JPA (Java Persistence API)

Contents

Content

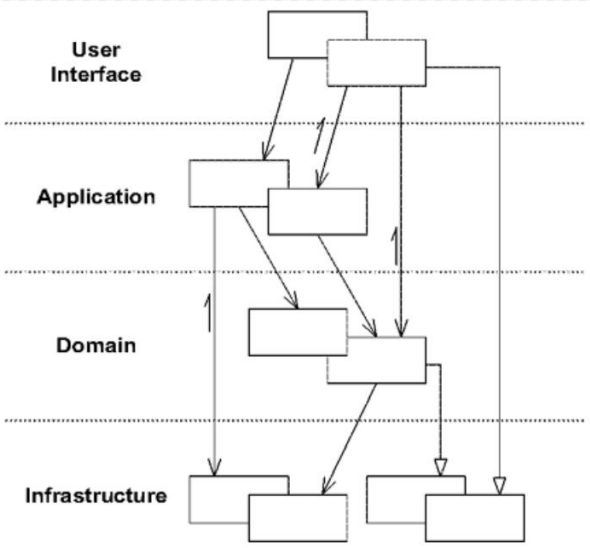
I. DDD의 이해

1. 개요

01. 레이어 아키텍처(Layered Architecture)

- 시스템을 유사한 책임을 지닌 레이어로 분해한 후 각각의 레이어가 하위의 레이어만 의존 하도록 하는 시스템을 구성 하는 것
- 목적
  : 관심사의 분리를 통한 결합도는 낮추고 재사용성을 높이고 유지 보수성 향상
- 적용 방식
  : 완화된 레이어 시스템 (relaxed layered system)
  - 하위 레이어만 의존해야만 하는 제약을 완화
  : 상속을 통한 레이어 구성 (layering through in inheritance)
  - 인터페이스나 클래스 상속을 받아 구현
- 적용
  : Model-View-Controller

일반적인 엔터프라이즈 애플리케이션의 레이어 구성



- **User Interface Layer**
  : 사용자에게 정보를 보여주고 사용자의 명령을 해석 하는 일을 책임 짐.

- **Application Layer**
  : 수행할 작업을 정의하고 표현하는 계층으로 업무 규칙이 포함 되지 않으며 작업을 조정 하고 아래 위치한 도메인 객체의 협력자에게 작업을 위임 함.

- **Domain Layer**
  : 업무 개념과 업무 상황에 관한 정보, 업무 규칙을 표현 하는 일을 책임짐. 상태 저장과 관련된 기술은 Infrastructure 위임

- **Infrastructure Layer**
  : 일반화된 기술 기능 제공, 메시지 전송, 도메인 영속화, UI에 위젯등.

02. 일반적인 레이어 분리 원칙

- Model-View Separation(Fowler POEAA, Larman AUP)
  : 모델(도메인 로직과 관련된 관심사)과 뷰(사용자 인터페이스와 관련된 관심사)는 별도의 레이어로 분리

- Clean and Thin View(Johnson J2EED)
  : 모델-뷰 분리원칙에 따라 모델과 뷰를 분리했다면 뷰는 오직 링크업과 화면 출력 로직을 포함해야 하며(Clean View), 시스템의 상태를 변경시킬 수 있는 비즈니스 로직을 포함해서는 안 된다(Thin View

- PI, Persistence Ignorance(Nilsson, ADDD)
  : 도메인 로직과 영속성 로직은 서로 다른 레이어로 분리, 도메인 객체를 데이터베이스 관련 인프라스트럭처에 독립적인 POJO(Plain Old Java Object)로 개발하고 DAO(Data Access Object) 패턴[Alur CORE]을 이용해서 인터페이스 하부로 영속성 메커니즘을 캡슐화하는 것

- Domain Layer Isolation(Evans, DDD)
  : 도메인 레이어는 비즈니스를 구성하는 핵심 개념과 중요한 정보, 준수해야 하는 비즈니스 규칙을 표현하는 곳이다. 시스템을 단순하고 유연한 상태로 유지하기 위해서는 도메인 레이어를 기술적인 이슈로부터 고립시켜야 한다

2. DDD (Domain Driven Design)

도메인 주도 개발은 Eric Evans가 2003년 출간한 Domain-Driven Design책으로 처음 소개한 방법론으로 "유용한 소프트웨어를 개발하고 싶다면 도메인 귀를 기울여라" 라는 슬로건으로 시작됨

01. 도메인 이란 ?

- 일반적인 요구사항, 전문 용어, 그리고 컴퓨터 프로그래밍 분야에서 문제를 풀기 위해 설계된 어떤 소프트웨어 프로그램에 대한 기능성을 정의하는 연구의 한 영역
- 소프트웨어로 해결하고자 하는 문제 영역
- 통신회사의 청약과 관련된 시식 = 도메인

02. 도메인 모델

- 특정 도메인을 개념적으로 표현한 것
- 도메인 모델을 사용하면 여러 관계자들(개발자, 기획자, 사용자 등)이 동일한 모습으로 도메인을 이해하고 도메인 지식을 공유하는 데 도움이 된다
- 모델의 각 구성 요소는 특정 도메인을 한정할 때 비로소 의미가 완전해지기 때문에, 각 하위 도메인마다 별도로 모델을 만들어야 한다.
- 애자일 개발 방식으로 반복 수행 하여 완성도를 높임

- 통신회사에 가입 처리시 : 고객 모델, 가입의 상태 관리를 하는 가입모델, 구매한 상품 모델

03. 도메인 주도 개발 이란 ?

- 도메인 전문가와 개발자 사이의 의사소통의 어려움은 해소 하기 위해 보편언어(Ubiquitous Languages)를 사용여 도메인과 구현을 충분히 만족 하는 모델을 만드는 것
- 설계와 구현은 계속된 수정 과정을 거친다.
- 도메인에 대한 역할과 책임을 부여 하기 위해서 경계를 설정 한다. (Bounded Context)

04. Bounded Context

- 독립적으로 서비스 될 떼 문제없는 업무 범위
- 역할과 책임 명확
- 도메인 : Bounded Context는 1:1이 이상적임
- 다른 Context 연결은 API통신으로만 한다.
- 주의할 점
  : 하위 도메인 모델이 뒤섞이지 않도록 한다.
  : 도메인이 섞이면 기능 확장이 어렵게 되고 서비스 경쟁력이 떨어짐
  : 개별 Bounded Context Package로 구성

3. 도메인 주도 설계 기본 요소

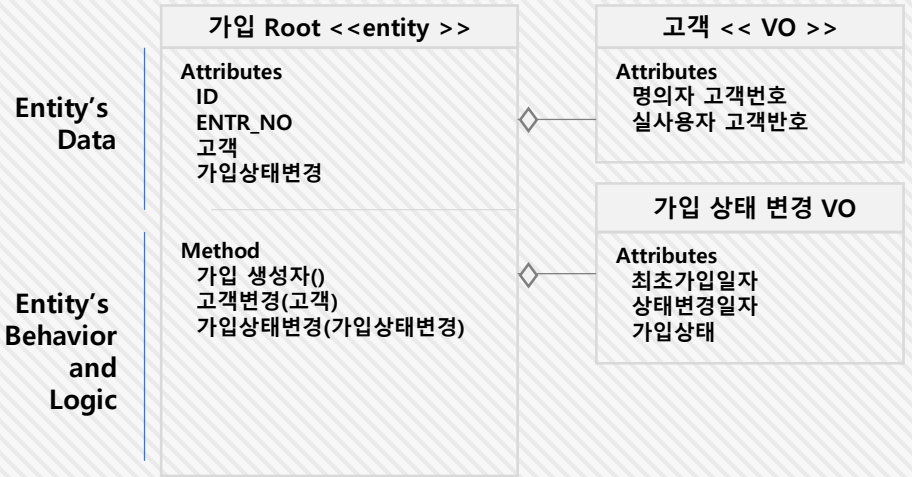
01. Entity, Value

- Entity

: 속성이 아인 식별성을 기준으로 정의되는 도메인 객체
: 식별자는 Entity 객체마다 고유해서 각 Entity는 서로 다른 식별자를 갖는다.
: 생성 시점에 필요한 것을 전달 한다.
: setXXX Method는 완전한 상태가 아닐 수 있으므로 사용 자제

- Value

: 식별상이 아닌 속성을 이용해 정의 되는 불변 객체
: 의미를 명확하게 표현 하거나 두 개 이상의 데이터가 개면적으로 하나인 경우 사용
; 값의 변경은 새로운 Value 객체를 할당
: 주민번호는 String로만 표현 되지 않음 -> 자료형 + 검증 로직

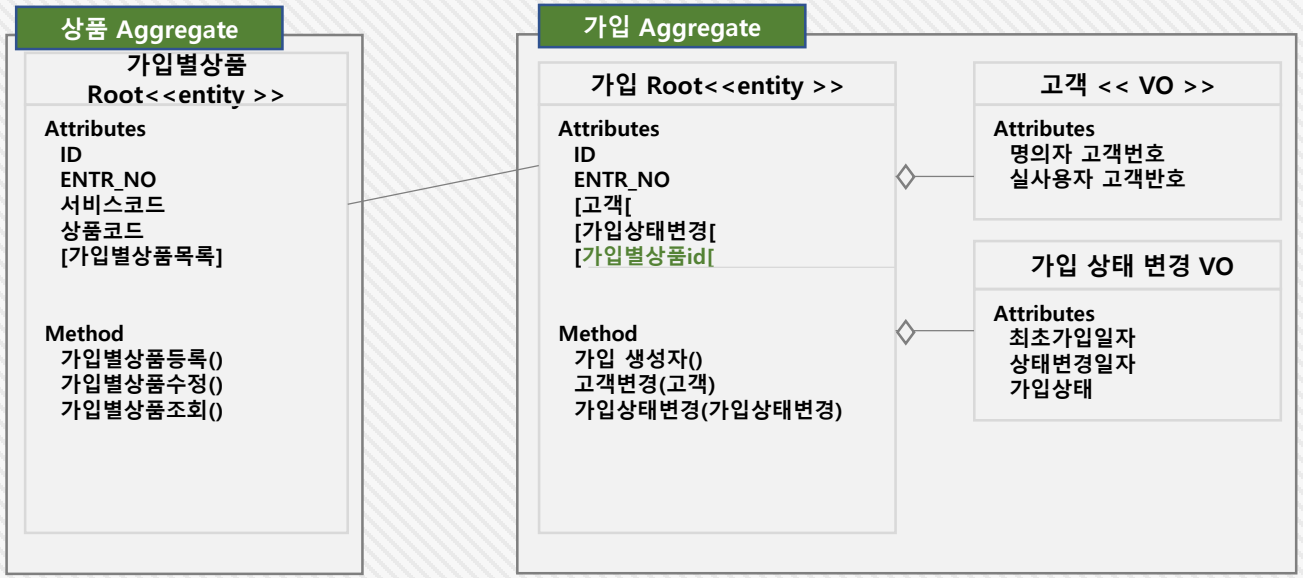


02. Aggregate

- 관련 객체를 하나로 묶음으로 데이터를 변경 하는 단위
- 루트 엔티티를 갖는다.

: 애그리커트가 제공해야 헐 도메인 기능 구현
: 내부 구현을 숨겨서 애그리거트 단위로 캡슐화
: 애그리커드에 속해 있는 Entity와 Value를 이용하여 기능 구현
: 애그리커트의 참조는 id를 이용한 참조 방식 사용
: 직접 참조 시 편리하지만 성능 및 확장 어려움 발생
: 외부에서 객체 참조 시 루트 애그리커트를 통해서 참조

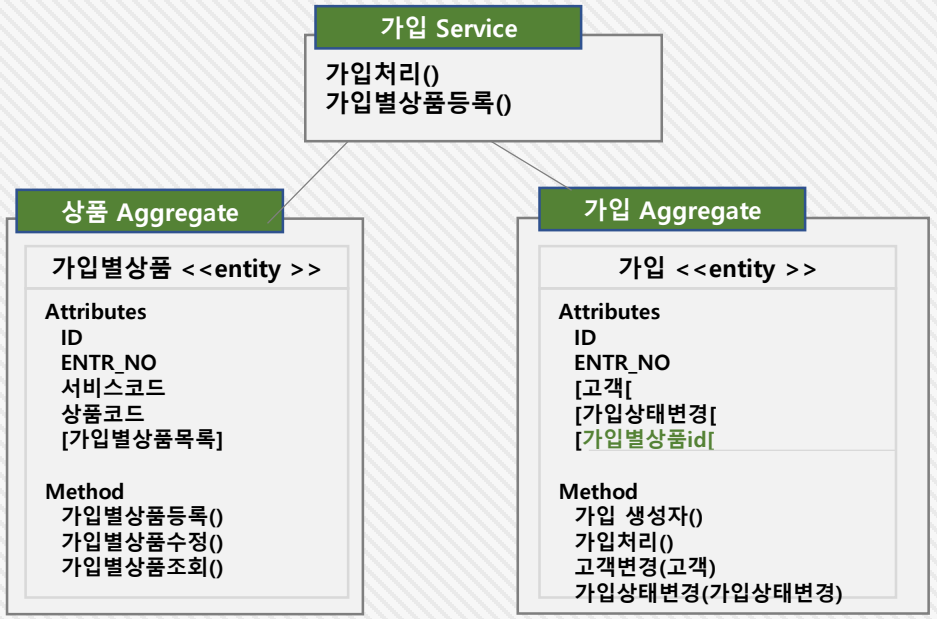
- Aggregate에 속해 객체는 유사 하거나 동일한 라이프 사이클을 갖는다.
- 한 애그리거트에 속한 객체는 다른 애그리거트에 속하지 않는다.
- 자기자신을 관리 할 뿐 다른 애그리커트를 관리 하지 않음



3. 도메인 주도 설계 기본 요소

03. Service

- 업무의 흐름
  : Use Case를 생각 해라
- 도메인 로직을 담당
- 상태 없이 로직만 구현
  : 연산에 영향을 주는 상태를 잦기 않는다,
- Transaction의 주체
- Domain Object애 위치 시키기 어려운 Operation으로 Domain 객체 호출
  : 가입 상태를 변경 하고 상품을 등록 한다면 가입상태변경 애그리거트와 상품 등록 애그리거트로 분리 하고 서비스에서 로직 구현
- Module(Package)
  : 인지 과부하 방지와 낮은 결합도와 높은 응집도를 가진다



04. Repository

- 구현을 위한 모델
- 애그리커드 단위로 도메인 객체를 저장하고 조회 하는 기능
  : 애그리커드에 대한 영속성 관리를 통한 일관성 유지
- 도메인 영역과 데이터 인프라스럭쳐 계층 분리하여 데이터 계층에 대한 결합도 낮추기 위한 방안

05. Factory

- 어떤 객체나 전체 AGGREATE를 생성하는 일이 복잡하다면 팩토리를 이용해 이것을 캡슐화 할 수 있다

Content

II. JPA

1. 기본 용어

- 자바 ORM기술에 대한 표준 명세로 자바 에서 제공하는 API

01. 영속성(Persistence)

- 데이터를 생성한 프로그램이 종료되더라도 사라지지 않는 데이터의 특성
- 메모리 상의 데이터를 파일시스템, 관계형 데이터 베이스 혹은 객체 데이터베이스 등을 활용 하여 영구적으로 저장 하여 영속성을 부여함
- 데이터를 데이터베이스에 저장 하는 방법
  - JDBC
  - Spring JDBC
  - Persistence Framework (Hibernate, Mybatis ...)

02. ORM(Object Relational Mapping)

- 객체는 객체대로 설계 하고, 관계형 데이터 베이스는 관계형 데이터베이스로 설계
- ORM 프레임워크가 중간에 매핑
- 객체와 데이터 베이스의 데이터를 자동으로 매핑(연결)해주는 것
- 객체는 Class를 사용 하고 관계형 데이터베이스는 테이블 사용
- Persistence API (JPA, Hibernate ..)

03. Persistence Layer

- Persistence Layer : 데이터에 영속성을 부여해 주는 계층
- Persistence Framework : JDBC프로그램의 복잡함이나 번거로움 없이 간단한 작업만으로 데이터 베이스와 연동 하여 안정정성을 보장 (JPA, Hibernate, Mybatis ..)
  - SQL Mapper
  - ORM

SQL Mapper

- 데이터 베이스 객체를 자바 객체로 매핑함으로써 객체 간의 관계를 바탕으로 SQL을 자동 생성 해 주지만 SQL Mapper는 SQL를 명시 해 주어야 함
- SQL <- 매핑 -> Object 필드
- Mybatis, JdbcTempletes

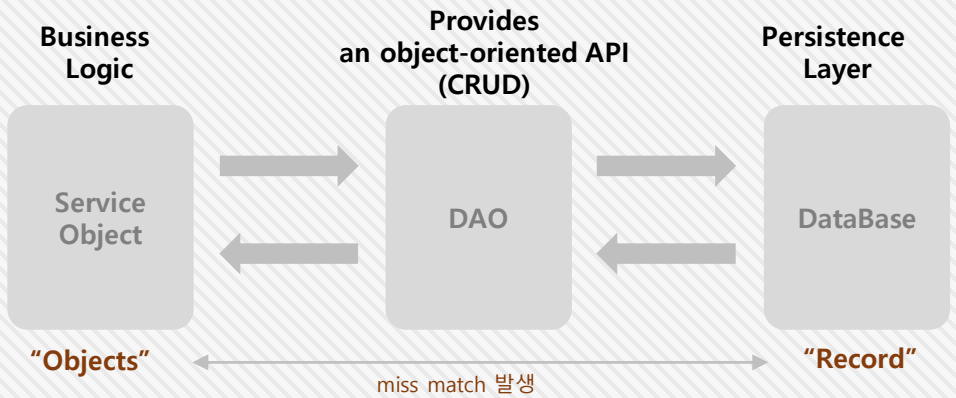
ORM

- 데이터 베이스 객체를 자바 객체로 매핑함으로써 객체 간의 관계를 바탕으로 SQL을 자동 생성 해서 자동으로 매핑(연결) 해 주는 것
- 데이터 베이스 데이터 <- 매핑 -> Object 필드
- SQL Query가 아닌 직관적인 코드(메서드)로 데이터 조작
- JPA, Hibernate

2. DAO, DTO, Entity 개념

DAO (Data Access Object)

- 실제로 DB에 접근하는 객체
- Service와 DB를 연결하는 고리 역할
- Object – SQL CRUD – DB Record



DTO (Data Transfer Object)

- 계층간 데이터 교환을 위한 객체(Java Beans)
- 로직을 가지고 있지 않는 순수한 데이터 객체, getter/setter 메소드만 가지고 있음
- Db에서 꺼낸 값을 임으로 변경할 필요가 없기 때문에 DTO Class에는 setter 없음 (대신 생성자에서 값을 할당)
- VO(Value Object)는 DTO와 동일한 개면 이지만 read only 속성을 갖음
  : VO는 특정한 비즈니스 값을 담는 객체, DTO는 Layer간의 통신 용도 객체

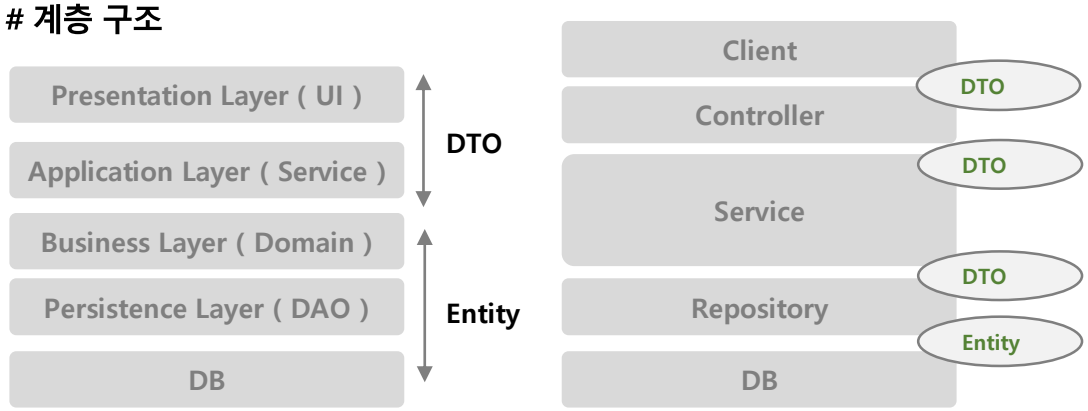
Entity Class – Domain Model

- 실제 DB의 테이블과 매칭될 Class, 즉 테이블과 링크될 Class
- 최대한 외부에서 Entity Class의 getter method를 사용 하지 않도록 해당 Class안에서 필요한 로직 Method 구현으로 Service Layer에서 사용

DTO 와 Entity 분리 이유

- View Layer와 DB Layer분리
- Entity Class의 변경은 여러 Class에 영향을 주는 반면에 View와 통신 하는 DTO는 요청과 응답에 따라서 변경이 되므로
- Domain Model에 Presentation을 위한 필드 추가 시 모델링의 순수성이 깨짐
- Presentation 요구사항들을 수용하기 위해서 여러 Domain Model 사용

계층 구조

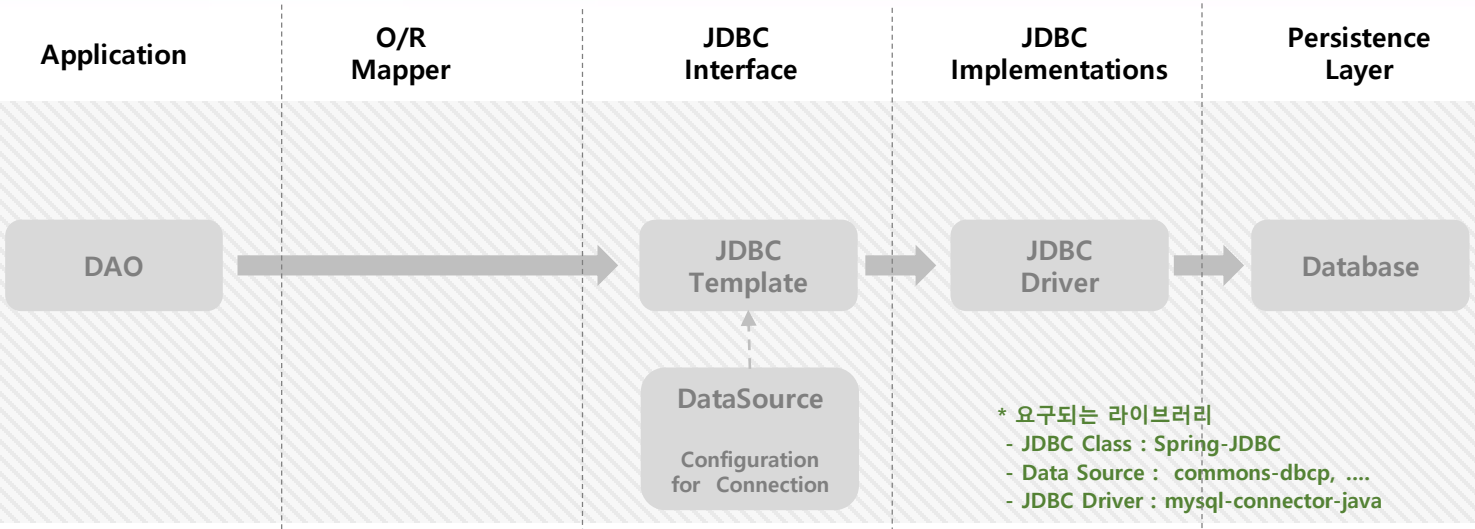


ABACUS

3. JDBC, JPA, MyBatis 차이점

01. JDBC (Java DataBase Connectivity)

- DB에 접근 할 수 있도록 JAVA에서 제공 하는 API
- 모든 JAVA의 Data Access 기술의 근간
- 데이터 베이스에서 CRUD



JAVA JDBC

1. db 연결
2. Statement 선언
3. insert, update, delete 수행
4. 조회
5. db 연결 해제

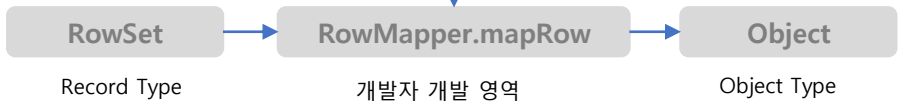
```
Connection connection = DriverManager.getConnection(URL, ID, PWD)
Statement statement = connection.createStatement()
statement.executeUpdate(쿼리String)
Result result = statement.executeQuery(쿼리실행)


result.next()를 이용해서 복수 건 처리
finally 에 자원 반납 처리
  - result.close()
  - statement.close()
  - connection.close()
commit, rollback 처리
```

