2020-12-14

SQL 중심적인 개발의 문제점

객체지향언어 관계형 DB -> SQL 중심적인 개발 (무한 반복 ,지루한 코드 CRUD)

필드 추가나 수정시 모든 SQL문을 수정을 해야한다.

SQL에 의존적인 개발을 피하기 어렵다 .

패러다임의 불일치

**\*객체를 자바 컬렉션에 저장 하듯이 DB에 저장할 수는 없을까?**

JPA –Java Persistence API

ORM?

Object-relational mapping(객체 관계 매핑)

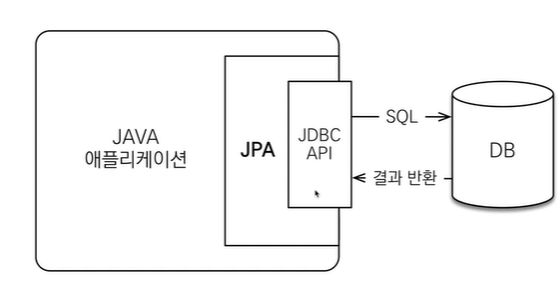
객체는 객체대로 설계

관계형 데이터베이스는 관계형 데이터 베이스대로 설계

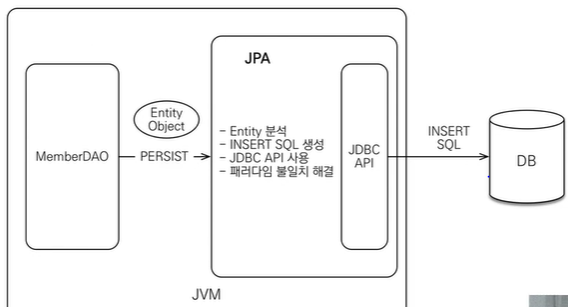
ORM 프레임워크가 중간에서 매핑

대중적인 언어에는 대부분 ORM 기술이 존재

JPA는 애플리케이션과 JDBC 사이에서 동작



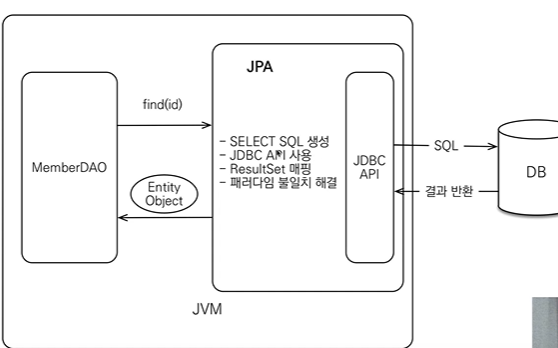
데이터 생성



\*패러다임 불일치 해결이 중요하다 .!

SQL을 JPA가 만들어주는것이다 .

데이터조회

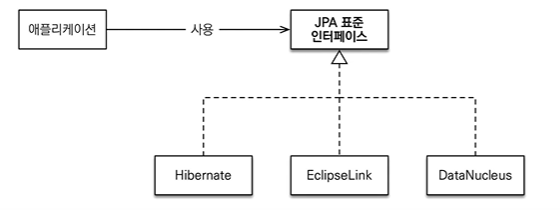


JPA 소개

EJB – 엔티티빈 (자바 표준) -> 하이버네이트(오픈 소스) -> JPA(자바 표준)

성능도 좋지않다 등등 -> 오픈소스를 통해서 많이 발전함 -> 하이버네이트정리해서 만듬

JPA는 인터페이스의 모음



JPA를 왜 사용해야 하는가 ?

SQL 중심적인 개발에서 객체 중심으로 개발-> 객체 상속관계 ,

생산성, 유지보수 -> CRUD가 만들어져있다 .,필드 추가시 모든 SQL이 알아서 수정된다.

패러다임의 불일치 해결

성능

데이터 접근 추상화와 벤더 독립성 //자유로운 객체 그래프탐색 가능

표준

JPA의 성능 최적화 기능

1차캐시와 동일설 보장 -> 약간의 조회 성능 향상

트랜잭션을 지원하는 쓰기 지연 em.persist // // transaction .commit

지연 로딩 : 객체가 실제 사용될 때 로딩 != ( 즉시 로딩 JOIN SQL로 한번에 연관된 객체까지 미리조회)

ORM은 객체와 RDB 두 기둥위에 있는 기술

2020-12-17

데이터 베이스 방언

JPA는 특정 데이터베이스에 종속 되어있지 않는다.

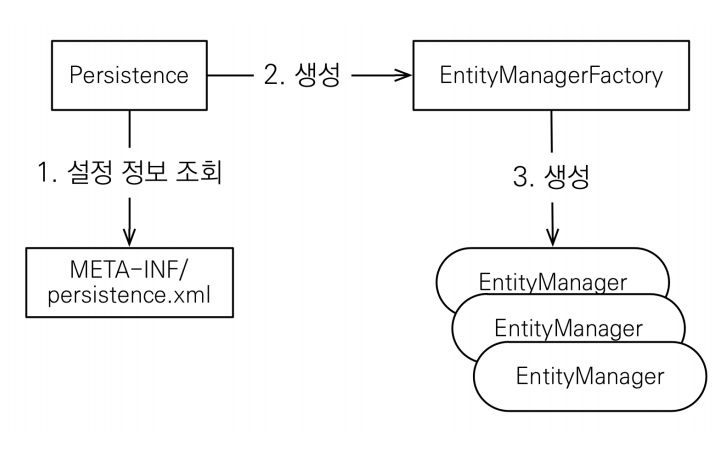
각각의 데이터베이스가 제공하는 sql 문법과 함수는 조금씩 다름

가변문자 :MySQL은 VARCHAR ,Oracle 은 VARCHAR2

문자열을 자르는 함수 :SQL 표준은 SUBSTRING(), Oracle은 SUBSTR

페이징 : MySQL은 LIMIT oracle은 ROWNUM

JPA 구동방식



엔티티 매니저 팩토리는 하나만 생성해서 애플리 케이션 전체에서 공유

엔티티 매니저는 쓰레드간에 공유 X (사용하고 버려야 한다.)

