JPA 프로그래밍

- JDBC만을 이용한 영속 계층

JDBC(Java Database Connectivity)는 Java에서 데이터베이스에 접속하고 데이터 처리 구현에 사용되는 Java API 이다. JDBC의 장점은 각각의 DBMS가 제공하는 JDBC 드라이버를 이용하여 DBMS의 종류와 상관없이 하나의 JDBC API를 이용하여 DB 작업을 처리할 수 있다는 것이다. 그러나 JDBC만을 이용하여 개발 할 경우에는 불편한 점이 꽤 많다.

- 간단한 SQL을 실행함에도 중복된 코드를 반복적으로 사용
- DB에 따라 일관성 없는 정보를 가진채로 Checked Exception(SQLException) 처리
- 연결과 같은 공유 자원을 제대로 반환하지 않으면 시스템의 자원 부족 현상 발생

그래서 이런 복잡하고 번거로운 작업을 해결하기 위해 데이터베이스와 연동되는 시스템을 빠르게 개발할 수 있는 Persistence Framework를 사용하기 시작하였다.

- SQL Mapper을 이용한 영속 계층

SQL Mapper는 객체와 SQL을 매핑하여 데이터를 객체화 하는 Persistence Framework 이다. 즉, 테이블과 객체 간의 관계를 매핑하는 것이 아닌 직접 작성한 SQL문의 결과와 객체의 필드를 매핑하는 것이 메인 컨셉이다. SQL Mapper를 이용하면 기존에 JDBC 만을 이용했을 때보다 코드가 간결해지고 유지보수성이 향상된다. 그러나 SQL Mapper가 JDBC 만을 사용했을 때보다 많은 불편함을 해소해 주었음에도 여전히 문제가 있다.

- SQL을 직접 작성하기에 특정 DB에 종속적
- 비슷한 CRUD SQL 작성 및 DAO를 반복적으로 개발
- 테이블 필드 변경 시 유지보수하기 힘듦

즉, 코드상에서는 SQL과 JDBC API를 분리했어도 논리적으로는 강한 의존성을 가지고 있게 되며 SQL에 의존적인 개발을 해야 하는 문제는 여전히 발생한다.

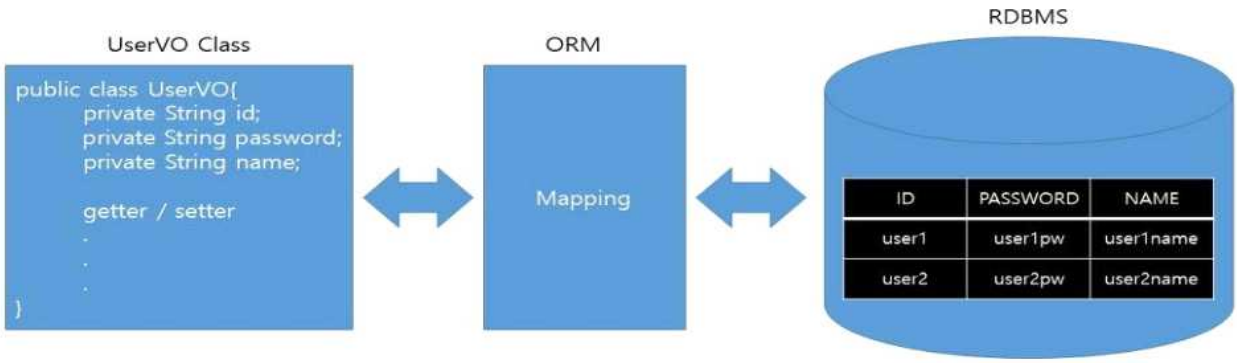
- ORM을 이용한 영속 계층

ORM이란 Object Relational Mapping, 객체-관계 매핑의 줄임말로 OOP에서 객체를 구현한 클래스와 RDBMS에서 사용하는 테이블을 자동으로 매핑하는 것을 의미한다. ORM은 객체 간의 관계를 바탕으로 정해진 메서드를 호출하면 관련 SQL문을 자동으로 생성하여 직관적으로 데이터를 조작하게 하는 방법으로 개발자의 불편함을 해소한다