Spring JDBC

1. jdbc template 선언
2. jdbc template 객체 생성
3. Jdbc api 실행
4. 조회 결과 처리

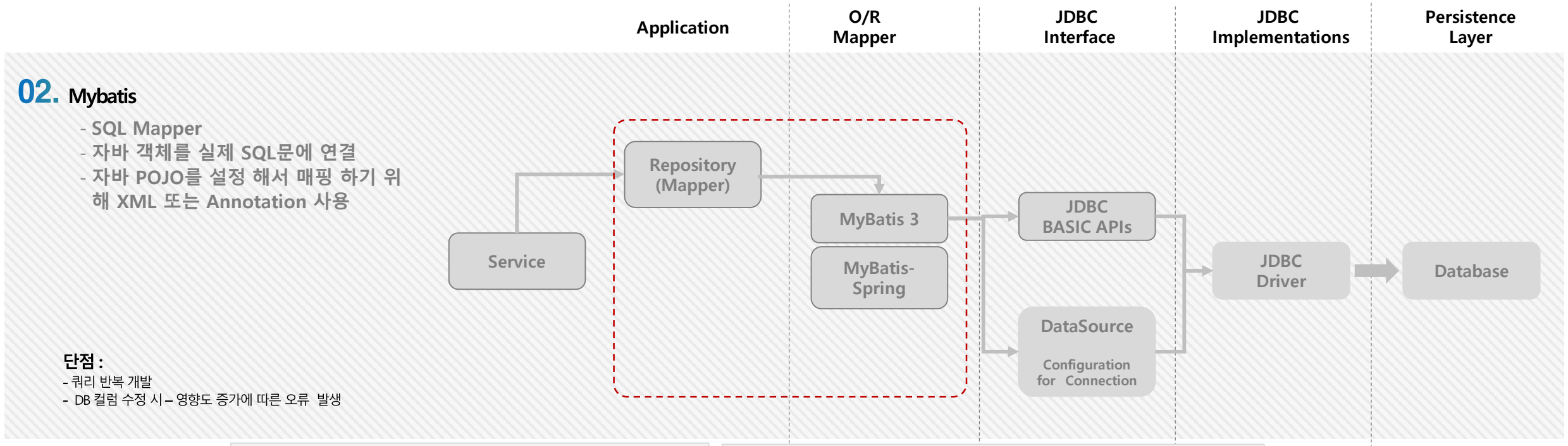
** 환경 설정
- Connection 설정 관리
- Configuration
  : DataSource 설정

```
private JdbcTemplate jdbcTemplate;

@Autowired
public void setDataSource(DataSource dataSource) {
   this.jdbcTemplate = new JdbcTemplate(datasource);
}

public List<Member> getMembers() {
    return jdbcTemplate.query("쿼리". new RowMapper<Member>() {
        public Member mapRow(ResultSet rs, int rowNum) throws SQLException {
            Member member = new Member();
            member.setXX(rs.getInt("id"));
             ....
            return member'
        }
    });
}
```



ABACUS

3. JDBC, JPA, MyBatis 차이점



02. Mybatis

- SQL Mapper
- 자바 객체를 실제 SQL문에 연결
- 자바 POJO를 설정 해서 매핑 하기 위해 XML 또는 Annotation 사용

단점 :

- 쿼리 반복 개발
- DB 컬럼 수정 시 – 영향도 증가에 따른 오류 발생

Spring Mybatis

1. 쿼리 작성
2. Mapper Class 작성
3. Service 작성

** 환경 설정
- Connection 설정 관리
- Mybatis 설정 관리
- Configuration
  : DataSource 설정

```
<mapper namespace="com.hhi.green.samp.dao.MemberDAO">
<!--  사용자 목록 조회  -->
 <select id="selectMember"
          parameterType="com.hhi.green.samp.dto.MemberDTO"
         resultType="com.hhi.green.samp.dto.MemberDTO">
    SELECT SEQ, ...
     FROM MEMBER
    WHERE SEQ = #{seq}
 </select>
.....
```

```
@Mapper
public interface MemberDAO {
   public MemberDTO selectMember(MemberDTO memberDto);
   public void insertMember(MemberDTO memberDto);
   public void deleteMember(Integer seq);
   public void updateMember(MemberDTO memberDto);
   public List<MemberDTO> selectMembers();
}
```

```
public MemberDTO selectMember(MemberDTO memberDto) throws BizException {
   return memberDAO.selectMember(memberDto);
}
```

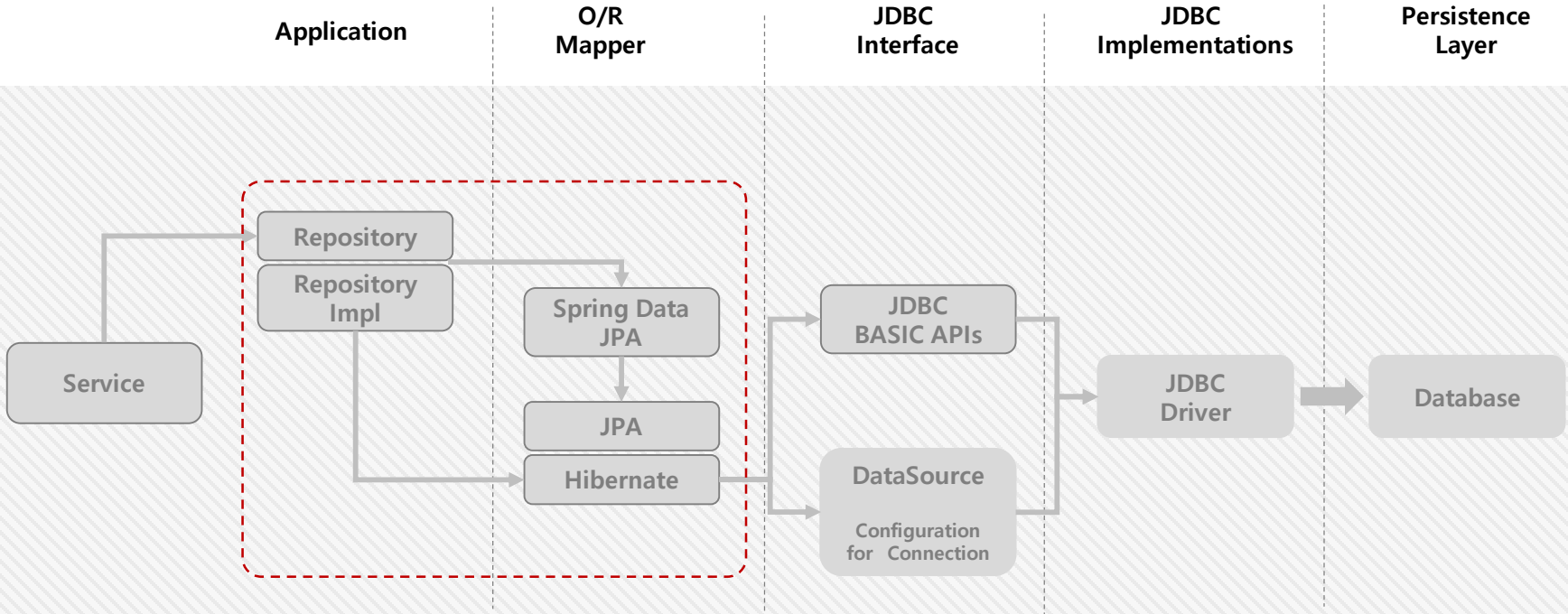
```
mybatis:
  mapper-locations: classpath*:mapper/**/*.xml
  configuration:
    log-prefix: hcfwJdbcLog.
```

```
datasource:
      #url: ENC(qsMZcX3awr8F/8mR4gSOM4HRpK5cAPR0r73YATmOiPeId77adM9h15aQAvOiX0Fb)
      #username: ENC(++870+xVkBmy675wHhUJSNMK3+Ctix0YlAcDkmnA0vzjESx1S73Xkw==)
      #password: ENC(LGC3fp4TgEjNFvDHK6ukjHJz5cPB9Pf5SzNFpgafPvk=)
      #driver-class-name: org.h2.Driver
```

3. JDBC, JPA, MyBatis 차이점

03. JPA

- 자바 ORM 기술에 대한 표준 API 명세, JAVA에서 제공 하는 API
- ORM을 사용하기 위한 표준 인터페이스를 모아둔 것
- 기존 EJB에서 제공되던 Entity Bean을 대체하는 기술
- 구성 요소
  - javax.persistence 페키지로 정의된 API
  - JPOL(Java Persistence Query Language)
  - 객체/관계 메타 데이터



 # 장점

- 객체 지형 개발이 가능 하며, 비즈니스 로직에 집중 할 수 있음
- 생산성, 유지 보수 용이
- 테이블 생성, 변경, 관리 쉬움
- 빠른 개발

 # 단점

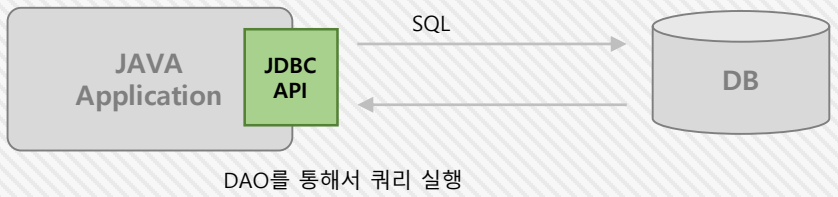
- 배우기 어렵다.
- 잘 이해하고 사용하지 않으면 데이터 손실 발생 (Persistence context)
- 잘 이해하고 사용자지 않으면 성능상 문제 발생

4. JPA

- JPA는 Application과 JDBC사이에서 JDBC API를 사용하여 SQL을 호출 하여 DB와 통신 한다.

01. SQL Base

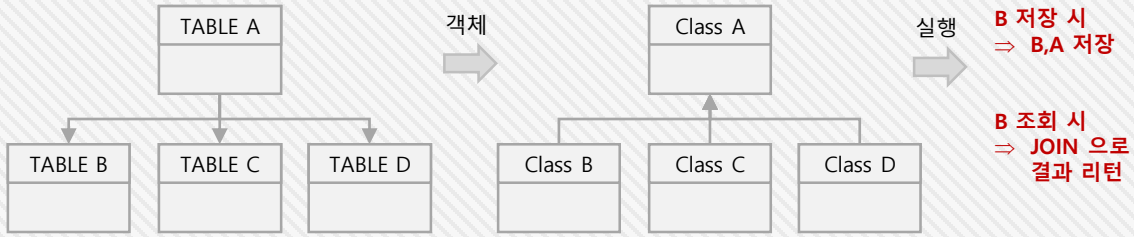
- JAVA Application에서 JDBC API를 사용하여 쿼리 실행



DAO를 통해서 쿼리 실행

02. 객체 관계

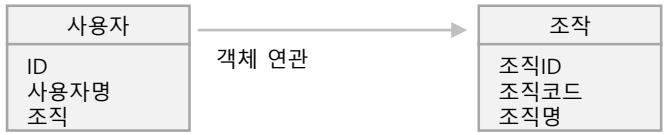
- 객체 관계 : 상속에 의해서 Bean Binding를 통한 쿼리 실행



DB 관계

사용자 JOIN 조직
조직 JOIN 사용자

JAVA 객체 관계 – 연관 방향에 따라서 참조됨



조직 = 사용자.get조직()

ABACUS

5. Persistence Context

- JPA는 Application과 Database사이에 영속성 컨텍스트(Persistence Context)라는 개념을 두고 관리 한다.

01. EntityManagerFactory

- EntityManager 인스턴스 관리
- EntityManagerFactory 생성시 커넥션 풀 생성
- 같은 EntityManagerFactory를 통해 EntityManager는 같은 Database에 접속
- 한 번 생성 후 Application 전체에 공유

02. EntityManager

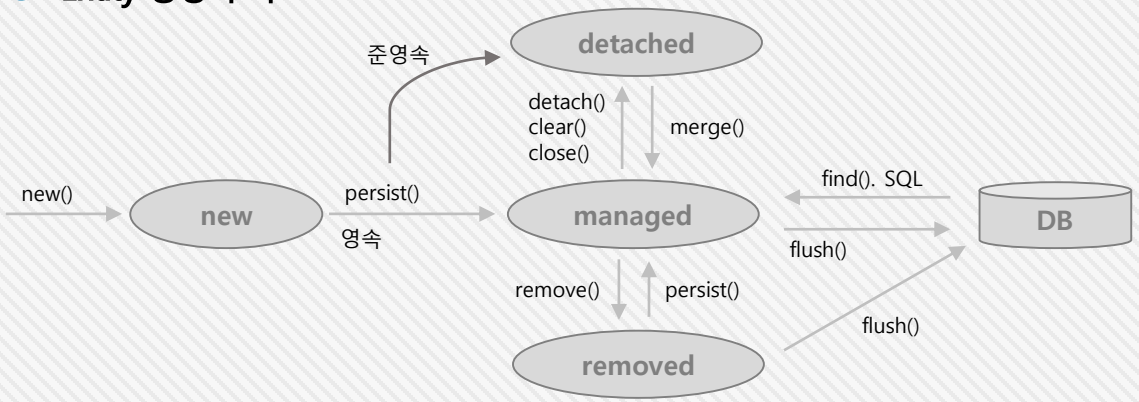
- 엔티티를 저장, 수정, 삭제, 조회 등 엔티티와 관련된 작업 수행

: em.find() : 엔티티 조회
: em.persist() : 저장
: em.remove() : 삭제
: em.flush() : 연속성 컨텍스트 내용을 db에 저장
: em.detach() : 준연속성 객체로 전환
: em. () : 연속 상태로 전환
: em.clear() : 영속성 초기화
: em.close() : 연속성 종료

```
@Repository
public class MyRepository {
   @PersistenceUnit
   EntityManagerFactory  emf;
}
```

@PersistenceUnit,
@PersistenceContext

03. Entity 생명 주기



1. New (비영속 객체)
   : Entity 객체가 DB에 반영 되지 않음, new 키워드를 사용해 생성한 Entity 객체

2. Managed (영속객체)
   : persist 메소드를 이용해 저장 한 경우와 db에서 조회한 경우로 Persistence Context가 관리 하는 상태로 해당 객체를 수정 했는지(자동 변경 감지)알아 냄

3. Removed (삭제 객체)
   : Managed 상태인 객체를 remove method를 이용해 삭제 한 경우 removed상태로 작업 단위가 종료되는 시점에 실제로 db 삭제

4. Detached(준 영속 객체)
   : 트랜잭션이 commit되었거나, clear, flush Method가 실행된 경우 Managed상태의 객체는 모두 Detached 상태가 됨, 이 상태에서는 db동기화를 보장 못함, merge() Method로 Managed 상태로 전환 필요

5. Persistence Context

- 논리적인 개념으로 눈에 보이지 않으며, "Entity를 영구 저장 하는 환경" 이라는 뜻으로 persist() 시점에 Entity를 Persistence Context에 저장 하는 것
- 버퍼링, 캐싱등으로 동일성(Identity), 쓰지 지연(Transactional Write-Behind), 변경감지(Dirty Checking)이 이점이 있음

01. new (비영속)

```
Member member = new Member();
member.setId("member1");
Member.setUserName("김길동");
```

02. managed (영속)

```
Member member = new Member();
member.setId("member1");
Member.setUserName("김길동");

EntityManager entityManager = entityManagerFactory,createEntityManager();
entityManager.getTransaction().begin();

// 객체를 저장한 상태 ( 영속 ) : db에 저장 하지 않음
entityManager.persist(member)
```

** 실제 저장을 위해서는 transaction.commit() 이 필요함

03. detached (준영속)

```
Member member = new Member();
member.setId("member1");
Member.setUserName("김길동");

EntityManager entityManager = entityManagerFactory,createEntityManager();
entityManager.getTransaction().begin();

entityManager.persist(member)

// 준영속 상태로 분리
entityManager.detach(member)
```

04. removed (삭제)

```
Member member = new Member();
member.setId("member1");
Member.setUserName("김길동");

EntityManager entityManager = entityManagerFactory,createEntityManager();
entityManager.getTransaction().begin();

entityManager.persist(member)

// 실제 db 삭제를 요청한 상태
entityManager.remove(member)
```

6. JPA 구조

- JPA는 Application과 JDBC사이에서 JDBC API를 사용하여 SQL을 호출 하여 DB와 통신 한다.

01. 쓰기

- 개발자가 DAO에서 객체를 저장
- JPA
  1) Entity 분석
  2) Insert SQL 생성
  3) JDBC API를 사용하여 SQL을 DB에서 실행

02. 조회

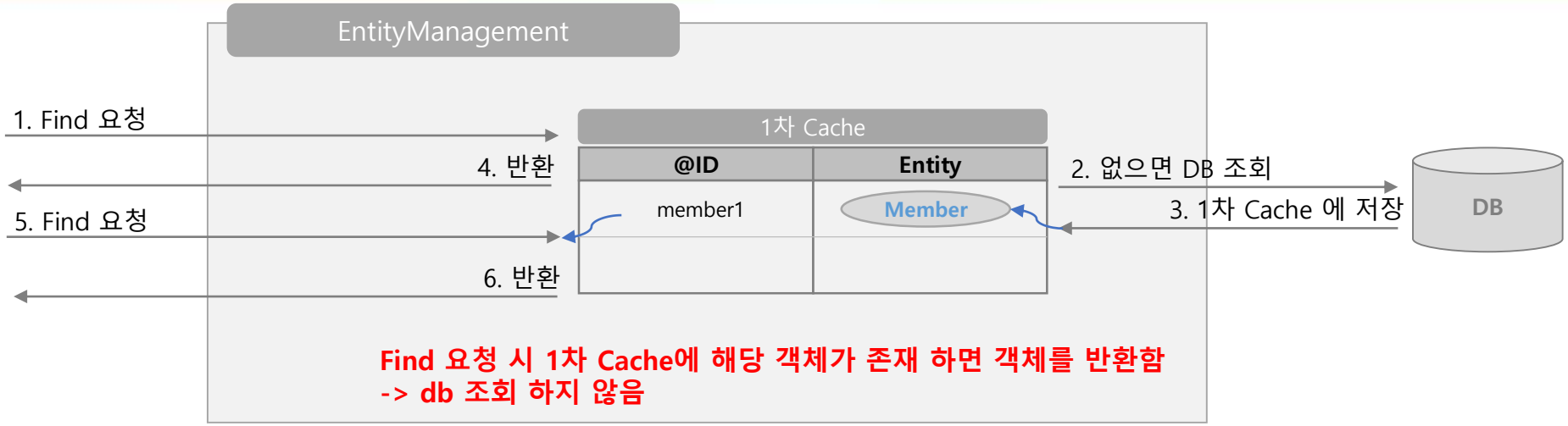
- 개발자가 객체의 PK 값을 JPA에 넘김
- JPA
  1) Entity 매핑 정보를 바탕으로 적절한 SELECT SQL 생성
  2) JDBC API를 사용하여 SQL을 DB에서 실행
  3) DB로 부터 결과를 받음
  4) 결과를 객체에 매핑

```java
@PersistenceContext
EntityManager entityManager;

@Override
@Transactional
public void run(ApplicationArguments args) throws Exception {

    Member member = new Member();
    member.setName("홍길동");
    entityManager.persist(member);
    entityManager.flush();
    entityManager.clear();

    logger.debug("Member : " + member.toString());
    Integer y = 1;
    Member member1 = entityManager.find(Member.class,y.longValue() );
    logger.debug("Member1 : " + member1.toString());
    Member member2 = entityManager.find(Member.class,y.longValue() );
    logger.debug("Member1 : " + member2.toString());
}
```

** 수정 및 삭제는 별도의 API가 존재 하지 않으며 commit이 발생하면 적용됨 **

```
Hibernate: call next value for hibernate_sequence
Hibernate: insert into member (name, id) values (?, ?)
[DEBUG] 2020-06-30 13:43:20.105 [restartedMain] JpaRunner - Member : Member(id=1, name=홍길동)
Hibernate: select member0_.id as id1_0_0_, member0_.name as name2_0_0_ from member member0_ where member0_.id=?
[DEBUG] 2020-06-30 13:43:20.122 [restartedMain] JpaRunner - Member1 : Member(id=1, name=홍길동)
[DEBUG] 2020-06-30 13:43:20.122 [restartedMain] JpaRunner - Member1 : Member(id=1, name=홍길동)
```

