Day 13 (2019-03-13)
1. JPA

(1) 왜 JPA를 사용하는가?

1) SQL을 직접 다룰 때 문제점

반복, 반복 그리고 반복

아래와 같은 Member 도메인이 있을 때 Mybatis 등의 Entity framework를 이용하여 Bottom-up을 통한 CRUD 개발 과정을 살펴봅시다.

Member.class

```
public class Member {
    private Long key;
    private String id;
    private String name;
    private String phone;
}
```

먼저 Member를 기반으로 CRUD를 수행하는 쿼리를 작성하여 memberMapper.xml을 작성합니다

memberMapper.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-mapper.dtd">

<mapper namespace="app.example.demo.member.domain.mapper.MemberMapper">
    <resultMap id="selectMemberResultMap" type="app.example.demo.member.domain.Member">
        <result column="MEMBER_KEY" property="key"/>
        <result column="MEMBER_ID" property="id"/>
        <result column="MEMBER_NAME " property="name"/>
        <result column="PHONE" property="phone"/>
    </resultMap>

   <insert id="insertMember">
     INSERT INTO MEMBER(MEMEBER_KEY, MEMBER_ID, MEMBER_NAME, PHONE) VALUES(#{Key}, #{name}, #{id},  #{phone})
   </insert>

   <select id="selectMembers" resultMap="selectMemberResultMap>
         SELECT
                MEMBER_KEY
         ,  MEMBER_ID
         ,        MEMBER_NAME
         ,        PHONE
         FROM MEMBER
         WHERE MEMBER_KEY
   </select>

   <select id="selectMember" resultMap="selectMemberResultMap>
         SELECT
                  MEMBER_KEY
         ,  MEMBER_ID
         ,        MEMBER_NAME
         ,        PHONE
         FROM MEMBER WHERE MEMBER_KEY = #{key}
   </select>

   <update id="updeteMember">
        UPDATE MEMBER
        SET
                MEMBER_NAME  = #{memberName}
        ,         PHONE                  = #{phone}
        WHERE MEMBER_KEY = #{memberKey}
   </update>

        <delete id="deleteMember">
         UPDATE MEMBER
         SET
                DEL_FLAG= 'Y'
          WHERE MEMBER_KEY = #{memberKey};
        <delete>
</mapper>
```

MemberMapper.xml을 매핑하기 위한 MemberMapper.java 인터페이스를 생성합니다

MemberMapper.java

```
public interface MembeMapperr {
        void insertMember(Member member);
        List<Member> selectMembers();
        Member selectMember(Long memberKey);
        void updateMember(Member member);
        void deleteMember(Long memberKey);
}
```

→ 도메인 객체가 늘어날수록 위에서 작업한 대로 쿼리와 파라미터 바인딩, 쿼리 엘리먼트를 매핑하기 위한 interface 그리고 결과 매핑 등을 반복해서 작성해야합니다

→ 사람이 작성한 쿼리는 단순 문자열이기때문에 Run-time에서야 오류를 발견할 수 있습니다 .따라서 쿼리로 작성한 CRUD에 대한 Unit test는 필수입니다
(위 예제 코드에서 발생 가능한 오류를 찾아보세요)

새로운 필드가 추가된다면?

만약 Member의 나이를 저장하는 필드를 추가해달라는 요청이 들어왔습니다. 그렇다면 우리는 어디 어디를 수정해야할까요?

Field 추가

age 필드가 추가 된 Member.class

```
public class Member {
    private Long key;
    private String id;
    private String name;
    private String phone;
    private int age;
}
```

INSERT문 수정

Insert 문

```
   <insert id="insertMember">
     INSERT INTO MEMBER(MEMBER_KEY, MEMBER_ID, MEMBER_NAME, PHONE, AGE)
     VALUES(#{Key}, #{id}, #{name}, #{phone}, #{age})
   </insert>
```

SELECT문 수정

Select 문

```
 <resultMap id="selectMemberResultMap" type="app.example.demo.member.domain.Member">
     <result column="MEMBER_KEY" property="key"/>
     <result column="MEMBER_ID" property="id"/>
     <result column="MEMBER_NAME" property="name"/>
     <result column="PHONE" property="phone"/>
     <result column="AGE" property="age" /> <!-- ResultMap    -->
 </resultMap>

