4-2 : JPA 개요

들어가기 전에

이번 시간에는...

JPA

JPA

JPA 등장배경

객체와 관계형DB의 패러다임의 불일치

DB 테이블은 id 값으로 관계 매핑
객체는 참조로 관계 매핑

DB테이블 특성 기준

객체를 테이블에 맞추어 모델링

```
class Member {
    String id;       //MEMBER_ID 컬럼 사용
    Long teamId;     //TEAM_ID FK 컬럼 사용 //**
    String username;//USERNAME 컬럼 사용
}

class Team {
    Long id;      //TEAM_ID PK 사용
    String name; //NAME 컬럼 사용
}
```

id값으로 객체 전체를 한 번에 불러올 수 없다는 단점
객체다운 모델링에 어긋남

객체 특성 기준

객체 모델링 저장

```
class Member {
    String id;       //MEMBER_ID 컬럼 사용
    Team team;       //참조로 연관관계를 맺는다. //**
    String username;//USERNAME 컬럼 사용
}

        member.getTeam().getId();

INSERT INTO MEMBER(MEMBER_ID, TEAM_ID, USERNAME) VALUES …
```

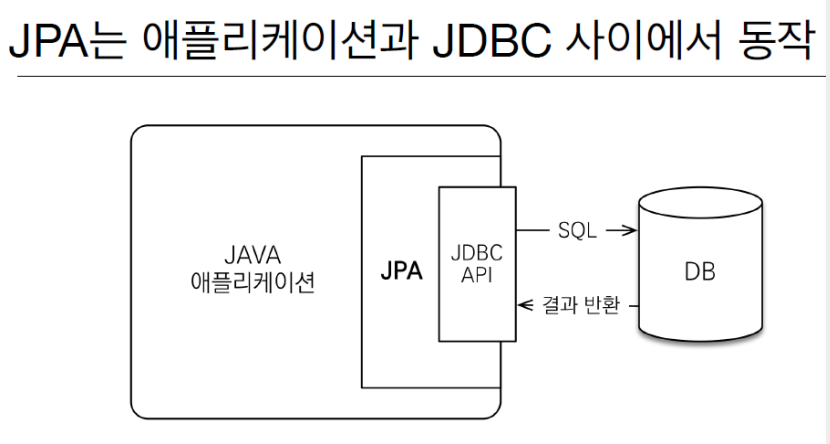
JOIN과 각 객체를 따로 만들어서 통합해야 하는 번거로움

JPA

JPA

자바 진영의 ORM 기술 표준

ORM 기술은
객체는 객체대로 설계하고, 관계형DB는 관계형DB 대로 설계
그 중간에서 ORM 프레임워크가 둘을 매핑



JPA는 인터페이스의 모음으로
대표적인 구현체에는 하이버네이트, EclipasLink, DataNucleus 등이 있다.

JPA

JPA를 사용하면?

회원 CRUD 예시

JDBC Template

회원 CRUD 중 R(조회)

```
// JDBC Template
String getUsersByIdQuery = "select * from USER where id = ?";
        Long getUsersByIdParams = id;
        return this.jdbcTemplate.query(getUsersByIdQuery,
                (rs, rowNum) -> new GetUserRes(
                        rs.getLong("id"),
                        rs.getString("userName")),
                getUsersByIdParams);
```

JPA

회원 CRUD

```
// JPA
entityManager.persist(user); // 저장
entityManager.find(User.class, 1L); // 조회
member.setName("수정할 이름"); // 수정
entityManager.remove(user); // 삭제
```