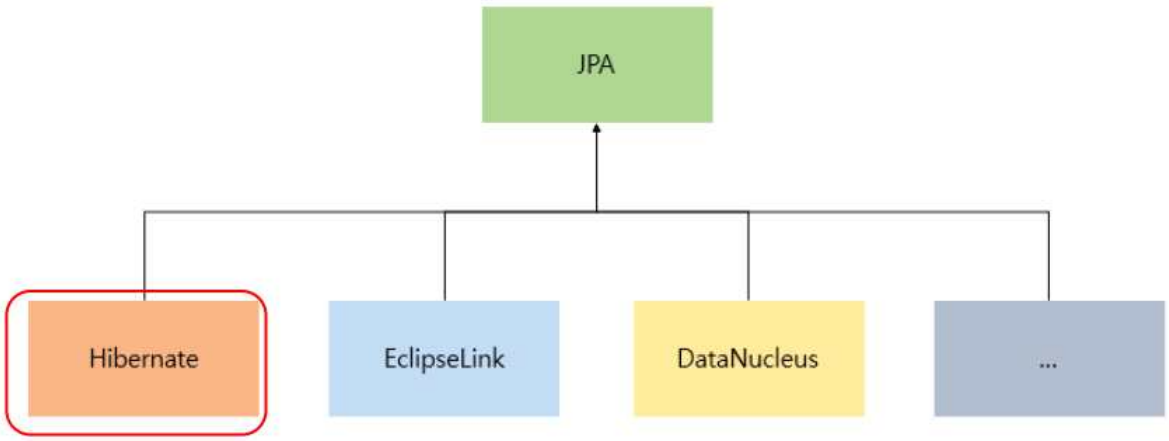
Java ORM의 대표적인 기술은 바로 JPA가 있다. 원래 JPA 기술이 나오기 전까지는 EJB(Entity Bean)이라는 Java 표준 기술이 있었다. 하지만 워낙 복잡하다 보니 Gavin King이라는 개발자가 하이버네이트라는 오픈 소스를 만들어냈고 Java 진영에서 Gavin King을 영입하여 하이버네이트 기반의 Java ORM 표준인 JPA를 만들었다.

JPA 란?

데이터베이스는 객체 구조와는 다른 데이터 중심의 구조를 가지므로 객체를 데이터베이스에 직접 저장하거나 조회할 수 없다. 따라서 개발자가 객체지향 애플리케이션과 데이터베이스 중간에서 SQL과 JDBC API를 사용해서 변환 작업 (객체 → 데이터베이스)을 직접 해주어야 한다.
JPA는 Java Persistence API 의 약자로서, 자바 객체와 관계형 데이터베이스의 데이터(테이블)를 매핑하기 위해 사용되는 ORM(Object-Relational Mapping) 기술의 Java 표준 명세이다.
클래스와 테이블은 기존부터 호환 가능성을 두고 만들어진 것이 아니므로 불일치가 발생하는데 이를 ORM을 통해서 객체 간의 관계를 바탕으로 SQL문을 자동 생성하여 불일치를 해결한다. ORM을 사용하면 SQL문을 구현할 필요 없이 객체를 통해 간접적으로 데이터베이스를 조작할 수 있게 된다.



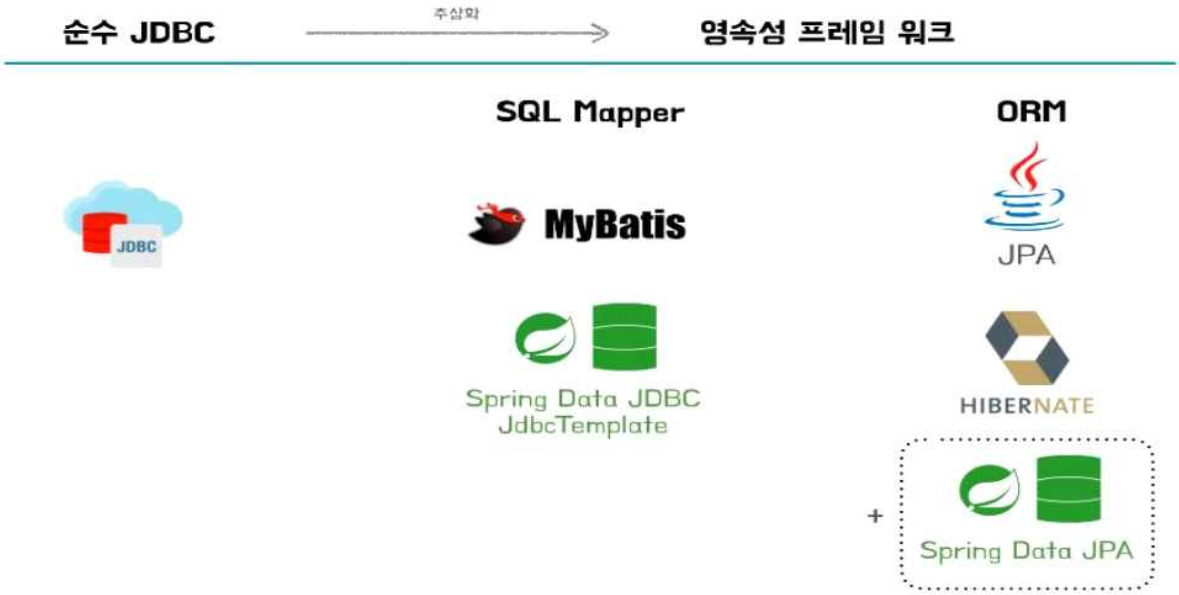
JPA 는 Java ORM에 대한 API 표준 명세이며, 인터페이스의 모음이다. 따라서 구현체가 없으므로, 사용하기 위해서는 ORM프레임워크를 선택해야 한다. 다양한 프레임워크가 존재하지만 가장 대중적인 것은 하이버네이트이다.



