08. 프록시와 연관관계 관리

프록시

기초

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 같음

- 사용하는 입장에서는 진짜 객체인지 프록시 객체인지 구분하지 않고 사용 (이론상)

- 프록시 객체는 실제 객체의 참조(target)를 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

- 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

- 영속성 컨텍스트의 도움을 받을 수 없는 준영속 상태일 때, 프록시를 초기화하면 문제 발생 (하이버네이트는 org.hibernate.LazyInitializationException 예외를 터트림)

프록시 확인

- 프록시 인스턴스의 초기화 여부 확인

PersistenceUnitUtil.isLoaded(Object entity)

- 프록시 클래스 확인 방법

Entity.getClass().getName() 출력

- 프록시 강제 초기화

Org.hibernate.Hibernate.intialize(entity)

즉시 로딩과 지연 로딩

단순히 member 정보만 사용하는 비즈니스 로직 : println(member.getName());

지연 로딩 LAZY을 사용해서 프록시로 조회

Member member = em.find(Member.class, 1L);

Team team = member.getTeam();

team.getName(); // 실제 team을 사용하는 시점에 초기화(DB 조회)

즉시 로딩 EAGER를 사용해서 함께 조회

- Member 조회시 항상 Team도 조회

- JPA 구현체는 가능하면 조인을 사용해서 SQL 한번에 함께 조회

프록시와 즉시 로딩 주의

- 가급적 지연 로딩만 사용

- 즉시 로딩을 적용하면 예상하지 못한 SQL이 발생

- 즉시 로딩은 JPQL에서 N+1 문제를 일으킴

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

- @OneToMany, @ManyToMany 는 기본이 지연 로딩

지연 로딩 활용

- Member와 Team은 자주 함께 사용 -> 즉시 로딩

- Member와 Order는 가끔 사용 -> 지연 로딩

- Order와 Product는 자주 함께 사용 -> 즉시 로딩

영속성 전이 : CASCADE

- 특정 엔티티를 영속 상태로 만들 때 연관된 엔티티도 함께 영속 상태로 만들고 싶은 경우

- 영속성 전이는 연관관계를 매핑하는 것과 아무 관련이 없음

- 엔티티를 영속화할 때 연관된 엔티티도 함께 영속화하는 편리함을 제공

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, @OneToMany만 가능

영속성 전이 + 고아 객체, 생명주기

- 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 Root 개념을 구현할 때 유용

09. 값 타입

\* JPA의 데이터 타입 분류

- 엔티티 타입

@Entity 로 정의하는 객체

데이터가 변해도 식별자로 지속해서 추적 가능

- 값 타입

Int , Integer, String처럼 단순히 값으로 사용하는 자바 기본 타입이나 객체

식별자가 없고 값만 있으므로 변경시 추적 불가

- 기본값 타입

생명주기를 엔티티의 의존

값 타입은 공유하면 x

절대 공유하지 않음, 기본 타입은 항상 값을 복사

Integer같은 래퍼 클래스나 String 같은 특수한 클래스는 공유 가능한 객체이지만 변경 x

- 임베디드 타입(복합 값 타입)

새로운 값 타입을 직접 정의할 수 있음

JPA는 임베디드 타입(embedded type)이라 함

주로 기본 값 타입을 모아서 만들어 복합 값 타입이라고도 함

Int, String과 같은 값 타입

@Embeddable : 값 타입을 정의하는 곳에 표시

@Embedded : 값 타입을 사용하는 곳에 표시

기본 생성자 필수

장점 – 재사용

높은 응집도

Period.isWork()처럼 해당 값 타입만 사용하는 의미있는 메소드를 만들 수 있음

임베디드 타입을 포함한 모든 값 타입은, 값 타입을 소유한 엔티티에

생명주기를 의존함

테이블 매핑

- 엔티티의 값

- 임베디드 타입을 사용하기 전과 후에 매핑하는 테이블은 같다.