JPA

JPA 동작 큰 그림

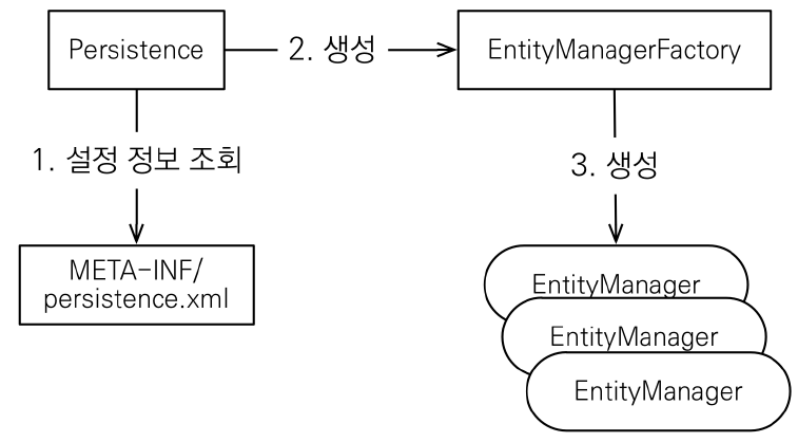
JPA 설정 정보

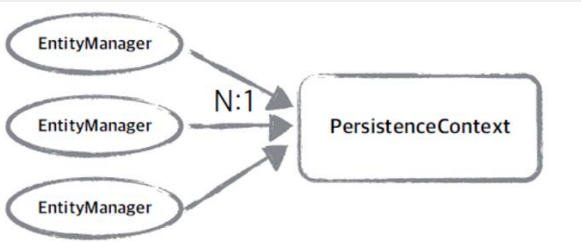
-> EntityManager가 여러 개 생성

-> 영속성 컨텍스트를 거쳐 JDBC API를 통해 DB와 통신

-> 사용한 EntityManager는 제거

-> WAS 종료 시, EntityManagerFactory 종료





플러시

플러시

영속성 컨텍스트의 변경 내용을 데이터베이스에 반영 하는 작업

특징

영속성 컨텍스트를 비우지 않고 유지

트랜잭션 작업 단위로 이루어짐

트랜잭션은 @Transactional로 쉽게 구현할 수 있다.(하위 메소드 모두 적용.)

동작 시점

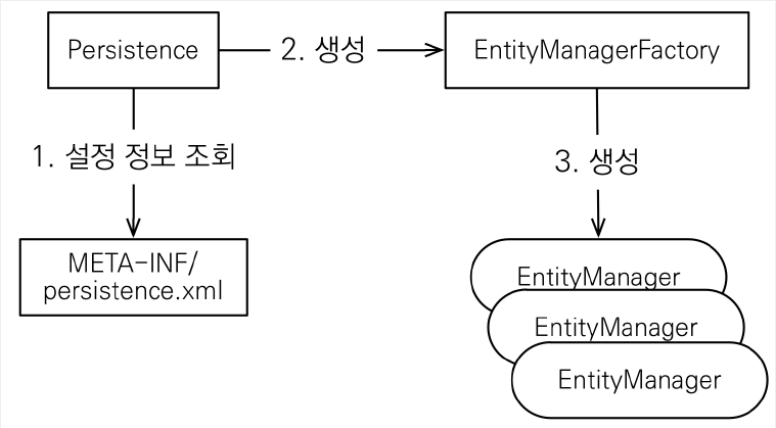
1. entityManager.flush() - 직접 호출

2. 트랜잭션 커밋(플러시 자동 호출)

3.JPQL 쿼리 실행 - 플러시 자동 호출 (기존 변경 사항을 적용 후, 쿼리를 실행해야 하기 때문)

JPA 동작 과정

JPA 구동 방식

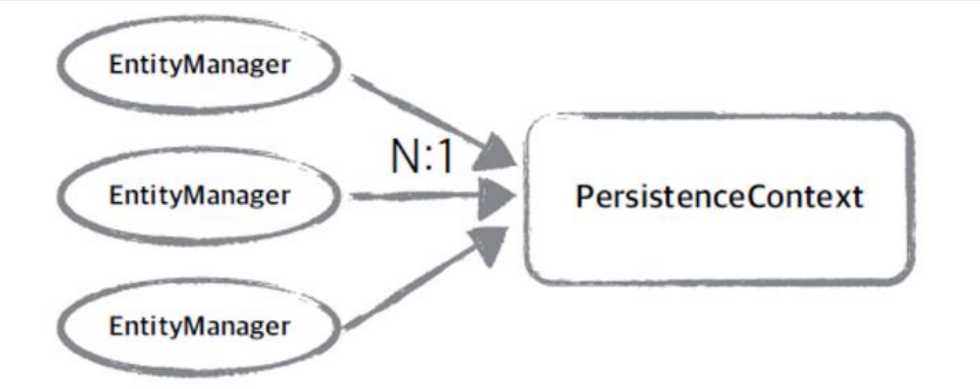


1. JPA 설정 정보를 바탕으로 EntityManagerFactory 생성
2. EntityManagerFactory는 하나만 생성해서 어플리케이션 전체에서 공유
3. EntityManagerFactory에서 각 쓰레드당 EntityManager를 생성하여 활용하고 사용 후 곧바로 버린다.
4. JPA의 모든 데이터 변경은 트랜잭션 안에서 실행된다.
5. WAS가 꺼질 때, EntityManagerFactory를 닫아준다.