Entity 란 DB 테이블의 데이터를 OOP 스럽게 다루기 위해 사용하는 DB 테이블의 모델링 클래스이이다. DB 테이블의 컬럼들에 매핑되는 필드를 선언하고 기본 생성자, setter, getter 등의 메서드 그리고 Primary Key에 매핑되는 Identity 역할의 Key 필드(@Id)를 선언한다.

Persistence Framework

JDBC 프로그래밍에서 경험하게 되는 복잡함이나 번거로움 없이 간단한 작업만으로 데이터베이스와 연동되는 시스템을 빠르게 개발할 수 있다. 일반적으로 SQL Mapper와 ORM으로 나눠진다.



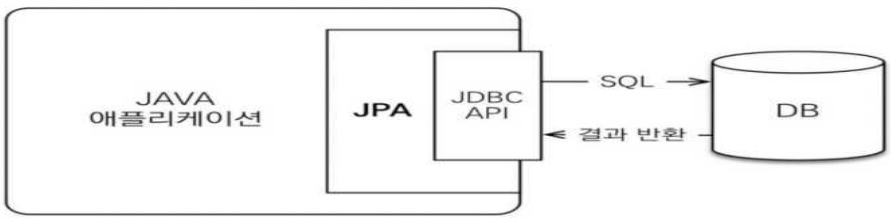
[SQL Mapper]

직접 작성한 SQL 문장으로 데이터베이스 데이터를 다룬다.
Mybatis, JdbcTemplates(Spring)

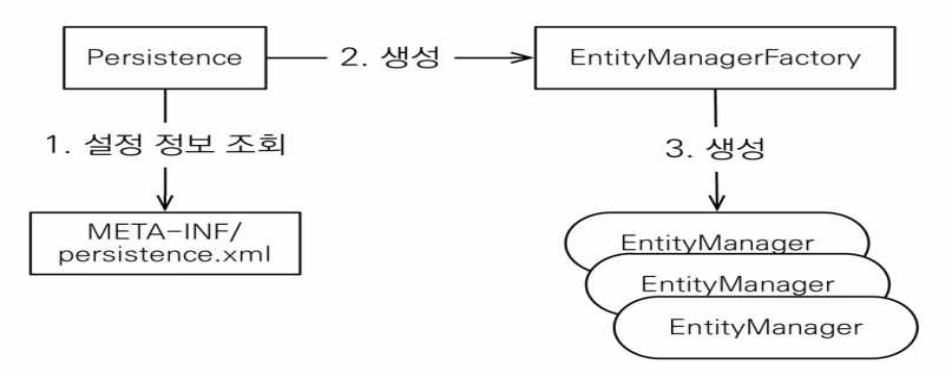
[ORM]

객체를 통해서 간접적으로 데이터베이스의 데이터를 다룬다.
객체와 관계형 데이터베이스의 데이터를 자동으로 맵핑시킨다.
JPA, Hibernate 등

JPA는 애플리케이션과 JDBC 사이에서 동작한다. JPA 내부에서 JDBC API를 사용하여 SQL을 호출하여 DB와 통신한다. 개발자가 ORM 프레임워크에 저장하면 적절한 INSERT SQL을 생성해 데이터베이스에 저장해주고, 검색을 하면 적절한 SELECT SQL을 생성해 결과를 객체에 매핑하고 전달한다.



EntityManager 객체 생성과 Entity 클래스 정의



1. persistence.xml 파일을 통해서 JPA 관련 정보를 설정한다.
2. EntityManagerFactory 객체를 생성한다.(<persistence-unit> 태그의 이름 정보 지정)
3. EntityManager 객체를 생성하여 Entity 객체를 영속성 컨텍스트(Persistence Context)를 통해 관리한다.

EntityManagerFactory

데이터베이스와 상호 작용을 위한 EntityManager 객체를 생성하기 위해 사용되는 객체로서
애플리케이션에서 한 번만 객체를 생성하고 공유해서 사용한다.
Thread-Safe 하므로 여러 스레드에서 동시에 접근해도 안전하다.
EntityManagerFactory 객체를 통해 생성되는 모든 EntityManager 객체는 동일한 데이터베이스에 접속한다.