- 객체와 테이블은 아주 세밀하게 매핑하는 것이 가능

- 잘 설계한 ORM 애플리케이션은 매핑한 테이블의 수보다

클래스의 수가 더 많음

- @AttributeOverride : 속성 재정의

임베디드 타입의 값이 null이면 매핑한 칼럼 값은 모두 null

- 값 타입과 불변 객체

값 타입 공유 참조

- 임베디드 타입 같은 값 타입을 여러 엔티티에서 공유하면 위험함

- 부작용 발생

값 타입 복사

- 값 타입의 실제 인스턴스인 값을 공유하는 것은 위험

- 대신 값(인스턴스)를 복사해서 사용

객체 타입의 한계

- 항상 값을 복사해서 사용하면 공유 참조로 인해 발생하는 부작용을 피할 수 있음

- 문제는 임베디드 타입처럼 직접 정의한 값 타입은 자바의 기본 타입이 아니라

객체 타입

- 자바 기본 타입에 값을 대입하면 값을 복사

- 객체 타입은 참조 값을 직접 대입하는 것을 막을 방법이 없음

- 객체의 공유 참조는 피할 수 없음

불변 객체

- 객체 타입을 수정할 수 없게 만들면 부작용을 원천 차단

- 값 타입은 불변 객체(immutable object)로 설계

- 불변 객체 : 생성 시점 이후 절대 값을 변경할 수 없는 객체

- 생성자로만 값을 설정하고 수정자(Setter)를 만들지 않으면 됨

- 참고 ; Integer, String은 자바가 제공하는 대표적인 불변 객체

- 값 타입의 비교

값 타입 : 인스턴스가 달라도 그 안에 값이 같으면 같은 것으로 봐야함

동일성 비교 : 인스턴스의 참조 값을 비교, == 사용

동등성 비교 : 인스턴스의 값을 비교, equals() 사용

값 타입은 a.equals(b)를 사용해서 동등성 비교를 해야 함

값 타입의 equals() 메소드를 적절하게 재정의(주로 모든 필드 사용)

- 값 타입 컬렉션

값 타입을 하나 이상 저장할 때 사용

@ElementCollection, @CollectionTable 사용

데이터베이스는 컬렉션을 같은 테이블에 저장할 수 없음

컬렉션을 저장하기 위한 별도의 테이블이 필요

영속성 전에(Cascade) + 고아 객체 제거 기능을 필수로 가진다고 볼 수 있음

제약사항

- 값 타입은 엔티티와 다르게 식별자 개념이 없음

- 값은 변경하면 추적이 어려움

- 값 타입 컬렉션에 변경 사항이 발생하면, 주인 엔티티와 연관된 모든 데이터를 삭제하고, 값 타입 컬렉션에 있는 현재 값을 모두 다시 저장한다.

- 값 타입 컬렉션을 매핑하는 테이블은 모든 칼럼을 묶어서 기본 키를 구성해야함

: null 입력 x, 중복 저장x

대안

- 실무에서는 상황에 따라 값 타입 컬렉션 대신에 일대다 관계를 고려

- 일대다 관계를 위한 엔티티를 만들고, 여기에서 값 타입을 사용

- 영속성 전이(Cascade) + 고아 객체 제거를 사용해서 값 타입 컬렉션 처럼 사용

10. 객체지향 쿼리 언어(JPQL)

객체 지향 쿼리 언어 – JPQL

조회 방법 : EntityManager.find(), 객체 그래프 탐색(a.getB().getC())

JPA를 사용하면 엔티티 객체를 중심으로 개발

문제는 검색 쿼리

검색을 할 때도 테이블이 아닌 엔티티 객체를 대상으로 검색

모든 DB 데이터를 객체로 변환해서 검색하는 것은 불가능

-> 검색 조건이 포함된 SQL 필요 !

