4장 엔티티 매핑 - 12월 3주차

기간 : 21.12.17 ~ 21.12.23

페이지 : 122 ~ 181p (4장 ~ 5장 5.3)

▼ 목차

4.1 @Entity 122p

- JPA를 사용해서 테이블과 매핑할 클래스에 필수로 붙여야 한다.
- @Entity가 붙은 클래스는 JPA가 관리하는 것으로, 엔티티라고 부른다.

속성	기능	기본값
name	- JPA에서 사용할 엔티티 이름을 지정 - 다른 패키지에 이름이 같은 엔티티 클래스가 있다면 이름을 지정해서 충돌하지 않도록 해야한다.	설정하지 않으면 클래스 이름을 그대로 사용

<주의사항>

- **기본 생성자는 필수**이다(파라미터가 없는 public 또는 protected 생성자). ⇒ JPA가 엔티티 객체를 생성할 때 기본 생성자를 사용
- final 클래스, enum, interface, inner 클래스에는 사용할 수 없다.
- 저장할 필드에 final을 사용하면 안 된다.

4.2 @Table 123p

- 엔티티와 매핑할 테이블을 지정
- 생략하면 매핑한 엔티티 이름을 테이블 이름으로 사용한다.

속성	기능	기본값
name	매핑할 테이블 이름	엔티티 이름을 사용
catalog	catalog 기능이 있는 데이터베이스에서 catalog를 매핑한다.	
schema	schema 기능이 있는 데이터베이스에서 schema 를 매핑한다.	
uniqueConstraints(DDL)	- DDL 생성 시에 유니크 제약조건을 만든다. - 2개 이상의 복합 유니크 제약조건도 만들 수 있다. - 참고로 이 기능은 스키마 자동 생성 기능을 사용해서 DDL을 만들때만 사용된다.	

4.4 데이터베이스 스키마 자동 생성 125p

- JPA는 클래스의 매핑정보와 데이터베이스 방언을 사용해서 데이터베이스 스키마를 생성한다.
  - 이렇게 생성된 DDL은 개발 장비에서만 사용한다.
  - 생성된 DDL은 운영서버에서는 사용하지 않거나, 적절히 다듬은 후 사용한다.
- 테이블 중심 -> 객체 중심

<persistence.xml> 에 아래 속성 추가

```
<property name="hibernate.hbm2ddl.auto" value="create">
```

- 이 속성을 추가하면 애플리케이션 실행 시점에 데이터베이스 테이블을 자동으로 생성한다.

hibernate.hbm2ddl.auto 속성

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

운영 장비에는 절대 create, create-drop, update 사용하면 안된다.

- 개발 초기 단계는 create 또는 update
- 테스트 서버는 update 또는 validate
- 스테이징과 운영 서버는 validate 또는 none

참고)

true로 설정하면 콘솔에 실행되는 테이블 생성 DDL을 출력할 수 있다.

```
<property name="hibernate.show_sql" value="true"/>
```

테이블 명이나 컬럼명을 언더스코어 표기법으로 매핑

```
<property name="hibernate.ejb.naming_strategy" value="org.hibernate.cfg.ImprovedNamingStrategy"/>
```

기존에 roleType으로 생성되었다면, 컬럼명이 role_type으로 생성된다.

4.5 DDL 생성 기능 129p

```
@Entity
@Table(name="MEMBER", uniqueConstraints = {@UniqueConstraint(
    name = "NAME_AGE_UNIQUE",
    columnNames = {"NAME", "AGE"}
)}) // 유니크 제약조건 추가
public class Member {
    //
    @Id
    @Column(name = "ID")
    private String id;

    @Column(name = "NAME", nullable = false, length = 10) // 제약조건 추가
    private String username;
}
```

- `nullable = false` : 자동 생성되는 DDL에 `not null` 제약조건을 추가할 수 있다.

생성된 DDL

```
ALTER TABLE MEMBER
  ADD CONSTRAINT NAME_AGE_UNIQUE UNIQUE (NAME, AGE)
```

- **DDL 생성 기능은 DDL을 자동 생성할 때만 사용되고 JPA의 실행 로직에는 영향을 주지 않는다.**

4.6 기본 키 매핑 131p

- JPA가 제공하는 데이터베이스 기본키 생성 전략은 직접 할당과 자동 생략 전략이 있다.
- 키 생성 전략을 사용하려면 아래 속성을 추가 해야한다.