EntityManager

Entity 객체를 관리하는 객체이다
데이터베이스에 대한 CRUD 작업은 모두 영속성 컨텍스트를 사용하는 EntityManager 객체를 통해 이루어진다. 동시성의 문제가 발생할 수 있으니 여러 스레드가 공유하지 않는다.
모든 데이터 변경은 트랜잭션 안에서 이루어져야 한다.(트랜잭션을 시작하고 처리해야 함)
-------- 데이터베이스의 논리적인 작업 단위

flush()

영속성 컨텍스트(Persistence Context)의 변경 내용을 데이터베이스에 반영한다.
일반적으로는 flush() 메서드를 직접 사용하지는 않고, Java 애플리케이션에서 커밋 명령이 들어왔을 때 자동으로 실행된다.

detach()

특정 Entity를 준영속 상태(영속 컨텍스트의 관리를 받지 않음)로 바꾼다.

clear()

Persistence Context를 초기화한다.

close()

Persistence Context를 종료한다.

merge()

준영속 상태의 엔티티를 이용해서 새로운 영속 상태의 엔티티를 반환한다.

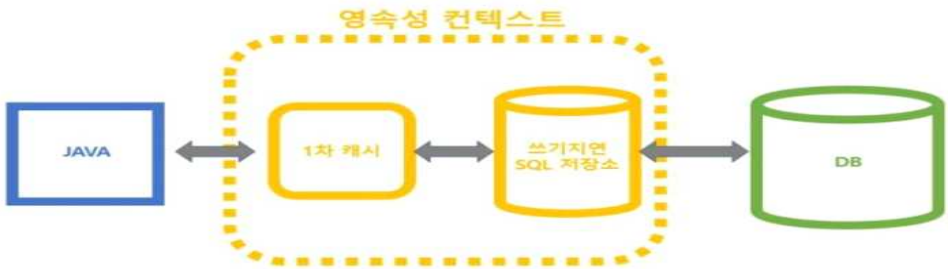
find()
식별자 값(Key 필드)을 통해 Entity를 찾는다.(DB 테이블의 데이터 또는 행을 찾는다.)
persist()
생성된 Entity 객체를 영속성 컨텍스트(Persistence Context)에 저장한다.
remove()
식별자 값을 통해 영속성 컨텍스트(Persistence Context)에서 Entity 객체를 삭제한다.

Entity
JPA에서 엔티티란 DB 테이블에 대응하는 클래스이고 엔티티 객체는 DB 테이블의 데이터(하나의 행)이다. @Entity 어노테이션이 붙은 클래스를 JPA에서는 엔티티라고 부르며, 이 엔티티는 영속성 컨텍스트에 담겨 EntityManager에 의해서 관리된다.

@Entity
클래스 레벨에 적용한다.
해당 클래스를 테이블과 매핑한다고 JPA에게 알려주는 역할을 한다.
해당 어노테이션이 적용된 클래스를 엔티티 클래스라고 한다
@Table
클래스 레벨에 적용한다.
엔티티 클래스에 매핑할 테이블 이름을 JPA에게 알려주는 역할을 한다.
@Table(name = "테이블이름")
해당 어노테이션을 생략하면 클래스 이름에서 모든 글자를 소문자로 변경하여
테이블을 생성한다.
@Id
필드 레벨에 적용한다.
엔티티 클래스의 필드를 테이블의 기본 키 (Primary Key)에 매핑한다.
해당 어노테이션이 적용된 필드를 식별자 필드라고 한다.
@Column
필드 레벨에 적용한다.
해당 필드를 테이블의 지정된 컬럼에 매핑한다.
name 속성으로 매핑할 컬럼의 이름을 지정하며 nullable, unique, length 등도 설정 가능하다.
@Column(name = "매핑할 컬럼명")
name 속성을 생략하거나 @Column 어노테이션을 생략하면 필드명을 그대로 사용해서 컬럼
명으로 매핑한다. 이름이 age인 필드에 매핑 정보를 생략하면, age 컬럼으로 자동 매핑한다.
대소문자를 구분하는 DB를 사용한다면 @Column 을 사용하여 명시적으로 매핑해야 한다.

영속성 컨텍스트(persistence context)

- 어플리케이션과 데이터베이스 사이에 존재하는 논리적인 개념으로 엔티티를 저장하고 관리하는 영역이다.
- EntityManager 객체당 한 개의 영속성 컨텍스트가 사용되며 EntityManager 객체를 통해서만 접근이 가능하다
- 영속성 컨텍스트는 엔티티들을 식별자(id) 값으로 구분하며 1차 캐시에 보관한다.
- 쓰기 지연을 지원하여 커밋하기 전까지는 DB 테이블에 저장하지 않고 SQL 저장소에 관련 SQL 문만 보관하며 트랜잭션을 커밋하는 시점에 영속성 컨텍스트를 플러시하여 보관되어 있던 SQL 문들을 DB 서버에 전송한 후 커밋 처리한다.