JPA는 SQL을 추상화한 JPQL이라는 객체 지향 쿼리 언어 제공

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

JPQL은 엔티티 객체를 대상으로 쿼리

SQL은 데이터 베이스 테이블을 대상으로 쿼리

테이블이 아닌 객체를 대상으로 검색하는 객체 지향 쿼리

SQL을 추상화해서 특정 데이터베이스 SQL에 의존 X

JPQL을 한마디로 정의하면 객체 지향 SQL

Criteria

문자가 아닌 자바 코드로 JPQL을 작성할 수 있음

JPQL 빌더 역할

JPA 공식 기능

너무 복잡하고 실용성 없어QueryDSL 사용 권장

QueryDSL

- 문자가 아닌 자바 코드로 JPQL을 작성할 수 있음

- JPQL 빌더 역할

- 컴파일 시점에 문법 오류를 찾을 수 있음

- 동적 쿼리 작성 편리함

- 단순하고 쉬움

- 실무 사용 권장

네이티브 SQL

- JPA가 제공하는 SQL을 직접 사용하는 기능

- JPQL로 해결할 수 없는 특정 데이터 베이스에 의존적인 기능

JDBC 직접 사용, SpringJdbcTemplate 등

- JPA를 사용하면서 JDBC 커넥션을 직접 사용하거나, 스프링 JdbcTemplate, 마이바티스 등을 함께 사용 가능

- 단 영속성 컨텍스트를 적절한 시점에 강제로 플러시 필요

JPQL – 기본 문법과 기능

- JPQL은 객체지향 쿼리 언어. 따라서 테이블을 대상으로 쿼리하는 것이 아니라 엔티티 객체를 대상으로 쿼리

- JPQL은 SQL을 추상화해서 특정데이터베이스 SQL에 의존하지 않음

- JPQL은 결국 SQL로 변환

JPQL 문법

- 엔티티와 속성은 대소문자 구분 가능

- JPQL 키워드 대소문자 구분 x

- 엔티티 이름 사용, 테이블 이름이 아님(Member)

- 별칭은 필수(m)

집합과 정렬

- GROUP BY, HAVING

- ORDER BY

TypeQuery : 반환 타입이 명확할 때 사용

Query : 반환 타입이 명확하지 않을 때 사용

결과 조회 API

- query.getResultList() : 결과가 하나 이상일 때, 리스트 반환 // 결과가 없으면 빈 리스트 반환

- query.getSingleResult() : 결과가 정확히 하나, 단일 객체 반환

프로젝션

- SELECT 절에 조회항 대상을 지정하는 것

- 프로젝션 대상 : 엔티티, 임베디드 타입, 스칼라 타입(숫자, 문자 등 기본 데이터 타입)

\* 여러 값 조회

1. Query 타입으로 조회
2. Object[] 타입으로 조회
3. New 명령어로 조회

- 단순 값을 DTO로 바로 조회

- 패키지 명을 포함한 전체 클래스 명 입력

- 순서와 타입이 일치하는 생성자 필요

페이징 API

- JPA는 페이징을 다음 두 API로 추상화

- setFirstResult(int startPosition) : 조회 시작 위치

- setMaxResults(int maxResult) : 조회할 데이터 수

조인

내부 조인, 외부 조인, 세타 조인

ON 절

- ON절을 활용한 조인(JPA 2.1부터 지원)

1. 조인 대상 필터링
2. 연관관계 없는 엔티티 외부 조인(하이버네이트 5.1 부터)

서브 쿼리

\*JPA 서브 쿼리 한계

- JPA는 WHERE, HAVING 절에서만 서브 쿼리 사용 가능

- SELECT 절도 가능(하이버네이트에서 지원)

- FROM 절의 서브 쿼리는 현재 JPQL에서 불가능

- 조인으로 풀 수 있으면 풀어서 해결

JPQL 타입 표현

문자, 숫자, Boolean, ENUM, 엔티티 타입

기타

SQL과 문법이 같은식, EXISTS, IN, AND, OR, NOT 등등…

조건식 – CASE

COALESCE : 하나씩 조회해서 null이 아니면 반환

NULLF : 두 값이 같으면 null 반환, 다르게 첫번째 값 반환

사용자 정의 함수 호출

- 하이버네이트는 사용전 방언에 추가해야 한다.