<select id="selectMembers" resultMap="selectMemberResultMap">
         SELECT
                  MEMBER_KEY
         ,  MEMBER_ID
         ,         MEMBER_NAME
         ,         PHONE
         ,          AGE /*    */
         FROM MEMBER
</select>

<select id="selectMember" resultMap="selectMemberResultMap">
    SELECT
                MEMBER_KEY
         ,  MEMBER_ID
         ,         MEMBER_NAME
         ,         PHONE
         ,          AGE /*    */
         FROM MEMBER
         WHERE MEMBER_KEY = #{key}
</select>
```

UPDATE문 수정

Update 문

```
<update id="updeteMember">
        UPDATE MEMBER
        SET
                MEMBER_NAME  = #{memberName}
        ,         PHONE                  = #{phone}
    ,   AGE                          = #{age}   /*    */
        WHERE MEMBER_KEY = #{memberKey}
</update>
```

만약 필드 추가가 아닌 필드의 제거라면?

→ 컬럼은 당연히 삭제할 것이므로 관련 쿼리와 ResultMap은 수정될 것입니다. 하지만 Model은 수정하지 않고 그대로 두는 케이스가 꽤 많습니다

연관관계에 있는 필드를 추가하려면?

Member가 속한 테이블과 관련된 정보를 함께 내려달라는 요청이 들어왔습니다. 어떻게 처리할까요?

먼저 Member 클래스에 Team과 관련된 필드를 추가해줍니다.

Team과 관련된 Field가 추가 된 Member.class

```
public class Member {
    private Long key;
    private String id;
    private String name;
    private String phone;
    private int age;
        private Long teamKey;
        private String teamId;
        private String teamName;
}
```

그에 따라서 Team과 Join 된 결과를 조회하는 쿼리와 ResultMap을 추가합니다.
(Select 절에서 선언된 프로젝션이 다르다면 새로운 resultMap을 만들기를 권장합니다)

Select query mapper

```
 <resultMap id="selectMemberWithTeamResultMap" type="app.example.demo.member.domain.Member">
     <result column="MEMBER_KEY" property="key"/>
     <result column="MEMBER_ID" property="id"/>
     <result column="MEMBER_NAME" property="name"/>
     <result column="PHONE" property="phone"/>
     <result column="AGE" property="age" />
     <result column="TEAM_KEY" property="teamKey" />
     <result column="TEAM_ID" property="teamId" />
     <result column="TEAM_NAME" property="teamName" />
 </resultMap>

<select id="selectMembersWithTeam" resultMap="selectMemberWithTeamResultMap>
         SELECT
                  ME.MEMBER_KEY
         ,  ME.MEMBER_ID
         ,         ME.MEMBER_NAME
         ,         ME.PHONE
         ,          ME.AGE
         ,  TE.TEAM_KEY
         ,  TE.TAEM_ID
         ,  TE.TAEM_NAME
         FROM MEMBER ME
         ,           TEAM TE
         WHERE
                TE.TEAM_KEy = ME.TEAM_KEY
</select>

<select id="selectMemberWithTeam" resultMap="selectMemberWithTeamResultMap>
    SELECT
                  ME.MEMBER_KEY
         ,  ME.MEMBER_ID
         ,         ME.MEMBER_NAME
         ,         ME.PHONE
         ,          ME.AGE
         FROM MEMBER ME
         ,           TEAM TE
         WHERE
                TE.TEAM_KEy = ME.TEAM_KEY
         AND MEMBER_KEY = #{key}
</select>
```

Mapper interface에 위 엘리먼트를 매핑하는 메소드를 선언합니다

```
public interface MembeMapperr {
        void insertMember(Member member);
        List<Member> selectMembers();
        List<Member> selectMembersWithTeam();
        Member selectMember(Long memberKey);
        Member selectMemberWithTeam(Long memberKey);
        void updateMember(Member member);
        void deleteMember(Long memberKey);
}
```

→ selectMember(s)WithTeam을 호출하면 Team과 관련된 정보가 정말 있는지 이 메소드를 사용하는 개발자는 어떻게 알까요?

최소 한번은 디버깅을 걸어서 확인하거나 쿼리를 직접 확인해보아야합니다. (**selectMemberWithTeam 쿼리는 Join은 되어있으나 결과에 Team과 관련된 정보에 대한 Select 구문이 없습니다**)

→ Member라는 도메인이 점점 Member는 상관없는 Field로 도배됩니다.
teamKey, teamId, teamName은 Member 클래스에 있는 것은 어울리지 않습니다. 객체지향적으로 보면 Member 클래스에서 Team 클래스 타입의 field를 가지고 있는 것이 훨씬 객체지향스럽겠지요.

Team 객체를 참조하는 Member.class

