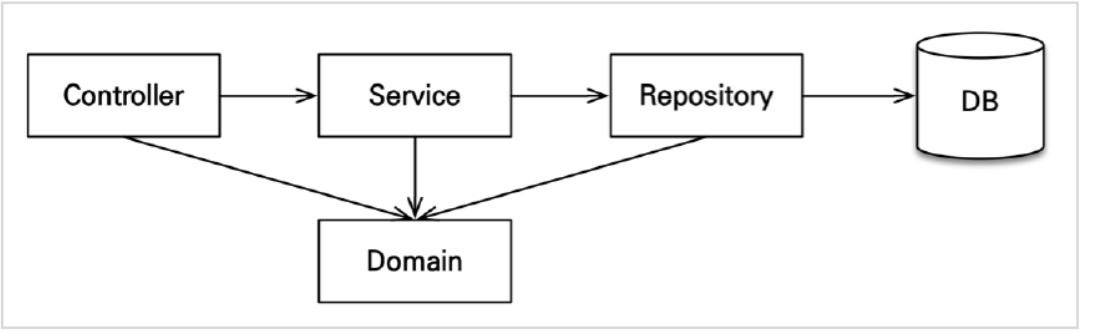
3.1 구현 요구사항

3.1.1 요구사항 범위

1. 회원 기능
   회원 등록
   회원 조회
2. 상품 기능
   상품 등록
   상품 조회
3. 주문 기능
   상품 주문
   주문 내역 조회
   주문 취소
4. 구현하지 않는 기능
   상품은 도서만 사용함
   카테고리는 사용하지 않음
   배송정보는 사용하지 않음

백명숙(msbaek@hucloud.co.kr)

3.2 애플리케이션 아키텍쳐



➢ 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

jpastudy.jpashop

- domain
- exception
- repository
- service
- web
- api

개발 순서: 서비스, 리포지토리 계층을 개발하고, 테스트 케이스를 작성해서 검증, 마지막에 웹 계층이 API 계층을 개발합니다.

3.3 JPA, Hibernate, Spring Data JPA 의 차이점

3.3.1 JPA 는 기술명세서이다.

JPA는 Java Persistence API의 약자로, 자바 어플리케이션에서 관계형 데이터베이스를 사용하는 방식을 정의한 API 명세입니다.
JPA는 단순히 명세이기 때문에 구현이 없고, 특정 기능을 하는 구현 라이브러리가 아닙니다.

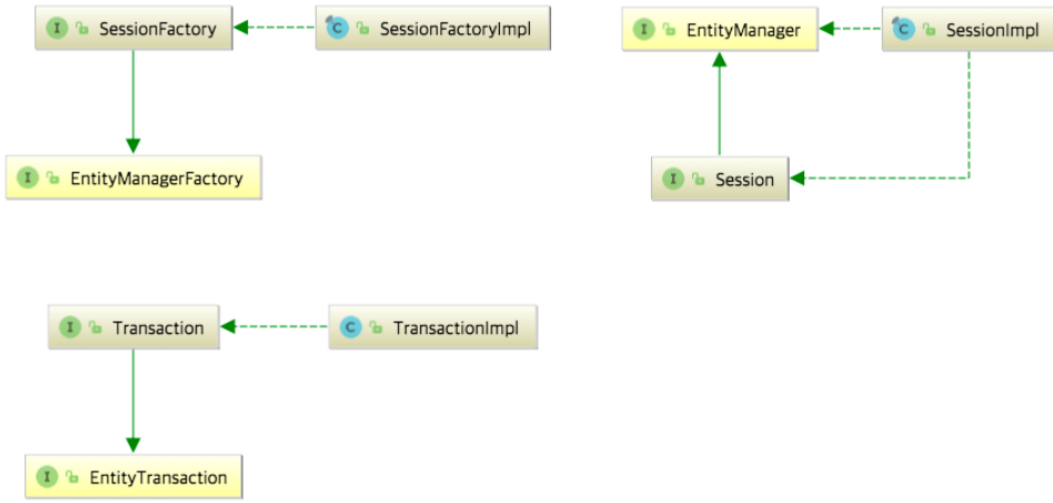
JPA를 정의한 javax.persistence 패키지의 대부분은 interface, enum, Exception, Annotation으로 이루어져 있습니다.
JPA의 핵심이 되는 EntityManager도 interface로 정의되어 있습니다.

```
1   package javax.persistence;
2
3   import ...
4
5   public interface EntityManager {
6
7       public void persist(Object entity);
8
9       public <T> T merge(T entity);
10
11      public void remove(Object entity);
12
13      public <T> T find(Class<T> entityClass, Object primaryKey);
14
15      // More interface methods...
16  }
```

3.3.2 Hibernate 는 JPA 의 구현체이다

Hibernate는 JPA 명세의 구현체입니다.

즉, JPA의 javax.persistence.EntityManager와 같은 인터페이스를 직접 구현한 라이브러리이다. JPA와 Hibernate는 마치 자바의 interface와 해당 interface를 구현한 class와 같은 관계입니다.



JPA의 핵심인 EntityManagerFactory, EntityManager, EntityTransaction을 Hibernate에서는 각각 SessionFactory, Session, Transaction으로 상속받고 각각 Impl로 구현하고 있습니다.

“Hibernate는 JPA의 구현체이다”로부터 도출되는 중요한 결론 중 하나는 JPA를 사용하기 위해서 반드시 Hibernate를 사용할 필요는 없습니다.

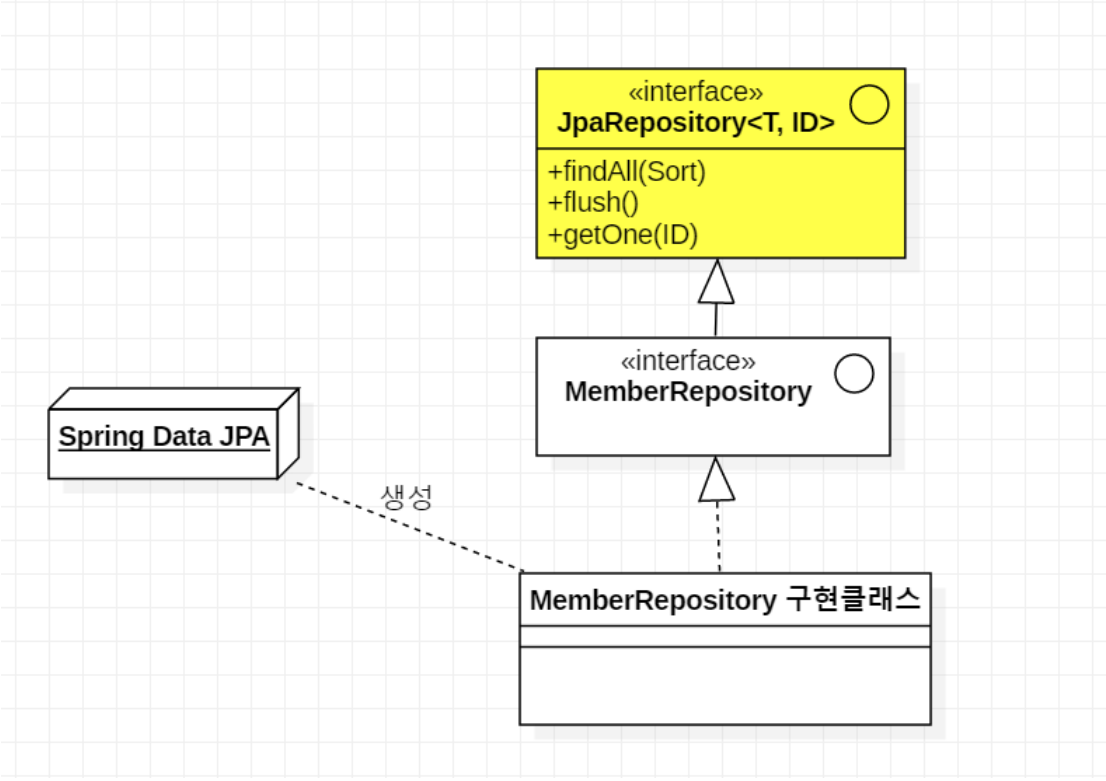
DataNucleus (https://github.com/datanucleus),

EclipseLink (https://www.eclipse.org/eclipselink/) 같은 다른 JPA 구현체를 사용해도 됩니다.

3.3.3 Spring Data JPA 는 JPA 를 사용하기 편하게 만들어 놓은 모듈이다

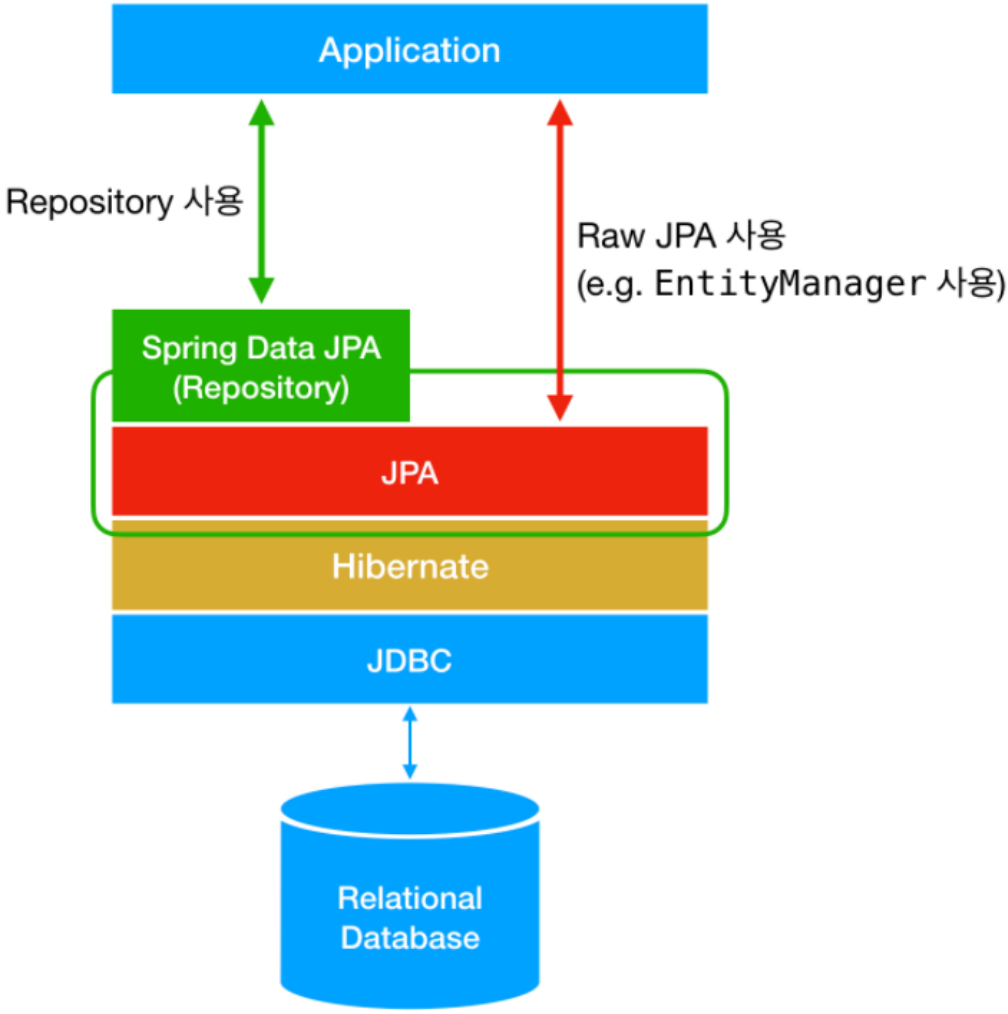
Spring Data JPA는 Spring에서 제공하는 모듈 중 하나로, 개발자가 JPA를 더 쉽고 편하게 사용할 수 있도록 해준다. 이는 JPA를 한 단계 추상화 시킨 Repository라는 인터페이스를 제공함으로써 이루어집니다.