영속성 컨텍스트

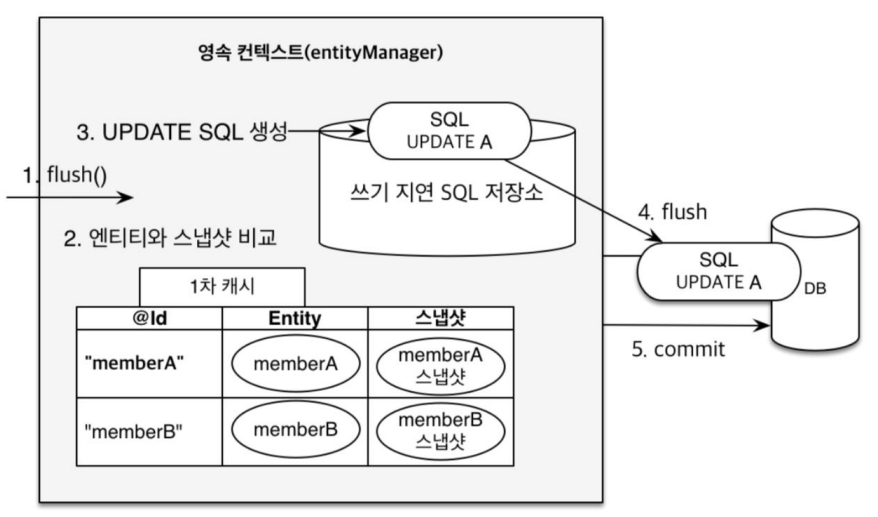
영속성 컨텍스트

엔티티를 영구 저장하는 환경
만들어진 엔티티 매니저를 통해 영속성 컨텍스트에 접근
엔티티 생성, 삭제 등 관리하는 공간



영속성 컨텍스트

영속성 컨텍스트 기능



1. 1차 캐시
영속성 컨텍스트에 영속된 객체들은 1차 캐시로써 사용된다.

2. 동일성 보장
1차 캐시로 동일객체 조회는 동일성을 보장한다.

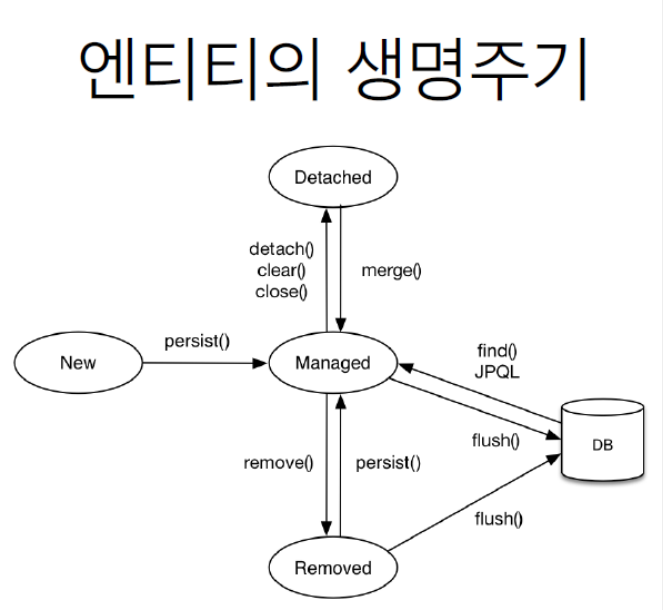
3. 트랜잭션을 지원하는 쓰기 지연
SQL을 쓰기 지연 SQL 저장소에 보관 후, 플러시 시점에 커밋한다.

4.변경 감지
영속성 컨텍스트의 1차 캐시에는 각 객체의 스냅샷을 함께 저장. flush()가 발생하면, 기존의 스냅샷과 객체를 비교하고, 차이가 발생하면 쓰기 지연 SQL 저장소에 update 쿼리를 추가함

5. 지연 로딩
데이터가 필요한 시점에 가져와 불필요한 쿼리를 절약

엔티티 생명주기

엔티티 생명주기



1. 비영속(new, transient)
영속성 컨텍스트와 전혀 관계가 없는 새로운 상태
객체를 이제 막 생성한 상태

2. 영속(managed)
영속성 컨텍스트에 관리되는 상태
새로운 객체를 persist, find 등을 한 상태

3. 준영속(detached)
영속성 컨텍스트에 저장되었다가 분리된 상태
영속된 객체를 detach() 했거나, 영속성 컨텍스트를 clear(), close()를 한 상태