JPA의 모든 데이터 변경은 트랜잭션 안에서 실행

JPQL

JPA를 사용하면 에티티 객체를 중심으로 개발

문제는 검색 쿼리

검색을 할 때도 테이블이 아니 엔티티 객체를 대상으로 검색

모든 DB데이터를 객체로 변환해서 검색하는 것은 불가능

애플리케이션이 필요한 데이터만 DB에서 불러오려면 결국 검색 조건이 포함된 SQL 이 필요

Jpa는 SQL을 추상화한 JPQL이라는 객체 지향 쿼리 언어 제공

SQL과 문법 유사, SELECT,FROM,WHERE,GROUP BY ,HAVING ,JOIN 지원

차이점

JPQL은 엔티티 객체를 대상으로 쿼리

SQL은 데이터 베이스 테이블을 대상으로 쿼리

영속성 관리

JAP 에서 가장 중요한 2가지

객체와 관계형 데이터베이스 매핑하기

영속성 컨텍스트

엔티티를 영구 저장하는 환경이라는 뜻.

EntitiyManager.persist(entity);

엔티티매니저,영속성 컨텍스트

영속성 컨텍스트는 논리적인 개념

눈에 보이지 않는다.

엔티티 매니저를 통해서 영속성 컨텍스트에 접근

엔티티의 생명주기

비영속

영속성 컨텍스트와 전혀 관계가 없는 새로운 상태 (객체만 생성한 상태 )

영속

영속성 컨텍스트에 관리되는 상태(em.persist())

준영속

영속성 컨텍스트에 저장되었다가 분리된 상태

영속-> 준영속

영속 상태의 엔티티가 영속성 컨텍스트에서 분리(detached)

영속성 컨텍스트가 제공하는 기능을 사용 못함

Em.detach(entity)-특정 엔티티만 준영속 상태로 전환

em.clear() –영속성 컨텍스트를 완전히 초기화

em.close()-영속성 컨텍스트를 종료

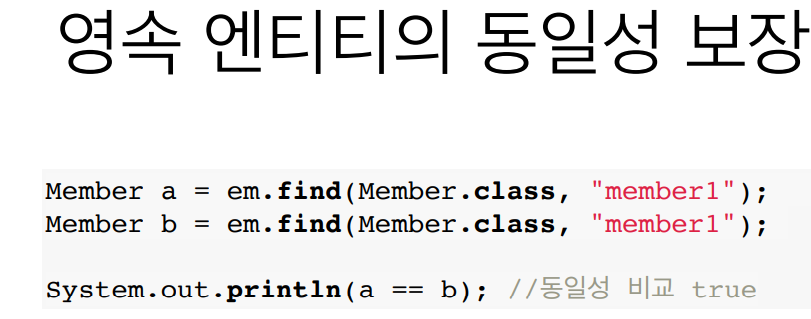
삭제

삭제된 상태

영속성 컨텍스트의 이점

1차 캐시: 똑같은 것을 두번 조회 할때는 캐시 에서 가져온다.

동일성(identity) 보장:



트랜잭션을 지원하는 쓰기 지연(transactional write- behind) :commit 하는 순간에 SQL문을 보낸다.

변경 감지(Dirty Checking): 컬렉션을 다루둣이 하기위해서 update()같은 것은 필요하지않는 다.

플러시 –영속성 컨텍스트의 변경내용을 데이터 베이스에 반영,

영속성 컨텍스트를 비우지 않음,영속성 컨텍스트의 변경 내용을 데이터

베이스에 동기화

트랜잭션이라는 작업 단위가 중요-> 커밋 직전에만 동기화 하면 됨

플러시 발생

변경감지,수정된 엔티티 쓰기 지연 SQL 저장소에 등록,

쓰기지연 SQL 저장소의 쿼리를 데이터 베이스에 전송

영속성 컨텍스트를 플러시 하는 방법

em.flush()-직접 호출 (1차캐시는 지워 지지 않는다 )

트랜잭션 커밋 –플러시 자동 호출

JPQL 쿼리 실행 –플러시 자동 호출

지연 로딩 (Lazy Loading)

2020-12-19

엔티티 매핑

객체와 테이블 매핑 :@Entitiy,@Table

@Entity

어노테이션이 붙은 클래스는 JPA가 관리, JPA를 사용해서 테이블과 매핑할 클래스는 필수

기본 생성자 필수 (파라미터가 없는 public 또는 protected 생성자)

Final 클래스,enum,interface,inner 클래스 사용 X

저장할 필드에 final 사용 X

@Table 은 엔티티와 매핑할 테이블 지정

Name ,catalog, schema,uniqueConstraints(DDL 생성 시에 유니크 제약 조건 생성 )

\*데이터 베이스 스키마 자동생성

DDL을 애플리케이션 실행 시점에 자동 생성

테이블 중심 -> 객체 중심

데이터베이스 방언을 활용해서 데이터 베이스에 맞는 적절한 DDL 생성

이렇게 생성된 DDL은 개발 장비에서만 사용

생성된 DDL은 운영서버에서는 사용하지않거나 적절히 다듬은후 사용

DDL 생성 기능 – 제약조건 추가 ( @Column(nullable = false,length=10))

유니크 제약조건 추가

DDL 생성기능은 DDL을 자동 생성할 때만 사용되고 JPA의 실행 로직에는 영향을 주지 않는다.

필드와 컬럼 매핑 :@Column

기본 키 매핑 :@Id

연관관계 매핑: @ManyToOne,@JoinColumn

2020-12-20

필드와 컬럼 매핑

@Colunm : 컬럼 매핑

속성

Name

Insertable,updateble

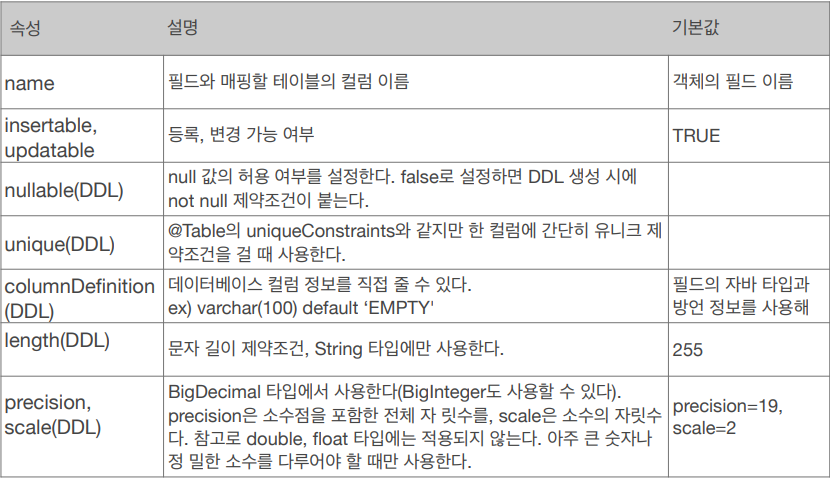
Nullable (not null)