사용자가 Repository 인터페이스에 정해진 규칙대로 메소드를 입력하면, Spring이 알아서 해당 메소드 이름에 적합한 쿼리를 수행하는 구현체를 만들어서 Bean으로 등록 해준다.



백명숙(msbaek@hucloud.co.kr)

3.3.4 JPA, Hibernate, Spring Data JPA 전반적인 개념



백명숙(msbaek@hucloud.co.kr)

제 4 장 회원 서비스 개발

4.1 JPQL(Java Persistence Query Language)

4.1.1 JPQL 이란?

JPQL은 엔티티 객체를 조회하는 객체지향 쿼리이다. 따라서 테이블을 대상으로 쿼리하는 것이 아니라 엔티티 객체를 대상으로 쿼리한다. 문법은 SQL과 유사하며 간결하다. JPQL은 결국 SQL로 변환된다.

또한 JPQL은 SQL을 추상화 해서 특정 데이터베이스에 의존하지 않는다는 특징이 있다. 데이터베이스 방언만 변경하면 JPQL을 수정하지 않아도 데이터베이스를 변경할 수 있다.

4.1.2 JPQL 기본 문법

JPQL도 SQL과 비슷하게 SELECT, UPDATE, DELETE문을 사용할 수 있다. 아래의 JPQL 문법을 보면 전체 구조는 SQL과 비슷한 것을 알 수 있다.

```
select_문 :: =
        select_절
        from_절
        [where_절]
        [groupby_절]
        [having_절]
        [orderby_절]

update_문 :: = update_절 [where_절]

delete_문 :: = delete_절 [where_절]
```

4.1.3 SQL 과 다른 JPQL 의 특징

```
SELECT m
FROM Member AS m
WHERE m.username = 'java'
```

1. 대소문자 구분

엔티티와 속성은 대소문자를 구분한다. 예를 들어, Member, username은 대소문자를 구분 해줘야 한다. 반면에 SELECT, FROM, WHERE 같은 JPQL 키워드는 대소문자를 구분하지 않아도 된다.

2. 엔티티 이름

JPQL에서 사용하는 Member는 클래스 명이 아니라 엔티티 명이다.
엔티티명은 @Entity(name="abc")로 지정할 수 있다. 엔티티 명을 지정하지 않으면 클래스 명을 기본값으로 사용한다.

3. 별칭은 필수

Member AS m을 보면 Member에 m이라는 별칭을 주었다. JPQL은 별칭을 필수로 사용해야 한다. AS를 생략해서 Member m 처럼 사용해도 된다.

4.1.4 TypedQuery 와 Query

JPQL을 실행하려면 쿼리 객체를 만들어야 한다. 쿼리 객체로는 TypedQuery와 Query가 있는데 반환할 타입을 명확하게 지정할 수 있으면 TypedQuery 객체를, 명확하게 지정할 수 없으면 Query 객체를 사용하면 됩니다.

1. TypedQuery 사용

```
TypedQuery<Member> query = em.createQuery("select m from Member m", Member.class);
List<Member> resultList = query.getResultList();
for (Member member : resultList) {
    System.out.println("member : " + member);
}
```

2. Query 사용

```
Query query = em.createQuery("select m.username, m.age from Member m");
List resultList = query.getResultList();
for (Object o : resultList) {
    Object[] result = (Object[]) o; // 결과가 둘 이상이면 Object[] 반환
    System.out.println("username : " + result[0]);
    System.out.println("age : " + result[1]);
}
```

4.1.5 파라미터 바인딩

파라미터 바인딩에는 이름 기준 파라미터와 위치 기준 파라미터가 있다. 위치 기준 파라미터보다는 이름 기준 파라미터가 더 명확하다.

1. 이름 기준 파라미터

이름 기준 파라미터는 파라미터를 이름으로 구분하는 방법이다. 이름 기준 파라미터는 앞에 : 를 사용한다.

```
String param = "java";
TypedQuery<Member> query = em.createQuery("select m from Member m where m.username = :username", Member.class);
query.setParameter("username", param);
List<Member> resultList = query.getResultList();
```

2. 위치 기준 파라미터

위치 기준 파라미터를 사용하려면 ? 다음에 위치 값을 주면 된다. 위치 값은 1부터 시작한다. 아래의 예시처럼 메소드 체인 방식으로 작성할 수도 있다.

```
String param = "java";
List<Member> members =
  em.createQuery("select m from Member m where m.username = ?1", Member.class)
  .setParameter(1, param)
  .getResultList();
```

4.1.6 프로젝션

SELECT 절에 조회할 대상을 지정하는 것을 프로젝션이라 한다. 프로젝션 대상에는 엔티티, 임베디드 타입, 스칼라 타입이 있다. 스칼라 타입은 숫자, 문자 등 기본 데이터 타입을 뜻한다.

1. 엔티티 프로젝션
   - 원하는 객체를 바로 조회
   - 엔티티 프로젝션으로 조회한 엔티티는 영속성 컨텍스트에서 관리

```
SELECT m FROM Member m          // 회원
SELECT m.team FROM Member m // 팀
```

2. 임베디드 타입 프로젝션
   - JPQL에서 임베디드 타입은 엔티티와 거의 비슷하게 사용된다.
   - 임베디드 타입은 엔티티 타입이 아닌 값 타입이다.
   - 직접 조회한 임베디드 타입은 영속성 컨텍스트에서 관리되지 않는다.

```
String query = "SELECT o.address FROM Order o";
List<Address> addresses = em.createQuery(query, Address.class)
                                    .getResultList();
```

⇨ 변환된 SQL

```
select
      order.city,
      order.street,
      order.zipcode
from
      Orders order
```

3. 스칼라 타입 프로젝션

숫자, 문자, 날짜와 같은 기본 데이터 타입들을 스칼라 타입이라 한다.

```
List<String> usernameList =
        em.createQuery("select username from Member m", String.class)
        .getResultList();
```

4. New 명령어

- SELECT 다음 NEW 명령어 사용하여 반환 받을 클래스 지정이 가능하며,
  이 클래스의 생성자에 JPQL 조회 결과를 넘겨줄 수 있음
- New 명령어를 사용한 클래스로 TypeQuery 사용이 가능하여 객체 변환 작업에 효율적
- New 명령어 사용 시 주의사항
  패키지 명을 포함한 전체 클래스 명 기입
  순서와 타입이 정확하게 일치하는 생성자 필요

```
public class UserDTO {
    private String username;
    private int age;

    public UserDTO(String username, int age) {
        this.username = username;
        this.age = age;
    }
    //...
}

TypeQuery<UserDTO> query =
    em.createQuery("SELECT new test.jpql.UserDTO(m.username, m.age)
                    FROM Member m", UserDTO.class);
List<UserDTO> resultList = query.getResultList();
```

4.1.7 JPQL 조인

1. INNER JOIN

JPQL 내부 조인은 SQL의 조인과 약간 다르다. JPQL 조인의 가장 큰 특징은 연관 필드를 사용한다는 것이다.
아래의 예시에서 m.team이 연관 필드이며, 연관 필드는 다른 엔티티와 연관관계를 가지기 위해 사용하는 필드를 뜻합니다.

```
String teamName = "teamA";
String query =
      "select m from Member m inner join m.team t where t.name = :teamName";
List<Member> memberList =
         em.createQuery(query, Member.class)
         .setParameter("teamName", teamName) .getResultList();
```

```
FROM Member m JOIN m.team t // 회원이 가지고 있는 연관 필드로 팀과 조인
FROM Member m JOIN Team t // 오류발생
```

2. OUTER JOIN

외부 조인은 기능상 SQL의 외부 조인과 같다. OUTER는 생략 가능해서 보통 LEFT JOIN으로 사용한다.

```
SELECT m FROM Member LEFT (OUTER) JOIN m.team t
```

3. FETCH JOIN

Fetch Join은 JPQL에서 성능 최적화를 위해 제공하는 기능이다.
연관된 엔티티나 컬렉션을 한 번에 같이 조회하는 기능이며, join fetch 명령어로 사용할 수 있다.
Fetch Join은 N + 1 문제를 해결하는 데 주로 사용되는 방법이다.

- 일반조인 vs Fetch 조인 차이점

회원(Member)과 팀(Team)이 지연 로딩(LAZY Loading)으로 설정되어 있다면

3.1 일반조인

```
SELECT m FROM m INNER JOIN m.team t
-- 실행 SQL
SELECT
    M.ID, M.AGE, M.TEAM_ID, M.NAME
FROM
    MEMBER M INNER JOIN TEAM T ON M.TEAM_ID = T.ID
```

-- Member 엔티티의 내용만 가져오게 된다. 조인된 테이블(TEAM)의 내용은 가져오지 않는다.

3.2 Fetch 조인

```
SELECT m FROM Member m JOIN FETCH m.team
-- 실행 SQL
SELECT
   M.*, T.*
FROM
   MEMBER M INNER JOIN TEAM T ON M.TEAM_ID = T.ID
```

-- 조인된 테이블(TEAM)의 컬럼내용도 가져온다.

=> TEAM 엔티티의 내용(필드)도 가져온다.

4.2 회원 Repository 개발

```
package jpastudy.jpashop.repository;

import jpastudy.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();
    }
}
```

[코드 4.1] MemberRepository.java

> 어노테이션 설명

@Repository : 스프링 Bean으로 등록, JPA 예외를 스프링 기반 예외로 변환
@PersistenceContext : 엔티티 매니저(EntityManager)를 주입해 준다.