4. 삭제(removed)
삭제된 상태
객체를 remove()한 상태

스키마 자동 생성

엔티티 매핑 – 스키마 자동 생성

JPA는 매핑한 클래스에 맞게 DDL을 자동 생성해준다
자동 생성 설정 값은 application.yml 파일에서 설정할 수 있다

옵션	설명
create	기존테이블 삭제 후 다시 생성 (DROP + CREATE)
create-drop	create와 같으나 종료시점에 테이블 DROP
update	변경분만 반영(운영DB에는 사용하면 안됨)
validate	엔티티와 테이블이 정상 매핑되었는지만 확인
none	사용하지 않음

```yaml
spring:
  application:
    name: demo
  jpa:
    database-platform: org.hibernate.dialect.MySQL5InnoDBDialect
    hibernate:
      ddl-auto: none
      # create, update, create-drop, none 등의 옵션이 있습니다.
      # create: 기존테이블 삭제 후 다시 생성
      # update: 변경된 부분만 반영
      # create-drop: create와 같으나 종료 시점에 테이블 DROP
      # none: 사용하지 않음
```

엔티티 매핑

엔티티 매핑 – 클래스와 테이블

클래스와 테이블은 @Entity, @Table 어노테이션으로 매핑한다

```
@NoArgsConstructor
@Getter
@Entity
@Table(name = "User")
public class User {


}
```

@Entity
Class를 Table과 매핑해주는 어노테이션
기본값으로 Class 이름과 동일한 테이블로 매핑된다.
기본 생성자를 필수로 생성해줘야 한다
final 클래스, inner 클래스, enum, interface에 사용 불가

@Table
엔티티와 매핑할 테이블을 지정해주는 어노테이션

엔티티 매핑

엔티티 매핑 - 필드와 컬럼

필드와 컬럼은 대표적으로 @Column, @Enumberated 어노테이션으로 매핑한다

```java
// 이메일
@Column(name = "email", nullable = false, length = 100)
private String email;

// 비밀번호
@Column(name = "password", nullable = false)
private String password;

// 닉네임
@Column(name = "nickname", nullable = false, length = 30)
private String nickname;

// 전화번호
@Column(name = "phoneNumber", length = 30)
private String phoneNumber;

// 상태
@Column(name = "status", nullable = false, length = 10)
private String status = "ACTIVE";
```

@Column

컬럼 명시, 컬럼에 대한 속성 값 설정에 사용
속성 값으로 컬럼 이름 명시, 등록 변경 가능 여부, null 허용 여부, unique키 설정, 길이 설정 등을 할 수 있다.

@Enumerated

자바 enum 타입을 매핑할 때 사용
enum 순서를 기준으로 매핑하는 ORDINAL 타입과 enum 이름을 기준으로 매핑하는 STRING 타입이 있다.
반드시 STRING 타입으로 사용할 것

엔티티 매핑

엔티티 매핑 – 기본키 매핑

기본키는 대표적으로 @Id, @GeneratedValue 어노테이션으로 매핑한다

```
@Id
@Column(name = "id", nullable = false, updatable = false)
@GeneratedValue(strategy = GenerationType.IDENTITY)
private int id;
```

@Id
기본키 컬럼임을 명시

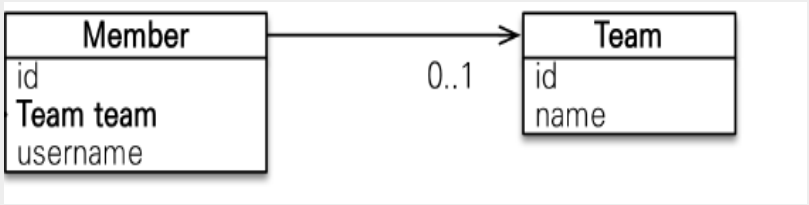
@GeneratedValue
기본키를 자동 생성해주는 설정
IDENTITY: 데이터베이스에 위임
SEQUENCE: 데이터베이스 시퀀스 오브젝트 사용
TABLE: 키 생성용 테이블 사용
AUTO: 방언에 따라 자동 지정

연관관계 매핑

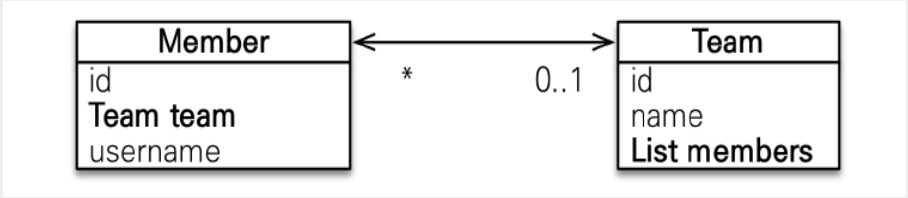
연관관계 매핑

연관관계 매핑에는 단방향과 양방향이 있다.
단방향은 연관관계를 맺고 있는 객체 중 하나의 객체에 관계를 매핑한 것
양방향은 연관관계를 맺고 있는 객체 양쪽에 관계를 매핑한 것