```
public class Member {
    private Long key;
    private String id;
    private String name;
    private String phone;
    private int age;
        Team team;
}
```

하지만 Mybatis를 사용하면 이런 구조를 가지면 ResultMap의 매핑 절차가 번거롭습니다

또한 Member 클래스를 Response 넘겨줄경우 Json이 그다지 아름답지 않습니다.
(Domain 영역의 Member와 별개로 DTO를 나누는 방법으로 해결은 가능합니다)

이런 문제 발생의 근원은 기본적으로 Java의 객체와 객체 연관관계(객체 그래프)가 RDB와 다름에 있습니다.
그러하여 어느정도 절충안을 두고 패러다임의 불일치를 맞춰주어야 하기에 이러한 문제가 생깁니다.

2) 모든 Domain은 변경이 필연적으로 발생한다

항상 사용자의 요구사항이 고정적이지 않듯이 요구사항과 관계없이 우리가 Domain에 대한 리팩토링을 진행하면서도 Domain은 바뀔 수 밖에 없습니다.

이러한 상황에서 위에 나열한 방법을 통핸 개발은 아래와 같은 단점이 있습니다

- Type safe 하지 않다
- 실수할 확률이 높다 (Ctrl+c, v, 오타, 문법 등)
- 간단한 변경에도 수정해야할 코드가 많다
- 단순 CRUD를 반복하는 데에도 꽤나 많은 시간을 요구한다

비즈니스로직에 중점을 두어 견고한 로직을 작성해야함에도 위의 문제점으로 인해 흐름을 잃기 쉽습니다.
또한 사람의 사고는 통상적으로 Context Switching이 유연하지 않아 더욱 주요 기능 개발에 중심을 두기 힘듭니다.

(2) JPA란?

1) 등장 배경

Java ORM(Object-Relational Mapping)의 표준 스펙으로 RDB와 Object 간에 발생하는 패러다임 불일치를 완화해줍니다.
Java에서 객체와 객체간의 연관관계를 유지하면 그것을 JPA가 알아서 RDB의 테이블과 테이블간의 연관관계로 바꾸어주는 역할을 하는것입니다.

예를들어서 Member 목록을 가진 Team이 있다고 할 때 1번 키를 가진 Team에 새로운 Member를 추가하는 것을 SQL로 개발하면 아래와 같습니다.

```
INSERT INTO MEMBER(MEMBER_KEY, MEMBER_NAME, MEMBER_ID, TEAM_KEY) values (...)
```

반면 Java로 이를 표현하면 아래와 같이 컨테이너에 객체를 추가하면 됩니다.