> 메서드

save() : 등록
findOne() : id로 1건 조회
findAll() : 전체조회
findByName() : name으로 조회

4.3 회원 서비스 개발

```
package jpastudy.jpashop.service;

import jpastudy.jpashop.domain.Member;
import jpastudy.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;

    // 회원 중복 검증
    private void validateDuplicateMember(Member member) {
        List<Member> findMembers = memberRepository.findByName(member.getName());
        if (!findMembers.isEmpty()) {
            throw new IllegalStateException("이미 존재하는 회원입니다.");
        }
    }
    // 회원가입
    @Transactional
    public Long join(Member member) {
        validateDuplicateMember(member); //중복 회원 검증
        memberRepository.save(member);
        return member.getId();
    }
```


백명숙(msbaek@hucloud.co.kr)

```
    //전체 회원 조회
    public List<Member> findMembers() {
            return memberRepository.findAll();
    }

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

}
```

[코드 4.2] MemberService.java

> 어노테이션 설명

@Service

@Transactional : 트랜잭션, 영속성 컨텍스트

readOnly=true : 데이터의 변경이 없는 읽기 전용 메서드에 사용하며, 영속성 컨텍스트를 flush 하지 않으므로 약간의 성능 향상(읽기 전용에는 다 적용)
데이터베이스 드라이버가 지원하면 DB에서 성능 향상

@Autowired

생성자 Injection 많이 사용, 생성자가 하나면 생략 가능

> 메서드

join() : 회원가입

findMembers() : 회원 전체조회

findOne() : 회원 1명 조회

4.3.1 필드 주입과 생성자 주입

- 필드 주입

```
public class MemberService {
    @Autowired
    MemberRepository memberRepository;
   }
}
```

- 생성자 주입

```
public class MemberService {
        private final MemberRepository memberRepository;

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

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

- Lombok의 @RequiredArgsConstructor

```
@RequiredArgsConstructor
public class MemberService {
        private final MemberRepository memberRepository;
...
}
```

- EntityManager 주입 : @RequiredArgsConstructor

```
@Repository
@RequiredArgsConstructor
public class MemberRepository {
    private final EntityManager em;
...
}
```

4.3.2 회원 서비스 (@ RequiredArgsConstructor 적용한 코드)

```
@Service
@Transactional(readOnly = true)
@RequiredArgsConstructor
public class MemberService {
  private final MemberRepository memberRepository;

  @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.");
    }
  }
```

[코드 4.2] MemberService.java

```
    // 전체 회원 조회
    public List<Member> findMembers() {
            return memberRepository.findAll();
    }
    public Member findOne(Long memberId) {
            return memberRepository.findOne(memberId);
    }
}
```

[코드 4.2] MemberService.java

백명숙(msbaek@hucloud.co.kr)

4.4 회원 기능 테스트

4.4.1 테스트 요구사항

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

```
package jpastudy.jpashop.service;


import jpastudy.jpashop.domain.Member;
import jpastudy.jpashop.repository.MemberRepository;
import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.transaction.annotation.Transactional;


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

```
@SpringBootTest
@Transactional
public class MemberServiceTest {
    @Autowired MemberService memberService;
    @Autowired MemberRepository memberRepository;

    @Test
    public void 회원가입() throws Exception {
        //Given
        Member member = new Member();
        member.setName("boot");
        //When
        Long saveId = memberService.join(member);
        //Then
        assertEquals(member, memberRepository.findOne(saveId));
    }
    @Test
    public void 중복_회원_예외() throws Exception {
        //Given
        Member member1 = new Member();
        member1.setName("boot");
        Member member2 = new Member();
        member2.setName("boot");
        IllegalStateException exception = Assertions.assertThrows(IllegalStateException.class, () ->
        {
                //When
                memberService.join(member1);
                memberService.join(member2); //예외가 발생해야 한다.
        });
        //Then
        assertEquals ("이미 존재하는 회원입니다. ",exception.getMessage());
    }
}
```

> 어노테이션 설명

@SpringBootTest : 스프링 부트 실행하고 테스트(없으면 @Autowired 다 실패)
@Transactional : 반복 가능한 테스트 지원, 각각의 테스트를 실행할 때마다 트랜잭션을 시작하고 **테스트가 끝나면 트랜잭션을 강제로 롤백**
(@Transactional 어노테이션은 테스트 케이스에서 사용될 때만 롤백되어 진다.)

> 메서드

회원가입 테스트
중복 회원 예외처리 테스트

제 5 장 상품 도메인 개발

5.1 상품 Entity 개발 (비즈니스 로직 추가)

```
package jpastudy.jpashop.domain.item;

import lombok.Getter;
import lombok.Setter;
import jpastudy.jpashop.exception.NotEnoughStockException;
import jpastudy.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;
```

[코드 5.1] Item.java

백명숙(msbaek@hucloud.co.kr)

```
    @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;
    }
}
```

> 비즈니스 로직

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.

5.1.1 사용자 예외 클래스

```
package jpastudy.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);
    }
}
```

[코드 5.2] NotEnoughStockException.java

백명숙(msbaek@hucloud.co.kr)

5.2 상품 Repository 개발

```
package jpastudy.jpashop.repository;

import jpastudy.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();
    }
}
```

[코드 5.3] ItemRepository.java

> 메서드

save()

: id 가 없으면 신규이므로 persist() 실행한다.

: id 가 있으면 이미 데이터베이스에 저장된 엔티티를 수정하므로, merge() 를 실행한다.

5.3 상품 Service 개발

```
package jpastudy.jpashop.service;

import jpastudy.jpashop.domain.item.Item;
import jpastudy.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);
    }
}
```

[코드 5.4] ItemService.java

제 6 장 주문 도메인 개발

6.1 주문 Entity 개발 (비즈니스 로직 추가)

```
package jpastudy.jpashop.domain.item;

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<>();
```


백명숙(msbaek@hucloud.co.kr)

```
@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);
}
```

[코드 6.1] Order.java

백명숙(msbaek@hucloud.co.kr)

```
    //== 비즈니스 로직 : 주문생성 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 불가능합니다.");
        }
        this.setStatus(OrderStatus.CANCEL);
        for (OrderItem orderItem : orderItems) {
                orderItem.cancel();
        }
    }
    //==비즈니스 로직 : 전체 주문 가격 조회 ==//
    public int getTotalPrice() {
        int totalPrice = 0;
        for (OrderItem orderItem : orderItems) {
                totalPrice += orderItem.getTotalPrice();
        }
        return totalPrice;
    }
}
```


백명숙(msbaek@hucloud.co.kr)

> 메서드

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

6.2 주문상품 Entity 개발 (비즈니스 로직 추가)

```
package jpastudy.jpashop.domain;

import lombok.Getter;
import lombok.Setter;
import javax.persistence.*;
import jpastudy.jpashop.domain.item.Item;

@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
```


백명숙(msbaek@hucloud.co.kr)

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

[코드 6.2] OrderItem.java

> 메서드

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

6.3 주문 Repository 개발

```
package jpastudy.jpashop.repository;

import jpastudy.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) { ... }
}
```

[코드 6.3] OrderRepository.java

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

6.4 주문 Service 개발

```
package jpastudy.jpashop.service;

import jpastudy.jpashop.domain.Delivery;
import jpastudy.jpashop.domain.Member;
import jpastudy.jpashop.domain.Order;
import jpastudy.jpashop.domain.OrderItem;
import jpastudy.jpashop.domain.item.Item;
import jpastudy.jpashop.repository.ItemRepository;
import jpastudy.jpashop.repository.MemberRepository;
import jpastudy.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;
```

[코드 6.4] OrderService.java

```
    /** 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);
    }
}
```

[코드 6.4] OrderService.java

주문 서비스는 주문 엔티티와 주문 상품 엔티티의 비즈니스 로직을 활용해서 주문, 주문 취소, 주문 내역 검색 기능을 제공한다.
참고: 예제를 단순화 하려고 한 번에 하나의 상품만 주문할 수 있도록 했습니다.

1. **주문**(order()): 주문하는 회원 식별자, 상품 식별자, 주문 수량 정보를 받아서 실제 주문 엔티티를 생성한 후 저장한다.
2. **주문 취소**(cancelOrder()): 주문 식별자를 받아서 주문 엔티티를 조회한 후 주문 엔티티에 주문 취소를 요청한다.
3. **주문 검색**(findOrders()): OrderSearch 라는 검색 조건을 가진 객체로 주문 엔티티를 검색한다. 자세한 내용은 뒤에서 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 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 스크립트패턴 (http://martinfowler.com/eaaCatalog/transactionScript.html)이라 한다.

6.5 주문 기능 테스트

테스트 요구사항

: 상품 주문이 성공해야 한다.

: 상품을 주문할 때 재고 수량을 초과하면 안 된다.

: 주문 취소가 성공해야 한다.

6.5.1 상품 주문 테스트

```
package jpastudy.jpashop.service;

import jpastudy.jpashop.domain.Address;
import jpastudy.jpashop.domain.Member;
import jpastudy.jpashop.domain.Order;
import jpastudy.jpashop.domain.OrderStatus;
import jpastudy.jpashop.domain.item.Book;
import jpastudy.jpashop.domain.item.Item;
import jpastudy.jpashop.exception.NotEnoughStockException;
import jpastudy.jpashop.repository.OrderRepository;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.Assertions;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.transaction.annotation.Transactional;

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

import static org.springframework.test.util.AssertionErrors.assertEquals;
```


백명숙(msbaek@hucloud.co.kr)

```
@SpringBootTest
@Transactional
public class OrderServiceTest {
    @PersistenceContext
    EntityManager em;
    @Autowired OrderService orderService;
    @Autowired OrderRepository orderRepository;

    @Test
    public void  상품주문() throws Exception {
        //Given
        Member member = createMember("회원1", new Address("서울", "성내로", "80"));
        Item item = createBook("스프링 부트", 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());
    }
```

[코드 6.5] OrderServiceTest.java

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.

```
private Member createMember(String name, Address address) {
    Member member = new Member();
    member.setName(name);
    member.setAddress(address);
    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;
}
```

[코드 6.5] OrderServiceTest.java

백명숙(msbaek@hucloud.co.kr)

6.5.2 상품 재고수량 초과 테스트

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

```
@Test
public void 상품주문_재고수량초과() throws Exception {
    //Given
    Member member = createMember("회원1", new Address("서울", "성내로", "80"));
    Item item = createBook("스프링 부트", 10000, 10); //이름, 가격, 재고
    int orderCount = 11; //재고보다 많은 수량
    NotEnoughStockException exception =
    Assertions.assertThrows(NotEnoughStockException.class, () -> {
            //When
            orderService.order(member.getId(), item.getId(), orderCount);
    });
    //Then
    Assertions.assertEqual("need more stock",exception.getMessage());
}
```

[코드 6.5] OrderServiceTest.java

//주문 orderService.order(memberId, itemId, count) ->
//주문상품 생성 orderItem.createOrderItem(item, item.getPrice(), count) ->
//주문 후 재고 감소 item.removeStock(count)

```
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;
    }
}
```

6.5.3 주문 취소 테스트

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

```
@Test
public void 주문취소() {
    //Given
    Member member = createMember("회원1", new Address("서울", "성내로", "180"));
    Item item = createBook("스프링 부트", 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());
}
```

[코드 6.5] OrderServiceTest.java

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.