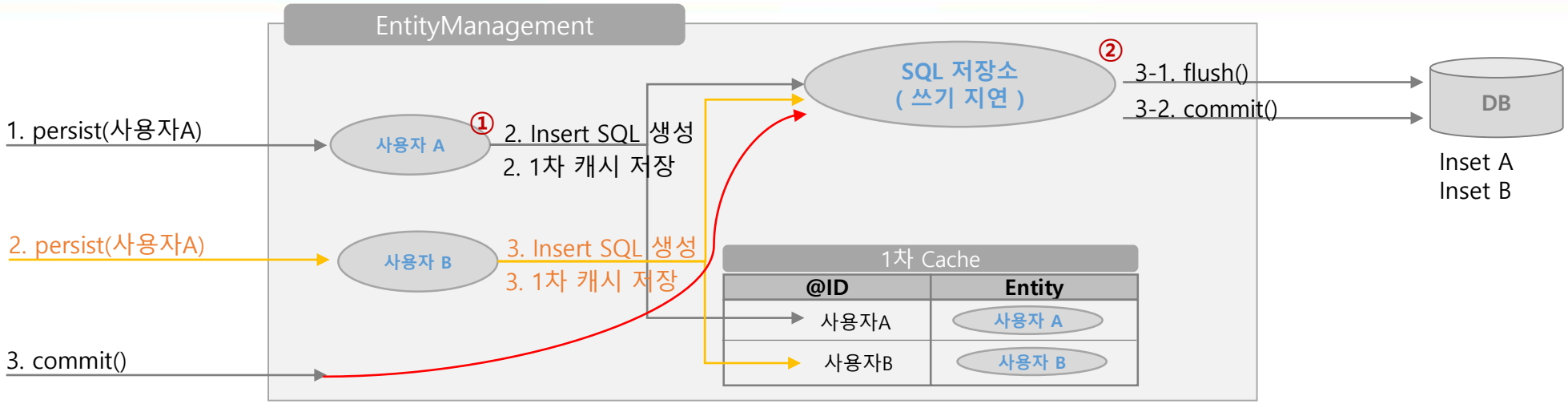
7. JPA 조회




```java
@PersistenceContext
EntityManager entityManager;

@Override
@Transactional
public void run(ApplicationArguments args) throws Exception {

    Member member = new Member();
    member.setName("홍길동");
    entityManager.persist(member);
    entityManager.flush();
    entityManager.clear();

    logger.debug("Member : " + member.toString());
    Integer y = 1;
    Member member1 = entityManager.find(Member.class,y.longValue() );
    logger.debug("Member1 : " + member1.toString());
    Member member2 = entityManager.find(Member.class,y.longValue() );
    logger.debug("Member1 : " + member2.toString());
}
```

```
Hibernate: call next value for hibernate_sequence
Hibernate: insert into member (name, id) values (?, ?)
[DEBUG] 2020-06-30 13:43:20.105 [restartedMain] JpaRunner - Member : Member(id=1, name=홍길동)
Hibernate: select member0_.id as id1_0_0_, member0_.name as name2_0_0_ from member member0_ where member0_.id=?
[DEBUG] 2020-06-30 13:43:20.122 [restartedMain] JpaRunner - Member1 : Member(id=1, name=홍길동)
[DEBUG] 2020-06-30 13:43:20.122 [restartedMain] JpaRunner - Member1 : Member(id=1, name=홍길동)
```

7. JPA 저장

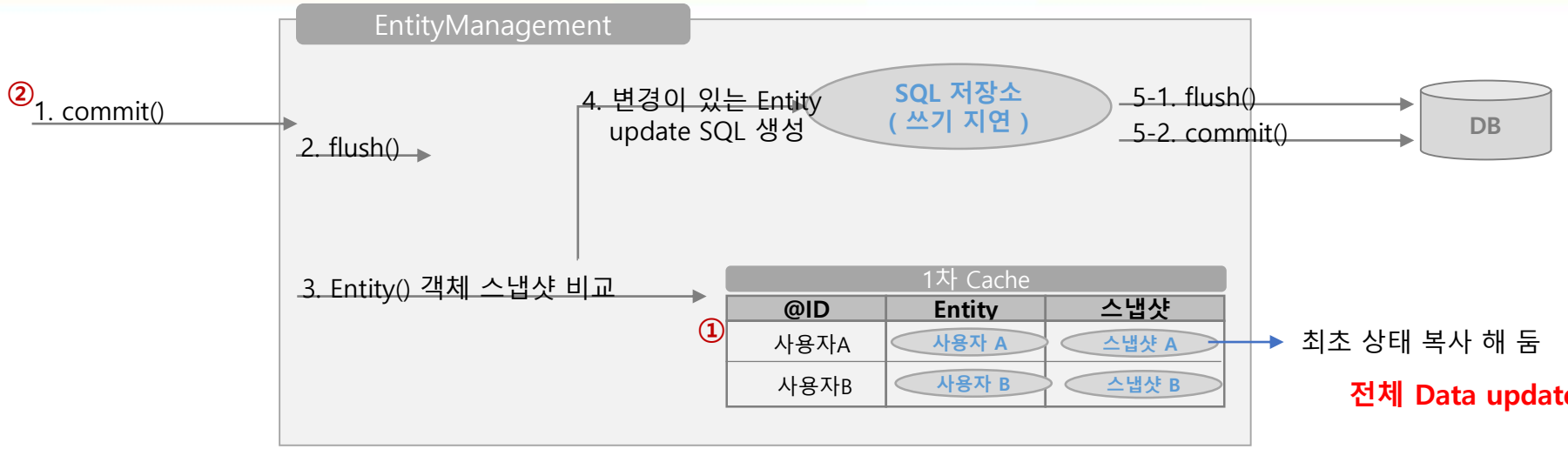



```
Member memberA = new Member();
memberA.setName("홍길동");

Member memberB = new Member();
memberB.setName("김길동");

①
entityManager.persist(memberA);  // 연속성 컨텍스트 저장
entityManager.persist(memberB);  // 연속성 컨텍스트 저장

②
entityManager.flush();

Integer y = 1;
Member member1 = entityManager.find(Member.class,y.longValue() );
logger.debug("Member1 : " + member1.toString());
memberA.setName("홍길동1");

entityManager.flush();
entityManager.clear();

Member member2 = entityManager.find(Member.class,y.longValue() );
logger.debug("Member2 : " + member2.toString());
```

②

```
Hibernate: call next value for hibernate_sequence
Hibernate: call next value for hibernate_sequence
Hibernate: insert into member (name, id) values (?, ?)
[TRACE] 2020-06-30 15:29:48.047 [restartedMain] BasicBinder - binding parameter [1] as [VARCHAR] - [홍길동]
[TRACE] 2020-06-30 15:29:48.048 [restartedMain] BasicBinder - binding parameter [2] as [BIGINT] - [1]
Hibernate: insert into member (name, id) values (?, ?)
[TRACE] 2020-06-30 15:29:48.049 [restartedMain] BasicBinder - binding parameter [1] as [VARCHAR] - [김길동]
[TRACE] 2020-06-30 15:29:48.049 [restartedMain] BasicBinder - binding parameter [2] as [BIGINT] - [2]
[DEBUG] 2020-06-30 15:29:48.058 [restartedMain] JpaRunner - Member1 : Member(id=1, name=홍길동)
Hibernate: update member set name=? where id=?
[TRACE] 2020-06-30 15:29:48.059 [restartedMain] BasicBinder - binding parameter [1] as [VARCHAR] - [홍길동1]
[TRACE] 2020-06-30 15:29:48.059 [restartedMain] BasicBinder - binding parameter [2] as [BIGINT] - [1]
Hibernate: select member0_.id as id1_0_0_, member0_.name as name2_0_0_ from member member0_ where member0_.id=?
[TRACE] 2020-06-30 15:29:48.063 [restartedMain] BasicBinder - binding parameter [1] as [BIGINT] - [1]
[TRACE] 2020-06-30 15:29:48.067 [restartedMain] BasicExtractor - extracted value ([name2_0_0_] : [VARCHAR]) - [홍길동1]
[DEBUG] 2020-06-30 15:29:48.070 [restartedMain] JpaRunner - Member2 : Member(id=1, name=홍길동1)
```

7. JPA 수정



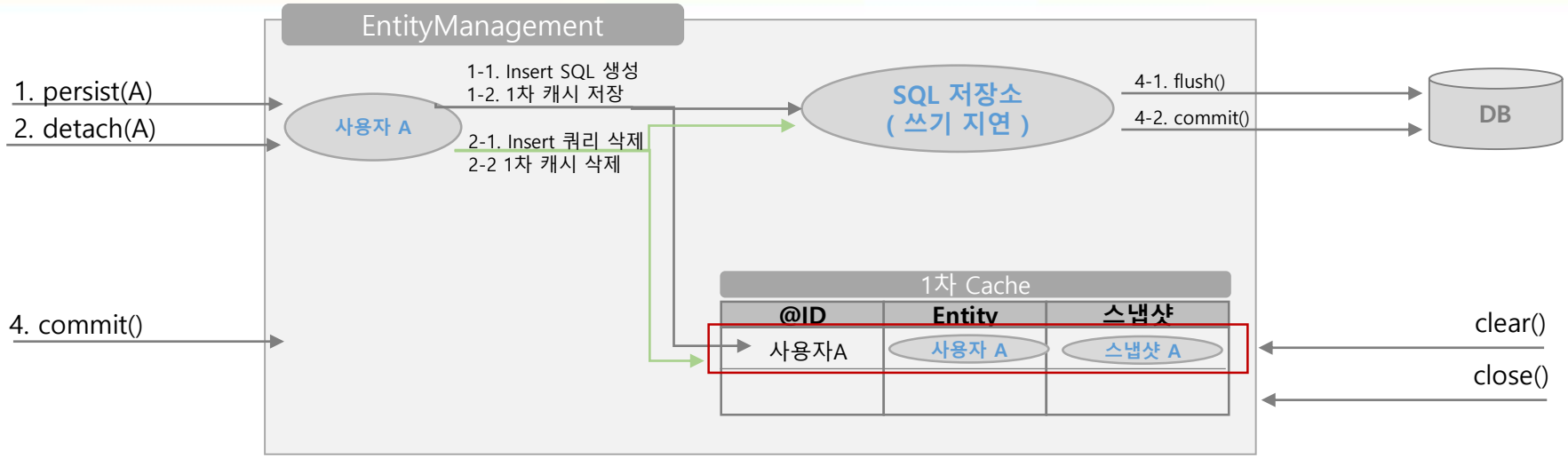
전체 Data update 발생 => 컬럼이 큰 경우
=> Dynamic update 사용

```
Member memberA = new Member();
memberA.setName("홍길동");

Member memberB = new Member();
memberB.setName("김길동");

entityManager.persist(memberA);  // 연속성 컨텍스트 저장
entityManager.persist(memberB);  // 연속성 컨텍스트 저장

entityManager.flush();

Integer y = 1;
Member member1 = entityManager.find(Member.class,y.longValue() );
logger.debug("Member1 : " + member1.toString());
memberA.setName("홍길동1");

entityManager.flush();
entityManager.clear();

Member member2 = entityManager.find(Member.class,y.longValue() );
logger.debug("Member2 : " + member2.toString());
```

① ②

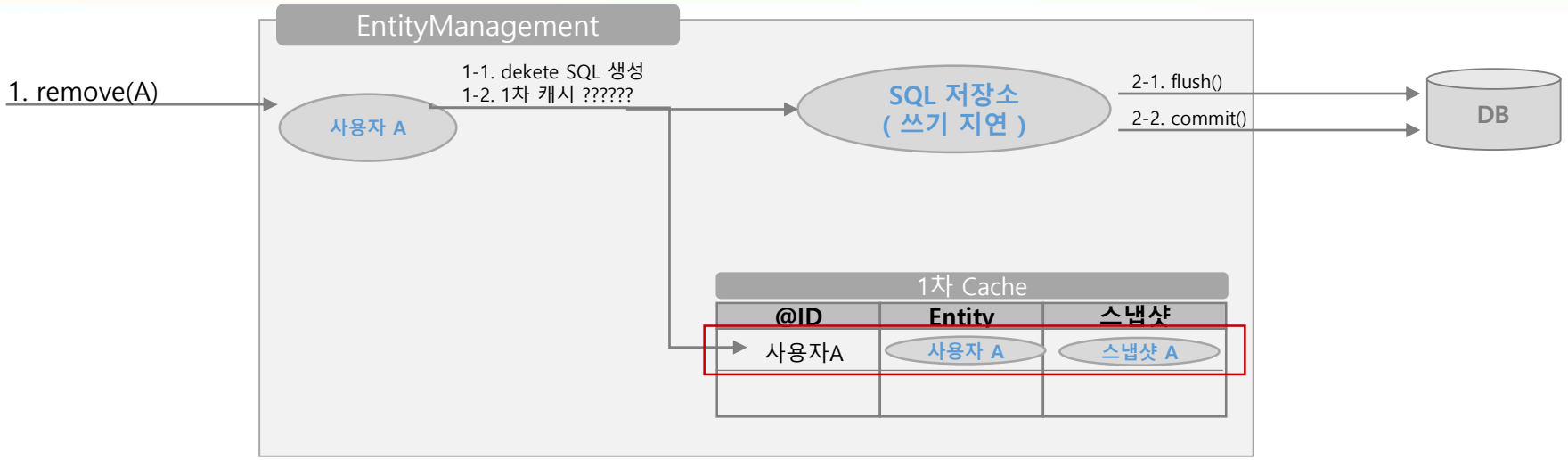
```
Hibernate: call next value for hibernate_sequence
Hibernate: call next value for hibernate_sequence
Hibernate: insert into member (name, id) values (?, ?)
[TRACE] 2020-06-30 15:29:48.047 [restartedMain] BasicBinder - binding parameter [1] as [VARCHAR] - [홍길동]
[TRACE] 2020-06-30 15:29:48.048 [restartedMain] BasicBinder - binding parameter [2] as [BIGINT] - [1]
Hibernate: insert into member (name, id) values (?, ?)
[TRACE] 2020-06-30 15:29:48.049 [restartedMain] BasicBinder - binding parameter [1] as [VARCHAR] - [김길동]
[TRACE] 2020-06-30 15:29:48.049 [restartedMain] BasicBinder - binding parameter [2] as [BIGINT] - [2]
① [DEBUG] 2020-06-30 15:29:48.058 [restartedMain] JpaRunner - Member1 : Member(id=1, name=홍길동)
② Hibernate: update member set name=? where id=?
[TRACE] 2020-06-30 15:29:48.059 [restartedMain] BasicBinder - binding parameter [1] as [VARCHAR] - [홍길동1]
[TRACE] 2020-06-30 15:29:48.059 [restartedMain] BasicBinder - binding parameter [2] as [BIGINT] - [1]
Hibernate: select member0_.id as id1_0_0_, member0_.name as name2_0_0_ from member member0_ where member0_.id=?
[TRACE] 2020-06-30 15:29:48.063 [restartedMain] BasicBinder - binding parameter [1] as [BIGINT] - [1]
[TRACE] 2020-06-30 15:29:48.067 [restartedMain] BasicExtractor - extracted value ([name2_0_0_] : [VARCHAR]) - [홍길동1]
[DEBUG] 2020-06-30 15:29:48.070 [restartedMain] JpaRunner - Member2 : Member(id=1, name=홍길동1)
```

7. JPA Detach




```
Member memberA = new Member();
memberA.setName("홍길동");

entityManager.persist(memberA);  // 연속성 컨텍스트 저장
entityManager.detach(memberA);
```

7. JPA Remove




```
Member memberA = new Member();
memberA.setName("홍길동");
entityManager.persist(memberA);  // 연속성 컨텍스트 저장
entityManager.flush();
entityManager.clear();

Integer y = 1;
Member memberB= entityManager.find(Member.class,y.longValue() );
logger.debug("memberB : " + memberB.toString());

entityManager.remove(memberB);
//memberB.setName("김길동");
//entityManager.persist(memberB);
entityManager.flush();
entityManager.clear();

memberB = entityManager.find(Member.class,y.longValue() );
if ( memberB == null ) {
    logger.debug("Member1 : 데이터 없음");
}
```

```
Hibernate: call next value for hibernate_sequence
Hibernate: insert into member (name, id) values (?, ?)
[TRACE] 2020-06-30 17:45:38.443 [restartedMain] BasicBinder - binding parameter [1] as [VARCHAR] - [홍길동]
[TRACE] 2020-06-30 17:45:38.443 [restartedMain] BasicBinder - binding parameter [2] as [BIGINT] - [1]
Hibernate: select member0_.id as id1_0_0_, member0_.name as name2_0_0_ from member member0_ where member0_.id=?
[TRACE] 2020-06-30 17:45:38.443 [restartedMain] BasicBinder - binding parameter [1] as [BIGINT] - [1]
[TRACE] 2020-06-30 17:45:38.458 [restartedMain] BasicExtractor - extracted value ([name2_0_0_] : [VARCHAR]) - [홍길동]
[DEBUG] 2020-06-30 17:45:38.458 [restartedMain] JpaRunner - memberB : Member(id=1, name=홍길동)
Hibernate: delete from member where id=?
[TRACE] 2020-06-30 17:45:38.458 [restartedMain] BasicBinder - binding parameter [1] as [BIGINT] - [1]
Hibernate: select member0_.id as id1_0_0_, member0_.name as name2_0_0_ from member member0_ where member0_.id=?
[TRACE] 2020-06-30 17:45:38.458 [restartedMain] BasicBinder - binding parameter [1] as [BIGINT] - [1]
[DEBUG] 2020-06-30 17:45:38.458 [restartedMain] JpaRunner - Member1 : 데이터 없음
```

8. JPA Entity 매핑

01. @Entity

- 테이블과 매핑할 Class
- Default : Class Name과 같은 보통 클래스와 같은 이름
- 변경할 때는 @Entity(name = "tableName")
- 기본 생성자 필수 (파라메터 없는 public or peotected)
- final 생성자, enum, interface, inner class 사용 금지
- 지정할 필드에 final 사용 금지

02. @Table

- 앤티티와 매핑할 Table 지정
- 생략시 엔티티이름을 테이블 이름으로 사용
- 속성
  : name : 매핑할 테이블 이름
  : catalog : catalog 기능이 있는 DB에서 catalog
  : schema : schema 기능이 있는 DB에서 schma
  : uniqueConstraints(DDL) : DDL 생성시 제약 조건

```
@Table(name="AAA",
       uniqueConstraints={@uniqueConstraints
           name = "AAA_UNIQE"
           column = { "id", "name" }}))
```

⇒ ALTER TABLE AAA ADD CONSTRAINT AAA_UNIQE UNIQE(ID, NAME)

03. @id

- 엔티티의 주키 (기본키 매핑)

04. @GenerateValue

```
@GenerateValue(strategy = GenerationType.IDENTITY)
```

- 엔티티의 주키 (기본키 매핑)
- 생성 전략과 생성기를 설정 할 수 있음, 기본 전력은 AUTO

05. @Column

- 맴버 변수와 DB Table의 컬럼에 매핑
- 속성
  : name : 객체명과 DB 컬럼명을 다르게 하고 싶은 경우, DB 컬럼명으로 설정할 이름을 name 속성
  : insertable : insert 여부 (true/false)
  : updatable : 컬럼을 수정했을 때 DB에 추가를 할 것인지 여부 (true/false), @Column(updatable = false)
  : nullable : @Column(nullable = false): NOT NULL 제약조건
  : unique : unique index (잘 사용 하지 않음) -> @Table 사용
  : columnDefinition : @Column(columnDefinition = "varchar(100) default 'EMPTY'"): 직접 컬럼 정보를 작성
  : length : 문자 길이 제약 조건, String Type에서만 사용
  : precision : 숫자가 엄청 큰 BigDecimal 의 경우 (Ex.private BigDecimal age;)

06. @Temporal

- 날짜 Type (java.util.Date, java.util.Calendar) 매핑
- 날짜 타입 사용 시 해당 annotation을 사용
- TemporalType enum class에는 세 가지가 존재.
  : DATE: 날짜, DB의 date 타입과 매핑, Ex. 2019-08-13
  : TIME: 시간, DB의 time 타입과 매핑, Ex. 14:20:38
  : TIMESTAMP: 날짜와 시간, DB의 timestamp 타입과 매핑, Ex. 2019-08-13 14:20:
- java 8의 LocalDate(date), LocalDateTime(timestamp) 을 사용할 때는 생략이 가능

07. @Transient

- 특정 필드를 컬럼에 매핑하지 않음.
- DB에 관계없이 메모리에서만 사용하고자 하는 객체에 해당 annotation을 사용
- 해당 annotation이 붙은 필드는 DB에 저장, 조회가 되지 않는다.

08. @Enumerated

- Enum Type 매핑, Enum 객체 사용 시 해당 annotation을 사용
- 속성
  value : EnumType.ORDINAL : enum 순서를 데이터베이스에 저장 (기본값),
  사용하지 않는다. (추가 , 수정 시 문제 발생 할 수 있음)
  EnumType.String : enum 이름을 데이터베이스에 저장

08. @Lob

- DB에서 varchar를 넘어서는 큰 내용을 넣고 싶은 경우에 해당 annotation을 사용 - @Lob에는 지정할 수 있는 속성이 없다.

매핑하는 필드의 타입에 따라 DB의 BLOB, CLOB과 매핑
필드의 타입이 문자열: CLOB -> String, char[], java.sql.CLOB
그 외 나머지: BLOB -> byte[], java.sql.BLOB

9. 기본 키 생성 전략

01. 기본 키 매핑 : AUTO

- 직접 할당
  : @Id만 사용

- 자동 생성 전략
  : DENTITY : 데이터 베이스에 위임
  : SEQUENCE : 데이터 베이스의 시퀀스 사용
  : TABLE : 키 생성용 테이블 사용
  : AUTO : 기본 설정 값 @GeneratedValue(strategy = GenerationType.AUTO)

02. DENTITY

- 본 키 생성을 데이터베이스에 위임
- @GeneratedValue(strategy = GenerationType.IDENTITY)
- MySQL, PostgreSQL, SQL Server DB2
- 커밋 시점에 Insert SQL 실행 한 이후에 얻을 수 있음
- 예외적으로 .persist()가 호출되는 시점에 db insert 발생
- 모아서 insert 하지 못함

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

03. SEQUENCE

- 데이터베이스 Sequence Object를 사용
- @GeneratedValue(strategy = GenerationType.SEQUENCE)
- Oracle, PostgreSQL, DB2, H2
- 테이블 마다 시퀀스 오브젝트를 따로 관리
  : @SequenceGenerator에 sequenceName 속성을 추가

```
@Entity
@SequenceGenerator(  name = "MEMBER_SEQ_GENERATOR",  // 식별자 생성기 이름
                     sequenceName = "MEMBER_SEQ",     // 매핑할 데이터베이스 시퀀스 이름
                     initialValue = 1,                // DDL 생성 시에만 사용됨
                     allocationSize = 1)              //시퀀스 한 번 호출에 증가하는 수 (성능 최적화에 사용)
public class Member {
    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE,
                    generator = "MEMBER_SEQ_GENERATOR")
     private Long id;
}
```

04. TABLE

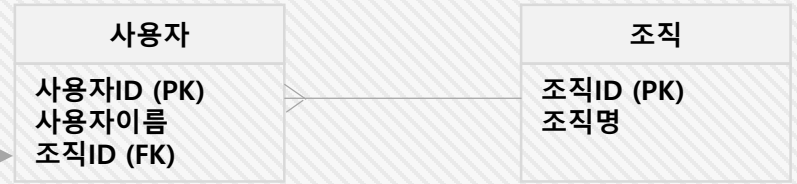
- 키 생성 전용 테이블을 하나 만들어서 데이터베이스 시퀀스
- @GeneratedValue(strategy = GenerationType.TABLE)
- 테이블 마다 시퀀스 오브젝트를 따로 관리
  : @TableGenerator 필요
- 모든 데이터 베이스 적용 가능 하나 성능상의 이슈 있음 -> 운영에 적합하지 않다

```
@Entity
@TableGenerator (  name = "MEMBER_SEQ_GENERATOR",
                   table = "MY_SEQUENCES", // 데이터베이스 이름
                   pkColumnValue = "MEMBER_SEQ",
                   allocationSize = 1)
public class Member {
    @Id
    @GeneratedValue(strategy = GenerationType.TABLE,
                    generator = "MEMBER_SEQ_GENERATOR")
    private Long id;
}
```

10. 단방향 연관

DB

JOIN을 통해서 양방향 관계
외래키를 통해서 연관



1. 여러 사용자가 하나의 조직을 가지고 있음
   : 사용자 입장에서는 다대일
   : @ManyToOne
   : @JoinColumn(name="조직ID")

1. 하나의 조직이 여러 사용자를 가지고 있음
   : 조직 입장에서는 일대다

연관관계매핑

객체 관계

1. 사용자는 조직을 없거나 하나의 조직을 가지고 있음

사용자는 조직을 소유 하지만 조직은 사용자를 소유 할 수 없다.
참조를 통한 연관

관계 설정 없이 개발

```
@Entity
public class 사용자 {
    private String 사용자ID;
    private String 사용자이름;
    private String 조직ID;
}
```

```
@Entity
public class 조직 {
    private String 조직ID;
    private String 조직이름;
}
```

```
@Service
public class 사용자조직 {
   public void 생성사용자조직() {
     사용자 user = new 사용자();
     조직 team = new 조직();
     team.set 조직이름("A팀");
     em.persist(team); <- 쿼리호출
     user.set조직ID(team.get조직ID()); <- 쿼리호출
     em.persist(user);
   }
}
```

```
@Service
public class 사용자조직 {
   public void 조회사용자조직() {
     사용자 user = em.find(사용자); <- 쿼리호출
     조직 team = em.find(조직, 사용자.get조직ID); <- 쿼리호출
    }
}
```

연관 관계 매핑 후 개발

```
@Entity
public class 사용자 {
    private String 사용자ID;
    private String 사용자이름;

    @ManyToOne
    @JoinColumn(name="조직ID")
    private 조직 조직;
}
```

```
@Entity
public class 조직 {

    @Id
    @Column(name="조직ID")
    private String 조직ID;
    private String 조직이름;
}
```

```
@Service
public class 사용자조직 {
   public void 생성사용자조직() {
     사용자 user = new 사용자();
     조직 team = new 조직();

     team.set 조직이름("A팀");
     user.set시용자이름("홍길동")
     user.set 조직(team)

     em.persist(user); <- 쿼리호출
   }
}
```

```
@Service
public class 사용자조직 {
   public void 조회사용자조직() {
       사용자 user = em.find(사용자); <- 쿼리호출
       조직 team = 사용자.get조직();
    }
}
```

10. 단방향 연관

01. @JoinColumn

- 조인컬럼은 외래키를 매핑
- name 속성에는 매핑할 외래키 이름을 지정
- 속성
  : name : 매핑할 외래 키 이름
- : 필드명 + _ + 참조하는 테이블의 기본키 컬럼명
  : refereced**C**olumnName : 외래키가 참조하는 대상 테이블의 컬럼명
  : 참조하는 테이블의 기본키 컬럼명
  : foreignKey : 외래키 제약조건을 직접 지정 가능
  : DDL 생성 때만 사용
  : unique, nullable, insertable, updatable, columnDefinition, table
  : @Column의 속성과 같다.