단방향 매핑



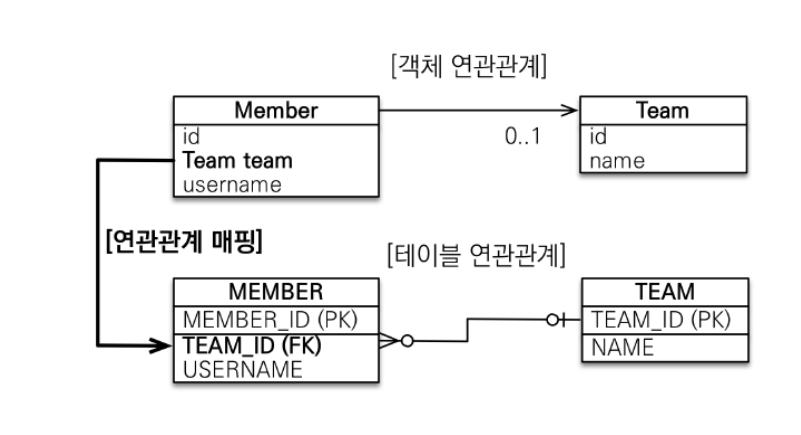
양방향 매핑



연관관계 매핑

단방향 연관관계 매핑

외래키가 있는 곳에 참조 객체를 멤버 변수로 선언
@JoinColumn 어노테이션을 통해 매핑할 컬럼 이름(FK키)을 명시



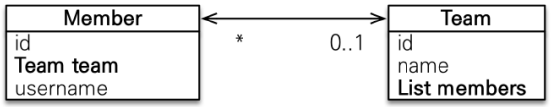

```java
@Entity
public class Member {
  @Id @GeneratedValue
  private Long id;

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

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

연관관계 매핑

양방향 연관관계 매핑

양쪽 객체에 관계를 맺고 있는 참조 객체를 멤버 변수로 선언
@JoinColumn 어노테이션을 통해 매핑할 컬럼 이름(FK키)을 명시
반대편 엔티티엔도 연관관계 매핑 후, mappedBy 속성 값 부여

[양방향 객체 연관관계]



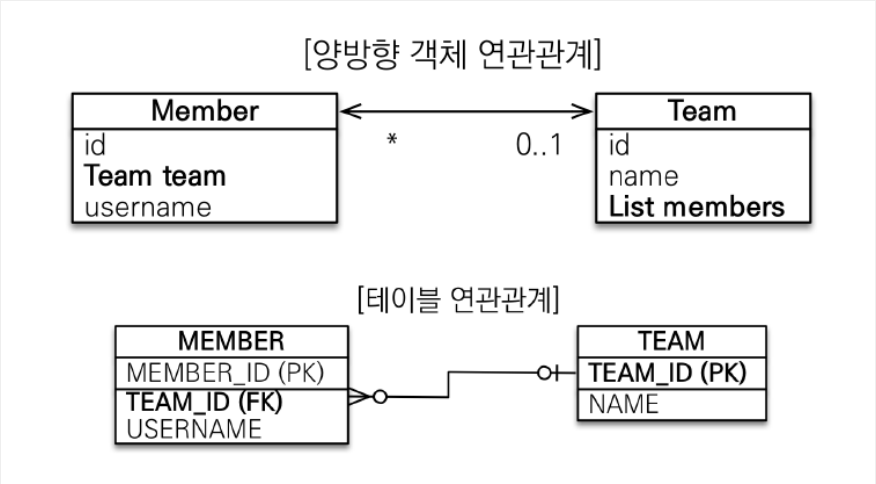

```java
@Entity
public class Member {
  @Id @GeneratedValue
  private Long id;

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

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

```java
@Entity
public class Team {
  @Id @GeneratedValue
  private Long id;

  private String name;

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

연관관계 매핑

양방향 연관관계 매핑 유의사항

연관관계 주인 설정
외래키를 가지고 있는 곳을 주인으로 설정



연관관계 매핑

양방향 연관관계 매핑 유의사항

연관관계 편의 메소드

양방향 매핑된 객체에 데이터를 추가, 수정하는 경우 항상 객체의 양쪽 다 값을 입력해줘야 한다.
매번 입력하는 경우를 연관관계 편의 메소드로 구현(선택 사항)

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

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

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

em.persist(member);
```

```
// Member 객체에서.. (선택1)
public void setTeam(Team team) {
  this.team = team;
  team.getMember().add(this); // 가짜 매핑도 값 추가
}
```