6.6 주문 검색 기능 개발

6.6.1 검색 조건 파라미터

```
package jpastudy.jpashop.domain;


@Getter @Setter
public class OrderSearch {
    private String memberName; //회원 이름
    private OrderStatus orderStatus;//주문 상태[ORDER, CANCEL]
}
```

[코드 6.6] OrderSearch.java

6.6.2 검색을 추가한 주문 Repository

```
package jpastudy.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
    }
}
```

[코드 6.7] OrderRepository.java

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

JPQL로 처리

```
public List<Order> findAllByString(OrderSearch orderSearch) {
    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);
    if (orderSearch.getOrderStatus() != null) {
                query = query.setParameter("status", orderSearch.getOrderStatus());
    }
    if (StringUtils.hasText(orderSearch.getMemberName())) {
        query = query.setParameter("name", orderSearch.getMemberName());
    }
  return query.getResultList();
}
```

[코드 6.8] OrderRepository.java

백명숙(msbaek@hucloud.co.kr)

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();
    }
```

[코드 6.9] OrderRepository.java

JPA Criteria는 JPQL과 같이 엔티티 조회를 기본으로 하며, 컴파일 시점에서 에러를 확인할수 있는 특징을 가지며, JPA 표준 스펙이지만 사용하기에 너무 복잡하다.

Querydsl로 처리

http://www.querydsl.com/

- 조건에 따라서 달라지는 동적 쿼리를 작성할 때 queryds1을 많이 사용한다.
- Querydsl은 SQL(JPQL)과 모양이 유사하면서 자바 코드로 동적 쿼리를 편리하게 생성할 수 있다.
- Querydsl은 JPQL을 코드로 만드는 빌더 역할을 한다.
- 복잡한 동적 쿼리를 많이 사용할 때 Querydsl을 사용하면 높은 개발 생산성을 얻으면서 동시에 쿼리 오류를 컴파일 시점에 빠르게 잡을 수 있다.
- 동적 쿼리가 아니라 정적 쿼리인 경우에도 아래와 같은 이유로 Querydsl을 사용하는 것이 좋다.
  - 직관적인 문법
  - 컴파일 시점에 빠른 문법 오류 발견
  - 코드 자동완성 및 코드 재사용
  - JPQL new 명령어와는 비교가 안될 정도로 깔끔한 DTO 조회를 지원한다.
- Querydsl is a framework which enables the construction of type-safe SQL-like queries for multiple backends including JPA, MongoDB and SQL in Java

build.gradle에 querydsl 설정 1

```
//querydsl 추가
buildscript {
	dependencies {
		classpath("gradle.plugin.com.ewerk.gradle.plugins:querydsl-plugin:1.0.10")
	}
}

plugins {
	id 'org.springframework.boot' version '2.4.10.RELEASE'
	id 'io.spring.dependency-management' version '1.0.11.RELEASE'
	id 'java'
}

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

apply plugin: "com.ewerk.gradle.plugins.querydsl"

configurations {
	compileOnly {
		extendsFrom annotationProcessor
	}
}

repositories {
	mavenCentral()
}
```


백명숙(msbaek@hucloud.co.kr)

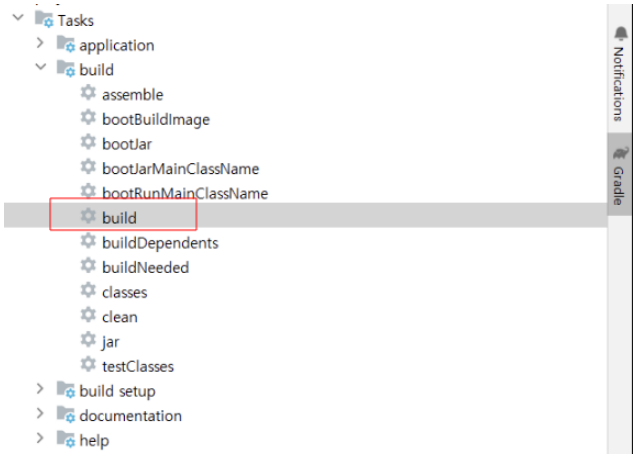
build.gradle에 querydsl 설정 2

```
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-web'
	implementation 'org.springframework.boot:spring-boot-devtools'
	implementation 'com.github.gavlyukovskiy:p6spy-spring-boot-starter:1.5.6'
	implementation 'com.fasterxml.jackson.datatype:jackson-datatype-hibernate5'
	compileOnly 'org.projectlombok:lombok'
	runtimeOnly 'com.h2database:h2'
	annotationProcessor 'org.projectlombok:lombok'
	testImplementation 'org.springframework.boot:spring-boot-starter-test'

	//querydsl 추가
	implementation 'com.querydsl:querydsl-jpa'
	//querydsl 추가
	implementation 'com.querydsl:querydsl-apt'
}
test {
	useJUnitPlatform()
}

//querydsl 추가
apply plugin: "com.ewerk.gradle.plugins.querydsl"
def querydslDir = "$buildDir/generated/querydsl"
querydsl {
	library = "com.querydsl:querydsl-apt"
	jpa = true
	querydslSourcesDir = querydslDir
}
```

build.gradle에 querydsl 설정 3

```
sourceSets {
        main {
            java {
                srcDirs = ['src/main/java', querydslDir]
            }
        }
}

compileQuerydsl{
        options.annotationProcessorPath = configurations.querydsl
}

configurations {
        querydsl.extendsFrom compileClasspath
}
```

Gradle 탭에서 build Task를 실행하면 generated 폴더 아래에 Q클래스가 생성 됩니다.



백명숙(msbaek@hucloud.co.kr)

Querydsl로 처리

```
import com.querydsl.core.types.dsl.BooleanExpression;
import com.querydsl.jpa.impl.JPAQueryFactory;
import org.springframework.util.StringUtils;

public List<Order> findAllByQuerydsl(OrderSearch orderSearch) {
    JPAQueryFactory query = new JPAQueryFactory(em);
    QOrder order = QOrder.order;
    QMember member = QMember.member;
    return query
        .select(order)
        .from(order)
        .join(order.member, member)
        .where(statusEq(orderSearch.getOrderStatus()),
        nameLike(orderSearch.getMemberName()))
        .limit(1000)
        .fetch();
}
private BooleanExpression statusEq(OrderStatus statusCond) {
    if (statusCond == null) {
            return null;
    }
    return QOrder.order.status.eq(statusCond);
}
private BooleanExpression nameLike(String nameCond) {
    if (!StringUtils.hasText(nameCond)) {
            return null;
    }
    return QMember.member.name.like(nameCond);
}
```

[코드 6.9] OrderRepository.java

백명숙(msbaek@hucloud.co.kr)

제 7 장 API 개발

7.1 API 개발 기본

7.1.1 회원 등록 API

```
package jpastudy.jpashop.api;

import jpastudy.jpashop.domain.Member;
import jpastudy.jpashop.service.MemberService;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.RequiredArgsConstructor;
import org.springframework.web.bind.annotation.*;
import javax.validation.Valid;
import java.util.List;
import java.util.stream.Collectors;

@RestController
@RequiredArgsConstructor
public class MemberApiController {
    private final MemberService memberService;

    @PostMapping("/api/v1/members")
    public CreateMemberResponse saveMemberV1(@RequestBody @Valid Member member){
        Long id = memberService.join(member);
        return new CreateMemberResponse(id);
    }
```

[코드 7.1] MemberApiController.java

```
    @Data
    static class CreateMemberResponse {
        private Long id;
        public CreateMemberResponse(Long id) {
            this.id = id;
        }
    }
}
```

[코드 7.1] MemberApiController.java

```
public class Member {

    @NotEmpty
    private String name;
}
```

[코드 7.2] Member.java

7.1.1.1 Version1 엔티티를 Request Body에 직접 매핑

- 문제점
  - 엔티티에 프레젠테이션 계층을 위한 로직이 추가된다.
  - 엔티티에 API 검증을 위한 로직이 들어간다. (@NotEmpty 등등)
  - 회원 엔티티를 위한 API가 다양하게 만들어져야 하는데 하나의 엔티티에 각각의 API를 위한 모든 요구사항을 담기는 어렵다.
  - 엔티티가 변경되면 API 스펙이 변한다.
- 결론
  - API 요청 스펙에 맞추어 별도의 DTO를 파라미터로 사용하는 것이 좋다.

7.1.1.2 Version2 엔티티 대신에 DTO를 Request Body에 매핑

```
/**
* 등록 V2: 요청 값으로 Member 엔티티 대신에 별도의 DTO를 받는다.
*/
@PostMapping("/api/v2/members")
public CreateMemberResponse saveMemberV2(@RequestBody @Valid
   CreateMemberRequest request) {
      Member member = new Member();
      member.setName(request.getName());
      Long id = memberService.join(member);
      return new CreateMemberResponse(id);
}

@Data
static class CreateMemberRequest {
      @NotEmpty
      private String name;
}
```

[코드 7.3] MemberApiController.java

- CreateMemberRequest 를 Member 엔티티 대신 RequestBody와 매핑한다.
- 엔티티와 프레젠테이션 계층을 위한 로직을 분리할 수 있다.
- 엔티티와 API 스펙을 명확하게 분리할 수 있다.
- 엔티티가 변해도 API 스펙이 변하지 않는다.
- 엔티티가 API 스펙에 노출되지 않는다.
- Member 엔티티에 있는 검증 어노테이션(@NotEmpty)은 제거한다.

7.1.2 회원 수정 API

```
/**
* 수정 API
*/
@PatchMapping("/api/v2/members/{id}")
public UpdateMemberResponse updateMemberV2(@PathVariable("id") Long id,
@RequestBody @Valid UpdateMemberRequest request) {
        memberService.update(id, request.getName());
        Member findMember = memberService.findOne(id);
        return new UpdateMemberResponse(findMember.getId(), findMember.getName());
}

@Data
static class UpdateMemberRequest {
        private String name;
}

@Data
@AllArgsConstructor
static class UpdateMemberResponse {
        private Long id;
        private String name;
}
```

[코드 7.4] MemberApiController.java

회원 수정 API updateMemberV2 은 회원 정보를 부분 업데이트 한다.
PUT은 전체 업데이트를 할 때 사용하는 것이고 부분 업데이트를 하려면 PATCH를 사용하는 것이 REST 스타일에 맞다.

