▼ 엔티티 매핑

객체와 테이블 매핑

- @Entity
  - jpa 가 관리, 엔티티라고 한다. jpa 사용해서 테이블과 매핑할 클래스는 @Entity 필수
  - **기본 생성자** 필수
  - final, enum, interface, inner 클래스 사용 x
  - 속성: name
    - jpa 에서 사용할 엔티티이름을 지정, 기본값: 클래스 이름을 그대로 사용, 같은 클래스 이름이 없으면 가급적 기본값을 사용한다.
- @Table
  - 엔티티와 매핑할 테이블 지정

데이터베이스 스키마 자동생성

DDL을 애플리케이션 실행 시점에 자동 생성

테이블중심->객체중심

데이터베이스 방언을 활용해서 데이터베이스에 맞는 적절한 DDL 생성
이렇게 **생성된 DDL은 개발 장비에서만 사용**

생성된 DDL은 운영서버에서는 사용하지 않거나, 적절히 다듬은 후 사용

- **hibernate.hbm2ddl.auto**
  - **create :**기존 테이블 삭제 후 다시 생성 drop → create
  - **create-drop :**create 와 같으나 종료시점에 테이블 drop
  - **update :** 변경분만 반영(운영 db에는 사용하면안됨)
  - **validate** : 엔티티와 테이블이 정상 매핑되었는지만 확인
  - **none** : 사용하지 않음

** 운영장비에는 절대 create, create-drop, update 사용하면 안된다

초기: create, update

테스트 서버: create, validate

스테이징과 운영서버: validate, none

필드와 컬럼 매핑

```
package hellojpa;
import javax.persistence.*;
import java.time.LocalDate;
import java.time.LocalDateTime;
import java.util.Date;
@Entity
public class Member {
    @Id
    private Long id;
    @Column(name = "name")
    private String username;
    private Integer age;
    @Enumerated(EnumType.STRING)
    private RoleType roleType;
    @Temporal(TemporalType.TIMESTAMP)
    private Date createdDate;
    @Temporal(TemporalType.TIMESTAMP)
    private Date lastModifiedDate;
    @Lob
    private String description;

    public  Member(){

    }
}
```

어노테이션	설명
@Column	컬럼 매핑
@Temporal	날짜 타입 매핑
@Enumerated	enum 타입 매핑
@Lob	BLOB, CLOB 매핑
@Transient	특정 필드를 컬럼에 매핑하지 않음(매핑 무시)

@Column

속성	설명	기본값
name	필드와 매핑할 테이블의 컬럼 이름	객체의 필드 이름
insertable, updatable	등록, 변경 가능 여부	TRUE
nullable(DDL)	null 값의 허용 여부를 설정한다. false로 설정하면 DDL 생성 시에 not null 제약조건이 붙는다.	
unique(DDL)	@Table의 uniqueConstraints와 같지만 한 컬럼에 간단히 유니크 제약조건을 걸 때 사용한다.	
columnDefinition (DDL)	데이터베이스 컬럼 정보를 직접 줄 수 있다. ex) varchar(100) default 'EMPTY'	필드의 자바 타입과 방언 정보를 사용해
length(DDL)	문자 길이 제약조건, String 타입에만 사용한다.	255
precision, scale(DDL)	BigDecimal 타입에서 사용한다(BigInteger도 사용할 수 있다). precision은 소수점을 포함한 전체 자 릿수를, scale은 소수의 자릿수다. 참고로 double, float 타입에는 적용되지 않는다. 아주 큰 숫자나 정 밀한 소수를 다루어야 할 때만 사용한다.	precision=19, scale=2

기본 키 매핑

@ID

@GeneratedValue

- 기본 키 매핑 방법
  - 직접 할당: @Id 만 사용
  - 자동 생성: @GeneratedValue(strategy=

    IDENTITY : 데이터 베이스에 위임, MYSQL
    SEQUENCE : 데이터베이스 시퀀스 오브젝트 사용, ORACLE,
    @SequenceGenerator 필요
    TABLE: 키 생성용 테이블 사용, 모든 DB 에서 사용, @TableGenerator 필요
    AUTO : 방언에 따라 자동 지정, 기본 값
- IDENTITY전략 - 특징
  - 기본 키 생성을 데이터 베이스에 위임
  - JPA 는 보통 트랜잭션 커밋시점에 INSERT SQL 실행
  - AUTO_CREMENT는 데이터베이스에 INSERT SQL 을 실행한 이후에 ID 값을 알 수 있음