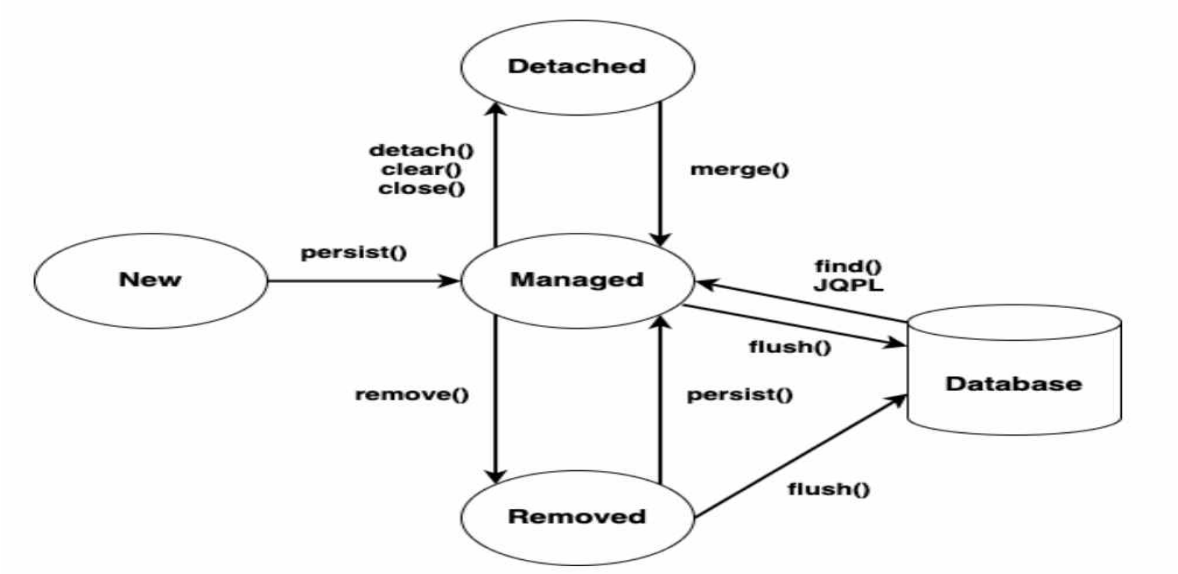


- 플러시한 다음에는 엔티티 상태들을 복사하여 저장한 스냅샷이 존재하며 스냅샷과 엔티티를 비교해 변경된 엔티티를 찾고, 있다면 수정 쿼리를 SQL 저장소에 보관한다.
- 영속성 컨텍스트에 존재하는 엔티티들의 플러시 처리 시점
  - entityManger.flush() 로 플러시 기능을 직접 호출
  - 트랜젝션 커밋(commit) 시 entityManger.flush() 메서드 자동 호출됨
  - JPQL 쿼리 실행 시 entityManger.flush() 메서가 자동 호출됨

엔티티 생명주기

Entity에는 아래와 같은 4가지의 상태가 존재한다.

1. 비영속(new/transient): 영속성 컨텍스트와 전혀 관계가 없는 상태
2. 영속(managed): 영속성 컨텍스트에 저장된 상태
3. 준영속(detached): 영속성 컨텍스트에 저장되었다가 분리된 상태
4. 삭제(removed): 삭제된 상태



엔티티 생성과 저장

1. Java 어플리케이션에서 어떤 엔티티 객체를 생성하여 JPA에게 데이터베이스 저장을 요구하면,
2. 만들어진 엔티티는 1차적으로 영속성 컨텍스트에 저장된다.(1차 캐시)
   그리고, 저장한 엔티티를 데이터베이스에 저장하기 위한 쿼리문을 생성시켜 쓰기 지연 SQL 저장소에 저장한다. 계속해서 엔티티를 넘기면 엔티티들과 쿼리문들은 차곡차곡 영속성 컨텍스트에 저장된다.
3. Java 어플리케이션에서 커밋 명령이 내려지면 영속 컨텍스트에는 자동으로 flush( )가 호출되고,
4. 영속성 컨텍스트의 변경내용을 데이터베이스와 동기(flush)화 한다.

-> SQL 저장소의 쿼리를 실행시킨다.

5. 마지막으로 데이터베이스에게 commit 쿼리문을 명령한다.