Unique 사용하기 보다는 ->@Table(uniqueConstraints = ..)을 사용

columnDefinition

length

precision,scale

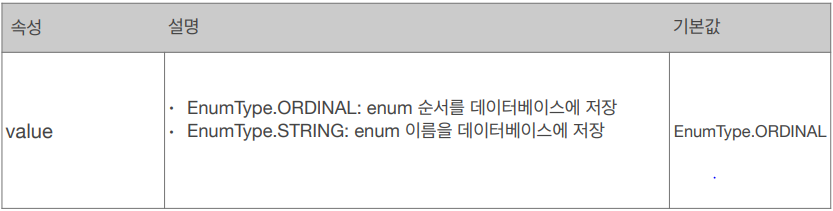


@Temporal : 날짜 타입 매핑 -> LocalDate,LocalDateTimer을 사용할떄는 생략 가능



@Enumerated: enum 타입 매핑

주의 ORDINAL 사용 X -> Enum에 다른 것을 추가시에 순서가 바뀔수도 있기 때문에 위험하다.



@Lob : BLOB,CLOB 매핑

지정할 수 있는 속성이 없다.

매핑하는 필드 타입이 문자면 CLOB매핑 , 나미저니느 BLOB매핑

@Transient : 특정 필드를 컬럼에 매핑하지 않음 (매핑 무시)

필드 매핑 X

데이터 베이스에 저장 X 조회 X

메모리상에서만 임시로 어떤 값을 보관하고 싶을 때 사용

기본키 매핑

@Id

직접 할당 할 때 사용

@GeneratedValue

IDENTITY : 데이터 베이스에 위임,MYSQL

데이터 베이스에 인서트를 해야 key값을 알수 있기 떄문에 쿼리를 미리 보내버린다.

그후 영속성 컨텍스트에 저장한다.

SEQUENCE: 데이터베이스 시퀀스 오브젝트 사용, ORACLE

@SequenceGenerator필요

영속성 컨텍스트에 넣기위해서 SEQUENCE이기 떄문에 시퀀스 번호를 가져온다.

allocationSize 50이라고 하게되면 한번에 50개를 다가져온후 메모리에서 사용한다.

51로만든후에 50개를 메모리상에 가져온다. 성능 최적화를 위함

TABLE:키 생성용 테이블 사용, 모든 DB에서 사용 ,단점은 성능이 조금 떨어진다.

@TableGenerator 필요

AUTO:방어넹 따라 자동 지정, 기본값

권장하는 식별자 권략

기본키 제약조건: Null아님, 유일,변하면 안된다.

미래까지 이 조건을 만족하는 자연키는 찾기 어렵다. 대리키를 사용하자

예를 들어 주민등록번호도 기본 키로 적절하지 않다 .

권장 -> Long형 + 대체키 + 키 생성전략 사용

데이터 중심 설계의 문제점

객체 설계를 테이블 설계에 맞춘방식

테이블의 외래키를 객체에 그대로 가져옴

객체 그래프 탐색이 불가능

찾모가 없으므로 UML도 잘못됨

2020-12-22

연관관계 매핑 기초

\*객체와 테이블 연관관계의 차이를 이해

\*객체의 참조와 테이블의 외래 키를 매핑

연관관계가 필요한 이유

객체 지향 설계의 목표는 자율적인 객체들의 협력 공동체를 만드는 것이다.

객체를 테이블에 맞추어 모델링 문제점

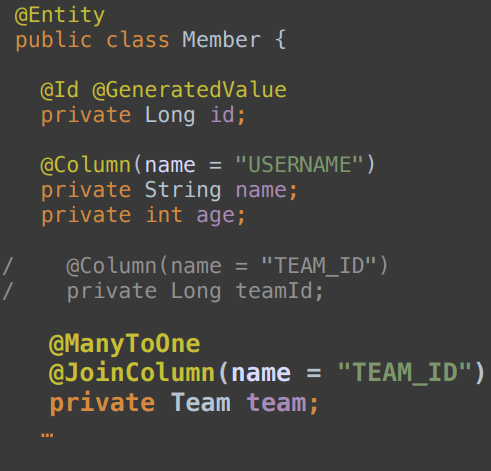
외래 키 식별자를 직접 다루게 된다.

식별자로 다시 조회, 객제 지향적인 방법이 아니다.

협력 관계를 만들 수 없다.

단방향 연관관계

객체를참조하기 위해서 외래키와 매핑을 시켜줘서 객체지향적으로 코딩 할 수 있다.



2020-12-22

양방향 연관관계와 연관관계의 주인



객체 연관관계 = 2개

회원 -> 팀 , 팀-> 회원

테이블 연관관계 =1개

회원<->팀의 연관관계 1개 하나만 있어도 양쪽을 다가져올 수 있다.

연관관계의 주인

객체의 두 관계중 하나를 연관관계의 주인으로 지정

연관관계의 주인만이 외래 키를 관리(등록, 수정)

주인이 아닌쪽은 읽기만 가능

주인은 mappedBy속성 사용 X

주인이 아니면 mappedBy 속성으로 주인 지정

외래 키가 있는 곳을 주인으로 정해라

필수 ! 주인에 값을 넣어라 하지만 앙쪽 넣어주는 것이 객체지향적으로 맞다.

순수 객체 상태를 고려해서 항상 양쪽에 값을 설정하자 .

연관관계 편의 메소드를 생성하자(예setter 아닌 다른 이름을 사용한다 )

양방향 매핑시에 무한 루프를 조심하자

예) toString(), Lombok, JSON 생성 라이브러리

단방향 매핑만으로도 이미 연관관계 매핑은 완료

양방향 매핑은 반대 방향으로 조회 기능이 추가 된 것 뿐

JPQL에서 역방향으로 탐색할 일이 많음

단방향 매핑을 잘 하고 양방향은 필요할 때 추가해도됨

다양한 연관관계 매핑

연관관계 매핑시 고려사항 3가지

다중성 -> 다대일 일대다 일대일 다대다(실무에서 쓰면 안된다)

단방향,양방향

테이블 -> 외래 키 하나로 양쪾 조인가능

사실 방향이라는 개념이 없음

객체 -> 참조용 필드가 있는 쪽으로만 참조 가능

한쪽만 참조하면 단방향

양쪽이 서로 참조하면 양방향

연관관계의 주인

테이블은 외래키 하나로 두테이블이 연관관계를 맺는다.