OR

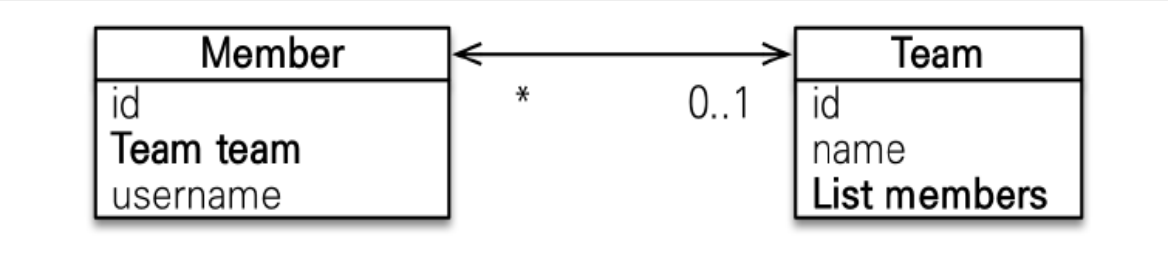
```
// Team 객체에서... (선택2)
public void addMember(Member member) {
  member.setTeam(this); // 연관관계 주인도 값 추가
  members.add(member);
}
```

연관관계 매핑

양방향 연관관계 매핑 유의사항

양방향 매핑 시 무한 루프를 유의

toString(), JSON 등의 생성 라이브러리 등에서 내부적으로 관계를 맺은 객체를 반복적으로 호출하면서 무한루프가 발생할 수 있다.
무분별한 toString 사용을 자제하고, Controller 단에서 DTO를 활용



toString() 메소드 무한 반복 호출

연관관계 매핑

다대일 연관관계

가장 많이 사용하는 연관관계

단방향 매핑: FK가 있는 테이블 매핑 클래스에 FK(id) 값 대신에 해당 객체를 명시.
@ManyToOne 어노테이션을 사용
양방향 매핑: FK가 있는 테이블 객체를 연관관계 주인으로 설정하고, 반대편 객체에 @OneToMany 활용

```java
@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;
```

```java
@Entity
public class Team {
  @Id @GeneratedValue
  private Long id;

  private String name;

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

연관관계 매핑

일대일 연관관계 매핑

주 테이블이나 대상 테이블 중에 외래키 선택 가능
(접근을 많이 하는 쪽에 두자!)
외래키에 데이터베이스 유니크 제약조건 추가

단방향 매핑: @OneToOne 어노테이션을 사용
양방향 매핑: @OneToOne 어노테이션을 사용 . mappedBy 속성을 사용

```java
@Entity
public class Member {
  @Id @GeneratedValue
  private Long id;

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

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

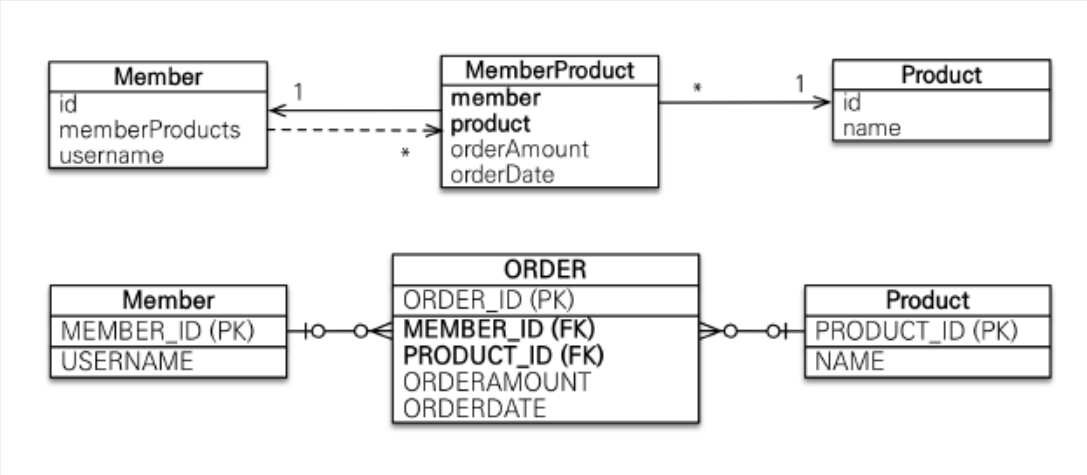
```java
@Entity
public class Team {
  @Id @GeneratedValue
  private Long id;

  private String name;

  @OneToOne(mappedBy = "team")
  Member member = new Member();
```