02. @ManyToOne

- 다대일()관계임을 명시.
- name 속성에는 매핑할 외래키 이름을 지정
- 속성
  : optional : false로 설정하면 연관된 엔티티가 항상 있어야 함
  : 기본값 : true
  : fetch : 글로벌 페치 전략 설정
  : FetchType.EAGER, FetchType.LAZY
  : cascade : 영속성 전이 기능 사용
  : targetEntity : 연관된 엔티티의 타입 정보를 설정. 거의 사용하지 않음

```
@Entity
public class 사용자 {
    private String 사용자ID;
    private String 사용자이름;
    @ManyToOne
    @JoinColumn(name="조직ID")
    private 조직 조직;
}
```

```
@Entity
public class 조직 {
    @Id
    @Column(name="조직ID")
    private String 조직ID;
    private String 조직이름;
}
```

```
@Service
public class 사용자조직 {
   public void 생성사용자조직() {
     사용자 user = new 사용자();
     조직 team = new 조직();

     team.set 조직이름("A팀");
     user.set사용자이름("홍길동")
     user.set 조직(team)

     em.persist(user); <- 쿼리호출
   }
}
```

```
@Service
public class 사용자조직 {
   public void 수정사용자조직() {
     조직 team = new 조직();

     team.set조직ID("ID1");
     team.set조직이름("A팀")

     em.persist(team) ;

     사용자 user = sm.find(사용자.class, "사용자id")
      user.set조직(team)  ;
   }
}
```

```
@Service
public class 사용자조직 {
   public void 연관제거사용자조직() {
       사용자 user = sm.find(사용자.class, "사용자id")
        user.set조직(null)  ;
   }
}
```

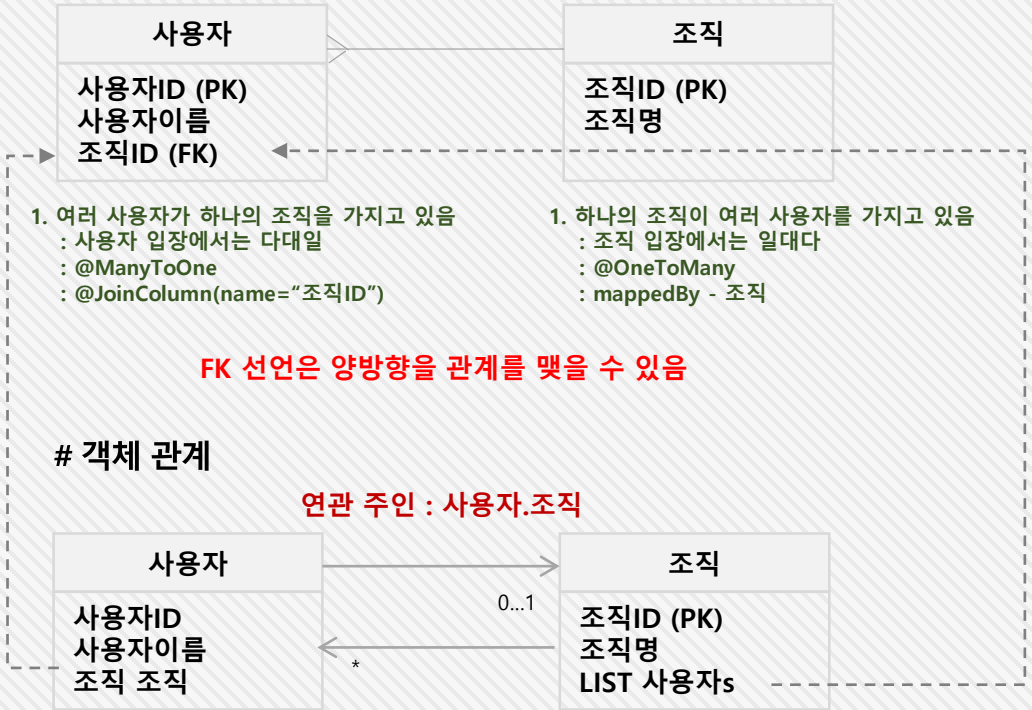
```
@Service
public class 사용자조직 {
   public void 삭제사용자조직() {

       사용자 user = sm.find(사용자.class, "사용자id")
       user.set조직(null)  ;
       em.remove(team);

   }
}
```

10. 양방향 연관

조직을 통해서 특정 조직에 속한 사용자 리스트 조회

JOIN을 통해서 양방향 관계
외래키를 통해서 연관

DB



FK 선언은 양방향을 관계를 맺을 수 있음

객체 관계

연관 주인 : 사용자.조직

1. 사용자 -> 조직 연관 관계 (단방향)
: 사용자.조직

객체는 양방향 없음

1. 조직 - > 사용자 연관관계 (단방향)
: 조직.사용자s

사용자의 조직값을 update 할 때 조직ID(FK)가 UPDATE ?
조직의 사용자s를 update 할 때 조직ID(FK)가 UPDATE ?

연관 관계의 주인

객체의 두 관계 중 하나를 주인(Owner)로 지정

주인만이 외래키를 관리(등록, 수정 삭제)할 수 있음.
주인이 아닌 쪽은 읽기만 가능

주인은 mappedBy가 없는 쪽 : JoinColumn 없음
주인은 @JoinColumn 사용

DB 에서 FK가 있는 곳이 주인 : 사용자,조직

값은 모두 넣어 준다.

01. @OneToMany

- 일대다(1:N) 매핑임을 명시.
- mappedBy = " 조직" : 양방향일 때 사용하는 반대쪽 매핑의 필드 이름

```
@Entity
public class 사용자 {
    private String 사용자ID;
    private String 사용자이름;
    @ManyToOne
    @JoinColumn(name="조직ID")
    private 조직 조직;
}
```

```
@Entity
public class 조직 {
    @Id
    @Column(name="조직ID")
    private String 조직ID;
    private String 조직이름;

    @oneToMany(mappedBy = "조직")
    private List<사용자> 사용자s = new ArrayList<사용자>{}
}
```

Content

III. JPA 연관관계

1. 연관 관계 매핑 시 고려 사항

- 엔티티들은 대부분 다른 엔티티와 연관 관계가 있으며 1:1, 1:N, N:1, N:N의 방향성을 가지고 있으면 연관 관계의 주인을 정해야 한다.

01. 다중성

- JPA에서는 어노테이션을 통해서 제공 하며 DB와 매핑을 하기 위해서 존재 하므로 데이터 베이스 관점에서 고민 해야 한다.
- 연관 관계
  : 다대일 (N : 1) - @ManyToOne
  : 일대다 (1 : N) - @OneToMany
  : 일대일 (1 : 1) - @OneToOne
  : 다대다 (N : N) - @ManyToMany

02. 객체는 단방향만 존재

- DB의 테이블은 외래키의 조인을 통해서 양방향성을 가진다.
- 객체는 단방향, 양방향 연관 관계를 설명 하지만 사실은 단방향이 두개 이다.
  : 가입은 여러 개의 상품을 가진다.
  : 상품을 구매한 가입자가 있다.

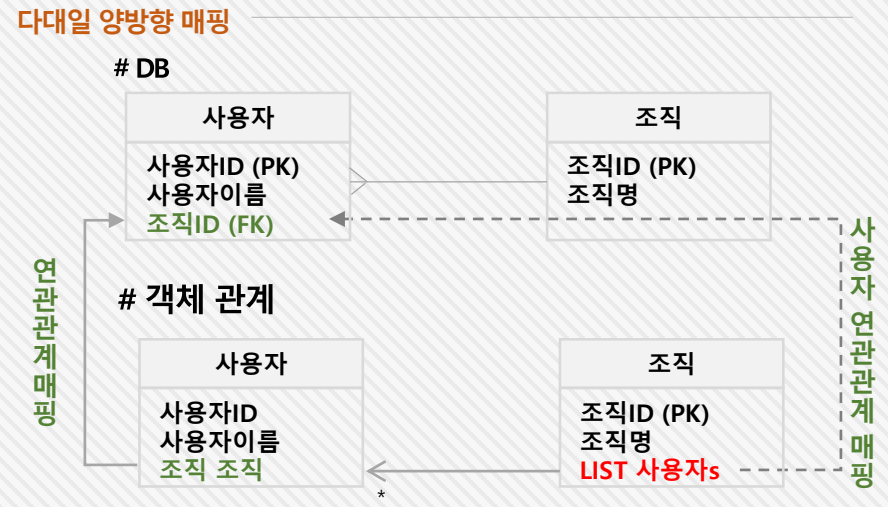
03. 연관 관계의 주인

- 객체의 양방향성은 A->B, B->A 처럼 2군데에서 일어 나므로 테이블의 외래키를 관리 할 객체를 선정 하는 것이 중요하다.
- 연관 관계의 주인은 외래키를 관리 하는 참조
- 주인 반대편은 외래키에 영향을 주지 않고 단순 읽기만 할 수 있다.

2. 다대일 [N : 1]

01. 다대일 [N : 1]

- 다(N)이 주인 객체
- DB 설계에서 N:1에서 N쪽에 외래키가 있어야 함.
- 객체로 전환 시 외래키가 있는 곳에 주키가 있는 객체를 참조 하면 된다.



01. 다대일 단방향 매핑

- 참조 객체에 @ManyToOne, @JoinColumn(name="외래키(참조키)" 사용

```
@Entity
public class 사용자 {
    private String 사용자ID;
    private String 사용자이름;

    @ManyToOne
    @JoinColumn(name="조직ID")
    private 조직 조직;
}
```

01. 다대일 양방향 매핑

- 다대일 단방향을 매핑 하고 반대쪽에서 단방향 매핑을 함
- 반대쪽 매핑은 복수 건은 가질 수 있으므로 Collection 객체를 추가 함.
- DB에는 영향을 주지 않음 (다대일 단방향 설정으로 주인이 결정 되어 있음)
- @OneToMany 사용, mappeby를 꼭 넣어 주어여 함 (조직 객체의 사용자s 맴버필드가 사용자 객체의 조직맴버 필드와 매핑이 됨)

```
@Entity
public class 조직 {
    @Id
    @Column(name="조직ID")
    private String 조직ID;
    private String 조직이름;

    @oneToMany(mappedBy = "조직")
    private List<사용자> 사용자s = new ArrayList<사용자>()
}
```

2. 다대일 [N : 1]

1. Spring Boot Application 에서 엔티티 객체를 생성 하고 시작 하면 @Entity 객체를 스캔 하여 Table 생성

```java
@SpringBootApplication
public class JpaApplication {

    public static void main(String[] args) {
        SpringApplication.run(JpaApplication.class, args);
    }

}
```

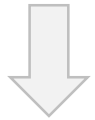
```java
@Entity
@Getter
@NoArgsConstructor
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private long userid;
    private String userNm;

    @ManyToOne(cascade={CascadeType.ALL})
    @JoinColumn(name="TEAM_ID")
    private Team team;

    @Builder
    public User(String userNm, Team team) {
        this.userNm = userNm;
        this.team = team;
    }

}
```

```java
@Entity
@Getter
@NoArgsConstructor
public class Team {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private long teamId;

    private String teamNm;

    @Builder
    public Team(String teamNm) {
        this.teamNm = teamNm;
    };

}
```

```java
public interface N1Repository extends JpaRepository<User, Long> {
}
```

application.yml

```yaml
spring:
  jpa:
    hibernate:
      ddl-auto: create
```



```
create sequence hibernate_sequence start with 1 increment by
1
Hibernate:
    create table team (
       team_id bigint not null,
        team_nm varchar(255),
        primary key (team_id)
    )
Hibernate:
    create table user (
       userid bigint not null,
        user_nm varchar(255),
        team_id bigint,
        primary key (userid)
    )
Hibernate:
    alter table user
       add constraint FKbmqm8c8m2aw1vgrij7h0od0ok
       foreign key (team_id)
       references team
```

2. 다대일 [N : 1]

2. Controller 생성 시점 초기 값 설정

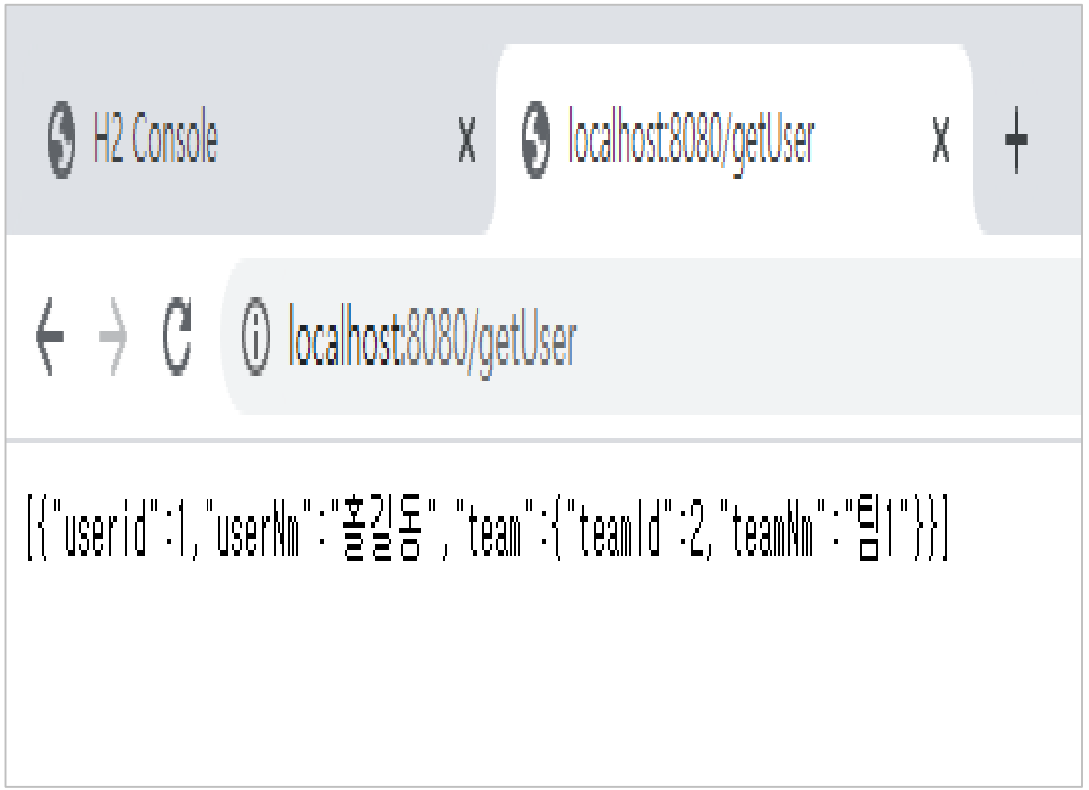
```java
@RestController
public class N1controller {

    @Autowired
    private N1Repository n1Repository;


    @PostConstruct
    @Transactional
    public void n1controller() {

        Team team = Team.builder()
                .teamNm("팀1")
                .build();

        User user = User.builder()
                .userNm("홀길동")
                .team(team)
                .build();

        n1Repository.save(user);

    }

    @GetMapping(value="/getUser")
    @ResponseBody
    public List<User> getUser() {
        return n1Repository.findAll();
    }
}
```

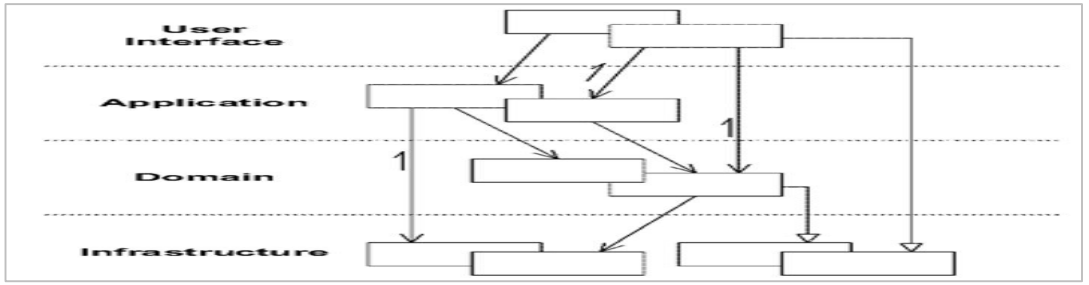
```
Hibernate:
    call next value for hibernate_sequence
Hibernate:
    call next value for hibernate_sequence
Hibernate:
    insert
    into
        team
        (team_nm, team_id)
    values
        (?, ?)
[TRACE] 2020-07-08 13:45:21.497 [restartedMain] BasicBinder - binding parameter [1] as [VARCHAR] - [팀1]
[TRACE] 2020-07-08 13:45:21.497 [restartedMain] BasicBinder - binding parameter [2] as [BIGINT] - [2]
Hibernate:
    insert
    into
        user
        (team_id, user_nm, userid)
    values
        (?, ?, ?)
[TRACE] 2020-07-08 13:45:21.497 [restartedMain] BasicBinder - binding parameter [1] as [BIGINT] - [2]
[TRACE] 2020-07-08 13:45:21.497 [restartedMain] BasicBinder - binding parameter [2] as [VARCHAR] - [홀길동]
[TRACE] 2020-07-08 13:45:21.497 [restartedMain] BasicBinder - binding parameter [3] as [BIGINT] - [1]
```

2. 다대일 [N : 1]

3. 결과 확인

```java
@RestController
public class N1controller {

    @Autowired
    private N1Repository n1Repository;


    @PostConstruct
    @Transactional
    public void n1controller() {

        Team team = Team.builder()
                .teamNm("팀1")
                .build();

        User user = User.builder()
                .userNm("홀길동")
                .team(team)
                .build();

        n1Repository.save(user);

    }

    @GetMapping(value="/getUser")
    @ResponseBody
    public List<User> getUser() {
        return n1Repository.findAll();
    }
}
```





ABACUS

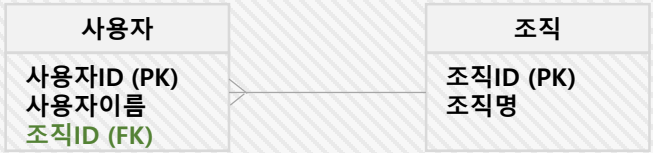
3. 일대다 [1 : N]

01. 일대다 [1 : N]

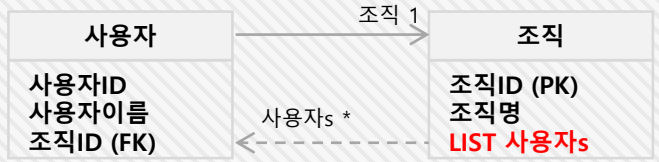
- 일(1)이 주인 객체
- 일 쪽에서 외래키 관리
- DB 설계에서 N:1에서 N쪽에 외래키가 있어야 함.
- 표준 스펙에서는 지원 하지만 권장하지 않음.