객체 양방향 관계는 A->B B->A 처럼 참조가 2군데이다

객체 양방향 관계는 참조가 2이다. 둘중 테이블의 외래키를 관리할 곳을 지정해야함

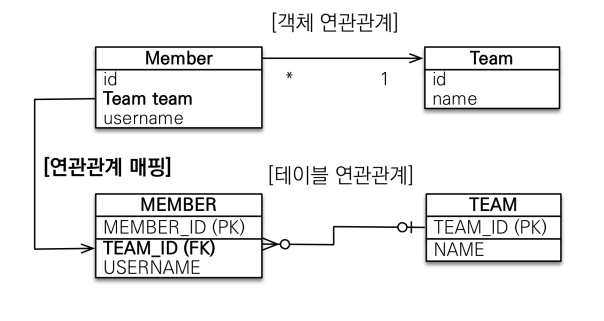
연관관계의 주인 -> 외래 키를 관리하는 참조

주인의 반대편 -> 외래키에 영향을 주지 않음 단순 조회만 가능

다대일

다대일 단방향

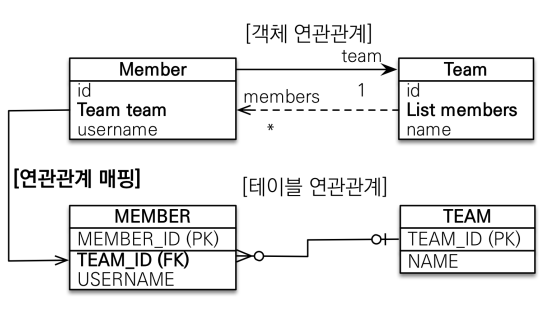
가장 많이 사용하는 연관관계



다대일 양방향

외래 키가 있는 쪽이 연관관계의 주인

양쪽을 서로 참조하도록 개발



일대다

일대다 단방향

1이 연관관계의 주인이다

테이블 일대다 관계는 항상 다 쪽에 외래 키가 있다.

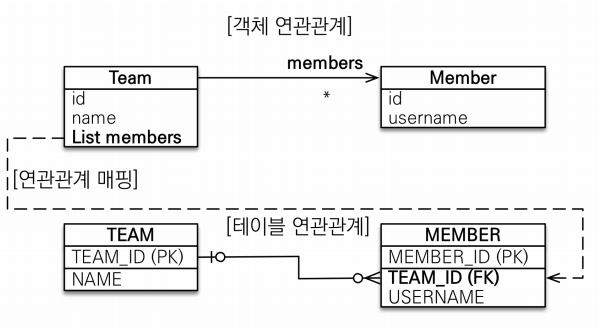
객체와 테이블의 차이 때문에 반대편 테이블의 외래 키를 관리하는 특이한 구조

@JoinColumn을 꼭 사용해야함, 그렇지 않으면 조인 테이블 방식을 사용함 (중간에 테이블을 하나 추가함)

단점 : 엔티티가 관리하는 외래키가 다른 테이블에 있음

연관관계 관리를 위해 추가로 UPDATE SQL 실행

일대다 단방향 매핑 보다는 다대일 양방향 매핑을 사용하자



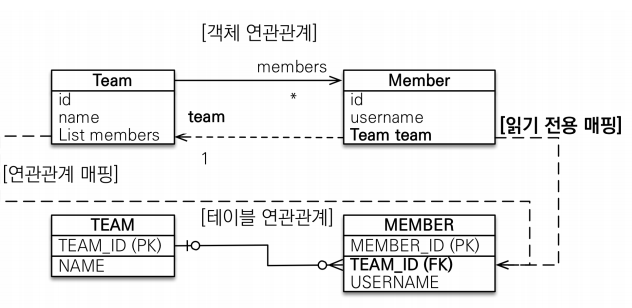
일대다 양방향

이런 매핑은 공식적으로 존재 X

@JoinColumn(insertable=false,updatable=false)

읽기 전용 필드를 사용해서 양방향 처럼 사용하는 방법

다대일 양방향을 사용하자 .



일대일

주 테이블이나 대상 테이블 중에 외래키 선택가능

주 테이블에 외래키

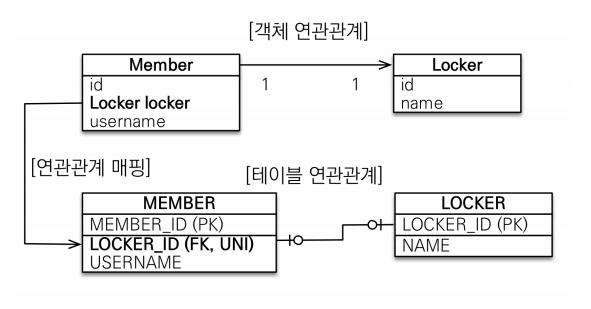
대상테이블에 외래키

외래키에 데이터베이스 유니크 제약조건 추가 (UNI)