- IDENTITY 전략은 em.persist()시점에 즉시 INSERT SQL 실행하고 DB에서 식별자를 조회

- SEQUENCE전략 - 특징
  - 데이터베이스 시퀀스는 유일한 값을 순서대로 생성하는 특별한 데이터베이스 오브젝트

SEQUENCE - @SequenceGenerator

- 주의: allocationSize 기본값 = 50

속성	설명	기본값
name	식별자 생성기 이름	필수
sequenceName	데이터베이스에 등록되어 있는 시퀀스 이름	hibernate_sequence
initialValue	DDL 생성 시에만 사용됨, 시퀀스 DDL을 생성할 때 처음 1 시작하는 수를 지정한다.	1
allocationSize	시퀀스 한 번 호출에 증가하는 수(성능 최적화에 사용됨 **데이터베이스 시퀀스 값이 하나씩 증가하도록 설정되어 있으면 이 값을 반드시 1로 설정해야 한다**	**50**
catalog, schema	데이터베이스 catalog, schema 이름	

- TABLE 전략
  - 키 생성 전용 테이블을 하나 만들어서 데이터베이스 시퀀스를 흉내내는 전략
  - 장점: 모든 데이터베이스에 적용가능/ 단점: 성능

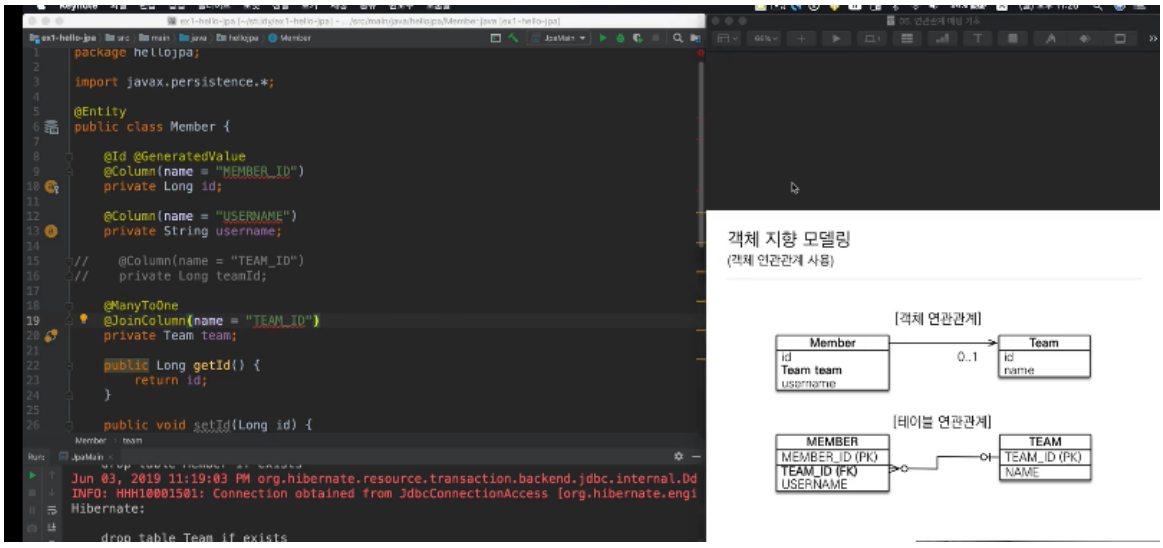
@TableGenerator - 속성

속성	설명	기본값
name	식별자 생성기 이름	필수
table	키생성 테이블명	hibernate_sequences
pkColumnName	시퀀스 컬럼명	sequence_name
valueColumnNa	시퀀스 값 컬럼명	next_val
pkColumnValue	키로 사용할 값 이름	엔티티 이름
initialValue	초기 값, 마지막으로 생성된 값이 기준이다.	0
allocationSize	시퀀스 한 번 호출에 증가하는 수(성능 최적화에 사용됨)	**50**
catalog, schema	데이터베이스 catalog, schema 이름	
uniqueConstraints(DDL)	유니크 제약 조건을 지정할 수 있다.	

- 권장하는 식별자 전략
  - 기본 키 제약 조건: null 아님, 유일, 변하면 안된다
  - 미래까지 이 조건을 만족하는 자연키는 찾기 어렵다. 대체키(대리키)사용하자
  - 주민등록번호도 기본키로 적절하지 않다
  - 권장: long형 + 대체키 + 키 생성전략 사용

실전 예제 - 1 요구사항 분석과 기본 매핑

▼ 연관관계 매핑 기초



연관관계가 필요한 이유

단방향 연관관계

```
@Entity
public class Member {
    @Id @GeneratedValue
    @Column(name="MEMBER_ID")
    private Long id;

    @Column(name="USERNAME")
    private String username;
// Many = member one = team

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

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

            Member member = new Member();
            member.setUsername("member1");
            member.setTeam(team);
            em.persist(member);
```