연관관계 매핑

다대다 연관관계

@ManyToMany 사용하며, @JoinTable로 연결 테이블 지정하여 구현

실무에서 사용 금지
다대다 매핑을 하면 연결 정보 외에 매핑에 추가 정보를 갖기 어려움
예측할 수 없는 쿼리가 나가게 됨



연관관계 매핑

상속관계 매핑

공통 매핑 정보가 필요할 때 사용
(created, updated, state 등)
테이블과 관계 없고, 단순히 엔티티의 공통 속성을 정의하는 것으로 사용
공통 속성으로 조회, 검색이 불가

공통 정보를 추상 클래스로 선언하고 @MappedSuperclass 어노테이션 명시

```java
// 선언
@MappedSuperclass
public abstract class BaseEntity {

  @Column(name = "createdAt") // 컬럼 이름도 설정 가능
  private LocalDateTime createed;
  private LocalDateTime updated;
  private String state;

  // getter, setter 등
}
```

```java
// 활용
public class Member extends BaseEntity  {

}
```

Spring Data JPA

Spring Data JPA

스프링 프레임워크 + JPA 기반 하에 JPA를 편리하게 사용하게 해주는 라이브러리

Spring Data JPA

Spring Data JPA 활용

Spring Data JPA는 JpaRepository를 상속받는 인터페이스만 구현해주면 곧바로 활용할 수 있다.
JpaRepository 인터페이스는 기본적인 CRUD 및 필요한 메서드를 제공

```
public interface MemberRepository extends JpaRepository<Member, Long> {
}
```

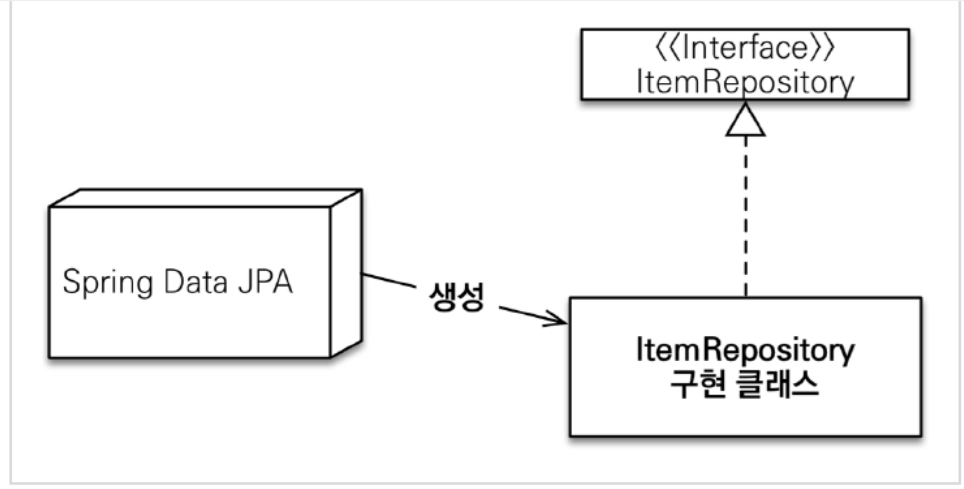
제네릭은 <엔티티 클래스 타입, 식별자(PK) 타입>으로 설정

Spring Data JPA

Spring Data JPA 원리

Spring Data JPA는 인터페이스만 구현해도
내부적으로 Spring Data JPA가 구현 클래스를 대신 생성해준다.

위와 같은 원리로 인터페이스만 구현해도 기본 구현 메서드를 활용할 수 있다.



Spring Data JPA

Spring Data JPA 주요메서드

save(S) : 새로운 엔티티는 저장하고 이미 있는 엔티티는 병합한다.

findAll(…) : 모든 엔티티를 조회한다. 정렬(Sort)이나 페이징(Pageable) 조건을 파라미터로 제공할 수 있다.

findById(ID) : 엔티티 하나를 조회한다. 내부에서 EntityManager.find() 호출

delete(T) : 엔티티 하나를 삭제한다. 내부에서 EntityManager.remove() 호출

쿼리 메서드

Spring Data JPA 쿼리 메서드 기능

주요 공통 메서드 외에 다른 메서드를 구현하고자 할 때 사용

Spring Data JPA 인터페이스를 상속 받아 추가 메서드를 구현하기에는
공통 메서드를 모두 오버라이딩 받아야하기 때문에 매우 비효율적

따라서, Spring Data JPA에서는 다음과 같은 쿼리 메서드 기능을 제공한다.