```
team.addMember(new Member());
```

→ JPA는 Persistence 영역을 Java의 Container로 추상화하는 개념부터 시작한 것입니다.

2) 자주사용하는 용어설명

Entity Manager Factory와 Entity Manager

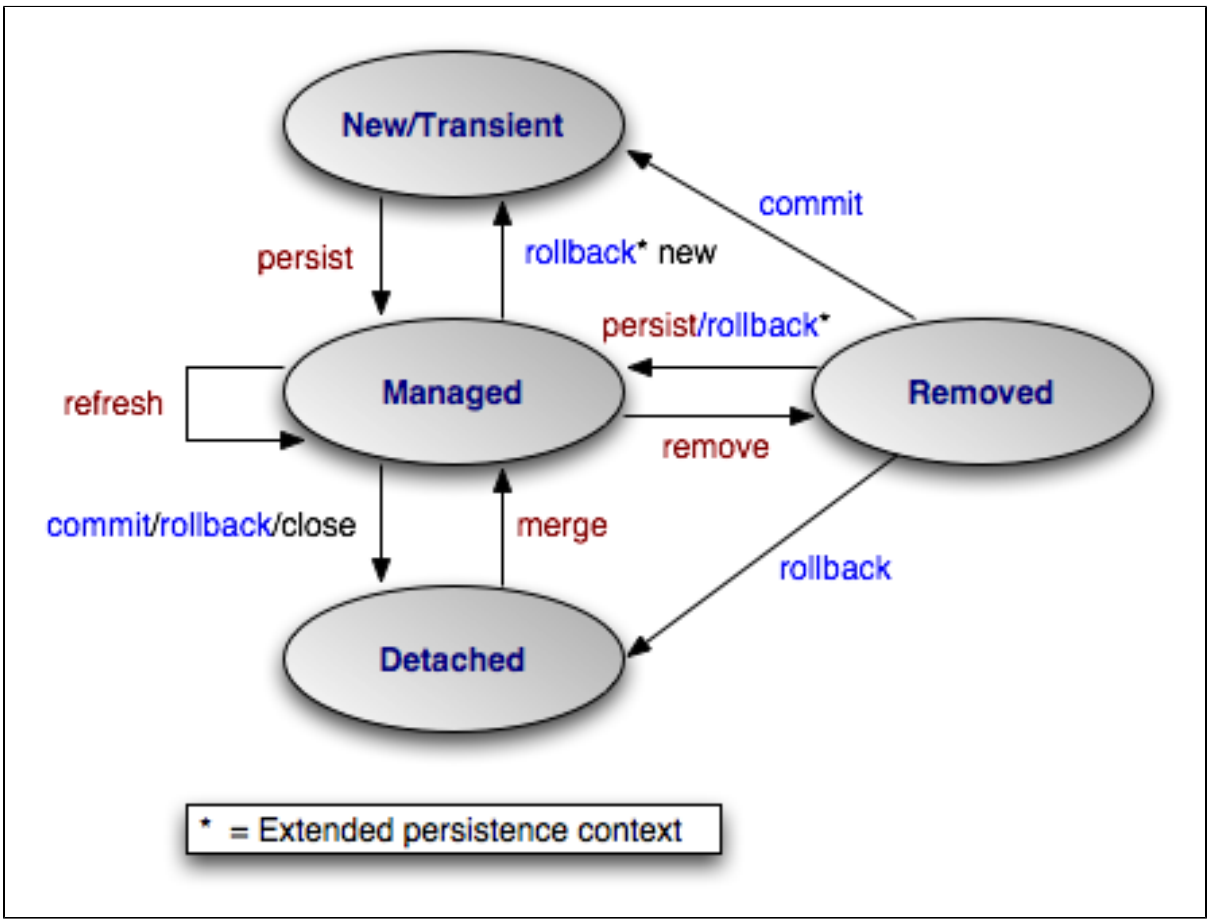
Entity Manager는 Entity를 저장하고 수정하고 삭제하고 조회하는 등 Entity를 관리하는 일을 합니다. 즉 가상의 데이터베이스라고 생각하면 됩니다.
하지만 Entity Manager는 여러 스레드가 동시에 접근하면 동시성 문제가 발생하므로 한 번에 하나의 스레드가 접근하도록 해야합니다.

Entity Manager Factory는 여러 스레드가 동시에 접근해도 안전한 영역으로 Entity Manager를 관리합니다.

보통 Entity Manger는 한 트랜잭션에서 유효합니다. Spring의 환경을 생각하면 하나의 Request Thread 혹은 Worker Thread의 Context마다 Entity Manager가 존재할 것입니다.

Entity 생명주기

상태	설명
비영속(new/transient)	영속성 컨텍스트와 전혀 관계가 없는 상태
영속(managed)	영속성 컨텍스트에 저장된 상태
준영속(detached)	영속성 컨텍스트에 저장되었다가 분리된 상태
삭제(removed)	삭제된 상태



Persistence Context(영속성 컨텍스트)

Entity를 영구 저장하는 환경으로 Entity Manage를 통해 Entity를 보관하고 관리하는 영역입니다.

영속성 컨텍스트에 저장된 Entity는 반드시 키 값이 있어야 합니다. 즉 Managed 상태인 Entity는 반드시 식별자 값을 보유해야합니다.

이러한 영속성 컨텍스트에 저장된 Entity는 보통 트랜잭션을 커밋하는 시점에 Entity들을 Database에 반영하는데 이것을 플러시(flush)라고 합니다.

아래는 영속성 컨텍스트가 Entity를 관리하면 얻는 이점입니다.

1차 캐시	• 영속성 컨텍스트를 통해 Entity를 조회하면 DB를 조회하기 전에 영속성 컨텍스트 내에 존재하는 1차캐시를 먼저 조회합니다 따라서 Database에 접근하는 횟수를 획기적으로 줄일 수 있습니다

동일성보장	• 영속성 컨텍스트에서 식별자를 통해 조회한 Entity는 언제나 동일합니다. • 동일성은 a == b 연산의 결과가 true입니다. (동등성은 a.equals(b) 또는 b.equals(a)가 true입니다) • REPEATABLE READ 수준의 트랜잭션의 격리수준을 Database가 아닌 Application 차원에서 얻을 수 있습니다
트랜잭션을 지원하는 쓰기 지연	• Mybatis 등을 이용하면 mapper의 메소드를 실행할 때마다 트랜잭션의 범위이더라도 Network call을 하므로 비용이 발생합니다 하지만 JPA에서는 Transaction을 commit 하는 시점까지 Insert SQL을 쓰기지연 SQL 저장소에 모아두었다가 한꺼번에 내보냅니다 • 만약 Rollback이 된다면 쓰기지연 SQL 저장소에 존재하는 Insert SQL은 지우면 그만입니다. 반면 Mybatis의 경우엔 Rollback을 위한 JDBC API를 호출해야합니다

변경 감지 (Dirty checking)

- 영속 상태의 Entity는 영속성 컨텍스트가 flush 될 때 변경 사항이 자동으로 감지됩니다.
- 만약 변경된 Entity가 존재한다면 쓰지기연 SQL 저장소에 Update 쿼리를 보관합니다.
  - Update 쿼리는 기본적으로 모든 field를 수정하도록 작성됩니다. 이것은 Application에서 한 번 동적으로 생성된 쿼리를 재사용하귀 위함과
    Database에 동일 쿼리를 보낼 때 파싱된 쿼리를 재사용하는 Query Cache를 활용하기 좋습니다.
  - 하지만 필드가 많거나(일반적으로 30개이상) 수정되는 내용이 너무 크면 수정된 데이터에 대해서만 동적인 Update SQL을 생성하는 설정이 필요합니다
- 변경 감지 기능을 통해서 변경이 있을 때만 수정일, 수정한사람을 변경하는 것과 Audit을 쌓는 기능도 쉽게 구현할 수 있습니다.
  아래는 PL/SQL에서 각 테이블마다 작성하는 p_EQUAL Function입니다.