엔티티 조회

1. Java 어플리케이션에서 JPA에게 데이터베이스 조회를 부탁하면, 1차적으로 영속성 컨텍스트에서 엔티티를 찾는다.
2. 있으면 Java 어플리케이션에 엔티티를 넘긴다.
3. 영속성 컨텍스트에 없는 엔티티 조회를 요구하면
4. 쿼리문을 사용해 데이터베이스에서 찾아와
5. 영속성 컨텍스트에 엔티티로 저장하고
6. Java 어플리케이션에 그 엔티티를 넘긴다.

엔티티 변경

JPA는 엔티티를 영속성 컨텍스트에 보관할 때, 현재의 상태를 복사해서 저장해 두는데,
이것을 스냅샷이라 한다.
1. Java 어플리케이션에서 커밋 명령이 들어오면, 영속 컨텍스트에는 자동으로 flush( )가 호출되고,
2. 엔티티와 스냅샷을 비교해서 변경된 엔티티를 찾는다.
3. 변경된 엔티티가 있으면 데이터베이스에 변경사항을 저장하기 위해 쿼리를 생성하고,
4. 영속성 컨텍스트의 변경내용을 데이터베이스와 동기(flush)화 한다(SQL 저장소의 쿼리를 실행시킨다).
5. 마지막으로 데이터베이스에게 commit 쿼리문을 명령한다.

이렇게 엔티티의 변경사항을 데이터베이스에 자동으로 반영하는 기능을 변경감지(Dirty Checking)라 한다.

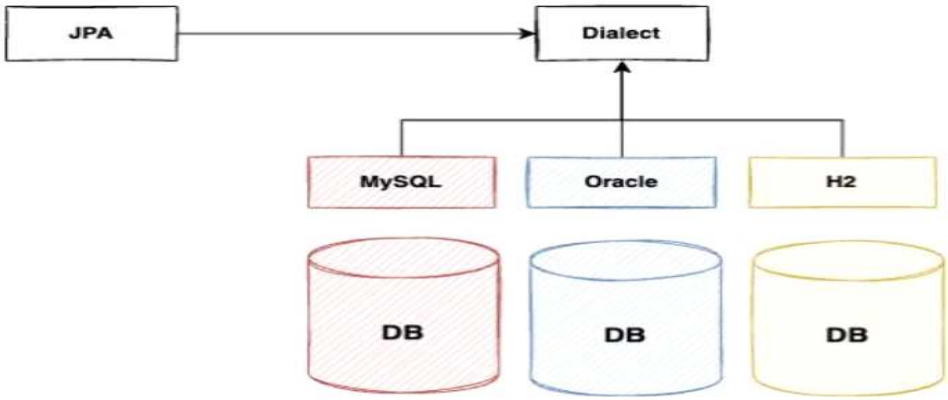
엔티티 삭제

앞의 과정과 마찬가지로, Java 어플리케이션에서 엔티티 삭제 명령이 들어오면,
엔티티를 찾고 쓰기 지연 SQL 저장소에 delete 쿼리를 생성한다.
그리고 Java 어플리케이션에서 커밋 명령이 들어오면, 자동으로 flush( )가 호출되고,
영속성 컨텍스트의 변경내용을 데이터베이스와 동기(flush)화 한다(SQL 저장소의 쿼리를 실행시킨다). 마지막으로 데이터베이스에게 commit 쿼리문을 명령한다

트랜잭션 관리

- JPA를 사용하면, 항상 트랜잭션 안에서 데이터를 변경(CUD)해야 한다.
- 트랜잭션 없이 데이터를 변경하면 예외가 발생한다.
- 트랜잭션을 시작하려면, 엔티티 매니저에서 트랜잭션 API를 받아와야 한다.

영속성 컨텍스트에서 관련 SQL을 데이터베이스에 전달할 때는 데이터베이스별 Dialect(방언)을 적용한다.
Dialect 관련 설정은 persistence.xml 에 다음 내용을 작성한다.



JPA 객체지향 쿼리 언어

- JPQL
- JPA Criteria
- QueryDSL
- 네이티브 SQL
- JDBC API 직접 사용, MyBatis, SpringJdbcTemplate 과 함께 사용

JPQL이란?

JPA 에서 엔티티 객체를 조회할 때 사용하는 객체지향 쿼리이다.
문법은 SQL과 비슷하고 ANSI 표준 SQL이 제공하는 기능을 유사하게 지원한다.
SQL은 데이터베이스 테이블을 대상으로 JPQL은 엔티티 객체를 대상으로 쿼리한다.
JPQL은 SQL을 추상화해서 특정 데이터베이스에 의존하지 않으며 실행 시 SQL로 변환된다

JPA 기본키 매핑

기본키(primary key)를 매핑하는 방법은 2가지로 직접 할당과 자동 생성이 있다.
직접 할당은 엔티티에 @Id 어노테이션만 사용해서 직접 할당하는 것이다.
자동 생성은 엔티티에 @Id와 @GeneratiedValue를 추가하고 원하는 키 생성 전략을 선택한다.
자동 생성 같은 경우에는 MySQL의 AUTO_INCREMENT 같은 기능으로 생성된 값을 기본키로 사용하는 것이다.