```
public class MemberService {
    private final MemberRepository memberRepository;
    /**
    * 회원 수정 (변경감지를 이용)
    */
    @Transactional
    public void update(Long id, String name) {
        Member member = memberRepository.findOne(id);
        member.setName(name);
    }
}
```

[코드 7.5] MemberService.java

7.1.3 회원 조회 API Version1

: 응답 값으로 엔티티를 직접 외부에 노출

```
/**
* 조회 V1: 응답 값으로 엔티티를 직접 외부에 노출한다.
*/
@GetMapping("/api/v1/members")
public List<Member> membersV1() {
    return memberService.findMembers();
}
```

[코드 7.6] MemberApiController.java

- 문제점
  - 엔티티에 프레젠테이션 계층을 위한 로직이 추가된다.
  - 기본적으로 엔티티의 모든 값이 노출된다.
  - 응답 스펙을 맞추기 위해 로직이 추가된다. (@JsonIgnore, 별도의 뷰 로직 등등)
  - 같은 엔티티에 대해 API가 용도에 따라 다양하게 만들어 지는데, 하나의 엔티티에 각각의 API를 위한 프레젠테이션 응답 로직을 담기는 어렵다.
  - 엔티티가 변경되면 API 스펙이 변한다.
  - 추가로 컬렉션을 직접 반환하면 항후 API 스펙을 변경하기 어렵다. (별도의 Result 클래스 생성으로 해결)
- 결론
  - 엔티티 대신에 API 응답 스펙에 맞추어진 별도의 DTO를 반환해야한다.

7.1.4 회원 조회 API Version 2

: 응답 값으로 엔티티가 아닌 별도의 DTO 사용

```
/**
* 조회 V2: 응답 값으로 엔티티가 아닌 별도의 DTO를 반환한다
*/
@GetMapping("/api/v2/members")
public Result membersV2() {
    List<Member> findMembers = memberService.findMembers();
    //엔티티 -> DTO 변환
    List<MemberDto> memberDtoList = findMembers.stream()
        .map(m -> new MemberDto(m.getName()))
        .collect(Collectors.toList());
    return new Result(memberDtoList);
}
@Data
@AllArgsConstructor
class Result<T> {
    private T data;
}
@Data
@AllArgsConstructor
class MemberDto {
    private String name;
}
```

[코드 7.6] MemberApiController.java

- 엔티티를 DTO로 변환해서 리턴한다.
- 엔티티가 변경 되어도 API 스펙이 변경되지 않는다.
- 추가로 Result 클래스로 컬렉션을 Wrapping 하기 때문에 추가로 필요한 필드를 확장할 수 있다.

7.2 API 개발 고급 – 지연로딩과 조회 성능 최적화

7.2.1 조회용 샘플 데이터 입력

API 개발 고급에서 연관관계가 설정된 샘플 데이터를 입력

userA
- JPA1 BOOK, JPA2 BOOK

userB
- SPRING1 BOOK, SPRING2 BOOK

```
package jpastudy.jpashop;

import jpastudy.jpashop.domain.*;
import jpastudy.jpashop.domain.item.Book;
import lombok.RequiredArgsConstructor;
import org.springframework.stereotype.Component;
import org.springframework.transaction.annotation.Transactional;
import javax.annotation.PostConstruct;
import javax.persistence.EntityManager;
@Component
@RequiredArgsConstructor
public class InitDb {
    private final InitService initService;

    @Component
    @Transactional
    @RequiredArgsConstructor
    static class InitService {
      private final EntityManager em;
    }
}
```

[코드 7.7] InitDb.java

〉InitDb 클래스 내부의 Static Inner 클래스인 InitService 클래스에 createMember(), createBook(), createDelivery() 메서드를 추가한다.

```
@Component
@Transactional
@RequiredArgsConstructor
static class InitService {
    private final EntityManager em;

    private Member createMember(String name, String city, String street, String zipcode) {
        Member member = new Member();
        member.setName(name);
        member.setAddress(new Address(city, street, zipcode));
        return member;
    }
    private Book createBook(String name, int price, int stockQuantity) {
        Book book = new Book();
        book.setName(name);
        book.setPrice(price);
        book.setStockQuantity(stockQuantity);
        return book;
    }
    private Delivery createDelivery(Member member) {
        Delivery delivery = new Delivery();
        delivery.setAddress(member.getAddress());
        return delivery;
     }

}//InitService
```

[코드 7.8] InitDb.InitService.java

```
static class InitService {
    public void dbInit1() {
        Member member = createMember("userA", "서울", "1", "1111");
        em.persist(member);
        Book book1 = createBook("JPA1 BOOK", 10000, 100);
        em.persist(book1);
        Book book2 = createBook("JPA2 BOOK", 20000, 100);
        em.persist(book2);
        OrderItem orderItem1 = OrderItem.createOrderItem(book1, 10000, 1);
        OrderItem orderItem2 = OrderItem.createOrderItem(book2, 20000, 2);
        Order order = Order.createOrder(member, createDelivery(member),orderItem1, orderItem2);
        em.persist(order);
    }//dbInit1

    public void dbInit2() {
        Member member = createMember("userB", "부산", "2", "2222");
        em.persist(member);
        Book book1 = createBook("SPRING1 BOOK", 20000, 200);
        em.persist(book1);
        Book book2 = createBook("SPRING2 BOOK", 40000, 300);
        em.persist(book2);
        Delivery delivery = createDelivery(member);
        OrderItem orderItem1 = OrderItem.createOrderItem(book1, 20000, 3);
        OrderItem orderItem2 = OrderItem.createOrderItem(book2, 40000, 4);
        Order order = Order.createOrder(member, delivery, orderItem1,orderItem2);
        em.persist(order);
    }//dbInit2
}//InitService
```

[코드 7.9] InitDb.InitService.java

```
public class InitDb {
    private final InitService initService;

    @PostConstruct
    public void init() {
        initService.dbInit1();
        initService.dbInit2();
    }

    static class InitService {
      private final EntityManager em;
    }
}
```

[코드 7.10] InitDb.java

- jpa.hibernate.ddl-auto: create 로 설정한다.
- h2-console 에서 데이터를 확인한다.

7.2.2 주문조회 Version1

지연 로딩(Lazy Loading)과 조회 성능 최적화

- 주문 + 배송정보 + 회원을 조회하는 API 만들기.
- 지연 로딩 때문에 발생하는 성능 문제를 단계적으로 해결하기.
- toOne(OneToOne, ManyToOne) 관계를 조회하고 최적하는 방법

1) 응답 값으로 엔티티를 직접 외부에 노출

```
package jpastudy.jpashop.api;
import jpastudy.jpashop.domain.Address;
import jpastudy.jpashop.domain.Order;
import jpastudy.jpashop.domain.OrderStatus;
import jpastudy.jpashop.repository.*;
import lombok.Data;
import lombok.RequiredArgsConstructor;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;
import java.time.LocalDateTime;
import java.util.List;
import static java.util.stream.Collectors.toList;
/**
* xToOne(ManyToOne, OneToOne) 관계 최적화
* Order, Order -> Member , Order -> Delivery
*/
@RestController
@RequiredArgsConstructor
public class OrderSimpleApiController {
    private final OrderRepository orderRepository;

    @GetMapping("/api/v1/simple-orders")
    public List<Order> ordersV1() {
        List<Order> all = orderRepository.findAll(new OrderSearch());
        return all;
    }
}
```

[코드 7.9] OrderSimpleApiController.java

⇨ 양방향 관계 문제 발생 -> @JsonIgnore로 해결하기

양방향 관계 문제를 해결하기 위하여 : @JsonIgnore 어노테이션 사용

```
import  com.fasterxml.jackson.annotation.JsonIgnore;


public class Member {
    @JsonIgnore
    @OneToMany(mappedBy = "member")
    private List<Order> orders = new ArrayList<>();
}
```

[코드 7.10] Member.java

```
public class OrderItem {
    @JsonIgnore
    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "order_id")
    private Order order; //주문
}
```

[코드 7.11] OrderItem.java

```
public class Delivery {
    @JsonIgnore
    @OneToOne(mappedBy=”delivery”,fetch = FetchType.LAZY)
    private Order order; //주문
}
```

[코드 7.12] Delivery.java

- Order가 Member를 참조할때는 지연 로딩이다. 실제 엔티티 대신에 프록시가 존재한다.
  실제로 private Member member = new ByteBuddyInterceptor();
- Jackson 라이브러리는 기본적으로 이 프록시 객체를 json으로 어떻게 생성해야 하는지 모르기 때문에 예외가 발생한다.

No serializer found for class org.hibernate.proxy.pojo.bytebuddy.ByteBuddyInterceptor and no properties discovered to create BeanSerializer 예외가 발생함

- Hibernate5Module 을 스프링 Bean으로 등록하면 해결할 수 있다.

- **Hibernate5Module 라이브러리 설치 및 Bean 등록**

```
implementation 'com.fasterxml.jackson.datatype:jackson-datatype-hibernate5'
```

[코드 7.13] build.gradle

- Hibernate5Module은 기본적으로 초기화 된 프록시 객체만 노출, 초기화 되지 않은 프록시 객체는 노출 하지 않는다.

```
@Bean
Hibernate5Module hibernate5Module() {
    return new Hibernate5Module();
}
```

[코드 7.14] Application.java

- 아래와 같이 설정하면 강제로 지연로딩이 가능하다.

```
@Bean
Hibernate5Module hibernate5Module() {
    Hibernate5Module hibernate5Module = new Hibernate5Module();
    hibernate5Module.configure(Hibernate5Module.Feature.FORCE_LAZY_LOADING,true);
    return hibernate5Module;
}
```

- Hibernate5Module.Feature.FORCE_LAZY_LOADING 설정은 Off 하고 강제로 Lazy Loading 하기

```
@GetMapping("/api/v1/simple-orders")
public List<Order> ordersV1() {
    List<Order> all = orderRepository.findAllByString(new OrderSearch());
    for (Order order : all) {
        order.getMember().getName(); //Lazy 강제 초기화
        order.getDelivery().getAddress(); //Lazy 강제 초기화
    }
    return all;
}
```

[코드 7.15] OrderSimpleApiController.java

* 주의사항

1. 엔티티를 직접 노출할 때는 양방향 연관관계 인 경우 한쪽을 @JsonIgnore 처리를 반드시 해야 한다. 처리하지 않으면 양쪽을 서로 호출하면서 무한 루프가 발생하기 때문이다.
2. 엔티티를 API 응답으로 외부로 노출하는 것은 좋지 않다.
API 응답으로는 DTO를 변환해서 반환 하는 것이 더 좋은 방법이다.
3. 지연 로딩(LAZY)을 피하기 위해 즉시 로딩(EAGER)으로 설정하면 안된다.
즉시 로딩 때문에 연관 관계가 필요 없는 경우에도 데이터를 항상 조회해서 성능 문제가 발생할 수 있다. 즉시 로딩으로 설정하면 성능 튜닝이 매우 어려워 진다.
4. 항상 지연 로딩을 기본으로 하고, 성능 최적화가 필요한 경우에는 패치 조인(fetch join)을 사용하는 것이 좋은 방법이다.

7.2.3 주문조회 Version2

: 엔티티를 DTO로 변환하여 외부에 노출하기