```
FUNCTION p_EQUAL (
        iRow  IN MST_USER%ROWTYPE
)
RETURN BOOLEAN
IS
        vBool BOOLEAN;
BEGIN
        vBool:= ROW.USER_KEY      = iRow.USER_KEY
                AND ROW.USER_ID       = iRow.USER_ID
                AND ROW.USER_NAME     = iRow.USER_NAME
                AND NVL(TO_CHAR( ROW.USER_NAME_ENG), COM_TYPE.NULL_STR) =
                        NVL(TO_CHAR(iRow.USER_NAME_ENG), COM_TYPE.NULL_STR)
                AND NVL(TO_CHAR( ROW.USER_PRIVS), COM_TYPE.NULL_STR) =
                        NVL(TO_CHAR(iRow.USER_PRIVS), COM_TYPE.NULL_STR)
                AND ROW.EMAIL         = iRow.EMAIL
                AND NVL(TO_CHAR( ROW.EMAIL_ALT), COM_TYPE.NULL_STR) =
                        NVL(TO_CHAR(iRow.EMAIL_ALT), COM_TYPE.NULL_STR)
                AND NVL(TO_CHAR( ROW.PHONE_MOBILE), COM_TYPE.NULL_STR) =
                        NVL(TO_CHAR(iRow.PHONE_MOBILE), COM_TYPE.NULL_STR)
                AND NVL(TO_CHAR( ROW.PHONE_OFFICE), COM_TYPE.NULL_STR) =
                        NVL(TO_CHAR(iRow.PHONE_OFFICE), COM_TYPE.NULL_STR)
                AND NVL(TO_CHAR( ROW.FAX), COM_TYPE.NULL_STR) =
                        NVL(TO_CHAR(iRow.FAX), COM_TYPE.NULL_STR)
                AND ROW.SITE_KEY      = iRow.SITE_KEY
                AND ROW.DEL_FLAG      = iRow.DEL_FLAG
/*                AND ROW.INPUT_TIME    = iRow.INPUT_TIME     */
/*                AND ROW.USER_KEY_UPD  = iRow.USER_KEY_UPD   */
        ;
        RETURN vBool;
END p_EQUAL;
```

지연로딩

- 우리가 사용할 Entity와 그 Entity가 참조하는 또 다른 Entity가 있을 때 실제 참조하는 Entity가 필요한 시점까지 조회 시점을 미룰 수 있습니다.
- Member 객체만 조회하여 비즈니스 로직 실행 중 member.getTeam() 호출하는 순간에 Member와 연관된 Team 정보를 조회하는 것입니다.
- Entity의 Relation Mapping(OneToOne, ManyToOne, ManyToMany 등)에 fetchType을 LAZY로 설정하여 적용 가능합니다.
  자세한 내용은 다음 세미나 시간에 Spring Data JPA를 다루며 살펴보겠습니다.

(3) Entity Mapping

1) @Entity

테이블과 매핑할 클래스를 Entity 애노테이션을 통해 선언합니다.

Entity 매핑 예제