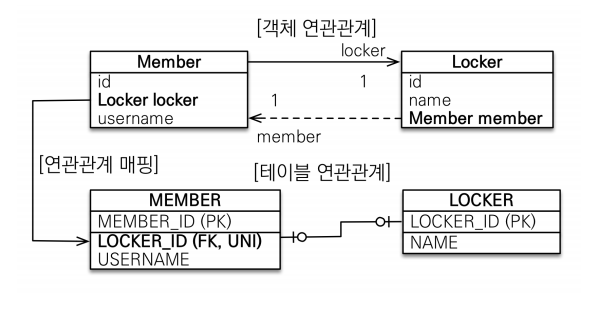
다대일 양방향 매핑 처럼 외래키가 있는 곳이 연관관계의 주인

반대편은 mappedBy적용

단방향



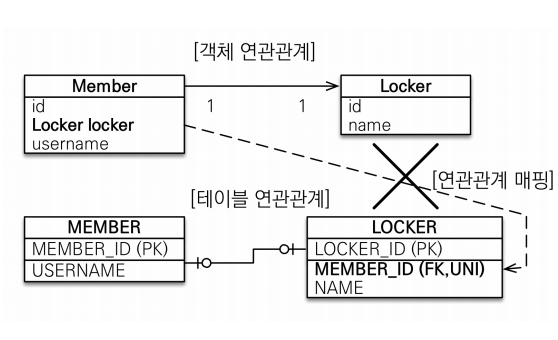
양방향

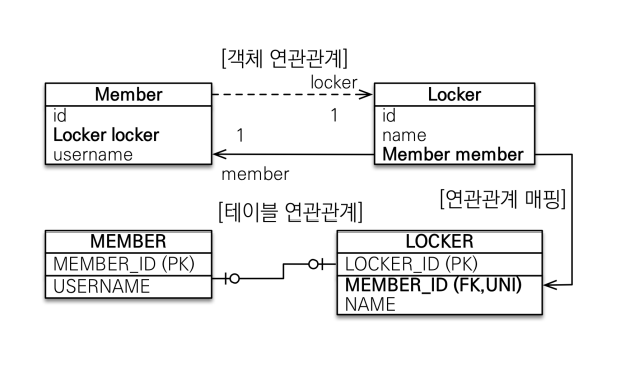


일대일:대상 테이블에 외래 키 단방향

단방향 관계는 JPA지원 X

양방향 관계는 지원





일대일 정리

주 테이블에 외래키

주 객체가 대상 객체의 참조를 가지는 것처럼 주테이블에 외래 키를 두고 대상 테이블을 찾음

객체지향 개발자 선호

JPA매핑 편리

장점 – 주테이블만 조회해도 대상 테이블에 데이터가 있는지 확인 가능

단점 – 값이 없으면 , 외래 키에 null 허용

대상 테이블에 외래 키

대상 테이블에 외래 키가 존재

전통적인 데이터 베이스 개발자 선호

장점 - 주 테이블과 대상 테이블을 일대일에서 일대다 관계로 변경할 때 테이블 구조 유지

단점 – 프록시 기능의 한계로 지연 로딩으로 설정해도 항상 즉시 로딩됨

다대다 -> 실무에서 쓰지않는 것이 좋다.