일대다 단방향 매핑

 # DB



 # 객체 관계



조직과 사용자는 일대다 관계
사용자입장에서는 팀을 참조 하지 않아도 된다는 설계가 객체 입장에서 나올 수 있음
조직의 사용자s가 변경 되면 db에 update 발생

01. 일대다 매핑

- @oneToMany, @JoinColumn(name="외래키(참조키)" 사용

```
@Entity
public class 사용자{
    private String 사용자ID;
    private String 사용자이름;
    private String 조직ID;
```

```
@Entity
public class 조작{
    private String 조작ID;
    private String 조직이름;

    @OneToMany
    @JoinColumn(name="조직ID")
    private List<사용자> 사용자s = new ArrayList<사용자>{}}
```

- 잘못 도메인 설계시 : 사용자 UPDATE 발생 : 성능 이슈
- @JoinColumn(name="조직ID", insertable = false, update=false)를 이용해서 update 방지

2. 일대다 [1 : N]

1. Application 생성 시점에 테이블 생성, 초기값 설정 값 : update 발생

```
Member member = new Member("홍길동");
member.addPhone(new Phone("010-9411-5479"));
member.addPhone(new Phone("010-8209-5479"));

memberRepository.save(member);
```

```
@Entity
public class Member {
  ...
@OneToMany(fetch=FetchType.EAGER, cascade =
CascadeType.ALL) // (1)
@JoinColumn(name="member_id")
private Collection<Phone> phone;


}
```

```
@Entity
public class Phone {
 @Column(name="member_id")
private int memberId;

}
```

```
    create table member (
       seq integer not null,
        name varchar(255),
        primary key (seq)
    )
Hibernate:

    create table phone (
       seq integer not null,
        member_id integer,
        no varchar(255),
        primary key (seq)
    )
```

```
...
    insert
    into
        member
        (name, seq)
    values
        (?, ?)
...
    insert
    into
        phone
        (member_id, no, seq)
    values
        (?, ?, ?)
...
    into
        phone
        (member_id, no, seq)
    values
        (?, ?, ?)
...
update
        phone
    set
        member_id=?
    where
        seq=?
....
Hibernate:
    update
        phone
    set
        member_id=?
    where
        seq=?
```

4. 일대다 [1 : 1]

01. 일대일 [1 : 1]

- 일(1)이 주인 객체로 외래키가 있는 곳
- DB에 유니크 제약 조건 필요

주 테이블 : 많이 접근 하는 테이블

주 테이블에 외래키
- 주 테이블만 조회 해도 대상 테이블 조회됨
- 주 테이블에 외래키에 null 허용
- DB 입장에서는 null 허용으로 잘못 된 설계

대상 테이블에 외래키
- 일대일 관계 에서 일대다 관계로 전환 용이
- 주 테이블에 많이 접근 하는데 양방향 설계를 해야 함
- 지연 로딩 사용 할 수 없음
- DB 입장에서는 바른 설계

일대일 주테이블 외래키 단방향 매핑

DB

객체 관계

일대일 주테이블 외래키 양방향 매핑

DB

객체 관계

01. 일대일 주 테이블 단방향 매핑 (사용자를 주테이블로)

- @OneToOne,
  @JoinColumn(name="외래키(참조키)" 사용

```
@Entity
public class 사용자{
    private String 사용자ID;
    private String 사용자이름;

    @OneToOne
    @JoinColumn(name="전화번호ID")
    private 전화번호 전화번호
```

```
@Entity
public class 전화번호{
    private String 전화번호ID;
    private String 전화번호;
```

02. 일대일 주 테이블 양방향 매핑 (사용자를 주테이블로)

- 주인 객체 : @OneToOne,
  @JoinColumn(name="외래키(참조키)" 사용
- 대상 객체 : @OneToOne(mappedby = "대상객체")
  => 읽기 전용

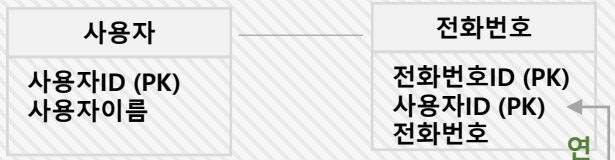
```
@Entity
public class 사용자{
    private String 사용자ID;
    private String 사용자이름;

    @OneToOne
    @JoinColumn(name="전화번호ID")
    private 전화번호 전화번호
```

```
@Entity
public class 전화번호{
    private String 전화번호ID;
    private String 전화번호;

    @OneToOne(mappedby = "전화번호")
    private 사용자 사용자
```

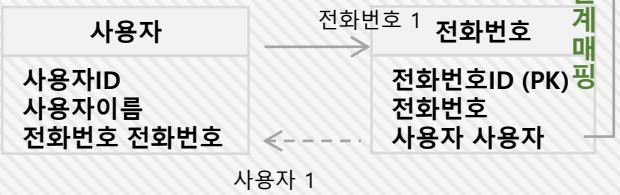
03. 일대일 대상 테이블 양방향 매핑

(사용자를 주테이블로)

DB



객체관계



5. 다대다 [N : N]

01. 다대다 [N : N]

- DB 입장에서는 존재 할 수 없음 : 연결 테이블 사용
- 객체 관계에서는 지원함
  : @ManyToMany, @JoinTable

 # DB

사용자	전화번호
사용자ID (PK) 사용자이름 전화번호ID (FK)	전화번호ID (PK) 전화번호 사용자ID (PK)

DB 해결 방안 처럼 객체 관계 해결

해결

사용자	사용자전화번호	전화번호
사용자ID (PK) 사용자이름	사용자ID (PK , FK) 전화번호ID (PK , FK)	전화번호ID (PK) 전화번호

 # 객체 관계

- 다대다 단방향

사용자	사용자전화번호	전화번호
사용자ID (PK) 사용자이름	사용자ID (PK , FK) 전화번호ID (PK , FK)	전화번호ID (PK) 전화번호

- 다대다 양방향

사용자	사용자전화번호	전화번호
사용자ID (PK) 사용자이름	사용자ID (PK, FK) 전화번호ID (PK , FK) 사용자 사용자 전화번호 전화번호	전화번호ID (PK) 전화번호

01. 다대다 단방향

- @ManyToMany, @JoinTable
- 사용자 전화번호 테이블 설정 되고 제약 조건 설정 됨

사용자	사용자전화번호	전화번호
사용자ID (PK) 사용자이름 List<전화번호> ..	사용자ID (PK , FK) 전화번호ID (PK , FK)	전화번호ID (PK) 전화번호

```
@Entity
public class 사용자{
    private String 사용자ID;
    private String 사용자이름;

    @ManyToMany
    @JoinTable(name="사용자전화번호")
    private List<전화번호> 전화번호s
```

02. 다대다 양방향

- @ManyToMany(mappedBy="대상객체") 설정
- 사용자 전화번호 테이블 설정 되고 제약 조건 설정 됨

사용자	사용자전화번호	전화번호
사용자ID (PK) 사용자이름 List<전화번호> ..	사용자ID (PK, FK) 전화번호ID (PK , FK) 사용자 사용자 전화번호 전화번호	전화번호ID (PK) 전화번호 List< 사용자 > ..

```
@Entity
public class 전화번호{
    private String 사용자ID;
    private String 사용자이름;

    @ManyToMany(mappedBy="전화번호")
    private List< 사용자 > 사용자s
```

03. 한계

- 중간 테이블이 생성 됨
- 중간 테이블에 다른 정보 삽입 불가
- 잘못된 쿼리 수행
- 사용 하면 안됨

극복

극복

- 중간 테이블을 엔티티로 승격
- 다대일 관계, 일대다 관계로 해결

6. 상속 관계 매핑

01. 상속 관계 매핑

- DB의 슈퍼타입 서브타입 논리 모델

 # DB 논리 모델

• 전략 :
@Inheritance(strategy= InheritanceType).
- default 전략은 SINGLE_TABLE.
- InheritanceType 종류
: JOINED
: SINGLE_TABLE
: TABLE_PER_CLASS
* 하위 구분
@DiscriminatorColumn(name="DTYPE")
* 구분값에 들어갈 값
@DiscriminatorValue("XXX")

 # 객체 상속 모델

02. 조인 테이블

- 조인 전략으로 테이블

 # DB 물리 모델

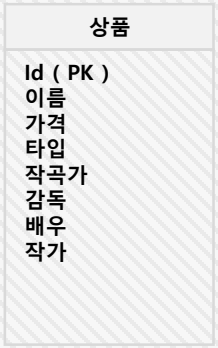

```
@Entity
@Inheritance(strategy= InheritanceType.JOINED)
@DiscriminatorColumn(name="DTYPE")
public abstract class 상품 {
  @Id
  @GeneratedValue
  private int id

}

@Entity
@DiscriminatorValue("A")  {
public class 앨범 extends 상품 {
   private String 작곡가
}

@Entity
@DiscriminatorValue("B")  {
public class 영하 extends 상품 {
    private String 감독;
    private String  배우
}
```

조회 시 조인을 많이 사용 하므로 성능 이슈가 있으며 조회 쿼리가 복잡 하고 insert가 두 번 발생

03. 단일 테이블

- 단일 테이블 통합

 # DB 물리 모델




```
@Entity
@Inheritance(strategy= InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="DTYPE")
public abstract class 상품 {
  @Id
  @GeneratedValue
  private int id

}

@Entity
@DiscriminatorValue("A")  {
public class 앨범 extends 상품 {
   private String 작곡가
}

@Entity
@DiscriminatorValue("B")  {
public class 영하 extends 상품 {
    private String 감독;
    private String  배우
}
```

조인이 발생 하지 않아서 성능이 빠르고 조회 쿼리가 단순 하다

04. 복수 테이블

- 각각의 테이블로 분리

 # DB 물리 모델


```
@Entity
@Inheritance(strategy= InheritanceType.TABLE_PER_CLASS)
public abstract class 상품 {
  @Id
  @GeneratedValue
  private int id

}

@Entity
public class 앨범 extends 상품 {
  private String 작곡가
}

@Entity
public class 영하 extends 상품 {
   private String 감독;
   private String  배우
}
```

Union 조인 필요

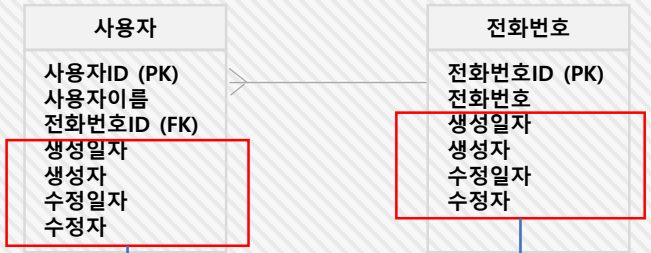
7. @MappedSuperclass

01. @MappedSuperclass

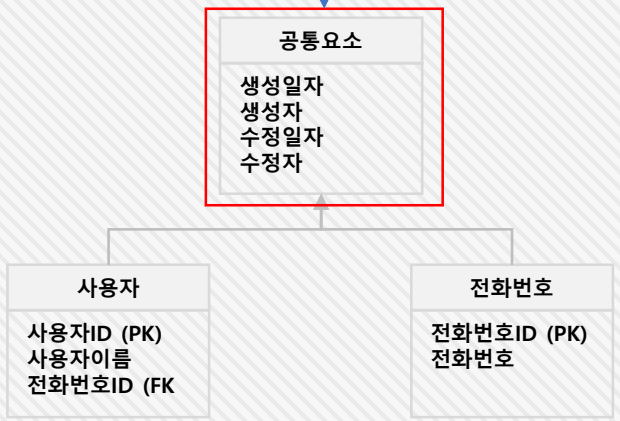
- 부모Class는 테이블에 매핑 하지 않고 자식 Class에게 매핑 정보 제공
- 추상 Class와 비슷, 테이블과 매핑 되지 않음
- 엔티티가 아니므로 조회가 불가함
- 공통 요소를 추상 Class 로 만들고 어노테이션 추가

@AttributeOverrides, @AttributeOverride : 매핑 정보 재정의
@AssociationOverrides, @AssociationOverride : 연관 관계 재정의
@Entity 객체는 @Entity, @MappedSuperclass 만 상속 가능

 # DB



 # 객체 관계




```
@Getter
@Setter
@MappedSuperclass
public abstract class 공통요소 {
  private String 생성자;
  private LocalDateTime 생성일자;
  private String 수정자;
  private LocalDateTime 수정일자;
}
```

```
@Entity
public class 사용자 extends 공통요소 {
  ...
}
```

```
@Entity
public class 전화번호 extends 공통요소 {
  ...
}
```

```
@Getter
@Setter
@MappedSuperclass
public abstract class 공통요소 {
  @Id, @GeneratedValue
  private Long id;
  private String name
}
```

```
@AttributeOverride(  name =id, column=@Column(name="MEMBER_ID" ))
public class 사용자 extends 공통요소 {
  ...
}
```

```
@Entity
@AttributeOverrides( {
      @AttributeOverride (name =id, column=@Column(name="MEMBER_ID") ),
      @AttributeOverride (name =name, column=@Column(name="MEMBER_NAME" )
public class 사용자 extends 공통요소 {
  ...
}
```

9. 복합키

01. 복합키 – 비식별관계 (Non-Identifying RelationShip)

- @IdClass – 식별자Class + Entity Class
  : 식별자, Entity 속성명 같아야 함
  : 식별자 Class는 Serializable 구현
  : equals, hasCode 구현
  : 기본 생성자가 있어야 함
  : 식별자 클래스는 public.

- Mandatory : Not Null – 연관관계 필수
- Optional : null 허용 – 연관관계 선택적

DB

가입별상품
int 가입상품누적번호 (PK) int 가입번호 (PK) String 서비스코드 String 상품코드

가입별 상품부가
Int 상품부가 (PK) int 가입상품누적번호 int 가입번호 String 대리점코드

@IdClass

가입별 상품 Key Class	가입별 상품 Calss
int 가입상품누적번호 (PK) →	int 가입상품누적번호
int 가입번호 (PK) →	int 가입번호
	String 서비스코드
	String 상품코드

```
@EqualsAndHashCode(onlyExplicitlyIncluded = true)
@AllArgsConstructor
@NoArgsConstructor
Public class 가입별상품key implements Serializable {

    @id
    @EqualsAndHashCode.Include
    private int 가입별상품누적번호;

    @id
    @EqualsAndHashCode.Include
    private int 가입번호
}
```

```
@Entity
@IdClass(가입별상품key.class)
Public class 가입별상품{
    private int 가입별상품누적번호;
    private int 가입번호

    private String 서비스코드;
    priavte String 상품코드;
}
```

```
@Entity
Public class 상품부가{
    @Id
    private int 상품부가;

    @ManyToOne
    @JoinColumns( { @JoinColumn(name="가입상품누적번호", refrencedColumn="가입상품누적번호"),
                    @JoinColumn(name="가입번호", refrencedColumn="가입번호") })
    private 가입별상품 가입별상품
}
```

02. 복합키 – 비식별관계 (Non-Identifying RelationShip)

- @EmbeddedId – 식별자Class + Entity Class
  : 식별자 클래스는 식별자 클래스에 기본키를 직접 매핑
  : 식별자 Class는 @Embeddedable 어노테이션
  : 식별자 Class는 Serializable 구현
  : equals, hasCode 구현
  : 기본 생성자가 있어야 함
  : 식별자 클래스는 public.

```
@EqualsAndHashCode(onlyExplicitlyIncluded = true)
@AllArgsConstructor
@NoArgsConstructor
@Embeddable
Public class 가입별상품key implements Serializable {

    @id
    @EqualsAndHashCode.Include
    private int 가입별상품누적번호;

    @id
    @EqualsAndHashCode.Include
    private int 가입번호
}
```

```
@Setter
@Getter
@Entity
Public class 가입별상품{
    @EmbeddedId
    가입별상품key 가입별상품key;

    private String 서비스코드;
    priavte String 상품코드;
}
```

03. 기타

- @IdClass가 db에 맞추어진 방법 이라면 @EmbeddedId는 객체지향적인 방법
- 복합 키에는 @GenerateValue를 사용 할 수 없습니다. 복합 키를 구성하는 여러 컬럼 중 하나에도 사용 할 수 없습니다.

9. 복합키

01. 복합키 – 식별 관계 (Identifying RelationShip)

- @IdClass – 식별자Class + Entity Class
  - : 식별자, Entity 속성명 같아야 함
  - : 식별자 Class는 Serializable 구현
  - : equals, hasCode 구현
  - : 기본 생성자가 있어야 함
  - : 식별자 클래스는 public.

@IdClass

```
@Entity
Public class 가입{
    @id
    @Column(name="ENTR_NO:")
    private int 가입번호
}
```

```
@Entity
@IdClass(가입별상품key.class)
Public class 가입별상품{
    @Id  @ManyToOne
    @JoinColumn(name="가입번호")
    private 가입 가입

    @Id
    @Column(name="가입별상품누적번호")
    private int 가입별상품누적번호;

    private String 서비스코드;
    priavte String 상품코드;
}
```

```
Public class 가입별상품key
        implements Serializable {

    private int 가입; // 가입별상품.가입

   // 가입별상품.가입별상품누적번호
    private int 가입별상품누적번호;

}
```

가입 Class

int 가입번호 (PK)

가입별 상품 Key Class

int 가입상품누적번호 (PK)
int 가입번호 (PK)

가입별 상품 Calss

int 가입상품누적번호 (PK)
int 가입번호 (PK. FK)
String 서비스코드
String 상품코드

```
@Entity
@IdClass(가입별 상품부가 key.class)
Public class 가입별상품부가 {
    @Id  @ManyToOne
    @JoinColumns{@JoinColumn(name="가입번호"),
                 @JoinColumn(name="가입별상품누적번호")})
    private 가입별상품 가입별상품

    @Id
    @Column(name="상품부가")
    private int 상품부가;

    private String 대리점코드;}
```

```
Public class 가입별상품부가key implements Serializable {

    private 가입별상품 가입별상품; // 가입별상품부가.가입
    private int 상품부가; // 가입별상품부가. 상품부가
}
```

- Mandatory : Not Null – 연관관계 필수
- Optional : null 허용 – 연관관계 선택적

 # DB

가입별상품

int 가입상품누적번호 (PK)
int 가입번호 (PK)
String 서비스코드
String 상품코드

가입별 상품부가

Int 상품부가 (PK)
int 가입상품누적번호 (PK, FK)
int 가입번호 (PK , FKV)
String 대리점코드

```
@Entity
Public class 가입별상품{
    @EmbeddedId
    가입별상품key 가입별상품key;

    @MapsId("가입")  @ManyToOne
    @JoinColumn(name="가입번호")
    private 가입 가입

    private String 서비스코드;
    priavte String 상품코드;
}
```

```
@Embeddable
Public class 가입별상품key implements
        Serializable {

    private int 가입; // @MapsId("가입")  매핑

    @Column(name="가입별상품누적번호")
    private int 가입별상품누적번호;
;
}
```

02. 복합키 – 비식별관계 (Non-Identifying RelationShip)

- @EmbeddedId – 식별자Class + Entity Class
  - : 식별자 클래스는 식별자 클래스에 기본키를 직접 매핑
  - : 식별자 Class는 @Embeddedable 어노테이션
  - : 식별자 Class는 Serializable 구현
  - : equals, hasCode 구현
  - : 기본 생성자가 있어야 함
  - : 식별자 클래스는 public.

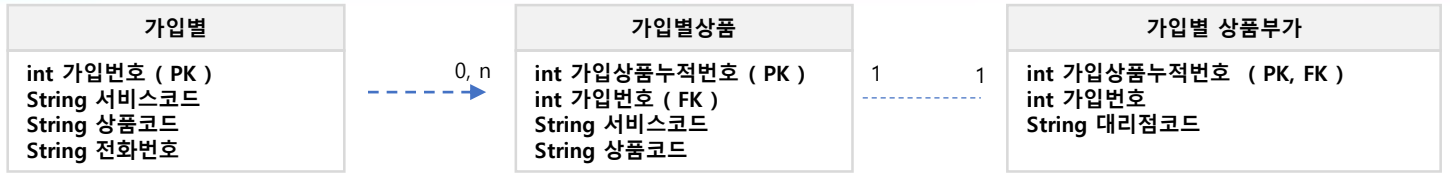
```
@Entity
Public class 가입{
    @id
    @Column(name="ENTR_NO:")
    private int 가입번호
}
```

```
@Entity
Public class 가입별상품부가 {
    @EmbeddedId
   가입별상품부가key 가입별상품부가key;

    @MapsId("가입별상품")  @ManyToOne
    @JoinColumns{@JoinColumn(name="가입번호"),
                 @JoinColumn(name="가입별상품누적번호")})
    private 가입별상품 가입별상품

    private String 서비스코드;
    priavte String 상품코드;
}
```

```
@Embeddable
Public class 가입별상품부가key implements
        Serializable {

    private 가입별상품 가입별상품; // @MapsId("가입별상품")

    @Column(name="상품부가")
    private int 상품부가;
;
}
```

9. 복합키 – 연관

 # DB




```
@Entity
Public class 가입{
    @id
    @GeneratedValue
    @Column(name="ENTR_NO:")
    private int 가입번호

    String 서비스코드
}
```

```
@Entity
Public class 가입별상품{

    @id
    @GeneratedValue
    @Column(name="가입상품누적번호")
    private int 가입상품누적번호


    @id
    @ManyToOne
    @JoinColumn(name="가입번호")
    private int 가입번호

    private String 서비스코드

    @OeeToOne(mappedBy = "가입별상품")
    private 가입별상품부가 가입별상품부가
}
```

```
@Entity
Public class 가입별상품부가{
    @Id
    private int 가입상품누적번호

    @MapsId
    @OneToOne
    @JoinColumn(name= "가입상품누적번호")
    private 가입별상품 가입별상품;

    private String 대리점코드
```

10. 프록시 & 즉시로딩, 지연로딩

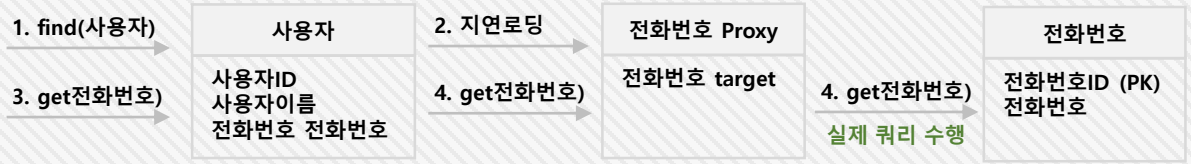
01. 프록시

- 가상 객체를 내부적으로 생성 하여 엔티티에 접근
- JPA는 find(), getReference() 제공
- getReference()가 가상 객체 조회
  : 실제 DB 조회 하지 않고 사용 시점에 DB 조회

준 영속 상태에서 getReference는 예외 발생
-> 트랜잭션 범위 밖에서 프록시 객체 조회 하는 경우
-> open-session-in-view 패턴

02. 지연로딩, 즉시 로딩

- 사용 시점에 쿼리 수행
- @ManyToOne(fetch = FetchType.LAZY)
  : 기본은 즉시 로딩 (EAGER)
  : 실제 개발은 지연로딩 사용



FetchType.EAGER
- @ManyToOne, @OneToOne
  : optional = false : 내부 조인
  : optional = true : 외부 조인
- @OneToMany, @ManyToMany
  : optional = false : 외부 조인
  : optional = true : 내부 조인

03. 영속성 전이

- 부모 객체 저장시 자식 객체 동시 저장
  : @OneToMany(mapprdBy="p" cascade = CasCadeTypr.PERSIST):

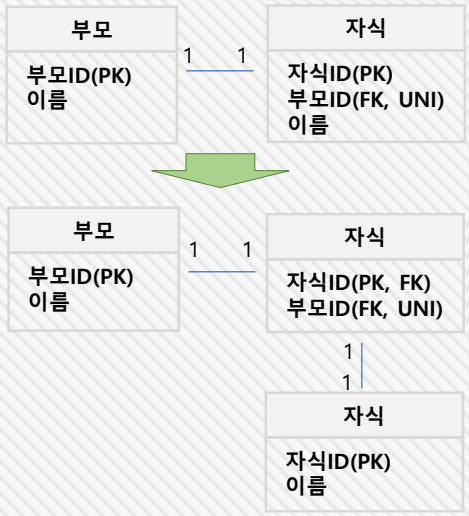
04. 고아 객체

- orphanRemoval = true
  : @OneToMany, @ManyToOne 만 사용
- 객체가 제거 되면 남아 있는 객체를 의미 하며 true로 설정 하면 자동 삭제
  : db 삭제 됨

11. JOIN

@JoinTable : name : 매핑할 조인 테이블 이름
joinColumn : 현재 엔티티를 참조하는 외래키
inverseoinColumn : 반대방향 엔티티를 참조하는 외래키