```
@Entity
public class Member {
}
```

2) @Table

매핑할 테이블 이름을 지정할 수 있습니다.

Table mapping 예제

```
@Table(name = "MEMBERS")
@Entity
public class Member {
}
```

3) Field Mapping

어노테이션	설명
@Id	테이블의 Primary Key를 지정합니다. 이때 Generated Value 어노테이션을 통해 자동생성되는 값임을 명시합니다.
@GeneratedValue	strategy 값으로 자동 생성할 전략을 지정합니다. 지원되는 전략은 AUTO, TABLE, SEQUENCE, IDENTITY가 있습니다. 또한 generator는 Sequence 전략을 사용할 때 사용할 SequenceGenerator를 명시하는 부분입니다.
@Column	필드와 Column을 매핑합니다. unique, nullable 등 값을 통해서 Schema를 매핑할 수 있습니다. 여러 Column에 의한 Unique 제약조건은 @Table 어노테이션에 uniqueConstraints 값을 설정하여 매핑할 수 있습니다
@Enumerated	Enum 유형의 필드를 매핑할 수 있습니다.
@Lob	Blob, Clob 컬럼에 매핑할 수 있습니다.
@Transient	Database에 매핑하지 않는 필드를 지정할 수 있습니다.

Field Mapping 예제

```
@Table(name = "MEMBERS")
@Entity
@SequenceGenerator(name = "MEMBER_SEQ", sequenceName = "MEMBER_SEQ")
public class Member {
        @Id
        @Column(name = "MEMBER_KEY")
        @GeneratedValue(strategy = GenerationType.AUTO, generator = "MEMBER_SEQ")
        private Long key;

        @Column(name = "MEMBER_ID", unique = true, nullable = false)
        private String id;

        @Column(name = "MEMBER_NAME", nullable = false)
        private String name;

        @Column(name = "PHONE")
        private String phone;


        @Enumerated(EnumType.STRING)
        private MemberType type;

        @Lob
        private String description;

        @Transient
        private String dummy;
}

public enum MemberType {
        ADMIN, USER
}
```

4) 스키마 자동 생성

JPA를 통해 설정한 Entity를 기반으로 Schema를 자동으로 생성할 수 있습니다.

옵션	설명
create	DROP + CREATE
create-drop	DROP+CREATE+DROP
update	테이블과 엔티티 매핑정보를 비교해서 변경 사항만 수정
validate	테이블과 엔티티 매핑정보를 비교해서 차이가 있으면 경고를 남기고 애플리케이션을 실행하지 않는다. (DDL을 수정하지 않는다.)
none	자동생성기능을 사용하지 않으려면 hibernate.hbm2ddl.auto 속성값 자체를 삭제하거나, 유효하지 않은 값(none)을 주면 된다.

개발 초기 환경이라면 Create 또는 Update를 사용하고 CI 또는 초기화 상태로 자동화 테스트를 진행하는 개발자 환경은 Create or Create-Drop을 사용합니다
보통 그외의 Staging, 운영 등 서버에서는 validate 또는 none으로 지정합니다

5) 연관관계 매핑

핵심 키워드

키워드	설명
방향성	단방향, 양방향 Member → Team 혹은 Team → Member로만 참조하는 것을 단방향이라고 하며 Member → Team 그리고 Team → Member로 참조하는 것을 양방향이라고 한다
다중성	다대일, 일대다, 일대일, 다대다 관계
연관관계의 주인	만약 Entity를 양방향 연관관계로 만들경우 연관관계의 주인을 지정해야 하는데 외래키를 관리할 Entity를 지칭한다고 생각하면 편하다

RDB의 테이블엔 단방향 방향성이 없습니다. 외래키를 가진 테이블이든 외래키의 의해 참조되는 테이블이든 외래키를 통해
방향에 상관없이 참조할 수 있기 때문입니다.

하지만 객체에서는 다릅니다. 양방향일 수도 있고 단방향일 수도 있습니다.
(정확하게 말하면 객체에서 양방향 관계는 없고 상반된 단방향 관계 두개입니다)

이전부터 사용한 Member - Team의 연관관계를 예로 설명을 진행하겠습니다.
여기서 Member는 Team에 속할수도 있고 속하지 않을수도 있으며 Member는 하나의 Team에만 속해야 하는 '**선택적 비식별 관계**'입니다.

키워드	설명
선택적 비식별 관계	?
선택적 식별 관계	?
필수적 식별 관계	?

이러한 관계를 기반으로 Entity Mapping을 하면 아래와 같습니다.

ManyToOne mapping 예제