관계형 데이터 베이스는 정규화된 테이블 2개로 다대다 관계를 표현할 수 없음

연결 테이블을 추가해서 일대다, 다대일 관계로 풀어내야함

@ManyToMany 사용

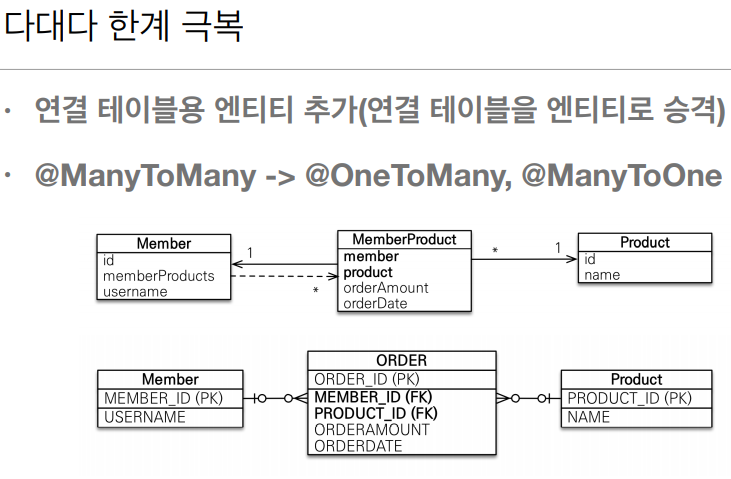
@JoinTable로 연결 테이블 지정

대다대 매핑: 단방향 , 양방향 가능

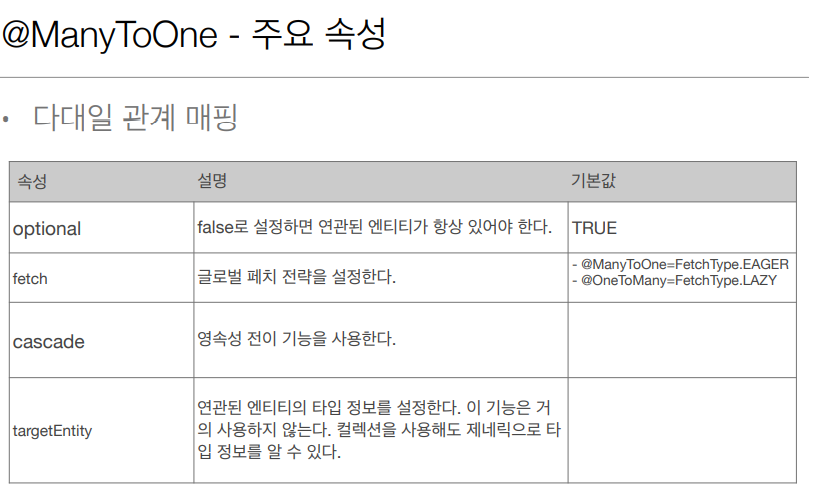
다대다 매핑의 한계

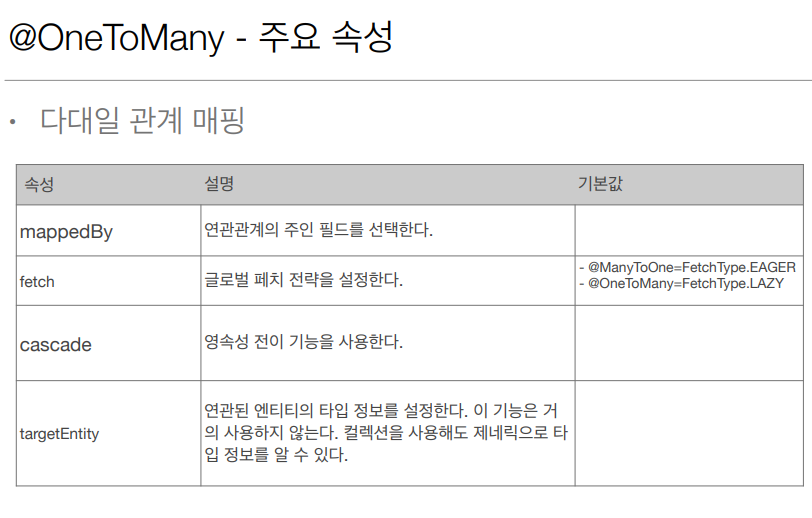
편리해 보이지만 실무에서 사용 X

연결 테이블이 단순히 연결만 하고 끝나지 않음









2020-12-28

고급 매핑

상속관계 매핑

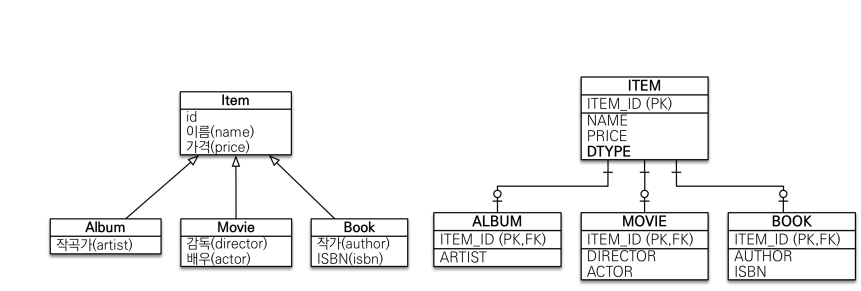
관계형 데이터베이스느 상속 관계가 없다.

슈퍼타입 서브타입 관계라는 모델링 기법이 객체 상송과 유사

상속관계 매핑 : 객체이 상속과 구조와 DB의 슈퍼타입 서브타입 관계를 매핑

슈퍼타입 서브타입 논리 모델을 실제 물리 모델로 구현하는 방법

각각 테이블로 변환 -> 조인 전략



장점

테이블 정규화

외래 키 참조 무결성 제약조건 활용 가능

저장공간 효율화

단점

조회시 조인을 많이 사용, 성능 저하

조회 쿼리가 복잡함

데이터 저장시 INSERT SQL 2번 호출

통합 테이블로 변환 -> 단일 테이블 전략

장점

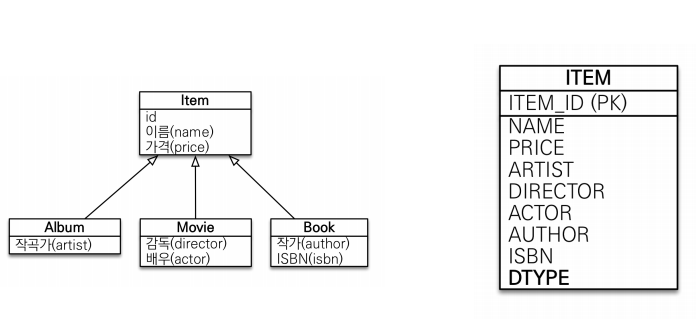
조인이 필요 없으므로 일반적으로 조회 성능이 빠름

조회 쿼리가 단순함

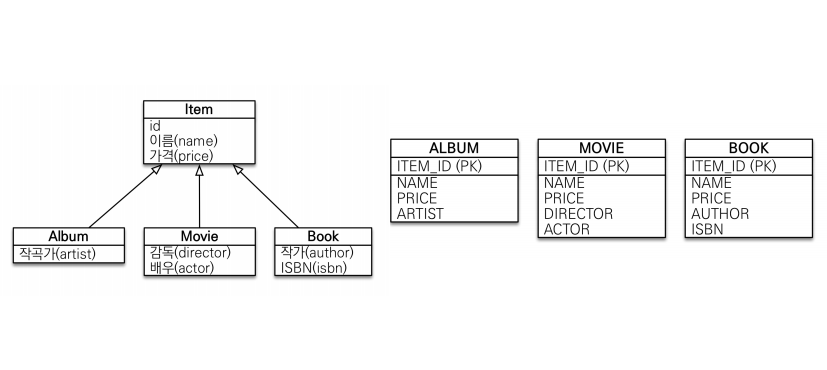
단점

자식 엔티티가 매핑한 컬럼은 모두 null 허용

단일 테이블에 모든 것을 저장하므로 테이블이 커질 수 있다. 상황에 따라서 조회 성능이 오히려 느려질 수 있다.



서브타입 테이블로 변환 -> 구현 클래스마다 테이블 전략



장점

서브 타입을 명확하게 구분해서 처리할 때 효과적

Not null 제약 조건 사용 가능

단점

여러 자식 테이블을 함께 조회할 때 성능이 느림(UNION SQL 필요 )

자식 테이블을 통합해서 쿼리하기 어려움

주요 어노테이션

@Inheritance(strategy = InheritanceType.XXX)

JOINED: 조인전략

SINGLE\_TABLE: 단일 테이블 전략

TABLE\_PER\_CLASS: 구현클래스마다 테이블 전략

* 조인전략시 입력 해주어야한다./단일 테이블 전략 일때는 DTYPE과 같은 것이 들어가져있다./ ,구현클래스마다 테이블은 사용할 필요가없다.

@DicriminatorColumn(name=”DTYPE”) 상위 클래스

@DiscriminatorValue(“XXX”) 하위 클래스

@MappedSuperclass

공통 매핑 정보가 필요할 때 사용 (예. Id,name)

상속관계 매핑 X

엔티티 X, 테이블과 매핑 X

부모 클래스를 상속 받는 자식클래스에 매핑 정보만 제공

조회 검색 불가 (em.find(BaseEntitiy ) 불가능)

직접 생성해서 사용할 일이 없으므로 추상 클래스 권장

테이블과 관계 없고, 단순히 엔티티가 공통으로 사용하는 매핑 정보를 모으는 역할

주로 등록일,수정일,등록자,수정자,같은 전체 엔티티에서 공통으로 적용하는 정보를 모을 때 사용

참고: @Entity 클래스는 엔티티나 @MappedSuperclass로 지 정한 클래스만 상속 가능

2020-19-29

프록시와 연관관계 관리

프록시

Em.find() = 데이터 베이스를 통해서 실제 엔티티 객체 조회

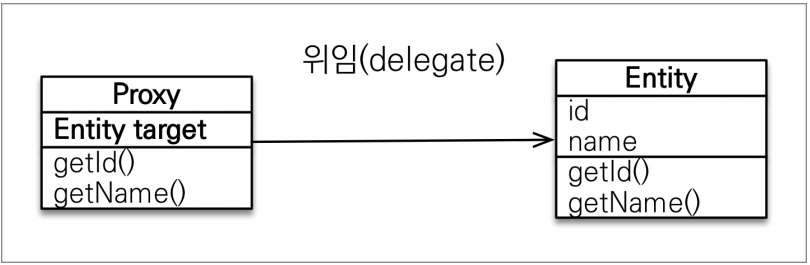
Em.getReference()= 데이터 베이스 조회를 미루는 가짜(프록시) 엔티티 객체 조회