양방향 연관관계와 연관관계의 주인

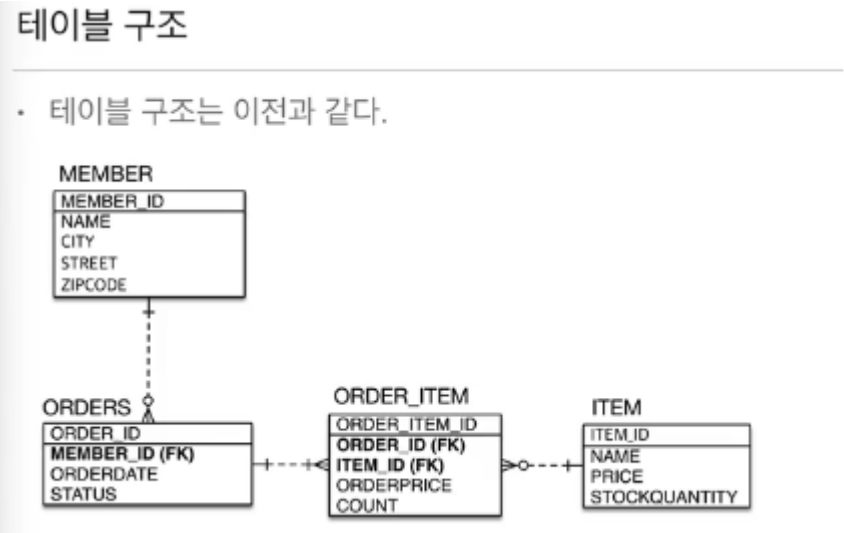
mapped by

객체와 테이블이 관계를 맺는 차이

- 객체 : 연관관계가 2개 {회원→팀 , 팀→회원} - 단방향
- 테이블 : 연관관계 1개 {회원 ↔ 팀} - 양방향
- 연관관계의 주인
    - 양방향 매핑 규칙
        - 객체의 두 관계 중 하나를 연관관계의 주인으로 지정
        - 연관관계의 주인만이 외래 키를 관리(등록, 수정)
        - 주인이 아닌쪽은 읽기만 가능
            - 주인은 mappedBy 속성을 사용하지 않는다
            - 주인이 아니면 mappedBy 속성으로 주인지정

        - 누구를 주인으로?
            - 외래키가 있는 곳을 주인으로 @ManyToOne : N 쪽이 연관관계의 주인!
- 양방향 연관관계 주의
    - 순수 객체 상태를 고려해서 항상 양쪽에 값을 설정하자

```
//JpaMain.java
Member member = new Member();
            member.setUsername("member1");
            // 연관관계의 주인에게만 값을 저장!
            member.setTeam(team);
// Member > setTeam
public void setTeam(Team team) {
        this.team = team;
        team.getMembers().add(this);
    }
```



jpabook 파일

▼ 다양한 연관관계 매핑

연관관계 매핑 시 고려사항 3가지

1. 다중성
2. 단방향, 양방향
    a. 테이블: 외래키 하나로 양쪽 조인 가능, 방향이라는 개념이 없음

b. 객체: 참조용 필드가 있는 쪽으로만 참조 가능, 한쪽만 참조하면 단방향, 양쪽이 서로 참조하면 양방향

3. 연관관계의 주인
   a. 외래키를 관리하는 참조, 주인의 반대편은 외래키에 영향을 주지 않음, 단순 조회만 가능

다대일 [N:1] @ManyToOne

단방향: 가장많이 사용하는 연관관계, 반대는 일대다

```
//단방향 (N 이 주인)
@ManyToOne
    @JoinColumn(name="TEAM_ID")
    private Team team;

// 양방향은 반대쪽(1인 곳)에 추가
@OneToMany(mappedBy = "team")
private List<Member> members = new ArrayList<>();
```

일대다 [1:N] @OneToMany

1 이 연관관계 주인

db에선 무조건 n 에 외래키가 들어가야해서 실제론 잘안씀

```
@OneToMany
@JoinColumn(name = "TEAM_ID")
private List<Member> members = new ArrayList<>();
```

객체와 테이블의 차이 때문에 반대편 테이블의 외래키를 관리하는 특이한 구조

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

단점: 엔티티가 관리하는 외래키가 다른 테이블에 있음, 연관관계 관리를 위해 추가로 update sql 실행

→ 일대다 단방향 매핑보다는 **다대일 양방향 매핑** 사용하자

일대일 [1:1] @OneToOne

일대일 관계는 반대도 일대일

주 테이블이나 대상 테이블 중에 외래 키 선택 가능

- 주 테이블에 외래 키
- 대상 테이블에 외래 키

외래키에 데이터베이스 유니크(UNI)제약조건 추가

일대일양방향: 다대일 양방향 매핑 처럼 외래키가 있는 곳이 연관관계 주인, 반대편의 mappedBy 적용

일대일: 대상 테이블에 외래 키 단방향 - 안된다, 양방향관계는 지원

다대다 [N:M] @ManyToMany - 쓰면안된다

관계형 데이터베이스는 정규화된 테이블 2개로 다대다 관계를 표현x

연결 테이블을 추가해서 일대다, 다대일 관계로 풀어내야함

객체는 컬렉션을 사용해서 객체 2개로 다대다 관계 가능

@ManyToMany / @JoinColumn