```
@Table(name = "MEMBERS")
@Entity
@Getter
@AllArgsConstructor
@NoArgsConstructor
@Builder
@SequenceGenerator(name = "MEMBER_SEQ", sequenceName = "MEMBER_SEQ")
public class Member {
        @Id
        @Column(name = "MEMBER_KEY")
        @GeneratedValue(strategy = GenerationType.AUTO, generator = "MEMBER_SEQ")
        private Long key;

        @Column(name = "MEMBER_ID", unique = true, nullable = false)
```

```
        private String id;

        @Column(name = "MEMBER_NAME", nullable = false)
        private String name;

        @Column(name = "PHONE")
        private String phone;

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


        public void setTeam(Team team){
          if(this.team != null){
              this.team.removeMember(this);
          }

      team.addMember(this);
          this.team = team;
        }
}
```

Member의 입장에서 다수의 Member가 하나의 Team에 매핑되는 형태이므로 Team에 @ManyToOne 어노테이션을 명시하였습니다.

또한 TEAM 테이블을 참조하기 위한 외래키를 지정하기 위해 @JoinColumn을 명시하였습니다.

OneToMany 매핑

```
@Table(name = "TEAM")
@Entity
@Getter
@AllArgsConstructor
@NoArgsConstructor
@Builder
@SequenceGenerator(name = "TEAM_SEQ", sequenceName = "TEAM_SEQ")
public class Team {

    @Id
    @Column(name = "TEAM_KEY")
    @GeneratedValue(strategy = GenerationType.AUTO, generator = "TEAM_SEQ")
    private Long key;

    @Column(name = "TEAM_NAME", nullable = false)
    private String name;

    @OneToMany(cascade = CascadeType.ALL, orphanRemoval = true, mappedBy = "team")
    private List<Member> members;


        public Team addMember(Member member) {
      if(this.members == null){
          this.members = new ArrayList<>();
      }

      this.members.add(member);
      return this;
        }

        public Team removeMember(Member target) {
      if(this.members == null){
          return this;
      }
      this.members.removeIf(member -> member.getKey().equals(target.getKey()));
      return this;
```

```
        }
}
```

하나의 Team에 여러 Member가 포함 될 수 있기 때문에 List<Member>에 @OneToMany 어노테이션을 명시하였습니다.

또한 mappedBy 속성을 통해 Member에서 "team" 이라는 필드를 통해 나(Team)와 매핑되었다 라고 명시합니다.
(헷갈리면 한쪽에 ManyToOne 이면 반대쪽은 OneToMany라고 생각하세요)

여기서 **mappedBy**라는 속성을 가진 Entity가 **연관관계의 주인이 '아닌' Entity**가 됩니다.
즉, 반대편에 있는 Member Entity가 연관관계의 주인이 되고 연관관계의 주인이기 때문에 외래키를 관리합니다.
주의할 점은 연관관계의 주인은 외래키의 관리자일 뿐 비즈니스로직 상에서 중요 위치에 있는 Entiry라고 판단하지 않아야 하는 것입니다.

생성된 테이블 스키마를 확인하면 MEMBER 테이블에 TEAM 테이블을 참조하기 위한 외래키 TEAM_KEY컬럼이 있는 것을 확인할 수 있습니다.

Cascade 옵션은 ALL, PERSIST, MERGE, REMOVE, REFRESH, DETACH 옵션이 있습니다. 각각 영속화에 대한 연쇄작용 옵션입니다.
예를들어 CascadeType이 ALL이면 Team Entity가 영속화 될 때 아직 영속화되지 않은 채 Team에 속한 Member 목록이 함께 영속화 됩니다.
또한 Team Entity가 삭제 될 때 포함 된 Member Entity들도 함께 삭제 됩니다.

orphanRemoval 옵션은 고아 Entity에 대한 자동 삭제 처리입니다. 예를 들어 member.setTeam(null)을 하면 연관관계가 끊긴 고아 Entity가 되며
이러한 Member는 영속화 되면서 삭제됩니다.

6) 양방향 연관관계의 주의점

흔히 발생하는 실수가 연관관계의 주인에는 값을 입력하고 연관관계의 주인이 아닌 곳에만 값을 입력하는 것입니다.

ManyToOne mapping 예제