- 기본키 자동 생성 전략 - IDENTITY

기본키 생성을 DB에 위임하는 전략이다.
MySQL, PostgreSQL, SQL Server, DB2에서 사용한다.
(MySQL의 AUTO_INCREMENT

- 기본키 자동 생성 전략 - SEQUENCE

유일한 값을 순서대로 생성하는 특별한 데이터베이스 오브젝트이다.
이 시퀀스를 사용해서 기본키를 생성하게 된다.
시퀀스를 지원하는 Oracle, PostgreSQL, DB2, H2 Database에서 사용할 수 있다.

JPA 연관 관계 매핑 기초

- 엔티티들은 대부분은 다른 엔티티들과 연관 관계를 맺는다.
- 데이터베이스 테이블은 외래키(FK)로 JOIN을 이용해서 관계 테이블을 참조
- 엔티티는 객체 참조를 이용해서 연관된 엔티티를 참조
- 연관 관계 매핑이란 데이터베이스 테이블의 외래 키(FK)를 객체의 참조와 매핑하는 것

즉, 데이터베이스 테이블의 외래키를 객체의 참조 관계로 매핑하는 것

[다대일의 단방향 연관 관계]

DB 테이블의 일(1), 다(N) 관계에서 외래키는 항상 다 쪽에 있다.
그러므로 객체 양방향 관계에서 연관 관계의 주인은 항상 다 쪽이다.
객체 간의 연관 관계에서는 단방향 관계이므로 member → team 조회는 가능하지만 반대의 경우는 할 수 없다. Member.team 필드를 통해서 팀을 알 수 있지만, 반대로 팀은 회원을 알 수 없다.

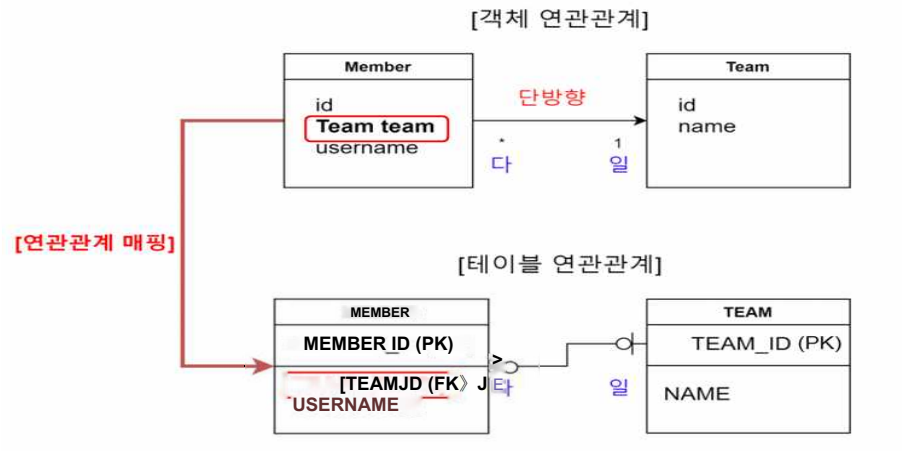
@ManyToOne 이란?

다대일(N:1) 관계라는 매핑 정보이다.

@JoinColumn 이란?

@JoinColumn은 외래키를 매핑할 때 사용한다.
name 속성에는 매핑 할 외래키 이름을 지정한다. 회원과 팀 테이블은 TEAM_ID 외래키로 연관 관계를 맺으므로 이 값을 지정한다.
생략할 수 있다. 생략한다면 외래키를 찾을 때 기본 전략을 사용하게 된다.
기본 전략 : 필드명 + _ + 참조하는 테이블의 기본키 컬럼명

회원 객체의 Member.team 필드와 회원 테이블의 MEMBER.TEAM_ID 외래키 컬럼이 매핑되는 것이다.




```
public class Member {
    @Id
    @Column(name = "MEMBER_ID")
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    private String username;
```