<persistence.xml>

```
<property name="hibernate.id.new_generator_mappings" value="true"/>
```

- 과거 버전과의 호환성을 유지하려고 기본값이 false로 되어있다.
- 참고로 이 옵션을 true 설정하면 키 생성 성능을 최적화하는 allocationSize 속성을 사용하는 방식이 달라진다. (뒤에서 설명)

4.6.1 기본 키 직접 할당 전략 132p

- 기본키를 애플리케이션에서 직접 할당한다.
- `@Id` 만 사용하면 된다.

▼ @Id 적용 가능 자바 타입

- 자바 기본형
- 자바 래퍼(Wrapper)형
- String
- java.util.Date
- java.sql.Date
- java.math.BigDecimal
- java.math.BigInteger

- 직접 할당은 `em.persist()` 로 엔티티를 저장하기 전에 애플리케이션에서 기본 키를 직접 할당하는 방법이다.

```
Board board = new Board();
board.setId("id1"); // 기본키 직접 할당
em.persist(board);
```

4.6.1 자동 생성 전략 133p~

- 대리키 사용 방식
- `@Id` 에 `@GeneratedValue` 를 추가하고 원하는 키 생성 전략을 선택

1. **IDENTITY** : 기본 키 생성을 데이터베이스에 위임한다.
2. **SEQUENCE** : 데이터베이스 시퀀스를 사용해서 기본키를 할당한다.
   - @SequenceGenerator 필요
3. **TABLE** : 키 생성 테이블을 사용한다. ⇒ 모든 DB에서 사용
   - @TableGenerator 필요
4. **AUTO** : 방언에 따라 자동 지정한다(기본값).

4.6.2 IDENTITY 전략 133p

- 기본 키 생성을 데이터베이스에 위임하는 전략
- 주로 MySQL, PostgreSQL, SQL Server, DB2에서 사용
  - ex) MySQL의 AUTO_INCREMENT
- `@GeneratedValue(strategy = GenerationType.IDENTITY)`

```
CREATE TABLE BOARD (
    ID INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    DATA VARCHAR(255)
);

INSERT INTO BOARD(DATA) VALUES('A');
INSERT INTO BOARD(DATA) VALUES('B');
```

- AUTO_INCREMENT 는 데이터베이스에 INSERT SQL을 실행한 이후에 ID 값을 알 수 있다.

IDENTITY 매핑 코드

```
@Entity
public class Board {
    //
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
}
```

IDENTITY 사용 코드

```
private static void logic(EntityManager em) {
    //
    Board board = new Board();
    em.persist(board);
    System.out.println("board.id = " + board.getId()); /
}
```

- 엔티티가 영속 상태가 되려면 식별자가 반드시 필요하다.

  그런데 IDENTITY 식별자 생성 전략은 엔티티를 데이터베이스에 저장해야 식별자를 구할 수 있으므로

  `em.persist()` 를 호출하는 즉시 INSERT SQL이 데이터베이스에 전달되고 (JPA는 보통 트랜잭션 커밋 시점에 INSERT SQL 실행)

  DB에서 식별자를 조회한다.

> 💡 따라서 이 전략은 트랜잭션을 지원하는 쓰기 지연이 동작하지 않는다.

4.6.3 SEQUENCE 전략 135p

- 시퀀스를 사용해서 기본 키를 생성한다.
  - 데이터베이스 시퀀스는 유일한 값을 순서대로 생성하는 특별한 데이터베이스 오브젝트이다.
- 시퀀스를 지원하는 오라클, PostgreSQL, DB2, H2 데이터베이스에서 사용할 수 있다.

시퀀스 DDL

```
CREATE TABLE BOARD (
    ID BIGINT NOT NULL PRIMARY KEY,
    DATA VARCHAR(255)
)

// 시퀀스 생성
CREATE SEQUENCE BOARD_SEQ START WITH 1 INCREMENT BY 1;
```

시퀀스 매핑 코드

```
@Entity
@SequenceGenerator(
    name = "BOARD_SEQ_GENERATOR",
    sequenceName = "BOARD_SEQ", // 매핑할 데이터베이스 시퀀스 이름
    initialValue = 1, allocationSize = 1)
public class Board {
    //
    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE,
                    generator = "BOARD_SEQ_GENERATOR")
    private Long id;
}
```

```
@Entity
public class Board {
    //
    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE,
                    generator = "BOARD_SEQ_GENERATOR")
    @SequenceGenerator(
        name = "BOARD_SEQ_GENERATOR",
        sequenceName = "BOARD_SEQ", // 매핑할 데이터베이스 시
        initialValue = 1, allocationSize = 1)
    private Long id;
}
```