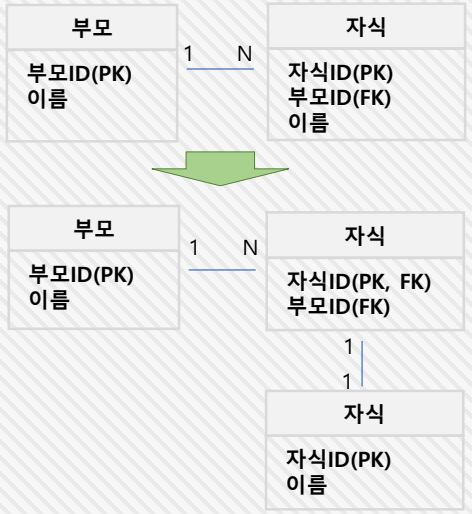
01. 일대일




```
@Entity
Public class 부모 {
  @id @GeneratedValue
  @Column(name="부모ID")
  private long 부모ID;
  private String 이름;
  @OneToOne
  @JoinTable(name="부모자식"
     JoinColumns=@JoinColumn(name="부모ID"),
     InverseJoinColumns=@JoinColumn="자식ID"))
  private 자식 자식;
}
```

```
@Entity
Public class 자식 {
      @id @GeneratedValue
      @Column(name="자식ID")
      private long 자식ID;
      private String 이름;
}
```

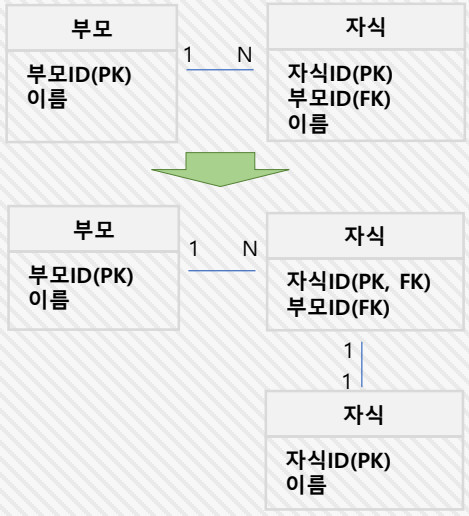
02. 일대다




```
@Entity
Public class 부모 {
  @id @GeneratedValue
  @Column(name="부모ID")
  private long 부모ID;
  private String 이름;
  @OneToMany
  @JoinTable(name="부모자식"
     JoinColumns=@JoinColumn(name="부모ID"),
     InverseJoinColumns=@JoinColumn="자식ID"))
  private LIst<자식> 자식 = new ArrayList<자식>();
}
```

```
@Entity
Public class 자식 {
      @id @GeneratedValue
      @Column(name="자식ID")
      private long 자식ID;
      private String 이름;
}
```

03. 다대일




```
@Entity
Public class 부모 {
  @id @GeneratedValue
  @Column(name="부모ID")
  private long 부모ID;
  private String 이름;
  @OneToMany(mapperdBy="부모")
  private LIst<자식> 자식 = new ArrayList<자식>();
}
```

```
@Entity
Public class 자식 {
      @id @GeneratedValue
      @Column(name="자식ID")
      private long 자식ID;

      @ManyToOne(optional=false)
      @JoinTable(name="부모자식"
      JoinColumns=@JoinColumn(name="자식ID"),
      InverseJoinColumns=@JoinColumn="부모ID"))
      private 부모 부모;
}
```

04. 다대다


```
@Entity
Public class 부모 {
  @id @GeneratedValue
  @Column(name="부모ID")
  private long 부모ID;
  private String 이름;

  @ManyToMany
  @JoinTable(name="부모자식"
     JoinColumns=@JoinColumn(name="부모ID"),
     InverseJoinColumns=@JoinColumn="자식ID"));
  private LIst<자식> 자식 = new ArrayList<자식>();
}
```

```
@Entity
Public class 자식 {
      @id @GeneratedValue
      @Column(name="자식ID")
      private long 자식ID;
      private String 이름;
}
```

12. Value Type

01. 값 타입

- Int, Integer, String처럼 단순히 값으로 사용하는 자바 기본 타입 or 객체
- 종류
  : 기본값 타입(basic value type)
  - 자바 기본 타입 (int, double ..)
  - 래퍼 Class (Integer ..)
  - String
  : 임베디드 타입 (embedded type)
  : 컬렉션 타입 (collection value type)

02. 기본값 타입

- 자바 기본 타입 (int, double ..), 래퍼 Class (Integer ..) , String

```
@Entity
Public class 엔티티 {
    @id @GeneratedValue₩
    private long id;
    private String 이름;
    prvate int 너이
}
```

03. 임베디드(복합) 타입

- 새로운 값 타입을 직접 정의해서 사용

• @Embedded : 값 타입을 정의 하는 곳에 표시
* @Embeddable : 값 타입을 사용하는 곳에 표시
* @Embeddable에서 @Embedded , @Entity를 사용 하여 참조 함
* @AttributeOverride : 속성 재 정의

 # DB 물리 모델

가입별상품

가입상품누적번호 (PK)
가입번호 (PK)
서비스코드
상품코드
상품상세코드
생성일자
생성자
수정일자
수정자

 # 객체 모델

가입별상품

int 가입상품누적번호 (PK)
int 가입번호
상품 상품
제어 제어

상품

String 서비스코드
String 상품코드
String 상품상세코드

제어

DATE 생성일자
String 생성자
DATE 수정일자
String 수정자

```
@Entity
Public class 가입별상품 {
    @Id  @GeneratedValue
    private long 가입상품누적번호
    private int 가입번호;

    @Embedded private 상품 상품;
    @Embedded private 제어 제어;

}
```

```
@Embeddable
Public class 상품 {
    private String 서비스코드;
    private String 상품코드;
    private String 상품상세코드;
}
```

```
@Embeddable
Public class 제어 {
    private String 생성일자;
    private String 생성자;
    private String 수정일자;
    private String 수정자;
}
```

 # 객체 모델 재정의

가입별상품

int 가입상품누적번호 (PK)
int 가입번호
상품 상품
제어 생성
제어 수정

상품

String 서비스코드
String 상품코드
String 상품상세코드

제어

DATE 생성일자
String 생성자

```
@Entity
Public class 가입별상품 {
    @Id  @GeneratedValue
    private long 가입상품누적번호
    private int 가입번호;

    @Embedded private 상품 상품;
    @Embedded private 제어 생성;
    @Embedded
    @AttributeOverrides({
        @AttributeOverride(name="생성일자", column="수정일자" ),
        @AttributeOverride(name="생성자", column="수정자" )})
    제어 수정;
}
```

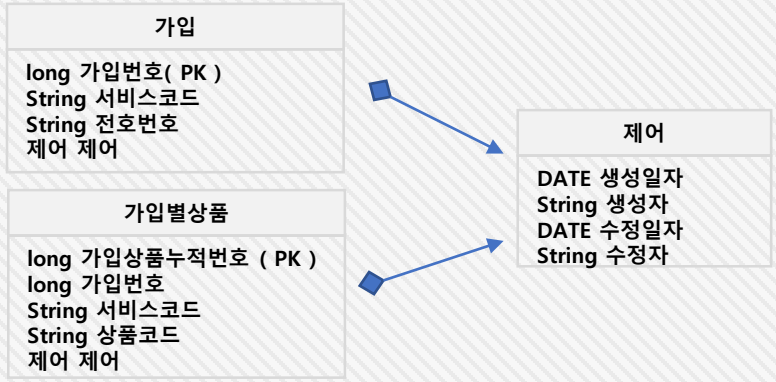
```
@Embeddable
Public class 상품 {
    private String 서비스코드;
    private String 상품코드;
    private String 상품상세코드;
}
```

```
@Embeddable
Public class 제어 {
    private String 생성일자;
    private String 생성자;
}
```

12. Value Type

04. 공유 참조 , 복사

- 공유 참조 사용시 값은 공유 됨



 # 공유

```
Public class 서비스 {
    가입 가입 = new 가입();
    가입별상품 가입별상품 = new 가입별상품();

    가입.set제어(new 제어("20201212",
                          "생성자",
                          "20201212",
                          "수정자"));
    제어 제어 = 가입.get제어();

    제어.set생성자("New생성자");
    가입별상품.set제어(제어);
}
```

 # 복사

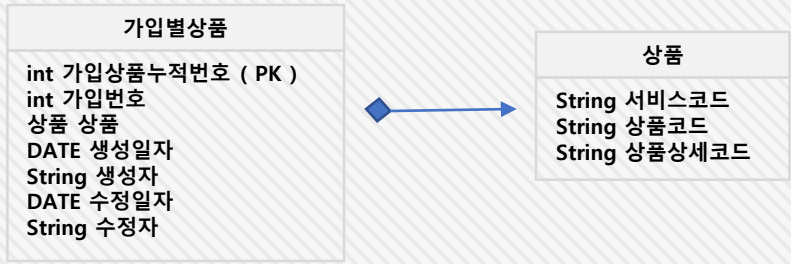
```
Public class 서비스 {
    가입 가입 = new 가입();
    가입별상품 가입별상품 = new 가입별상품();

    가입.set제어(new 제어("20201212",
                          "생성자",
                          "20201212",
                          "수정자"));
    제어 제어 = 가입.get제어();

    제어 가입상품제어 = 제어.clone();
    가입별상품.set제어(가입상품제어);
}
```

- 가입의 생성자가 생성자에서 New 생성자로 변경됨
- 임베디드 타입처럼 직접 정의한 Value Type은 자바의 기본 타입(primitive type)이 아니라 객체 타입 이다.

- 복잡하게 생각 하지 말고 생성자에 의해서만 객체의 값을 생성 할 수 있도록 설계 하면 쉽게 문제를 해결 할 수 있다. Value Type은 생성자에 의해서만 값을 변경 하도록 한다.
  => immnutable Object (불변 객체)

05. Collection Type

- @ElementCollection, @CollectionTable




```
@Entity
Public class 가입별상품 {
    @Id  @GeneratedValue
    private long 가입상품누적번호
    private int 가입번호;

    @ElementCollection
    @CollectionTable( name = '제어",
                      joinColumn = @JoinColumn(name="가입상품누적번호"))
    private List<상품> 상품s = new ArrayList<상품>
}
```

```
@Embeddable
Public class 상품 {
    private String 서비스코드;
    private String 상품코드;
    private String 상품상세코드;
}
```

13. Entity Type & Value Type

01. Entity Type

- 식별자 @Id가 있다
- 생명 주기가 있다.
- 공유 할 수 있다
  : 가입 엔티티를 다른 엔티티에서 참조 할 수 있다.

02. Value Type

- 식별자가 없다
- 생명주기가 엔티티에 의존 한다.
- 공유하지 않는 것이 안전 하다. (불변 객체)

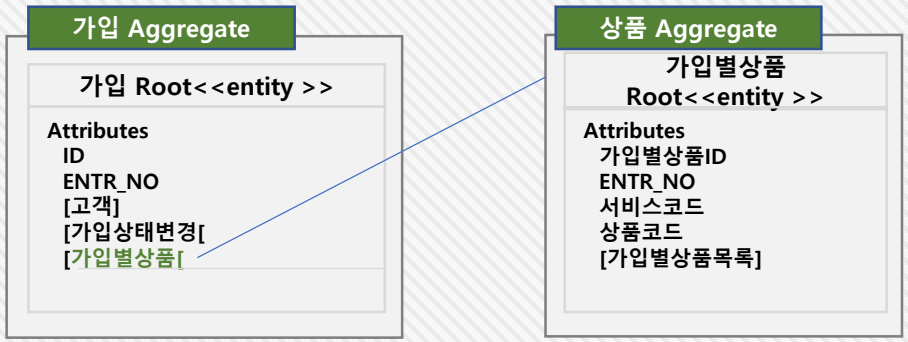
Domain Driven Design 설계를 이해 해야 한다.

Content

IV. Aggregate

4. Aggregate 참조

01. Aggregate 객체 참조



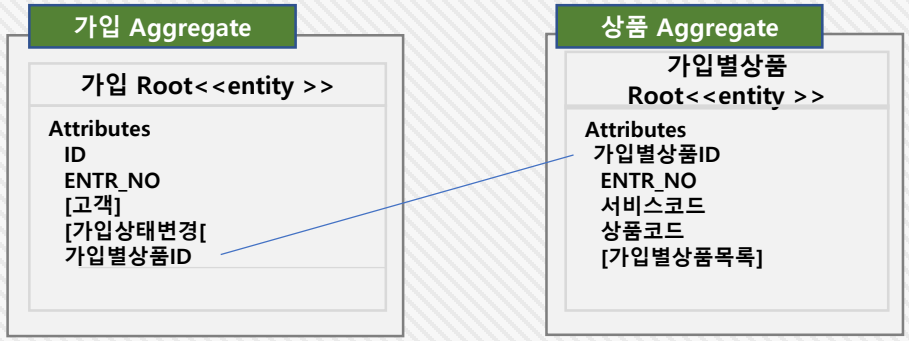

```
class 가입Root {
  long Id
  ....
  가압별상품Root 가입별상품Root ;

}
```

```
class 가입별상품Root {
  ...

}
```

생각해 볼 문제

- 가입 Aggregate에서 상품 Aggregate를 수정 하는 것이 맞는가 ?
  => 책임과 역할에 대한 문제
- 즉시 로딩을 할 것인가? , 지연로딩을 할 것 인가 ?
  => 업무에 따라서 선택 하여야 함
- 도메인 별로 확장을 할 경우 확장이 용이 한가 ?
  => 학장 용이 하지 않음
- 모델의 결합도와 응집도는 ?
  => 결합도가 높고 응집도는 낮음
- 모델의 복잡도는 낮은가 ?
  => 모델 복잡

02. Aggregate 객체 ID 참조




```
class 가입Root {
  long Id
  ....
  long 가입별상품ID;

}
```

```
class 가입별상품Root {
  ...

}
```

생각해 볼 문제

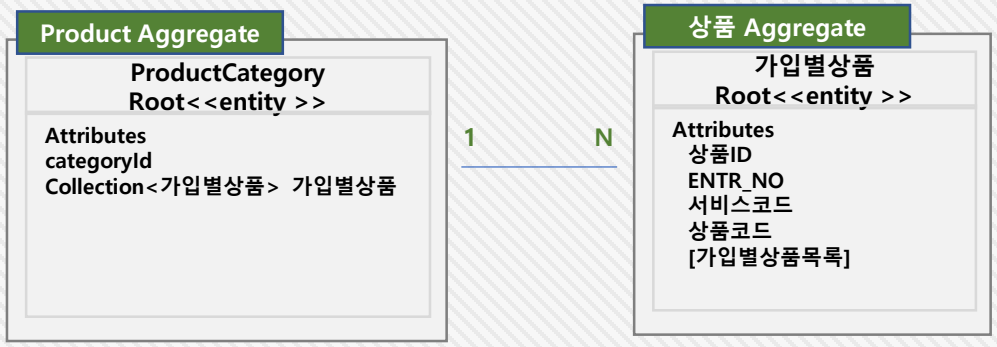
- 가입 Aggregate에서 상품 Aggregate를 수정 하는 것이 맞는가 ?
  => 원칙적으로 수정 금지
- 즉시 로딩을 할 것인가? , 지연로딩을 할 것 인가 ?
  => 고민할 필요 없이 필요시 직접 로딩
- 도메인 별로 확장을 할 경우 확장이 용이 한가 ?
  => 확장 용이
- 모델의 결합도와 응집도는 ?
  => 결합도는 낮아지고 응집도는 높아짐
- 모델의 복잡도는 낮은가 ?
  => 객체 참조 보다 낮음