```
Member member1 = new Member();
Member member2 = new Member();
memberRepository.saveAll(Arrays.asList(member1, member2));

Team team = new Team();
team.getMembers().add(member1);
team.getMembers().add(member2);

teamRepository.save(team);
```

위 예제 코드를 보면 **연관관계의 주인인 Member**에 team을 set하는 코드가 없습니다. 따라서 MEMBER 테이블에 TEAM_KEY가 없는 ROW들만 쌓인 결과가 타나납니다

그렇다면 연관관계의 주인에만 값을 저장하면 될까요?

ManyToOne mapping 예제

```
Team team = new Team();
teamRepository.save(team);

Member member1 = new Member();
Member member2 = new Member();
member1.setTeam(team);
```

```
member2.setTeam(team);

memberRepository.saveAll(Arrays.asList(member1, member2));
```

이럴 경우엔 정상적으로 Table에 값은 저장되지만 객체 상태가 일관성있지 않습니다. 분명 Team Entity에 Member Entity를 추가하였으나 team.getMembers()를 호출하면 비어있기 때문입니다.

따라서 이러한 연관관계 설정에서 발생하는 문제를 방지하고자 아래와 같이 연관관계 편의 메소드를 작성합니다.

연관관계 편의 메소드가 적용된 Member.class

```
@Table(name = "MEMBERS")
@Entity
@Getter
@AllArgsConstructor
@NoArgsConstructor
@Builder
@SequenceGenerator(name = "MEMBER_SEQ", sequenceName = "MEMBER_SEQ")
public class Member {
        @Id
        @Column(name = "MEMBER_KEY")
        @GeneratedValue(strategy = GenerationType.AUTO, generator = "MEMBER_SEQ")
        private Long key;

        @Column(name = "MEMBER_ID", unique = true, nullable = false)
        private String id;

        @Column(name = "MEMBER_NAME", nullable = false)
        private String name;

        @Column(name = "PHONE")
        private String phone;

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

        //
        public void setTeam(Team team){
          if(this.team != null){
              this.team.removeMember(this);        //
          }

      team.addMember(this);               //
          this.team = team;
        }
}
```

방어코드가 적용된 Team.class

```
@Table(name = "TEAM")
@Entity
@Getter
@AllArgsConstructor
@NoArgsConstructor
@Builder
@SequenceGenerator(name = "TEAM_SEQ", sequenceName = "TEAM_SEQ")
public class Team {

    @Id
    @Column(name = "TEAM_KEY")
    @GeneratedValue(strategy = GenerationType.AUTO, generator = "TEAM_SEQ")
    private Long key;

    @Column(name = "TEAM_NAME", nullable = false)
```

```
    private String name;

    @OneToMany(cascade = CascadeType.ALL, orphanRemoval = true, mappedBy = "team")
    private List<Member> members;

        public Team addMember(Member member) {
      if(this.members == null){
          this.members = new ArrayList<>();
      }

      this.members.add(member);
      return this;
        }

        public Team removeMember(Member target) {
      if(this.members == null){
          return this;
      }
      this.members.removeIf(member -> member.getKey().equals(target.getKey()));
      return this;
        }
}
```

연관관계 편의메소드는 연관관계 주인 측에서만 작성하는 것이 속 편합니다.

만약 양쪽에서 작성한다면... 예를 들어 Team의 addMember에서 member.setTeam을 호출하게 되면 메소드가 순환 호출 구조를 이루어 변경 사항에 대한 연쇄 수정이 일어날 수 있고
제대로 validation 로직을 수행하지 않으면 상호호출로 인한 무한루프에 빠질 수 있기 때문입니다.