- @SequenceGenerator 를 사용해서 BOARD_SEQ_GENERATOR 라는 시퀀스 생성기를 등록
- sequenceName 속성의 이름으로 BOARD_SEQ를 지정 ⇒ JPA는 이 시퀀스 생성기를 실제 데이터베이스의 BOARD_SEQ 시퀀스와 매핑한다.
- `generator = "BOARD_SEQ_GENERATOR"` : 시퀀스 생성기(BOARD_SEQ_GENERATOR)를 선택

시퀀스 사용 코드

```
private static void logic(EntityManger em) {
    //
    Board board = new Board();
    em.persist(board);
    System.out.println("board.id = " + board.getId()); // board.id = 1
}
```

- `em.persist()` 를 호출할 때 먼저 데이터베이스 시퀀스를 사용해서 식별자를 조회한다.
- 조회한 식별자를 엔티티에 할당한 후에 엔티티를 영속성 컨텍스트에 저장한다.
- 이후 트랜잭션을 커밋해서 플러시가 일어나면 엔티티를 데이터베이스에 저장한다.

💡 **IDENTITY 전략**은 먼저 엔티티를 데이터베이스에 저장한 후에 식별자를 조회해서 엔티티의 식별자에 할당한다.

@SequenceGenerator 속성

속성	기능	기본값
name	식별자 생성기 이름	필수
sequenceName	데이터베이스에 등록되어 있는 시퀀스 이름	
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	

매핑할 DDL

```
create sequence [sequenceName]
start with [initialValue] increment by [allocationSize]
```

SequenceGenerator.allocationSize 의 기본값이 50인 것에 주의하기

`create sequence [sequenceName] start with 1 increment by 50` 이므로 시퀀스를 호출할 때마다 값이 50씩 증가한다.

4.6.4 TABLE 전략 139p

- 키 생성 전용 테이블을 하나 만들고 여기에 이름과 값으로 사용할 컬럼을 만들어 데이터베이스 시퀀스를 흉내내는 전략
- 테이블을 사용하므로 모든 데이터베이스에 적용할 수 있다.

TABLE 전략 키 생성 DDL

```
create table MY_SEQUENCES (
    sequence_name varchar(255) not null,
    next_val bigint,
    private key (sequence_name)
)
```

- sequence_name 컬럼을 시퀀스 이름으로 사용
- next_val 컬럼을 시퀀스 갑사으로 사용

TABLE 전략 매핑 코드

TABLE 전략 매핑 사용 코드

```
@Entity
@TableGenerator(
    name = "BOARD_SEQ_GENERATOR",
    table = "MY_SEQUENCES", // 키 생성용 테이블로 매핑할 테이블
    pkColumnValue = "BOARD_SEQ", allocationSize = 1
)
public class Board {
    //
    @Id
    @GeneratedValue(strategy = GenerationType.TABLE
                    generator = "BOARD_SEQ_GENERATOR")
    private Long id;
}
```

```
private static void logic(EntityManger em) {
    //
    Board board = new Board();
    em.persist(board);
    System.out.println("board.id = " + board.getId()); /
}
```

MY_SEQUENCES 테이블

sequence_name	next_val
BOARD_SEQ	2
MEMBER_SEQ	10
PRODUCT_SEQ	50
...	...

- @TableGenerator.pkColumnValue 에서 지정한 "BOARD_SEQ" 가 컬럼명으로 추가되었다.
- 키 생성기를 사용할 때마다 next_val 컬럼 값이 증가한다.
- 참고) MY_SEQUENCES 테이블에 값이 없으면 JPA 가 값을 INSERT 하면서 초기화하므로 값을 미리 넣어둘 필요는 없다.

@TableGenerator 속성

속성	설명	기본값
name	식별자 생성기 이름	필수
table	키생성 테이블명	hibernate_sequences
pkColumnName	시퀀스 컬럼명	sequence_name
valueColumnName	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.	

매핑할 DDL, 데이블명 {table}

{pkColumnName}	{valueColumnName}
{pkColumnValue}	{initialValue}

- TABLE 전략은 값을 조회하면서 SELECT 쿼리를 사용하고 다음 값으로 증가시키기 위해 UPDATE 쿼리를 사용한다.

4.6.5 AUTO 전략 142p

- 선택한 데이터베이스 방언에 따라 IDENTITY, SEQUENCE, TABLE 전략 중 하나를 자동으로 선택한다.

```
@Entity
public class Board {
    //
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
}
```

```
@Entity
public class Board {
    //
    @Id
    @GeneratedValue
    private Long id;
}
```

- AUTO를 사용할 때 SEQUENCE 나 TABLE 전략이 선택되면 시퀀스나 키 생성용 테이블을 미리 만들어 두어야 한다.