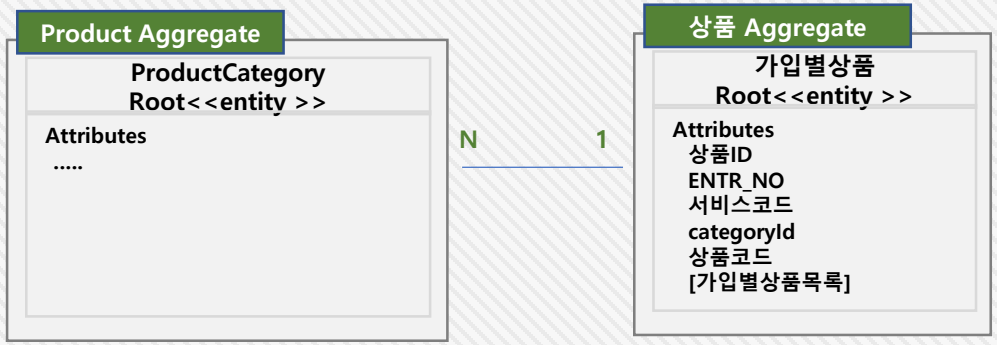
5. Aggregate 연관

01. 1:N, N:1

생각해 볼 문제

- 가입자가 자지고 있는 상품 중 요금 상품을 모두 조회 해야 한다면

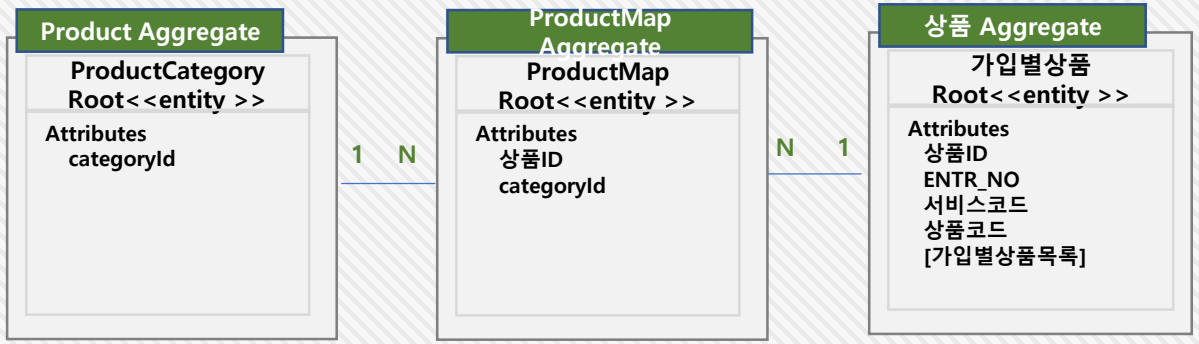




02. N:M

생각해 볼 문제

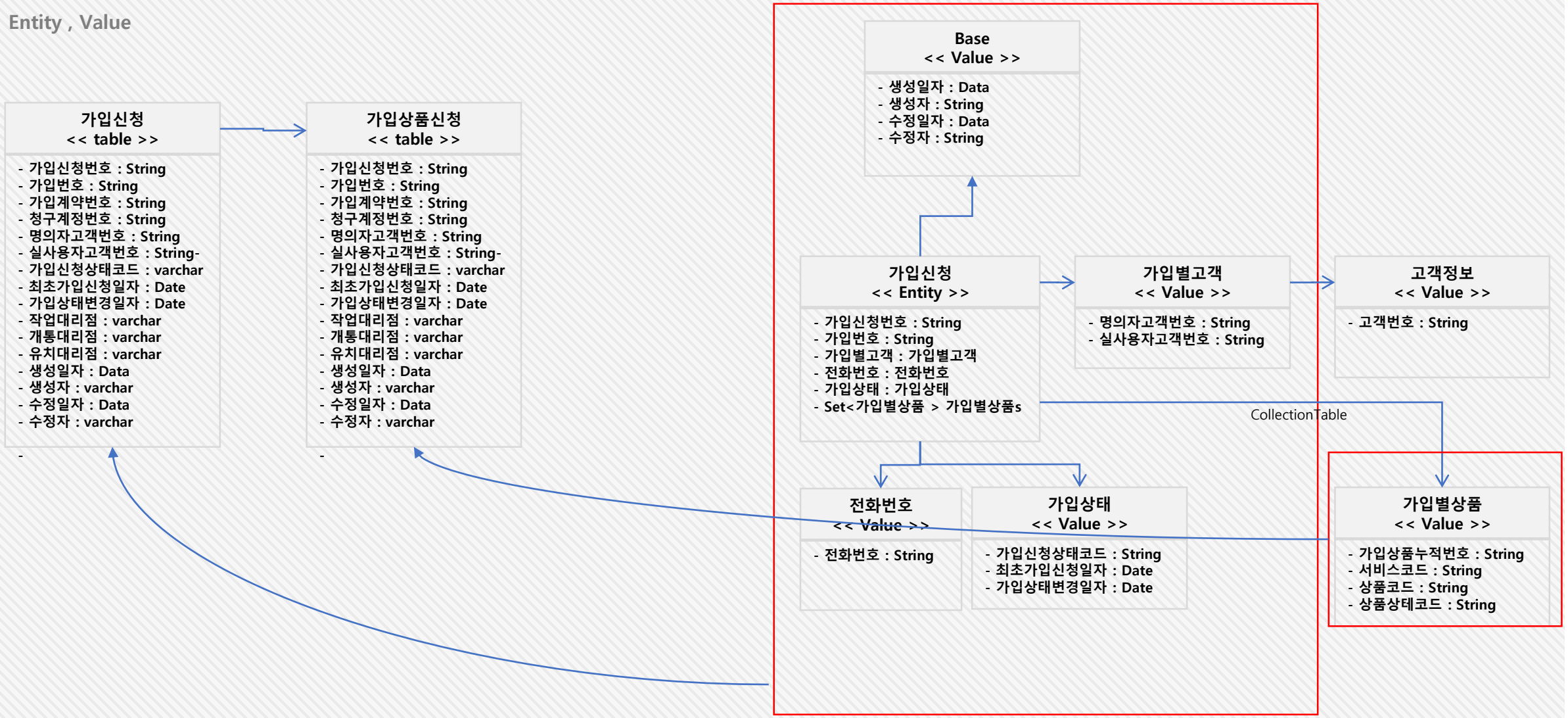
- 가입자가 자지고 있는 상품 중 요금 상품을 모두 조회 해야 한다면



ABACUS

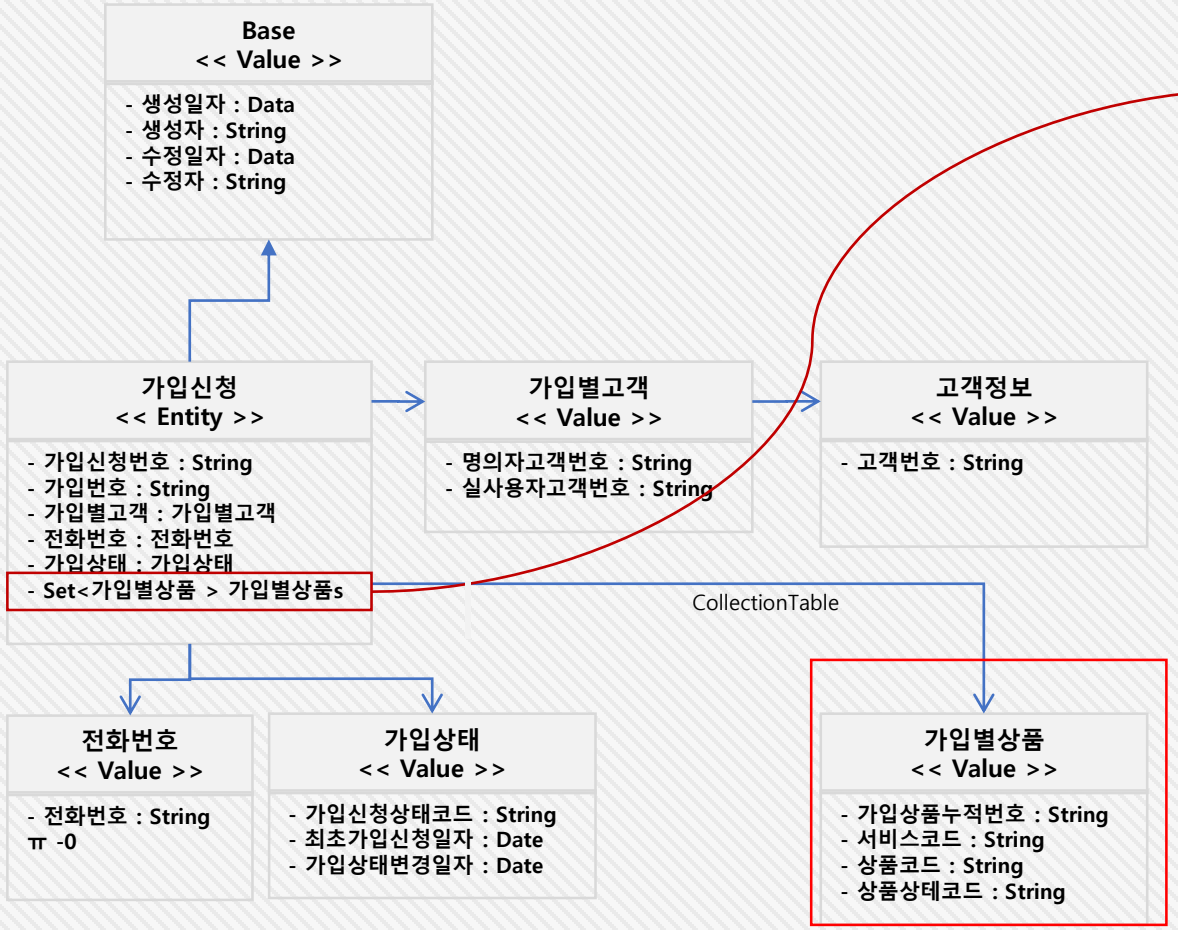
6. Aggregate에서 Entity, Value

01. Entity , Value



7. Aggregate에서 CollectionTable

01. @ElementCollection, @CollectionTable

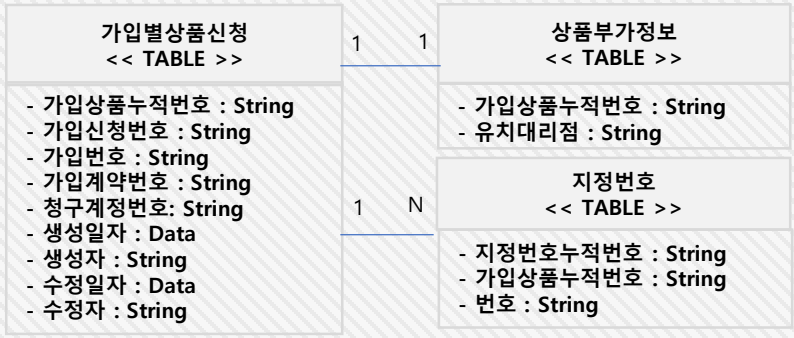


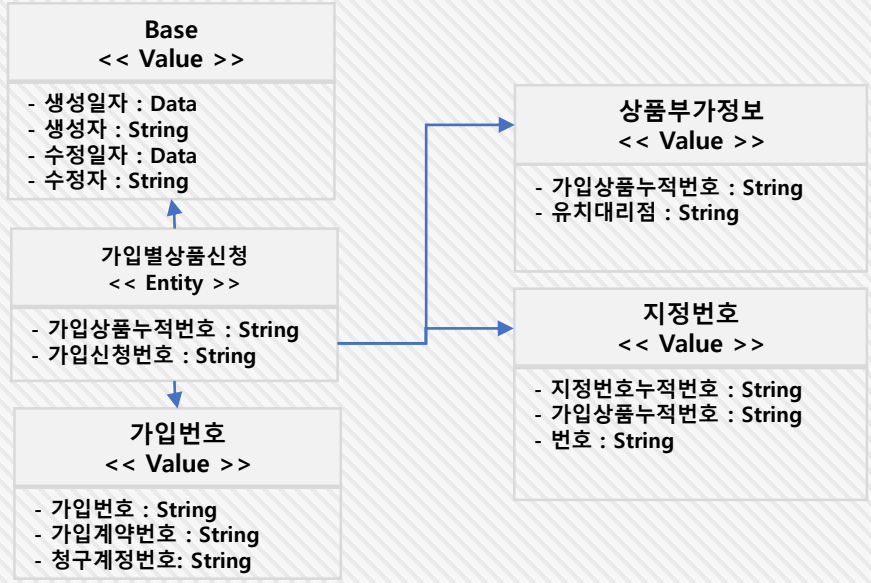

```
@ElementCollection(fetch = FetchType.LAZY)
@CollectionTable(name = "TB_SB_ENTR_RQST_HIST",
    joinColumns = { @JoinColumn(name = "ENTR_RQST_NO", referencedColumnName =
"ENTR_RQST_NO")})
@OrderColumn(name="ENTR_RQST_HIST_DTTM", columnDefinition = "date")
private List<EntrRqstHistVO> entrRqstHistVos;
```

: 값 객체 Collection에 새로운 값이 추가되거나 기존 값이 변경될 시 All Delete and Re-insert
: 이를 완화시키기 위해서 @OrderColumn을 추가하는 방법이 있긴하나 완벽하지는 않음

8. Aggregate에서 SecondaryTable

01. @SecondaryTable,@SecondaryTables






```java
@Entity
@Table(name="TB_SB_SVC_BY_ENTR_RQST")
@SecondaryTable(
        name = "TB_SB_SVC_ADDV_RQST",
        pkJoinColumns = @PrimaryKeyJoinColumn(name="ENTR_SVC_RQST_SEQ")
)
public class TbSbEntrSvcRqst  {

    @EmbeddedId
    private EntrSvcRqstKeyVO entrSvcRqstSeq;

    @Embedded
    private EntrSvcNumVO entrSvcNumVO;

    @Embedded
    private EntrSvcAddvVO entrSvcAddvVO;

    @OneToMany
    @JoinColumn(name="ENTR_SVC_RQST_SEQ")
    @OrderColumn(name="ENTR_SVC_FTR_RQST_SEQ")
    private Collection<EntrSvcFtrRqstVO> entrSvcFtrRqstVOs ;
```

```java
@Embeddable
@Table(name="TB_SB_SVC_ADDV_RQST")
@Getter
@NoArgsConstructor
@AllArgsConstructor
@EqualsAndHashCode
public class EntrSvcAddvVO extends BaseEntity {

    @Column(name="DLR_CD")
    private String dlrCd;

    @Column(name="WORK_DLR_CD")
    private String workDlrCd;
}
```

```java
@Entity
@Table(name="TB_SB_SVC_FTR_RQST")
@Getter
@NoArgsConstructor
@AllArgsConstructor
@EqualsAndHashCode
public class EntrSvcFtrRqstVO  {

    @EmbeddedId
    private EntrSvcFtrRqstKeyVO entrSvcFtrRqstKeyVO;

    @Column(name="FTR_CD" )
    private String ftrCd;


    @Column(name="FTR_VAALUE" )
    private String ftrValue;

}
```

Content

V. Spring Data JPA

1. 개요
2. Open Session In View
3. 지연로딩에 따른 LazyInitializationException
4. 영속성 컨텍스트 범위

1. 개요

Spring Data JPA는 실제로 Hibernate, Jboss, EclipseLink와 같은 JPA 제공자가 아니고 라이브러리 또는 프레임 워크로 Spring Data의 프로젝트

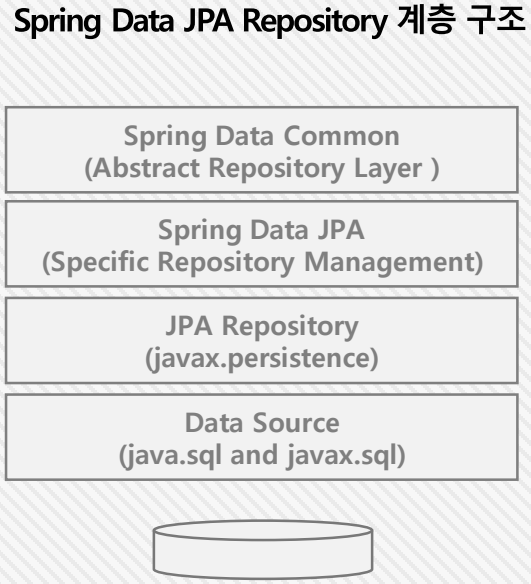
01. Spring Data JPA

- Spring Data Repository 확장하여 Repository를 구축
- CRUDRepository, JPARepository에서 제공하는 내장 CRUD 메소드를 사용
- 메소드 이름 규칙을 사용하여 메소드를 작성하거나 @Query 어노테이션이있는 조회를 제공하는 메소드를 작성
- 페이지 매김, 슬라이싱, 정렬 및 감사 지원
- XML 기반 엔터티 매핑 지원
- Specification<T> and QueryDsl 제공

02. Spring Data JPA 에서 지원 하는 쿼리

- JPQL : 문자열 기반의 쿼리 언어
- Criteria Query : CriteriaBulider을 사용한 쿼리 생성로 동적 쿼리를 사용하기 위한 JPA 라이브러리 제공
- Native Query : JPA에서 SQL 쿼리 실행
- RDBMS 저장 프로시저 지원

03. Spring Data JPA Repository 계층 구조



데이터 저장소 특정 스프링 데이터 프로젝트 간에 공유되는 인프라

JPA Repository를 생성하기 위해 구현 된 **Spring Data Repository Interface**

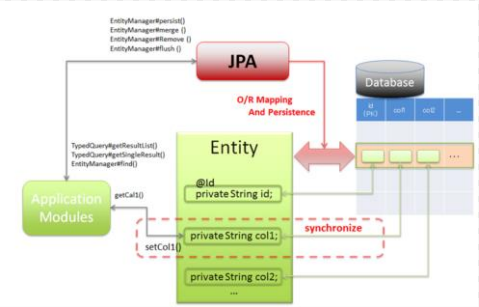
JPA 지속성 **API**를 구현하는 **JPA** 제공자

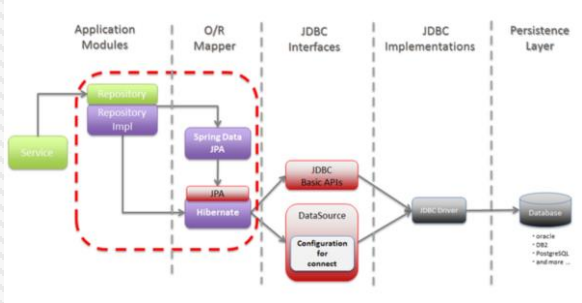
2. Open Session In View

영속성 컨텍스트를 뷰 렌더링이 끝나는 시점까지 개방한 상태를 유지 하는 것

01. 영속성 컨텍스트

- 도메인 레이어 객체들이 하부의 데이터 저장소와 영속성 매커니즘에 대해 알지 않아도 되는 투명한 영속성을 제공하는 프레임워크로 Hibernate사용
- Hibernate의 session객체가 영속성 컨텍스트를 관리함
- Transaction는 쪼갤 수 없는 업무처리의 단위로 영속성 컨텍스는 1:1:로 연결됨
- 하나의 Transaction동안 수정된 객체의 모든 상태는 영속성 컨텍스트 내에 저장되고 종료 될 때 데이터 저장소와 동기 됨





03. Open Session In View 를 이해 하기 전 알아야 할 개념

- Transaction 범위
  : Spring에서 @Transactional을 사용 하면 자동으로 Session이 열리고 객체의 변경 추적이 되기 시작되고 종료 시점에 쿼리 실행을 하여 데이터 저장소와 동기
- 영속성 객체 상태
  : Persistence, Detached, Transient, Removed
- 패치전략
  : 쿼리 수행과 관련된 객체와 연관 관계를 맺고 있는 객체나 컬렉션을 어느 시점에 가져올지에 대한 전략으로 EAGER(즉시로딩), LAZY(지연로딩)가 있음
  : EAGER : 데이터를 즉시 가져 오는 전략
  : LAZY : 데이터를 처음 액세스 될 때 가져오는 전략
- 프록시
  : 지연 패치 전략에 따라 연관 관계를 맺고 있는 객체와 컬렉션에 있은 실제 객체 처럼 위장된 프록시 객체 생성 후 실제 엔티티에 접근 할 때 영속성 컨텍스트에 생성 요청 하는 것을 프록시 초기화라고 하며 한번만 초기화 됨

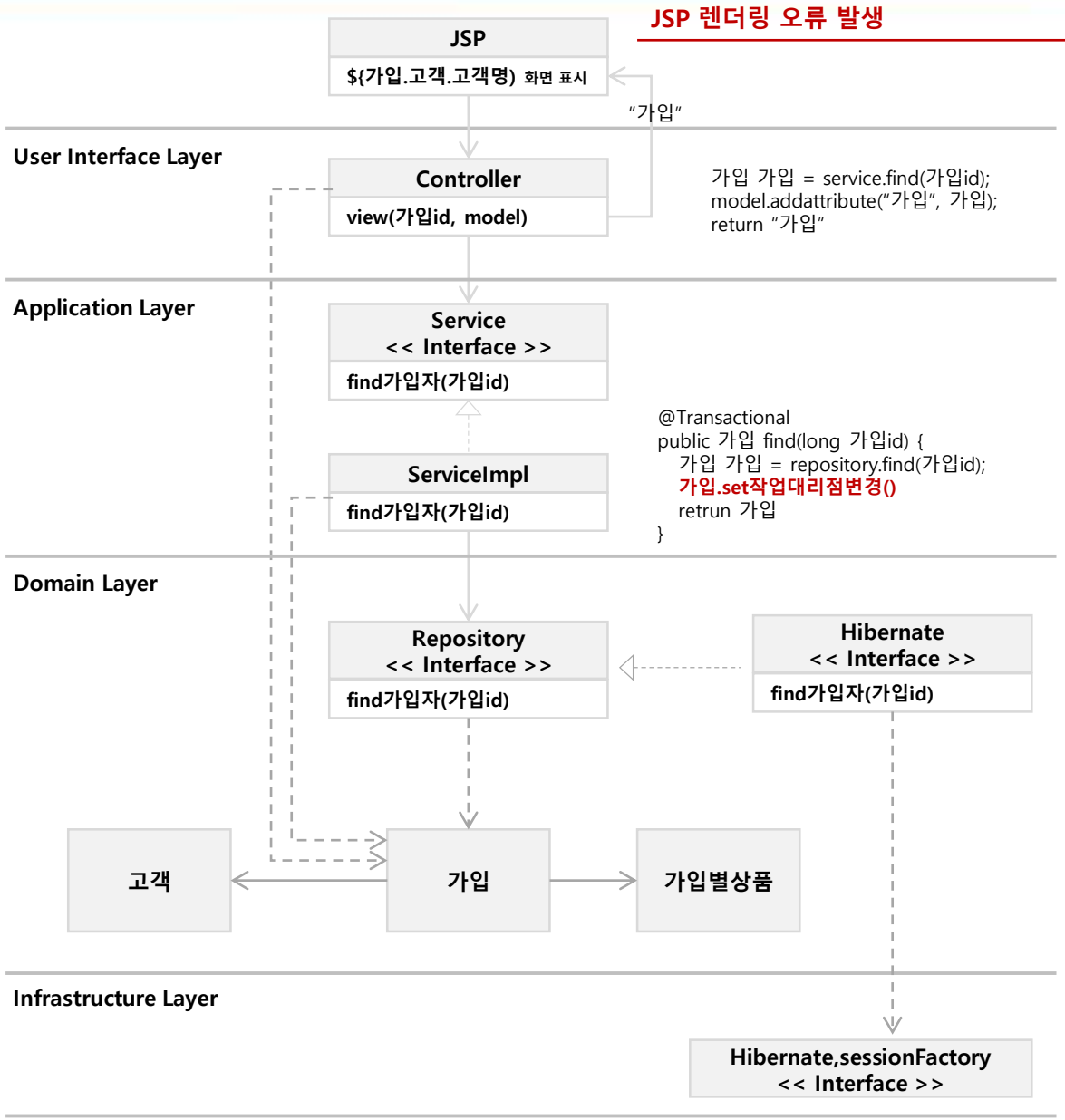
04. Open Session In View

- Service Layer는 Application의 Transaction경계를 정의하는 역할을 하게 되고, 이로인해 발생하는 문제가 Open Session In View을 등장 하게 됨
  : View Layer에서 연관 객체를 사용하려 할 때 발생하는 LazyInitialzationException

해결방안

1. 뷰 렌더링에 필요한 객체 그래프 모두 로딩 : EAGER Fetch
   -> View와 강한 결합 -> 관심사의 분리 원칙 위배
2. POJO FAÇADE Pattern : 새로운 객체를 통해 프록시를 초기화한 후 사용자 인터페이스로 반환 하는 방법
   -> View에 사용 하는 객체를 모두 로드 하는 방식
3. Open Session In View : 뷰 렌더링 시점에 영속성 컨텍스트가 존재 하지 않기 때문에 Detached 객체의 프록시를 초기화 할 수 없다면 영속성 컨텍스트를 오픈 된 채로 뷰 렌더링 시점까지 유지 하는 것
   -> 서블릿 필터 시작 시에 Hibernate Session을 열고 Transaction 시작하고 종료 시점에 커밋
   -> JDBC Connection 보유 시간 증가
4. Spring Open Session In View : FlushMode 와 ConnectionReleaseMode의 조정을 통해 전통적인 서블릿 필터의 단점을 보안 하는 OpenSessionInViewFilter 와 OpenSessionInViewInterceptor 를 제공
   -> OpenSessionInViewFilter 는 필터 내에서 Session을 오픈하지만 트랜잭션은 시작하지 않음

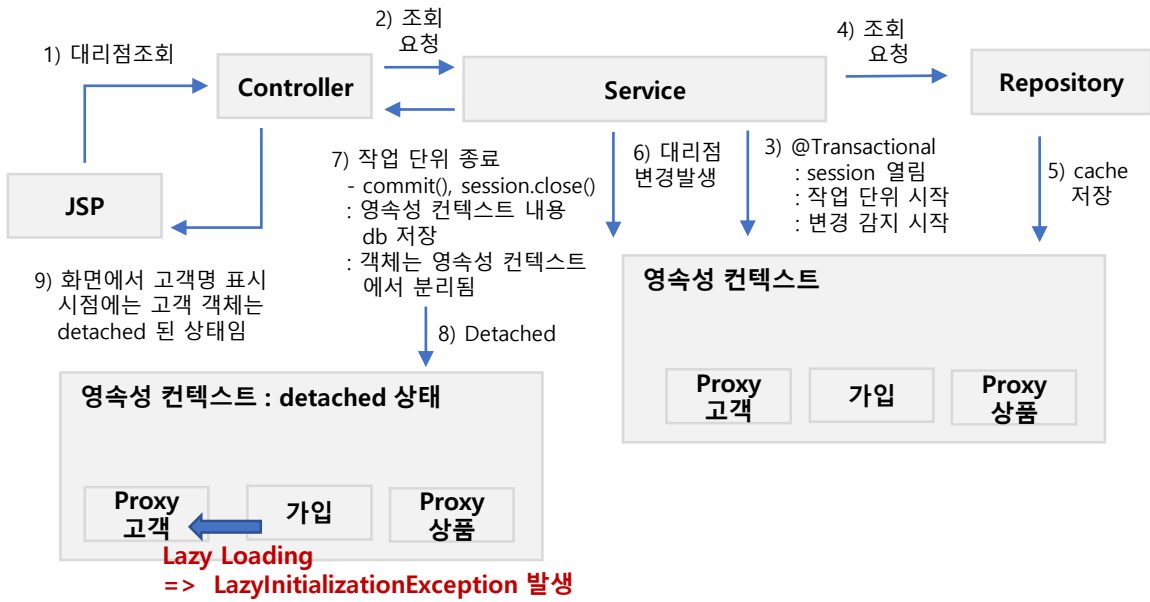
3. 지연로딩에 따른 LazyInitializationException



JSP 렌더링 오류 발생 → org.hibernate.LazyInitializationException: proxy – no Session

- Hibernate의 session은 JSP의 1차 캐쉬 역할을 함
  즉 영속성 컨텍스트 내에 저장된 후 작업 단위가 종료 시점에 DB 동기화
- 기본적으로 Hibernate는 **Lazy Loading, Transactional Write-behind**

내부 동작 구조



해결 방법 : ThreadLocal Session 패턴 => Session per request

1. 지연 로딩을 하지 않는다.
   (QueryDSL 사용)
   : Repository 재활용성 감소
   : Repository 복잡도 증가
   : View와 영속성 관심사 강한 결합

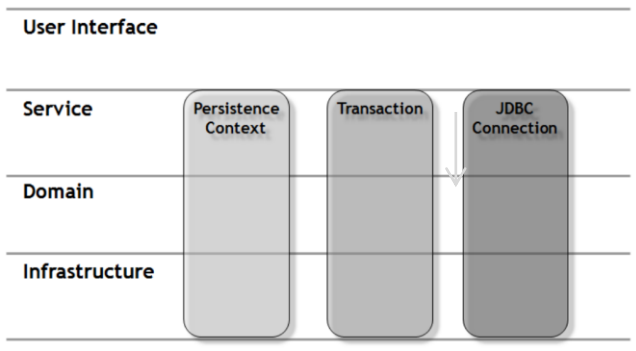
2. POJO FAÇADE 패턴
   : 모든 프록시를 초기화
   : getHibernateTemplate().initialize() 사용
   : 개발 난이도 증가

3. OPEN SESSION IN VIEW 패턴
   : 작업 단위를 요청 시작 부터 뷰 렌더링 시점까지 확장
   : Filter 사용 -> Transaction 관리
   : Spring 의 OpenSessionInViewFilter 설정