```
/**
* V2. 엔티티를 조회해서 DTO로 변환
* - 단점: 지연로딩으로 쿼리 N번 호출
*/
@GetMapping("/api/v2/simple-orders")
public List<SimpleOrderDto> ordersV2() {
    List<Order> orders = orderRepository.findAll(new OrderSearch());
    List<SimpleOrderDto> result = orders.stream()
            .map(o -> new SimpleOrderDto(o))
            .collect(Collectors.toList());
    return result;
}
@Data
static class SimpleOrderDto {
        private Long orderId;
        private String name;
        private LocalDateTime orderDate;
        private OrderStatus orderStatus;
        private Address address;

        public SimpleOrderDto(Order order) {
            orderId = order.getId();
            name = order.getMember().getName();   //Lazy 강제 초기화
            orderDate = order.getOrderDate();
            orderStatus = order.getStatus();
            address = order.getDelivery().getAddress(); //Lazy 강제 초기화
    }
}
```

[코드 7.16] OrderSimpleApiController.java

* 주의사항

1. N + 1 문제가 발생한다.
   - N + 1 문제란 하나의 쿼리를 수행하는데 조회하는 Row의 개수만큼 관계가 있는 Row의 갯수 만큼의 쿼리가 추가적으로 발생하는 문제를 말한다.

쿼리가 총 1 + N + N번 실행된다. (version1과 쿼리갯수 결과는 같다)

: order 조회 1번(order 조회 결과 수가 N이 된다.)

: order -> member 지연 로딩 조회 N 번

: order -> delivery 지연 로딩 조회 N 번

예) 현재는 order의 결과가 2개이므로 1 + 2 + 2 (총5개) 쿼리가 실행된다.

1(order) + 2(member) + 2(delivery)

백명숙(msbaek@hucloud.co.kr)

7.2.4 주문조회 Version3

7.2.4.1 Fetch Join 이란?

1) N + 1의 해결방법
2) 기존 SQL의 조인이 아니고, JPQL의 성능 튜닝을 위해 제공되는 조인이다.
3) 연관된 엔티티와 컬렉션을 한번에 함께 조회하는 기능이다.
4) Fetch Join을 하면 연관된 엔티티는 프록시가 아니라 실제 엔티티이므로 조회 할 때 지연로딩(Lazy Loading)이 일어나지 않는다.
5) join fetch 명령어 사용

- Member를 조회하면서 연관된 Team도 함께 조회한다.

```
//JPQL
select m from Member m join fetch m.team

//SQL
select m.* t.* from Member m inner join Team t on m.team_id = t.id;
```

[MEMBER]

ID (PK)	NAME	TEAM_ID (FK)
1	회원1	1
2	회원2	1
3	회원3	2
4	회원4	null

[TEAM]

ID (PK)	NAME
1	팀A
2	팀B
3	팀C

FK -> PK
(MEMBER.TEAM_ID -> TEAM.ID)

[MEMBER JOIN TEAM]

ID (PK)	NAME	TEAM_ID (FK)	ID (PK)	NAME
1	회원1	1	1	팀A
2	회원2	1	1	팀A
3	회원3	2	2	팀B

- fetch join은 내부적으로 inner join을 사용하므로 Team이 있는 회원만 조회되고, Team이 없는 회원은 누락된다.

: 엔티티를 DTO로 변환하고, Fetch Join으로 최적화하여 외부에 노출하기

```
public List<Order> findAllWithMemberDelivery() {
    return em.createQuery(
        "select o from Order o" +
        " join fetch o.member m" +
        " join fetch o.delivery d", Order.class)
        .getResultList();
}
```

[코드 7.17] OrderRepository.java

```
/**
* V3. 엔티티를 조회해서 DTO로 변환(fetch join 사용함)
* fetch join으로 쿼리 1번 호출
*/
@GetMapping("/api/v3/simple-orders")
public List<SimpleOrderDto> ordersV3() {
    List<Order> orders = orderRepository.findAllWithMemberDelivery();
    List<SimpleOrderDto> result = orders.stream()
            .map(o -> new SimpleOrderDto(o))
            .collect(Collectors.toList());
    return result;
}
```

[코드 7.18] OrderSimpleApiController.java

*** 주의사항**

1. 엔티티를 Fetch Join을 사용해서 1번의 쿼리로 조회한다.
2. Fetch Join으로 order -> member, order -> delivery 는 이미 조회 된 상태이므로 지연로딩이 발생하지 않는다.

7.2.5 주문조회 Version4

: JPA에서 DTO로 바로 조회하기

```
package jpastudy.jpashop.repository.order.simplequery;


import jpastudy.jpashop.domain.Address;
import jpastudy.jpashop.domain.OrderStatus;
import lombok.Data;
import java.time.LocalDateTime;

@Data
public class OrderSimpleQueryDto {
    private Long orderId;
    private String name;
    private LocalDateTime orderDate;
    private OrderStatus orderStatus;
    private Address address;

    public OrderSimpleQueryDto(Long orderId, String name, LocalDateTime orderDate,
    OrderStatus orderStatus, Address address) {
        this.orderId = orderId;
        this.name = name;
        this.orderDate = orderDate;
        this.orderStatus = orderStatus;
        this.address = address;
    }
}
```

[코드 7.19] OrderSimpleQueryDto.java

백명숙(msbaek@hucloud.co.kr)

```
package jpastudy.jpashop.repository.order.simplequery;

import lombok.RequiredArgsConstructor;
import org.springframework.stereotype.Repository;
import javax.persistence.EntityManager;
import java.util.List;

@Repository
@RequiredArgsConstructor
public class OrderSimpleQueryRepository {
    private final EntityManager em;

    public List<OrderSimpleQueryDto> findOrderDtos() {
        return em.createQuery(
            "select new
            jpastudy.jpashop.repository.order.simplequery.OrderSimpleQueryDto(o.id, m.name,
            o.orderDate, o.status, d.address)" +
            " from Order o" +
            " join o.member m" +
            " join o.delivery d", OrderSimpleQueryDto.class)
            .getResultList();
    }
}
```

[코드 7.20] OrderSimpleQueryRepository.java

```
/**
* V4. JPA에서 DTO로 바로 조회
* - 쿼리 1번 호출
* - select 절에서 원하는 데이터만 선택해서 조회
*/
@RestController
@RequiredArgsConstructor
public class OrderSimpleApiController {
    private final OrderSimpleQueryRepository orderSimpleQueryRepository;

    @GetMapping("/api/v4/simple-orders")
    public List<OrderSimpleQueryDto> ordersV4() {
        return orderSimpleQueryRepository.findOrderDtos();
    }
}
```

[코드 7.21] OrderSimpleApiController.java

* 주의사항

1. 일반적인 SQL을 사용할 때처럼 원하는 값을 선택해서 조회한다.
2. new 명령어를 사용해서 JPQL의 결과를 DTO로 즉시 변환한다.
   SELECT 절에서 원하는 데이터를 직접 선택한다.
3. Repository 재사용성이 떨어진다. API 스펙에 맞춘 코드가 Repository에 포함되는 단점이 있다.

* 요약정리

엔티티를 DTO로 변환하거나, DTO로 조회하는 두가지 방법은 각각 장단점이있다.
엔티티로 조회하면 Repository 재사용성도 좋고, 개발도 단순해진다.
권장하는 방법은 다음과 같다.

쿼리 방식 선택 권장 순서

1. 우선 엔티티를 DTO로 변환하는 방법을 선택한다.
2. 필요하면 Fetch Join으로 성능을 최적화 한다. 대부분의 성능 이슈가 해결된다.
3. API 스펙에 맞추어진 DTO로 직접 조회하는 방법을 사용한다.
4. 최후의 방법은 JPA가 제공하는 네이티브 SQL이나 스프링 JDBC Template을 사용해서 SQL을 직접 사용한다.

7.3 API 개발 고급 – 컬렉션 조회 성능 최적화

주문내역에서 주문한 상품 정보를 추가로 조회하자.
Order 기준으로 컬렉션인 OrderItem 와 Item 을 조회한다.

7.3.1 주문 조회 Version 1

: 응답 값으로 엔티티를 직접 외부에 노출

: toMany(OneToMany) 컬렉션인 일대다 관계를 조회하고 최적화 하는 방법

```
package jpastudy.jpashop.api;


@RestController
@RequiredArgsConstructor
public class OrderApiController {
        private final OrderRepository orderRepository;


       @GetMapping("/api/v1/orders")
      public List<Order> ordersV1() {
          List<Order> all = orderRepository.findAll(new OrderSearch());
          for (Order order : all) {
                order.getMember().getName(); //Lazy 강제 초기화
                order.getDelivery().getAddress(); //Lazy 강제 초기화
                List<OrderItem> orderItems = order.getOrderItems();
                orderItems.stream().forEach(o -> o.getItem().getName()); //Lazy 강제초기화
          }
          return all;
    }
}
```

[코드 7.22] OrderApiController.java

- orderItem, item 관계를 직접 초기화 하면 Hibernate5Module 설정에 의해 엔티티를 JSON으로 생성한다.
- 양방향 연관관계이면 무한 루프가 발생하지 않도록 한곳에 @JsonIgnore 를 추가해야 한다.
- 엔티티를 직접 노출하므로 좋은 방법이 아니다.

7.3.2 주문 조회 Version 2

: 엔티티를 DTO로 변환하기

```
@GetMapping("/api/v2/orders")
public List<OrderDto> ordersV2() {
        List<Order> orders = orderRepository.findAll(new OrderSearch());
        List<OrderDto> result = orders.stream()
                        .map(o -> new OrderDto(o))
                        .collect(Collectors.toList());
       return result;
}
@Data
static class OrderDto {
    private Long orderId;
    private String name;
    private LocalDateTime orderDate;
    private OrderStatus orderStatus;
    private Address address;
    private List<OrderItemDto> orderItems;

    public OrderDto(Order order) {
        orderId = order.getId();
        name = order.getMember().getName();
        orderDate = order.getOrderDate();
        orderStatus = order.getStatus();
        address = order.getDelivery().getAddress();
        orderItems = order.getOrderItems().stream()
                        .map(orderItem -> new OrderItemDto(orderItem))
                        .collect(Collectors.toList());
    }
} //static class OrderDto
```

[코드 7.23] OrderApiController.java

```
@Data
static class OrderItemDto {
    private String itemName; //상품 명
    private int orderPrice; //주문 가격
    private int count; //주문 수량

    public OrderItemDto(OrderItem orderItem) {
        itemName = orderItem.getItem().getName();
        orderPrice = orderItem.getOrderPrice();
        count = orderItem.getCount();
    }
} //static class OrderItemDto
```

[코드 7.23] OrderApiController.java

- 지연 로딩으로 너무 많은 SQL을 실행한다.
- SQL 실행 수

  order 1번

  member , address N번(order 조회 수 만큼)

  orderItem N번(order 조회 수 만큼)

  item N번(orderItem 조회 수 만큼)

* 참고: 지연 로딩은 영속성 컨텍스트에 있으면 영속성 컨텍스트에 있는 엔티티를 사용하고 없으면 SQL을 실행한다. 따라서 같은 영속성 컨텍스트에서 이미 로딩한 회원 엔티티를 추가로 조회하면 SQL을 실행하지 않는다.

7.3.3 주문 조회 Version 3

: 엔티티를 DTO로 변환하기 – fetch join 최적화

: OrderRepository에 추가