4.6.6 기본키 매핑 정리 143p

- 엔티티를 영속 상태로 만들려면 식별자 값이 반드시 있어야 한다.
- `em.persist()` 를 호출한 직후에 발생하는 일을 식별자 할당 전략별로 정리하면
  - 직접 할당 : `em.persist()` 를 호출하기 전에 애플리케이션에서 직접 식별자 값을 할당해야 한다.
  - IDENTITY : 데이터베이스 엔티티를 저장해서 식별자 값을 획득한 후 영속성 컨텍스트에 저장한다. (IDENTITY 전략은 테이블에 데이터를 저장해야 식별자 값을 획득할 수 있다.)
  - SEQUENCE : 데이터베이스 시퀀스에서 식별자 값을 획득한 후 영속성 컨텍스트에 저장한다.
  - TABLE : 데이터베이스 시퀀스 생성용 테이블에서 식별자 값을 획득한 후 영속성 컨텍스트에 저장한다.

권장하는 식별자 선택 전략

- 기본키 제약 조건
  - null 허용 x
  - 유일해야한다.
  - 변하면 안 된다.
- 자연키
  - 비즈니스에 의미가 있는 키
  - ex) 주민등록번호, 이메일, 전화번호
- 대리키(대체키)
  - 비즈니스와 관련 없는 임의로 만들어진 키
  - ex) 오라클 시퀀스, auto_increment, 키생성 테이블 사용
- 자연키보다는 대리키를 권장한다. 비즈니스 환경은 언젠가 변하기 때문
  - 그럴듯해 보이는 주민번호(null 아니고, 유일하며 변하지 않는다)도 여러 가지 이유로 변경될 수 있다.
    - ex) 정부 정책이 변경되면서 법적으로 주민번호를 저장할 수 없게 되었다.
- 권장 : Long 형 + 대체키 + 키 생성전략 사용

4.7 필드와 컬럼 매핑 : 레퍼런스 145p

- JPA가 제공하는 필드와 컬럼 매핑용 어노테이션

필드와 컬럼 매핑 분류

매핑 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)

4.7.1 @Column 145p

- 객체 필드를 테이블 컬럼에 매핑한다.

속성	기능	기본값
name	필드와 매핑할 테이블의 컬럼 이름	
nullable	null 값의 허용 여부를 설정한다.	true

- nullable 속성
  - false 로 설정하면 DDL 생성 시에 not null 제약조건이 붙는다.
  - true 로 설정하면 null 허용
- insertable, updatable 속성은 데이터베이스에 저장되어 있는 정보를 읽기만 하고 실수로 변경하는 것을 방지하고 싶을 때 사용
- 속성 관련하여 145p , https://ttl-blog.tistory.com/114 참고

- @Column 을 생략하면 대부분 @Column 속성의 기본값이 적용되는데, 자바 기본 타입일 때는 nullable 속성에 예외가 있다.

```
// @Column 생략
int data1; // 자바 기본 타입

생성된 DDL : data1 integer not null
```

- 자바 기본 타입에는 null 값을 입력할 수 없다.

```
// @Column 생략
Integer data2; // 객체타입

생성된 DDL : data2 integer
```

- 객체 타입일 때만 null 값이 허용된다.

```
@Column
int data3; // 자바 기본 타입

생성된 DDL : data3 integer
```

- @Column은 `nullable = true` 가 기본값이므로 not null 제약조건을 설정하지 않는다.

⇒ 자바 기본 타입에 @Column을 사용하면 `nullable = false` 로 지정하는 것이 안전

4.7.2 @Enumerated 148p

- 자바의 enum 타입을 매핑할 때 사용.

enum 클래스

```
enum RoleType {
    ADMIN, USER
}
```

enum 이름으로 매핑

```
@Enumerated(EnumType.STRING)
private RoleType roleType;
```

enum 사용

```
member.setRoleType(RoleType.ADMIN); // DB에 문자 ADMIN으로 저장된다.
```

value

- `EnumType.ORDINAL`: enum 순서를 데이터베이스에 저장 (기본값)

  ex) enum에 정의된 순서대로 ADMIN은 0, USER는 1 값이 데이터베이스에 저장된다. ⇒ 권장하지 않음
- `EnumType.STRING`: enum 이름을 데이터베이스에 저장

  ex) enum 이름 그대로 ADMIN은 'ADMIN', USER는 'USER'라는 문자로 데이터베이스에 저장된다.