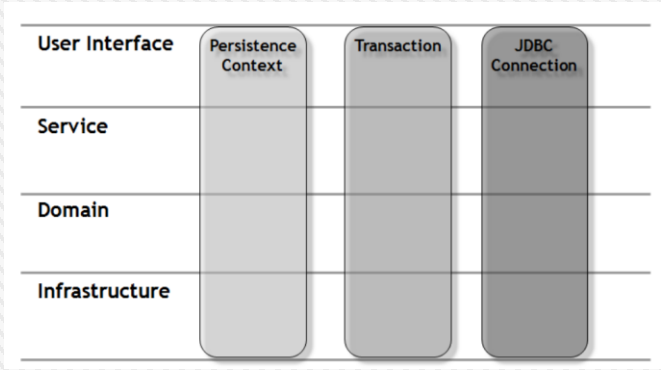
ABACUS

4. 영속성 컨텍스트 범위

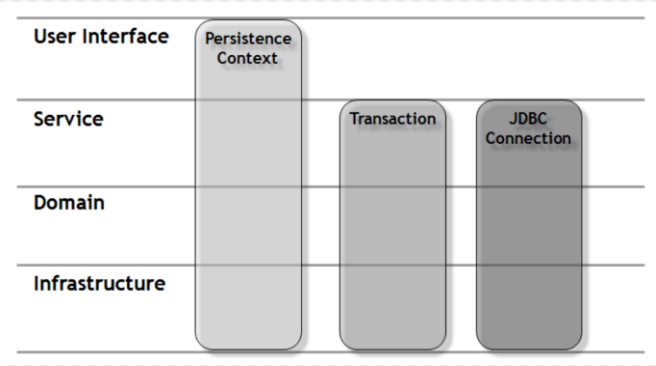
01. 일반적인 영속성 컨텍스트, 트랜잭션, 커넥션 범위



02. Filter 사용 한 영속성 컨텍스트, 트랜잭션, 커넥션 범위

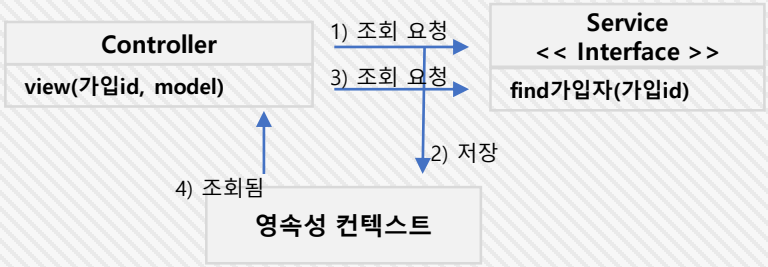


03. Spring 의 OpenSessionInViewFilter 영속성 컨텍스트, 트랜잭션, 커넥션 범위



1. 영속성 컨텍스트와 트랜잭션의 경제가 틀림
2. 트랜잭션에 대한 일관성 있는 뷰를 제공 하지만 영속성 컨텍스트에 대한 일관성 있는 뷰는 제공 하지 않음
   => Controller에서 영속성 컨텍스트가 활성화됨
   => Transaction 경계 외부에서 영속성 컨텍스트 내에 저장
3. OpenSessionInViewFilter=false 설정
   : Transaction 별로 Session 생성
   : 서블릿 필터 시작 시에 Session 을 오픈하지 않는다. 대산 SessionFactory 자체를 ThreadLocal 에 저장한 후 컨트롤러로 요청 처리를 위임
4. 기본적으로 HibernateTransactionManager는@Transactional 시점에 ThreaLocal에 저장 되어 있는 SessionFactory를 사용하여 Session 오픈
   : singleSession=true -> 서블릿 필터에서 Session 공유
   : singleSession=fa;se -> Transactional 시점에 Session 공유
   - JDBC Connection 증가됨

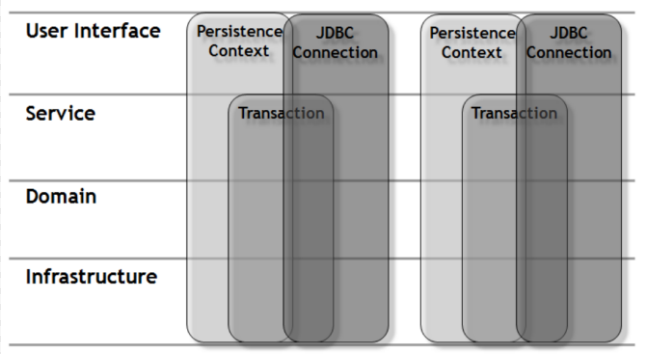
Controller 에서 두 번 호출



즉 1), 2) 에서 조회한 객체는 동일함
만약 2)번에서 변경이 있었으면 Transaction은 두 번 실행

OpenSessionInViewFilter을 사용하지 않으면 다른 객체에 대한 작업
OpenSessionInViewFilter을 사용하면 동일 객체에 대한 작업

singleSession=fa;se 영속성 컨텍스트, 트랜잭션, 커넥션 범위



ABACUS

Content

VI. 쿼리언어

1. 개요
2. JPQL – Runing JPA Query
3. JPQL – Query Parameters In JPA
4. JPQL – SELECT 절 - Projection
5. JPQL – Query 구조
   (Path Expression, SubQuery .. 등)
6. JPQL – Named 쿼리
7. QueryDSL

참고 :
Object DB : https://www.objectdb.com/java/jpa/query
QueryD니 : http://www.querydsl.com/

1. 개요

테이블이 아닌 객체를 대상으로 검색하는 객체 지향 쿼리로 SQL을 추상화해서 특정 DBMS SQL에 의존 하지 않는다.

01. JPA가 공식 지원하는 기능

- JPQL(Java Persistence Query Language)
  : Entity Object를 조회 하는 객체 지향 쿼리
  : SQL을 추상화해서 특정 DBMS에 의존 하지 않음
  : Entity 직접 조회, 묵시적 조회, 다양성 지원으로 SQL보다 간결함
- Criteria Query
  : JPQL을 편하게 작성하도록 도와주는 API, Builder Class 모음
  : 문자가 아닌 프로그램 코드로 JPQL을 작성
  -> 컴파일 시점에 오류 발견
  -> 동적 쿼리를 작성 하기 편하다.
  -> JPA 2.0 부터 적용 됨
  : 프로그램 코드로 쿼리가 한눈에 들어 오지 않음
- Native Query
  : 직접 Query 사용
- Named Query
  : @NamedQuery, @NamedQuery annotation을 사용해서 정적 쿼리 실행

02. JPA 개발시 알아야 할 것

- QueryDSL
  : 정적 타입을 이용해서 SQL과 같은 쿼리를 생성할 수 있도록 해 주는 프레임워크
- MyBatis
  : SQL Mapper 프레임워크
- JDBC

03 JPA Query API

- Query
  : 변환 타입을 명확하게 지정할 수 없을 때
  : 조회 대상이 하나이면 Object로 반환 (select c.firstname, from Cust c)
  : 조회 대상이 여러 개 이면 Objects로 반환 (select c.firstname, c.lastname from Cust c)

- Type Query
  : 변환할 타입을 명확하게 지정할 수 있을 때

```
Query q1 = em.createQuery("SELECT c.firstname, c.lastname FROM Cust c");
List resultList = q1.getResultList();
for (Object o: resultList) {
   Object[] result  = (Object[]) o;
   System.out.println("firstname:" + result[0];
   System.out.println("lastname:"+ resulr[1];
   ....
}


TypedQuery<Cust> q2 = em.createQuery("SELECT c FROM Cust c", Cust.class);
List<Cust> resultList = q2.getResultList();
for(Cust c : resultList) {
   System.out.println("Cust" + cust);
}
```

2. JPQL – Runing JPA Query

객체 지향 쿼리 언어로 엔티티 객체를 대상으로 쿼리 하며 특정 DBMS에 의존 하지 않으나 결국 SQL로 변환 한다.

01. 기본 문법

- 대소문자 구분
  : JPQL Keyword는 대소문자를 구분 하지 않음

- @Entity이름 사용
  : Class 명이 아닌 @Entity(name="xxx")로 지정한 이름

- Identification variable 사용 (Alias)
  : SELECT CUST.CUST_NO FROM CUSTOMER AS CUST
  : hibernate는 HQL(Hibernate Query Language)를 사용 하므로 SELECT CUST_NO FROM CUSTOMER AS CUST에서 CUST.CUST_NO를 CUST_NO로 사용 해도 됨

- SELECT 문

```
SELECT ... FROM ...
[WHERE ...]
[GROUP BY ... [HAVING ...]]
[ORDER BY ...]
```

- UPDATE 문

```
UPDATE .... SET ..... [ WHERE .... ]
```

- DELETE 문

```
DELETE FROM .... [ WHERE .... ]
```

02. JPA 쿼리 실행Query

- createQuery
  : JPQL 문을 실행 하기 위햐서 Query Instance생성
- Select Query
  : entitymanager.createQuery(string query);
  : Query.getSingleResult : 정확히 하나의 결과 객체가 예상:
  -> 없으면 : NonUnuqueResultException, NoResultException
  : Query.getResult : 복수의 결과가 예상
- Select TypeQuery
  : entitymanager.createQuery(string query, retrun.class[돌려받을 객체]);
  : Query.getSingleResult : 정확히 하나의 결과 객체가 예상
  -> 없으면 : NonUnuqueResultException, NoResultException
  : Query.getResult : 복수의 결과가 예상
- UPDATE or DELECT
  : Query.executeUpdate : UPDATE , DELETE 쿼리만 실행
  : entitymanager.createOuery(String Query).executeUpdate();

3. JPQL – Query Parameters In JPA

쿼리 매개 변수를 사용 하면 다른 매개 변수 값(인수)으로 동일한 쿼리를 여러 번 실행하면 반복 된 쿼리 컴파일이 필요하지 않으므로 재사용 됨.

01. 이름 기준 매개 변수 (: name)

- WHERE 조건 값에 :name 사용

: setParameter를 이용해서 바인딩,
: Method Chain 방식으로 설계되어 연속으로 사용 할 수 있음

```
public Country getCountryByName(EntityManager em, String name) {
   TypedQuery<Country> query = em.createQuery(
       "SELECT c FROM Country c WHERE c.name = :name", Country.class);
   return query.setParameter("name", name).getSingleResult();
}
```

02. 위치 기준 매개변수 (? index)

- WHERE 조건 값에 ? index 사용

: setParameter를 이용해서 바인딩,
: Method Chain 방식으로 설계되어 연속으로 사용 할 수 있음

```
public Country getCountryByName(EntityManager em, String name) {
   TypedQuery<Country> query = em.createQuery(
       "SELECT c FROM Country c WHERE c.name = ?1", Country.class);
   return query.setParameter(1, name).getSingleResult();
}
```

03. Criteria 쿼리 매개변수

- JPA Criteria API를 사용 하여 Object(유형 또는 수퍼 인터페이스) 사용

04. Literals 매개변수 (+ name +)

- WHERE 조건 값에 + name + 사용

: 쿼리에 문자열이 포함 되어서 재 사용 할 수 없음

```
public Country getCountryByName(EntityManager em, String name) {
   TypedQuery<Country> query = em.createQuery(
       "SELECT c FROM Country c WHERE c.name = '" + name + "'",
       Country.class);
   return query.getSingleResult();
}
```

05. 페이징 API

- getFirstResult(int startPosition)

: 조회 시작 위치(0부터 시작한다)

- getMaxResults(int maxResult)

: 조회할 데이터 수

```
TypedQuery<Cust> query = em.createQuery("SELECT c FROM Cust c ORDER BY c.custNm DESC",
                                        Cust.class);
query.setFirstResult(10)'
query.setMaxResults(20);
Query.getResultList();
```

4. JPQL – SELECT 절 - Projection

SELECT 절에 조회 대상을 지정 하는 것을 의미 하며 엔티티, 임베디드, 스칼라 타입(숫자, 문자 등 기본 데이터 타입)이 있음

01. 엔티티 타입

- 엔티티를 지정 하는 것으로 연속성 컨텍스트에서 관리됨

```
SELECT c FROM Cust c;
SELECT c.custNm FROM Cust c
```

02. 임베디드 타입

- 임베디드 타입은 객체가 아닌 값이므로 조회의 시작점이 될 수 없음
- 영속성 컨텍스트에서 관리 되지 않음

```
String addressQuery = "SELECT a FROM Address a";   -> 잘못된 쿼리

String addressQuery = "SELECT c.address FROM Cust c";
```

03. 스칼라 타입

- 숫지, 문자, 날짜와 같은 기본 데이터 타입

```
List<String> custNm = em.createQuery("SELECT custNm FROM Cust c", String.class)
                        .getResultList();

: SELECT DISTINCT custNm FROM Cust c;
```

04. 여러 값 조회

- Query를 사용, 영속성 컨텍스트 관리 됨

```
Query query = em.createQuery("SELECT c.custNm, c.age FROM Cust c");
```

```
List resultList = query.getResultList();
Iterator iterator = resultList.iterator()'
while (iterator.hasNext()) {
    Object[] row = (Object[]) iterator.next();
    System.out.println((String) row[0]);
    System.out.println((Integer) row[1])
}
```

```
List<Object[]> resultList = query.getResultList();
for(Object[] row : resultList ) {
    System.out.println((String) row[0]);
    System.out.println((Integer) row[1])
}
```

```
Query query = em.createQuery("SELECT c.cust, c.address, c.count FROM Cust c");
```

```
List<Object[]> resultList = query.getResultList();
for(Object[] row : resultList ) {
    Cust  r  = (Cust) row[0];
    Address  a = (Address) row[1];
    int count = (integer) row[2];
}
```

05. DTO로 변환

- Full Package를 지정, 조회 순서와 일치하는 생성자 필요
- TypedQuery 사용

```
TypedQuery<CustDTO> query = em.createQuery("SELECT NEW co.kr.cust..CustDTO(c.custNm,
                                                                           c.age) FROM Cust c", CustDTO.class);
```

```
List<CustDTO> resultList = query.getResultList();
```

https://www.objectdb.com/java/jpa/query/jpql/select

5. JPQL – Query 구조 (Path Expression, SubQuery .. 등)

.(점)을 찍어서 객체 그래프를 탐색 하는 것으로 묵시적 조인

01. 용어

- 상태필드 (state field) : 값 저장 필드
- 연관필드 (association field) : 연관관계 , 임베디드 필드
  : 단일 연관 : @ManyToOne, @OneToOne
  : 복합 연관 : @OneToMany, @ManyToMany

```
SELECT c.custNm FROM Cust c
                JOIN c.ENTR e
```

명시적 조인 : join을 직접 적어 주는 것
묵시적 조인 : 경로 표현식에 의해서 묵시적으로 join

02. 묵시적 조인

```
SELECT e.products FROM Entr e;

- 컬랜션 값 연관 탐색
SELECT s.svcNm FROM Entr e JOIN e.products s
```

03. 서브 쿼리

- WHERE, HAVING 절에만 사용, SELECT, FROM 절 사용 불가
- Hibernate 에서는 SELECT 절에 사용 가능
- 서브 쿼리 함수
  : [NOT] EXISTS (subquery)
  : [ALL | ANY | SOME] (subquery)
  : [NOT] IN (subquery)

05. 조건식

- 논리식 : ANF, OR, NOT
- 비교식 : = , > , >= , < , <=, <>
- Between, IN, Like, NULL 비교
- 컬랙션의 맴버 식
- 스칼라 식 : 숫자, 문자, 날짜
- CASE 식, 사용자 정의 함수 등 .....

6. JPQL – Named 쿼리

Application 로딩 시점에 JPQL 문법을 체크 하여 미리 파싱 하야서 제사용 하므로 성능상의 이점이 있음, @NameQuery 또는 xml 문서 작성

01. 동적 쿼리
- em.createQuery("select ...") 처럼 문자로 완성해서 직접 넘기는 것

02. 정적 쿼리
- 변경 할 수 없으며 미리 정의한 쿼리에 이름을 부여해서 필요 할 때 사용

```
// NamedQuery 설정
@Entity @NamedQuery (
      name ="Member.findByUsername",
      query = "select m from Member m where m.username = :username")
public class Member{ ... }

// NamedQuery 사용시
List<Member> resultList = em.createNamedQuery("Member.findByUsername", Member.class)
                                    .setParameter("username", "corn").getResultList();


// 다중 NamedQuery 설정
@Entity
NamedQueries({
      @NamedQuery ( name ="Member.findByUsername",
                    query = "select m from Member m where m.username = :username"),
      @NamedQuery ( name ="Member.findByTeamId",
                    query = "select m from Member m where m.teamId = :teamId") })
public class Member{ ... }
```

7. QueryDSL

문자가 아닌 코드를 사용해서 문자 쿼리와 비슷하게 개발 할 수 있는 프로젝트
Querydsl이 제공하는 플루언트(Fluent) API를 이용해서 쿼리를 생성

01. 개요

- Open Source Project로 HQL(hibernate Query Language)을 코드로 작성할 수 있도록 해주는 프로젝트에서 현재는 JPA, JDP, JDBC, 몽고 DB 등을 지원
- 필요 라이브러리
  : querydsl-jpa : QueryDSL JPA 라이브러리
  : querydsl-apt : 쿼리 타입을 생성 할 때 필요한 라이브러리
- Querydsl은 JPQL과 Criteria 쿼리를 모두 대체할 수 있다
- 생성

02. 장점

- IDE의 코드 자동 완성 기능 사용
- 문법적으로 잘못된 쿼리를 허용하지 않음
- 도메인 타입과 프로퍼티를 안전하게 참조할 수 있음
- 도메인 타입의 리팩토링을 더 잘 할 수 있음

03. Maven 통합

- pom.xml에 환경 설정을 해야 함
  : mvn compile 을 하면 target/generated-sources에 Qxxxx.java로 시작하는 쿼리 타입

```
<dependency>
   <groupId>com.querydsl</groupId>
   <artifactId>querydsl-apt</artifactId>
  <version>${querydsl.version}</version>
  <scope>provided</scope>
</dependency>
<dependency>
  <groupId>com.querydsl</groupId>
  <artifactId>querydsl-jpa</artifactId>
 <version>${querydsl.version}</version>
</dependency>
```

```
<project>
 <build>
 <plugins>
  ...
  <plugin>
   <groupId>com.mysema.maven</groupId>
   <artifactId>apt-maven-plugin</artifactId>
   <version>1.1.3</version>
   <executions>
    <execution>
     <goals>
      <goal>process</goal>
     </goals>
     <configuration>
      <outputDirectory>target/generated-sources/java</outputDirectory>
      <processor>com.querydsl.apt.jpa.JPAAnnotationProcessor</processor>
     </configuration>
    </execution>
   </executions>
  </plugin>
  ...
 </plugins>
 </build>
</project>
```

ABACUS

7. QueryDSL - 사용하기

01. 구현체 생성

- 변수와 Query 구현체 생성

: Customer와 동일한 패키지에 QCustomer라는 이름을 가진 쿼리 타입 생성

```
@Entity
public class Customer {
    private String firstName;
    private String lastName;

    public String getFirstName(){
        return firstName;
    }

    public String getLastName(){
        return lastName;
    }

    public void setFirstName(String fn){
        firstName = fn;
    }

    public void setLastName(String ln) {
        lastName = ln;
    }
}
```

 # 결과 조회

- **uniqueResult() : 오직 한 건만 조회. 없으면 null, 복수건이면 오류 발생**
- **singleResult(): uniqueResult()와 동일 단 하나이상이면 처음 한 건만 반환됨**
- **List() " 하나이상, 없으면 빈 Collection 반환**

02. 쿼리 타입 사용 하기

- JAP API를 사용 하려면 JPAQuery 인스턴스 사용

```
JPAQuery query = new JPAQuery(entityManager);
```

- firstName 프로퍼티가 Bob인 Customer를 조회

```
QCustomer customer = QCustomer.customer;
JPAQuery query = new JPAQuery(entityManager);
Customer bob = query.from(customer        // 쿼리 대상 (소스)
  .where(customer.firstName.eq( " Bob"))  // 조회 조건
  .uniqueResult(customer);                // 1개 결과 리턴
```

- 여러 소스로부터 쿼리를 만들고 싶다면

```
QCustomer customer = QCustomer.customer;
QCompany company = QCompany.company;
query.from(customer, company);

query.from(customer)
    .where(customer.firstName.eq("Bob"), customer.lastName.eq("Wilson"));

query.from(customer)
    .where(customer.firstName.eq("Bob").and(customer.lastName.eq("Wilson")));

// JPQL 쿼리 사용
from Customer as customer
    where customer.firstName = "Bob" and customer.lastName = "Wilson"

query.from(customer)
    .where(customer.firstName.eq("Bob").or(customer.lastName.eq("Wilson")));
```

- product.price.between(10000,2000)
- customer.name.contains("홍길") => like '%홍길%'
- customer.name.startWith("홍길") => like '홍길%'

7. QueryDSL – JOIN, 일반용법

01. JOIN

- JPQL의 INNER JOIN, JOIN, LEFT JOIN, RIGHT JOIN

```
QCat cat = QCat.cat;
QCat mate = new QCat("mate");
QCate kitten = new QCat("kitten");
query.from(cat)
    .innerJoin(cat.mate, mate)
    .leftJoin(cat.kittens, kitten)
    .list(cat);

JPQL :
    from Cat as cat
    inner join cat.mate as mate
    left outer join cat.kittens as kitten

=========================================================
// ON
query.from(cat)
    .leftJoin(cat.kittens, kitten)
    .on(kitten.bodyWeight.gt(10.0))
    .list(cat);

JPQL :
    from Cat as cat
    left join cat.kittens as kitten
    on kitten.bodyWeight > 10.

// FETCH
query.from(cat)
    .leftJoin(cat.kittens, kitten).fetch()
    .on(kitten.bodyWeight.gt(10.0))
    .list(cat);
```

02. 일반 용법

- from: 쿼리 소스

- innerJoin, join, leftJoin, rightJoin, on: 조인.
  조인 메서드에서 첫 번째 인자는 조인 소스
  두 번재 인자는 대상(별칭).

- where: 쿼리 필터(조회조건)를 추가한다.
  가변인자나 and/or 메서드를 이용함 .

- groupBy: 가변인자 형식의 인자를 기준으로 그룹.

- having: Predicate 표현식을 이용해서 "group by" 그룹핑의 필터.

- orderBy: 정렬 표현식을 이용해서 정렬 순서를 지정.
  숫자나 문자열에 대해서는 asc()나 desc()를 사용
  OrderSpecifier에 접근하기 위해 다른 비교 표현식을 사용.

- limit, offset, restrict: 결과의 페이징을 설정.
  limit은 최대 결과 개수,
  offset은 결과의 시작 행,
  restrict는 limit과 offset을 함께 정의.

7. QueryDSL – 정렬, 그룹핑, 삭제, 수정, 서브쿼리

01. 정렬

- .asc(), .desc()

```
QCustomer customer = QCustomer.customer;
query.from(customer)
    .orderBy(customer.lastName.asc(), customer.firstName.desc())
    .list(customer);

JPQL :
    from Customer as customer
    order by customer.lastName asc, customer.firstName desc
```

02. 그룹핑

- groupBy, having(결과제한)

```
query.from(customer)
    .groupBy(customer.lastName)
    .list(customer.lastName);

JPQL :
    select customer.lastName
    from Customer as customer
    group by customer.lastName
```

03. 삭제

- 삭제 후 삭제된 엔티티 개수 리턴

```
QCustomer customer = QCustomer.customer;

// delete all customers
new JPADeleteClause(entityManager, customer).execute();

// delete all customers with a level less than 3
new JPADeleteClause(entityManager,
                    customer).where(customer.level.lt(3)).execute();
```

04. 수정

- 생성자의 두번째 파라미터는 수정할 엔티티 대상
- 수정된 엔티티 개수 리턴

```
QCustomer customer = QCustomer.customer;
// rename customers named Bob to Bobby
new JPAUpdateClause(session, customer).where(customer.name.eq("Bob"))
    .set(customer.name, "Bobby")
    .execute();;
```

05. 서브쿼리

- JPASubQuery를 사용
- 서브쿼리를 만들기 위해 from 메서드로 쿼리 파라미터를 정의,
- unique나 list를 이용

```
QDepartment department = QDepartment.department;
QDepartment d = new QDepartment("d");
query.from(department)
    .where(department.employees.size().eq(
        new JPASubQuery().from(d).unique(d.employees.size().max())
    )).list(department);
```

THANKS

ABACUS

www.iabacus.co.kr

Tel. 82-2-2109-5400

Fax. 82-2-6442-5409