```
public List<Order> findAllWithItem() {
    return em.createQuery(
    "select distinct o from Order o" +
            " join fetch o.member m" +
            " join fetch o.delivery d" +
            " join fetch o.orderItems oi" +
            " join fetch oi.item i", Order.class)
        .getResultList();
}
```

[코드 7.24] OrderRepository.java

```
@GetMapping("/api/v3/orders")
public List<OrderDto> ordersV3() {
    List<Order> orders = orderRepository.findAllWithItem();
    List<OrderDto> result = orders.stream()
            .map(o -> new OrderDto(o))
            .collect(Collectors.toList());
    return result;
}
```

[코드 7.25] OrderApiController.java

백명숙(msbaek@hucloud.co.kr)

⇨ Fetch Join으로 SQL이 1번만 실행됨

1) distinct 를 사용한 이유는 1대다 조인이 있으므로 데이터베이스 row가 증가한다. 그 결과 같은 order 엔티티의 조회 수도 증가하게 된다.

2) JPA의 distinct는 SQL에 distinct를 추가하여 같은 엔티티가 조회되면, 애플리케이션에서 중복을 걸러준다. order가 컬렉션 fetch join 때문에 중복 조회 되는 것을 막아준다.
   - SQL의 distinct는 중복된 결과를 제거하는 명령
   - JPQL의 distinct는 2가지 기능 제공
     1. SQL에 DISTINCT를 추가한다.
     2. 어플리케이션에서 엔티티 중복을 제거한다.

3) Fetch Join의 단점

   : Fetch Join을 사용하면 페이징이 불가능하다. 하이버네이트는 경고 로그를 남기면서 모든 데이터를 DB에서 읽어오고, 메모리에서 페이징 해버린다. (매우 위험)

   : firstResult / maxResults specified with collection fetch; applying in memory!

   : 둘 이상의 컬렉션은 페치 조인 할 수 없다.
   - 컬렉션 페치 조인은 1개만 사용할 수 있다. 컬렉션 둘 이상에 페치 조인을 사용하면 안된다. 데이터가 부정확하게 조회될 수 있다.

7.3.4 주문 조회 Version 3.1

: 엔티티를 DTO로 변환하기 – 페이징 문제 해결

⇨ 컬렉션을 fetch join 하면 페이징이 불가능 하다.

1) 컬렉션을 페치 조인하면 일대다 조인이 발생하므로 데이터가 예측할 수 없이 증가한다.
2) 일다대에서 일(1)을 기준으로 페이징을 하는 것이 목적이다. 그런데 데이터는 다(N)를 기준으로 row가 생성된다.
3) Order를 기준으로 페이징 하고 싶은데, 다(N)인 OrderItem을 조인하면 OrderItem이 기준이 되어 버린다.

⇨ 페이징 + 컬렉션 엔티티 조회 문제 해결

1) **ToOne**(OneToOne, ManyToOne) 관계를 모두 페치조인 한다.
   ToOne 관계는 row수를 증가 시키지 않으므로 페이징 쿼리에 영향을 주지 않는다.
2) **ToMany**(OneToMany, ManyToMany) 컬렉션은 지연 로딩으로 조회한다.
3) 지연 로딩 성능 최적화를 위해 hibernate.default_batch_fetch_size , @BatchSize 를 적용한다.
   : hibernate.default_batch_fetch_size는 글로벌 설정
   : @BatchSize: 개별 최적화 (컬렉션은 필드에, 엔티티는 클래스에 선언)
   이 옵션을 사용하면 컬렉션이나, 프록시 객체를 한꺼번에 설정한 size 만큼 IN 쿼리로 조회한다.

결론: ToOne 관계는 페치조인으로 쿼리 수를 줄여서 해결하고,
ToMany 관계는 hibernate.default_batch_fetch_size 로 최적화 하면 된다.

```yaml
spring:
  jpa:
    properties:
      hibernate:
        default_batch_fetch_size: 1000
```

[코드 7.26] main/resources/application.yml

참고: default_batch_fetch_size 의 크기는 적당한 사이즈를 골라야 하는데, 100~1000 사이의 값을 선택하는 것을 권장합니다. 이 전략을 SQL IN 절을 사용하는데, 데이터베이스에 따라 IN 절 파라미터를 1000으로 제한하기도 한다. 1000으로 잡으면 한번에 1000개를 DB에서 애플리케이션으로 로드하므로 DB에 순간 부하가 증가할 수 있다. 하지만 애플리케이션은 100이든 1000이든 결국 전체 데이터를 로딩해야 하므로 메모리 사용량이 같다.

: OrderRepository에 추가

```
public List<Order> findAllWithMemberDelivery(int offset, int limit) {
    return em.createQuery(
            "select o from Order o" +
                    " join fetch o.member m" +
                    " join fetch o.delivery d", Order.class)
            .setFirstResult(offset)
            .setMaxResults(limit)
            .getResultList();
}
```

[코드 7.27] OrderRepository.java

```
@GetMapping("/api/v3.1/orders")
public List<OrderDto> ordersV3_page(
    @RequestParam(value = "offset", defaultValue = "0") int offset,
    @RequestParam(value = "limit", defaultValue= "100") int limit) {
    List<Order> orders = orderRepository.findAllWithMemberDelivery(offset,limit);
    List<OrderDto> result = orders.stream()
            .map(o -> new OrderDto(o))
            .collect(Collectors.toList());
    return result;
}
```

[코드 7.28] OrderApiController.java

7.3.5 주문 조회 Version 4

: JPA에서 DTO 직접 조회

ToOne(N:1, 1:1) 관계들을 먼저 조회하고, ToMany(1:N) 관계는 각각 별도로 처리한다.

1) ToOne 관계는 조인해도 데이터 row 수가 증가하지 않는다.
- ToOne 관계는 조인으로 최적화 하기 쉬우므로 한번에 조회한다.

2) ToMany(1:N) 관계는 조인하면 row 수가 증가한다.
- ToMany 관계는 최적화 하기 어려우므로 findOrderItems() 같은 별도의 메서드로 조회한다.

```
package jpastudy.jpashop.repository.order.query;
import com.fasterxml.jackson.annotation.JsonIgnore;
import lombok.Data;
@Data
public class OrderItemQueryDto {
    @JsonIgnore
    private Long orderId; //주문번호
    private String itemName;//상품명
    private int orderPrice; //주문가격
    private int count; //주문수량

    public OrderItemQueryDto(Long orderId, String itemName, int orderPrice, int count) {
        this.orderId = orderId;
        this.itemName = itemName;
        this.orderPrice = orderPrice;
        this.count = count;
    }
}
```

[코드 7.29] OrderItemQueryDto.java

백명숙(msbaek@hucloud.co.kr)

```
package jpastudy.jpashop.repository.order.query;
import jpastudy.jpashop.domain.Address;
import jpastudy.jpashop.domain.OrderStatus;
import lombok.Data;
import lombok.EqualsAndHashCode;
import java.time.LocalDateTime;
import java.util.List;

@Data
@EqualsAndHashCode(of = "orderId")
public class OrderQueryDto {
    private Long orderId;
    private String name;
    private LocalDateTime orderDate;
    private OrderStatus orderStatus;
    private Address address;
    private List<OrderItemQueryDto> orderItems;

    public OrderQueryDto(Long orderId, String name, LocalDateTime orderDate,
    OrderStatus orderStatus, Address address) {
        this.orderId = orderId;
        this.name = name;
        this.orderDate = orderDate;
        this.orderStatus = orderStatus;
        this.address = address;
    }
}
```

[코드 7.30] OrderQueryDto.java

백명숙(msbaek@hucloud.co.kr)

```
package jpastudy.jpashop.repository.order.query;
import lombok.RequiredArgsConstructor;
import org.springframework.stereotype.Repository;
import javax.persistence.EntityManager;
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;

@Repository
@RequiredArgsConstructor
public class OrderQueryRepository {
    private final EntityManager em;

    /**
     * 1:N 관계(컬렉션)를 제외한 Order, Member, Delivery를 한번에 조회
     */
    private List<OrderQueryDto> findOrders() {
        return em.createQuery(
                "select new
                jpastudy.jpashop.repository.order.query.OrderQueryDto(o.id, m.name, o.orderDate,
                o.status, d.address)" +
                        " from Order o" +
                        " join o.member m" +
                        " join o.delivery d", OrderQueryDto.class)
                .getResultList();
    } //findOrders
```

[코드 7.31] OrderQueryRepository.java

백명숙(msbaek@hucloud.co.kr)

```
    /**
    * 1:N 관계인 orderItems 조회
    */
    private List<OrderItemQueryDto> findOrderItems(Long orderId) {
          return em.createQuery(
                  "select new
                  jpastudy.jpashop.repository.order.query.OrderItemQueryDto(oi.order.id, i.name,
                  oi.orderPrice, oi.count)" +
                  " from OrderItem oi" +
                  " join oi.item i" +
                  " where oi.order.id = : orderId", OrderItemQueryDto.class)
                        .setParameter("orderId", orderId)
                        .getResultList();
     } //findOrderItems

     public List<OrderQueryDto> findOrderQueryDtos() {
       //루트 조회(toOne 코드를 모두 한번에 조회)
       List<OrderQueryDto> result = findOrders();
       //루프를 돌면서 컬렉션 추가(추가 쿼리 실행)
       result.forEach(o -> {
             List<OrderItemQueryDto> orderItems = findOrderItems(o.getOrderId());
             o.setOrderItems(orderItems);
       });
       return result;
     } //findOrderQueryDtos

}//class
```

[코드 7.31] OrderQueryRepository.java

백명숙(msbaek@hucloud.co.kr)

```
@RestController
@RequiredArgsConstructor
public class OrderApiController {
    private final OrderQueryRepository orderQueryRepository;

    @GetMapping("/api/v4/orders")
    public List<OrderQueryDto> ordersV4() {
        return orderQueryRepository.findOrderQueryDtos();
    }
}
```

[코드 7.32] OrderApiController.java

7.3.6 주문 조회 Version 5

: JPA에서 DTO 직접 조회 – Java8 Stream API를 사용한 최적화

```
private List<Long> toOrderIds(List<OrderQueryDto> result) {
    return result.stream()
            .map(o -> o.getOrderId())
            .collect(Collectors.toList());
}
private Map<Long, List<OrderItemQueryDto>> findOrderItemMap(List<Long> orderIds) {
    List<OrderItemQueryDto> orderItems = em.createQuery(
            "select new
            jpastudy.jpashop.repository.order.query.OrderItemQueryDto(oi.order.id, i.name,
            oi.orderPrice, oi.count)" +
            " from OrderItem oi" +
            " join oi.item i" +
            " where oi.order.id in :orderIds", OrderItemQueryDto.class)
                .setParameter("orderIds", orderIds)
                .getResultList();
    return orderItems.stream().collect(Collectors.groupingBy(OrderItemQueryDto::getOrderId));
}

public List<OrderQueryDto> findAllByDto_optimization() {
   //toOne 코드를 모두 한번에 조회
   List<OrderQueryDto> result = findOrders();
   //orderItem 컬렉션을 MAP 한방에 조회
   Map<Long, List<OrderItemQueryDto>> orderItemMap =
            findOrderItemMap(toOrderIds(result));
   //루프를 돌면서 컬렉션 추가(추가 쿼리 실행X)
   result.forEach(o -> o.setOrderItems(orderItemMap.get(o.getOrderId())));
   return result;
}
```

[코드 7.31] OrderQueryRepository.java

백명숙(msbaek@hucloud.co.kr)

- ToOne 관계들을 먼저 조회하고, 여기서 얻은 식별자 orderId로 ToMany 관계인 OrderItem 을 한꺼번에 조회
- MAP을 사용해서 매칭 성능 향상