특징

실제 클래스를 상속 받아서 만들어짐

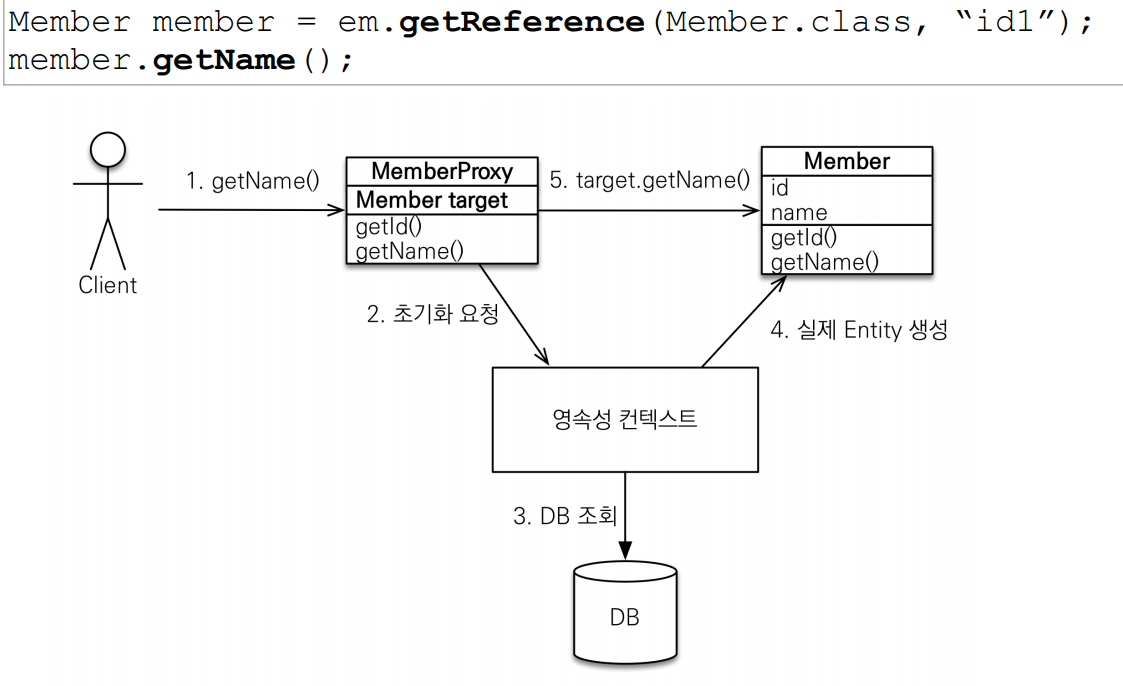
실제 클래스와 겉 모양이 같다.

사용하는 입장에서는 진짜 객체인지 프록시 객체인지 구분하지 않고 사용하면 됨(이론상)



프록시 객체는 실제 객체의 참조를 보관

프록시 객체를 호출하면 프록시 객체는 실제 객체의 메소드 호출

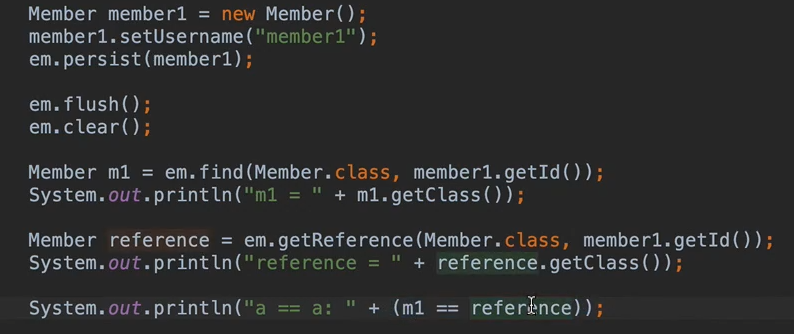


프록시 객체는 처음 사용할 때 한 번만 초기화

프록시 객체를 초기화 할 때, 프록시 객체가 실제 엔티티로 바뀌는 것은 아님, 초기화 되면 프록시 객체를 통해서 실제 엔티티에 접근가능

프록시 객체는 원본 엔티티를 상속받음, 따라서 타입 체크시 주의해야함 (== 비교 실패 , 대신 instance of 사용)

영속성 컨텍스트에 찾는 엔티티가 이미 있으면 em.getReference()를 호출해도 실제 엔티티 반환



\*JpA는 영속성 컨텍스트에서 가져오든 항상 == 은 true를 반환해야한다.

영속성 컨텍스트의 도움을 받을 수 없는 준 영속 상태 일 때, 프록시를 초기화 하면 문제 발생

(하이버네이트는 org.hibernate.LazyInitializationException 예외를 터트림)

프록시확인

프록시 인스턴스의 초기화 여부 확인 PersistenceUnitUtil.isLoaded(Object entity)

프록시 클래스 확인 방법 entity.getClass().getName() 출력(..javasist.. or HibernateProxy…)

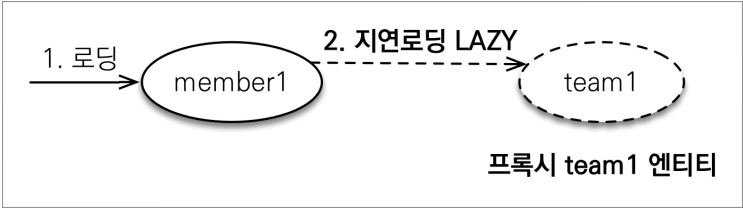
프록시 강제 초기화 org.hibernate.Hibernate.initialize(entity); -> 하이버네이트에서 제공하는 것.