4.7.3 @Temporal 149p

- 날짜 타입(java.util.Date, java.util.Calendar)을 매핑할 때 사용

- `TemporalType.DATE` : 날짜, 데이터베이스 date 타입과 매핑 ex) 2013-10-11
- `TemporalType.TIME` : 시간, 데이터베이스 time 타입과 매핑 ex) 11:11:11
- `TemporalType.TIMESTAMP` : 날짜와 시간, 데이터베이스 timestamp 타입과 매핑 ex) 2021-02-02 11:11:11
- 기본값 : TemporalType 은 필수로 지정해야 한다.

```
@Temporal(TemporalType.DATE)
private Date date; // 날짜

@Temporal(TemporalType.TIME)
private Date time; // 시간

@Temporal(TemporalType.TIMESTAMP)
private Date timestamp; // 날짜와 시간
```

```
// 생성된 DDL
date date,
time time,
timestamp timestamp,
```

- 자바의 Date 타입에는 년월일 시분초가 있지만

  데이터베이스에는 date(날짜), time(시간), timestamp(날짜와 시간) 라는 세 가지 타입이 별도로 존재한다.
- @Temporal을 생략하면 자바의 Date와 가장 유사한 timestamp로 정의된다.

참고

timestamp 대신에 datetime을 예약어로 사용하는 데이터베이스도 있다(데이터베이스 방언 덕분에 애플리케이션 코드는 변경하지 않아도 된다).

- datetime: MySQL
- timestamp : H2, 오라클, PostgreSQL

4.7.4 @Lob 150p

- 데이터베이스 BLOB, CLOB 타입과 매핑한다.

속성정리

@Lob에는 지정할 수 있는 속성이 없다.

대신에 매핑하는 필드 타입이 문자면 `CLOB` 로 매핑하고 나머지는 `BLOB` 로 매핑한다.

- CLOB : String, char[], java.sql.CLOB
- BLOB : byte[], java.sql.BLOB

```
@Lob
private String lobString;

@Lob
private byte[] lobByte;
```

오라클

```
lobString clob,
lobByte blob,
```

MySQL

```
lobString longtext,
lobByte longblob,
```

PostgreSQL

```
lobString text,
lobByte oid,
```

4.7.5 @Transient 151p

- 이 필드는 매핑하지 않는다.
- 데이터베이스에 저장하지 않고 조회하지도 않는다.
- 주로 메모리상에서만 객체에 임시로 어떤 값을 보관하고 싶을 때 사용한다.

```
@Transient
private Integer temp;
```

4.7.6 @Access 151p

JPA 가 엔티티 데이터에 접근하는 방식을 지정

필드 접근

- `AccessType.FIELD` 로 지정한다.
- 필드에 직접 접근한다.
- 필드 접근 권한이 private이어도 접근할 수 있다.

```
@Entity
@Access(AccessType.FIELD)
public class Member {
    //
    @Id
    private String id;
```

```
    private String data1;
    private Stting data2;
}
```

- @Id 가 필드에 있으므로 `@Access(AccessType.FIELD)` 로 설정한 것과 같다.
- 따라서 `@Access` 는 생략해도 된다.

프로퍼티 접근

- `AccessType.PROPERTY` 로 지정한다.
- 접근자(Getter)를 사용한다.

```
@Entity
@Access(AccessType.PROPERTY)
public class Member {
    //
    private String id;

    private String data1;
    private Stting data2;

    @Id
    public String getId() {
        return id;
    }

    @Column
    public String getData1() {
        return data1;
    }

    public String getData2() {
        return data2;
    }
}
```

- @Id 가 프로터피에 있으므로 `Access(AccessType.PROPERTY)` 로 설정한 것과 같다.
- 따라서 `@Access` 는 생략해도 된다.

필드 접근 방식 + 프로퍼티 접근 방식

```
@Entity
public class Member {
    //
    @Id
    private String id;

    @Transient
    private  String firstName;

    @Transient
    private String lastName;

    @Access(AccessType.PROPERTY)
    public String getFullName() {
        return firstName + lastName;
    }
}
```

- @Id가 필드에 있으므로 기본은 필드 접근 방식을 사용하고
  getFullName() 만 프로퍼티 접근 방식을 사용한다.
- 따라서 회원 엔티티를 저장하면 회원 테이블의 FULLNAME 컬럼에 firstName + lastName 의 결과가 저장된다.