- 사용하는 DB 방언을 상속받고, 사용자 정의 함수를 등록한다

경로 표현식

- .(점)을 찍어 객체 그래프를 탐색하는 것

상태 필드(state field) : 단순히 값을 저장하기 위한 필드

연관 필드(association field) : 연관관계를 위한 필드

단일 값 연관 필드 : @ManyToOne, @OneToOne, 대상이 엔티티

컬렉션 값 연관 필드 : @OneToMany, @ManyToMany, 대상이 컬렉션

경로 표현식 특징

상태 필드 : 경로 탐색의 끝, 탐색 x

단일 값 연관 경로 : 묵시적 내부 조인(inner join) 발생, 탐색 o

컬렉션 값 연관 경로 : 묵시적 내부 조인 발생, 탐색 x

FROM 절에서 명시적 조인을 통해 별칭을 얻으면 별칭을 통해 탐색 가능

명시적 조인 : join 키워드 직접 사용

Select m from Member m join m.team t

묵시적 조인 : 경로 표현식에 의해 묵시적으로 SQL 조인 발생 (내부 조인만 가능)

Select m.team from Member m

경로 탐색을 사용한 묵시적 조인 시 주의사항

- 항상 내부 조인

- 컬렉션은 경로 탐색의 끝, 명시적 조인을 통해 별칭을 얻어야함

- 경로 탐색은 주로 SELECT, WHERE 절에서 사용하지만 묵시적 조인으로 인해 SQL의 FROM(JOIN) 절에 영향을 줌

페치 조인(fetch join)

- SQL 조인 종류 x

- JPQL에서 성능 최적화를 위해 제공하는 기능

- 연관된 엔티티나 컬렉션을 SQL 한 번에 함께 조회하는 기능

- join fetch 명령어 사용

엔티티 페치 조인

- 회원을 조회하면서 연관된 팀도 함께 조회(SQL에 한 번에)

- SQL 을 보면 회원 뿐만 아니라 팀(T.\*) 도 함께 SELECT

컬렉션 페치 조인

- 일대다 관계, 컬렉션 페치 조인

페치 조인과 DISTINCT

- SQL의 DISTINCT는 중복된 결과를 제거하는 명령

- JPQL의 DISTINCT 2가지 기능 제공

1. SQL에 DISTINCT를 추가

2. 애플리케이션에서 엔티티 중복 제거

JPQL - 다형성 쿼리

TYPE : 조회 대상을 특정 자식으로 한정

TREAT : 자바의 타입 캐스팅과 유사, 상속 구조에서 부모 타입을 특정 자식 타입으로 다룰 때 사용, FROM, WHERE, SELECT(하이버네이트 지원) 사용

JPQL – 엔티티 직접 사용

- JPQL 에서 엔티티를 직접 사용하면 SQL에서 해당 엔티티의 기본 키 값을 사용

Named 쿼리

- 정적 쿼리

미리 정의해서 이름을 부여해두고 사용하는 JPQL

정적 쿼리

어노테이션, XML에 정의

애플리케이션 로딩 시점에 초기화 한 후 재서용

애플리케이션 로딩 시점에 쿼리를 검증

-Named 쿼리 환경에 따른 설정

XML이 항상 우선권을 가진다

애플리케이션 운영 환경에 따라 다른 XML을 배포할 수 있다.

-벌크 연산

영속성 컨텍스트를 무시하고 데이터베이스에 직접 쿼리

- 벌크 연산을 먼저 실행

- 벌크 연산 수행 후 영속성 컨텍스트 초기화