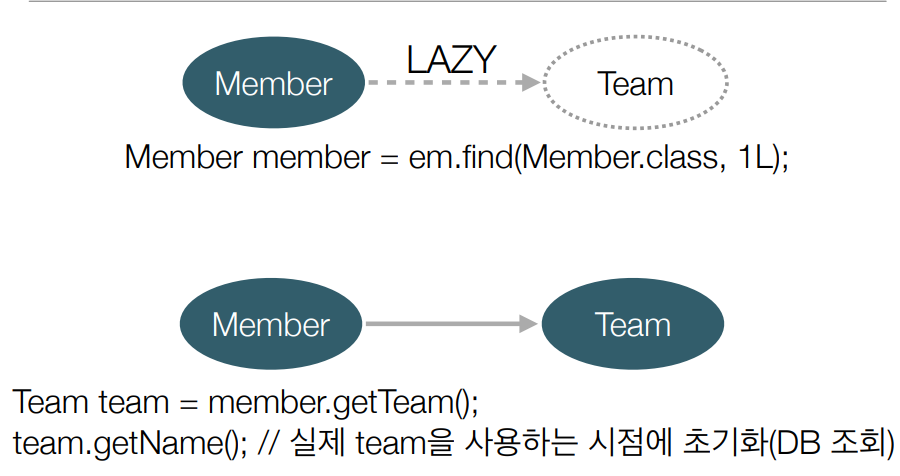
참고: JPA 표준은 강제 초기화 없음 강제 호출: member.getName()

즉시 로딩과 지연로딩

지연 로딩

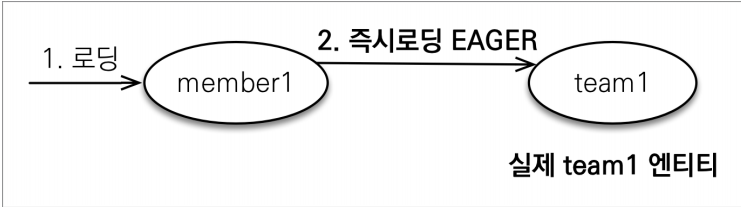
@ManyToOne(fetch = FetchType.Lazy)

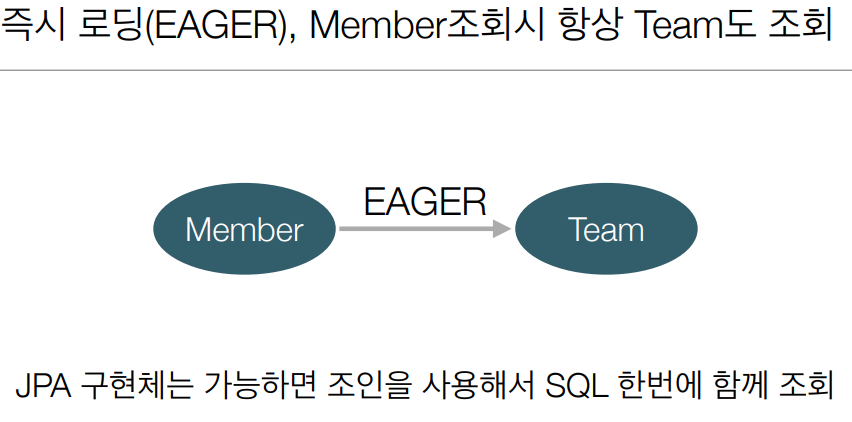




즉시 로딩

@ManyToOne(fetch=FetchType.EAGER)





프록시와 즉시로딩 주의

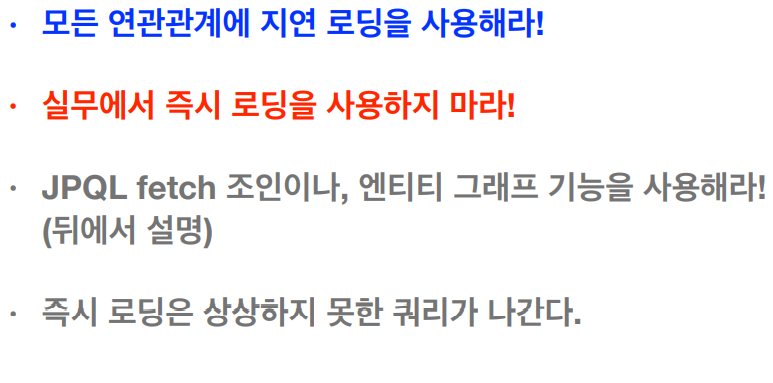
가급적 지연 로딩만 사용(특히 실무에서)

즉시 로딩을 적용하면 예상하지 못한 SQL이 발생

즉시 로딩은 JPQL에서 N+1 문제를 일으킨다.

@ManyToOne, @OneToOne은 기본이 즉시 로딩 -> LAZY로 설정

@OneToMany, @ManyToMany는 기본이 지연 로딩



영속성 전이 : CASCADE

특정 엔티티를 영속 상태로 만들 때 연관된 엔티티도 함께 영속상태로 만들고 싶을 때 .

예: 부모 엔티티를 저장할 때 자식 엔티티도 함께 저장.

@OneToMany(mappedBy="parent", cascade=CascadeType.PERSIST)

영속성 전이는 연관관계를 매핑하는 것과 아무 관련이 없음

엔티티를 영속화 할 때 연관된 엔티티도 함께 영속화 하는 편리함을 제공할 뿐.

종류

ALL: 모두 적용

PERSIST: 영속

REMOVE: 삭제

MERGE: 병합

REFRESH: REFRESH

DETACH: DETACH

고아 객체

고아 객체 제거 : 부모 엔티티와 연관관계가 끊어진 자식 엔티티를 자동으로 삭제

orphanRemoval = true

Parent parent1 = em.find(Parent.class, id); parent1.getChildren().remove(0); //자식 엔티티를 컬렉션에서 제거

DELETE FROM CHILD WHERE ID=?

주의

참조가 제거된 엔티티는 다른 곳에서 참조하지 않는 고아 객체로 보고 삭제하는 기능

참조하는 곳이 하나일 때 사용해야함!

특정 엔티티가 개인 소유할 때 사용

@OneToOne,@OneToMany 만 가능

개념적으로 부모를 제거하면 자식은 고아가 된다. 따라서 고 아 객체 제거 기능을 활성화 하면, 부모를 제거할 때 자식도 함께 제거된다. 이것은 CascadeType.REMOVE처럼 동작한다.

영속성 전이 + 고아 객체 , 생명주기

CascadeType.ALL + orphanRemovel=true

스스로 생명주기를 관리하는 엔티티는 em.persist()로 영속화, em.remove()로 제거

두 옵션을 모두 활성화 하면 부모 엔티티를 통해서 자식의 생명 주기를 관리할 수 있음

도메인 주도 설계(DDD)의 Aggregate Root개념을 구현할 때 유용

글로벌 페치 전략 설정

모든 연관관계를 지연 로딩으로

@ManyToOne, @OneToOne은 기본이 즉시 로딩이므로 지연 로딩으로 변경