```
@GetMapping("/api/v5/orders")
public List<OrderQueryDto> ordersV5() {
    return orderQueryRepository.findAllByDto_optimization();
}
```

[코드 7.32] OrderApiController.java

7.4 API 개발 고급 정리

7.4.1 Entity 를 직접 사용한 조회

Ver1 - Entity를 조회해서 그대로 반환
Ver2 - Entity 조회 후 DTO로 변환
Ver3 - fetch 조인으로 쿼리 수 최적화
Ver3.1 - 컬렉션 페이징과 한계 돌파
컬렉션은 fetch 조인시 페이징이 불가능
: ToOne 관계는 fetch 조인으로 쿼리 수 최적화
: 컬렉션은 fetch 조인 대신에 지연 로딩을 유지하고,
: hibernate.default_batch_fetch_size, @BatchSize 로 최적화

7.4.2 DTO 를 직접 사용한 조회

Ver4 - JPA에서 DTO를 사용하여 직접 조회
Ver5 - 컬렉션 조회 최적화 - 일대다 관계인 컬렉션은 IN 절을 활용해서 메모리에 미리 조회해서 최적화

7.4.3 권장하는 방식

1. 엔티티 조회 방식으로 우선 접근
: ToOne관계는 fetch 조인으로 쿼리 수를 최적화
: ToMany관계의 컬렉션 fetch 조인으로 최적화(페이징 처리 않됨)
- 페이징이 필요하면 hibernate.default_batch_fetch_size , @BatchSize 로 최적화

2. 엔티티 조회 방식으로 해결이 안되면 DTO 조회 방식 사용
3. DTO 조회 방식으로 해결이 안되면 NativeSQL or 스프링 JdbcTemplate

엔티티 조회 방식 vs DTO 직접조회 방식

- 엔티티 조회 방식은 JPA가 많은 부분을 최적화 해주기 때문에, 단순한 코드를 유지하면서, 성능을 최적화 할 수 있다.
  즉, fetch 조인이나, hibernate.default_batch_fetch_size , @BatchSize 같이 코드를 거의 수정하지 않고, 옵션만 약간 변경해서, 다양한 성능 최적화를 시도할 수 있다.
- DTO를 직접 조회하는 방식은 성능을 최적화 하거나 성능 최적화 방식을 변경할 때 많은 코드를 변경해야 하며, SQL을 직접 다루는 것과 유사하다.

DTO 직접조회 방식 내부에서의 선택

- V4는 코드가 단순하다. 특정 주문 한건만 조회하면 이 방식을 사용해도 성능이 잘 나온다. 예를 들어서 조회한 Order 데이터가 1건이면 OrderItem을 찾기 위한 쿼리도 1번만 실행하면 된다.
- V5는 코드가 복잡하다. 여러 주문을 한꺼번에 조회하는 경우에는 V4 대신에 이것을 최적화한 V5 방식을 사용해야 한다.

- 예를 들어서 조회한 Order 데이터가 1000건인데, V4 방식을 그대로 사용하면, 쿼리가 총 1 + 1000번 실행된다. 1은 Order 를 조회한 쿼리고, 1000은 조회된 Order의 row 수 이다.
- V5방식으로 최적화 하면 쿼리가 총 1 + 1번만 실행된다. 상황에 따라 다르겠지만 운영 환경에서 100배 이상의 성능 차이가 날 수 있다.

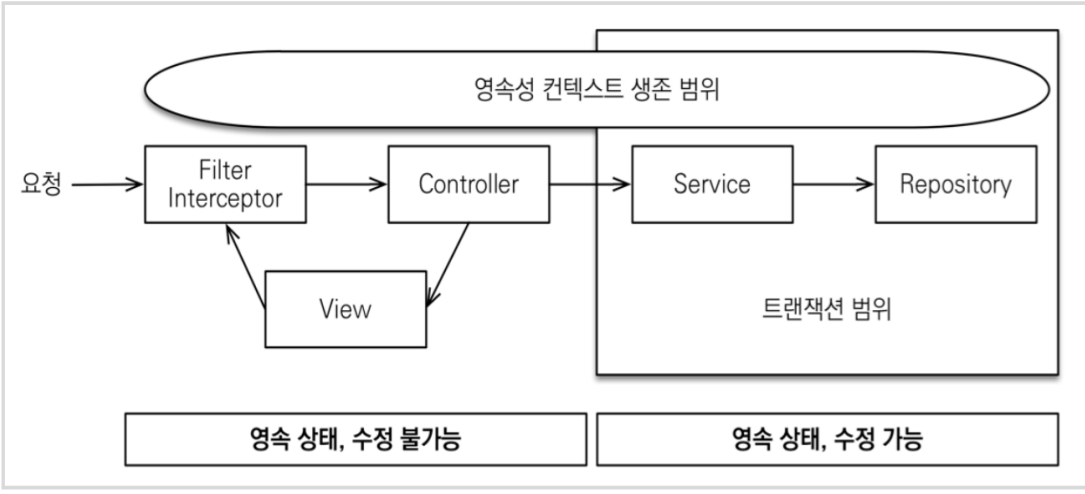
7.5 OSIV 와 성능 최적화

7.5.1 OSIV 란?

- Open Session In View의 약자로 View 계층까지 Session 영역을 열어 둔다는 의미이다.
- OSIV는 세션(JPA에서는 Entity Manager)의 생명 주기를 설정할 수 있는 옵션이다.
- Entity Manager는 데이터베이스 커넥션의 생명 주기를 같이 가지고 있다. Entity Manager가 생성될 때 데이터 베이스 커넥션을 생성하여 Entity Manager가 사라질 때 데이터 베이스의 커넥션은 반환된다.
- Entity Manager는 트랜잭션 시작시 생성된다.

spring.jpa.open-in-view: true (기본값)

: 데이터베이스 커넥션 시작 시점부터 API 응답이 끝날 때까지 영속성 컨텍스트와 데이터베이스 커넥션을 유지한다.



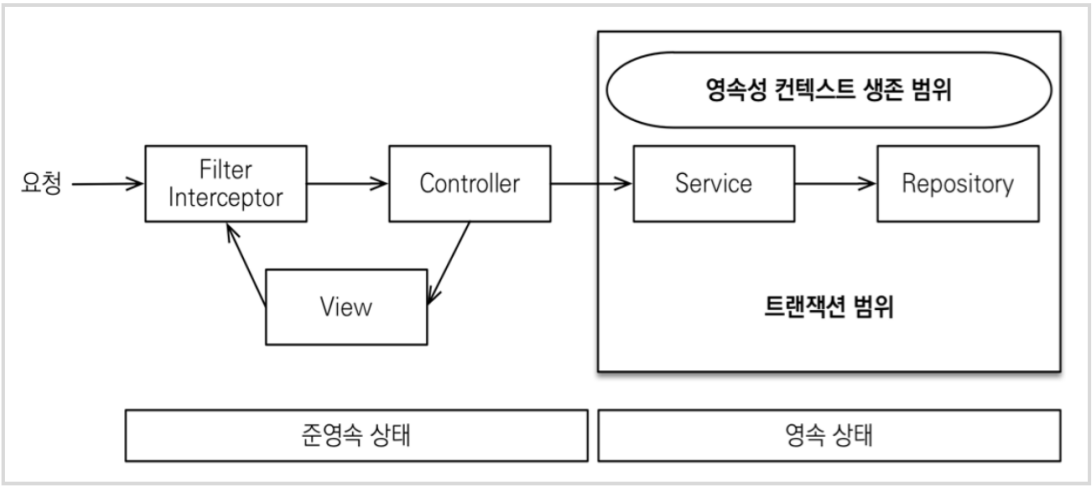
장점: 컨트롤러에서 지연로딩이 가능하다. (코딩이 편하다)

단점: 커넥션을 물고 있는 주기가 길어지고 커넥션을 빨리 반환하지 않으면 커넥션풀로 인해 장애로 이어질 수 있다.

- 트랜잭션 시작처럼 최초 데이터베이스 커넥션 시작 시점부터 API 응답이 끝날 때까지 영속성 컨텍스트와 데이터베이스 커넥션을 유지하므로 View Template 이나 API 컨트롤러에서 지연 로딩이 가능하다. (지연 로딩은 영속성 컨텍스트가 살아 있어야 가능하고, 영속성 컨텍스트는 기본적으로 데이터베이스 커넥션을 유지한다.)
- 이 전략은 너무 오랜 시간동안 데이터베이스 커넥션 리소스를 사용하기 때문에, 실시간 트래픽이 중요한 애플리케이션에서는 커넥션이 모자랄 수 있다. 이것은 결국 장애로 이어질 수도 있다.

spring.jpa.open-in-view: false

: 트랜잭션 시작시 엔티티 매니저, 데이터베이스 커넥션을 얻어오고 트랜잭션 종료시 반환된다.



장점: 데이터 베이스 커넥션 반환 주기가 빨라져 성능이 향상된다.

단점: 지연로딩을 위해 서비스단에서 강제로 지연로딩을 하여 프록시를 초기화 해줘야 하기 때문에 서비스단의 복잡도가 커진다.

- OSIV를 끄면 트랜잭션을 종료할 때 영속성 컨텍스트를 닫고, 데이터베이스 커넥션도 반환한다. 따라서 커넥션 리소스를 낭비하지

않는다.

- OSIV를 끄면 모든 지연로딩을 트랜잭션 안에서 처리해야 한다. 따라서 지금까지 작성한 많은 지연 로딩 코드를 트랜잭션 안으로 넣어야 하는 단점이 있다.
- view template에서 지연로딩이 동작하지 않는다. 결론적으로 트랜잭션이 끝나기 전에 지연 로딩을 강제로 호출해 두어야 한다.

7.5.2 OSIV Off 일 때 복잡성을 관리하는 방법

핵심비지니스 로직과 화면이나 API에 맞춰진 로직을 분리하자.

OrderService

: OrderService: 핵심 비즈니스 로직

: OrderQueryService: 화면이나 API에 맞춘 서비스
(주로 읽기 전용 트랜잭션 사용)

- 보통 Service 계층에서 트랜잭션을 유지한다. 두 Service 모두 트랜잭션을 유지하면서 지연 로딩을 사용할 수 있다.
- 실시간 API 업무는 OSIV를 끄고, 관리자 업무 (ADMIN) 처럼 커넥션을 많이 사용하지 않는 업무는 OSIV를 켜는 것이 편하다.