주요 쿼리 메서드 기능 종류

1. 메서드 이름으로 쿼리 생성
2. @Query
3. @EntityGraph

쿼리 메서드

Spring Data JPA 쿼리 메서드 기능
– 메서드 이름으로 쿼리 생성

메서드 이름을 분석해서 JPQL 쿼리를 실행
조건이 간단하고, 짤막한 쿼리 생성 시 효과적

- **조회**: find…By ,read…By ,query…By get…By
- **COUNT**: count…By 반환타입 long
- **EXISTS**: exists…By 반환타입 boolean
- **삭제**: delete…By, remove…By 반환타입 long
- **DISTINCT**: findDistinct, findMemberDistinctBy
- **LIMIT**: findFirst3, findFirst, findTop, findTop3

공식 문서: https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#jpa.query-methods.query-creation

쿼리 메서드

Spring Data JPA 쿼리 메서드 기능
– 메서드 이름으로 쿼리 생성

메서드 이름을 분석해서 JPQL 쿼리를 실행
조건이 간단하고, 짤막한 쿼리 생성 시 효과적

```
public interface MemberRepository extends JpaRepository<Member, Long> {
    List<Member> findByUsernameAndAgeGreaterThan(String username, int age);
}
```

쿼리 메서드

Spring Data JPA 쿼리 메서드 기능
– @Query

@Query 어노테이션을 활용
JPA Repository에 메서드와 쿼리를 동시에 정의하여 활용할 수 있다.

기본적으로 메서드 이름 쿼리 생성으로 사용 후, 조건이 많아지면 @Query를 주로 사용

```
public interface MemberRepository extends JpaRepository<Member, Long> {
    @Query("select m from Member m where m.username= :username and m.age = :age")
    List<Member> findUser(@Param("username") String username, @Param("age") int age);
}
```

쿼리 메서드

Spring Data JPA 쿼리 메서드 기능
– @Query

@Query 어노테이션을 활용
JPA Repository에 메서드와 쿼리를 동시에 정의하여 활용할 수 있다.

Entity 조회
+
파라미터 바인딩

```
public interface MemberRepository extends JpaRepository<Member, Long> {
    @Query("select m from Member m where m.username= :username and m.age = :age")
    List<Member> findUser(@Param("username") String username, @Param("age") int age);
}
```

DTO 조회

```
// DTO로 직접 조회
// DTO 패키지 경로를 모두 적어줘야 한다는 번거로움이 있다. -> QueryDSL 사용으로 해결 가능.
@Query("select new study.datajpa.dto.MemberDto(m.id, m.username, t.name) " +
        "from Member m join m.team t")
List<MemberDto> findMemberDto();
```

쿼리 메서드

Spring Data JPA 쿼리 메서드 기능
– @EntityGraph

연관된 엔티티들을 SQL 한 번에 조회하는 방법
@EntityGraph 어노테이션을 활용하여 Fetch Join을 간단하게 구현할 수 있는 방법

```
//JPQL + 엔티티 그래프
@EntityGraph(attributePaths = {"team"})
@Query("select m from Member m")
List<Member> findMemberEntityGraph();

//메서드 이름으로 쿼리에서 특히 편리하다.
@EntityGraph(attributePaths = {"team"})
List<Member> findByUsername(String username)
```

과제 안내

Spring Boot 강의 수강(총 약 12시간 분량)

Section6. Spring과 Spring Boot로 JPA와 Hibernate 시작하기

Section8. Spring Boot와 Spring Framework, Hibernate로 Java REST API 생성하기

Section7. Spring Framework, Spring Boot, Hibernate로 웹 애플리케이션 만들기(선택사항)

과제 안내

Spring Data JPA로 리팩토링

3주차에 구현한 REST API 코드를 Spring Data JPA를 활용하여 다시 재구현

과제 유의사항

최대한 Spring Data JPA를 활용하여 구현(JDBC Template 지양)

N + 1 설정(지연 로딩, Fetch Join, BatchSize 설정) 1개 이상씩 구현

Spring Data JPA 쿼리 메서드 기능 1개 이상씩 구현

최대한 Optional, Lambda 식을 활용해서 구현해보기

Hello World!