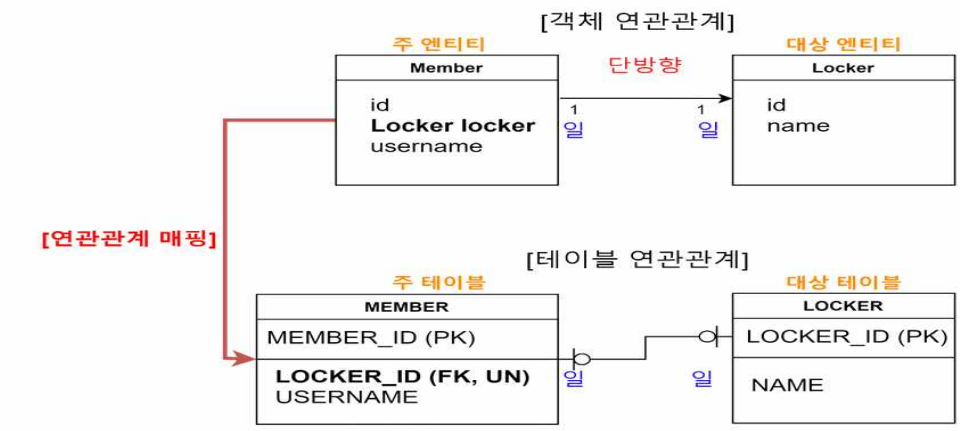
```
    // 연관 관계 매핑
    @ManyToOne
    @JoinColumn(name = "TEAM_ID")
    private Team team;

    @OneToOne
    @JoinColumn(name = "LOCKER_ID")
    private Locker locker;
}

public class Team {
    @Id
    @Column(name = "TEAM_ID")
    private String id;
    private String name;
}
```

TEAM_ID 와 JOIN

LOCKER_ID 와 JOIN

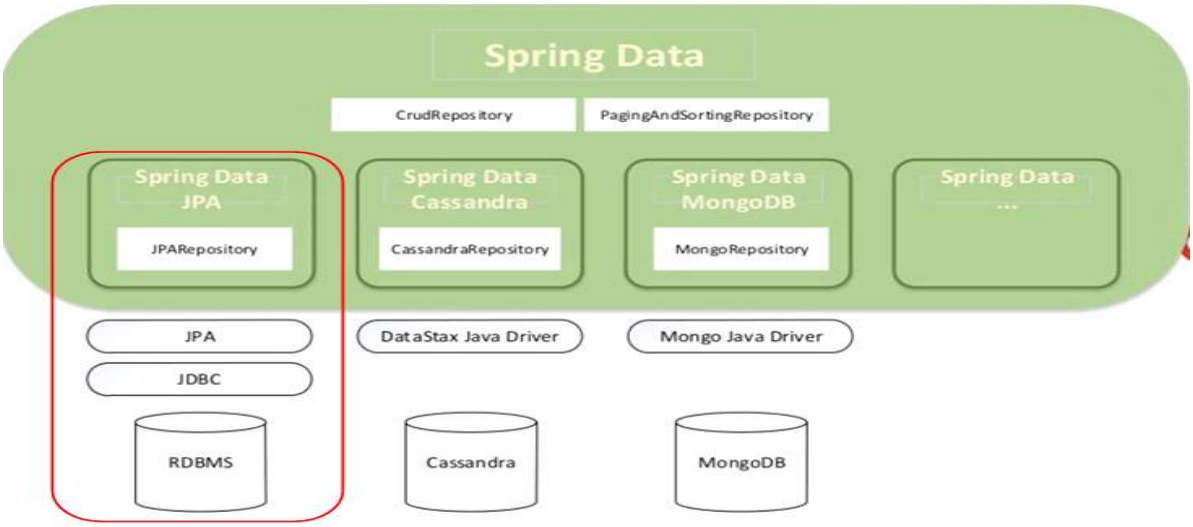
[일대일의 연관 관계]

일대일 관계는 양쪽이 서로 하나의 관계만 맺는다.
ex) 회원은 하나의 사물함만 사용하고, 사물함도 하나의 회원에 의해서만 사용된다.
이때, '주 테이블 = 회원', '대상 테이블 = 사물함'이다.
테이블은 주 테이블이든 대상 테이블이든 외래키 하나만 있으면 양쪽으로 조회할 수 있다.
일대일 관계는 주 테이블이나 대상 테이블 둘 중 어느 곳이나 외래키를 가질 수 있으며 주 테이블이나 대상 테이블 중, 누가 외래키를 가질지 선택해야 한다. 주 테이블에 외래키가 있는 경우 주 객체가 대상 객체를 참조하는 것처럼, 주 테이블에 외래키를 두고 대상 테이블을 참조한다.
외래키를 객체 참조와 비슷하게 사용할 수 있다. 주 테이블이 외래키를 가지고 있으므로, 주 테이블만 확인해도 대상 테이블과 연관 관계가 있는지 알 수 있다



Spring Data

Spring Data의 목적은 기본 데이터 저장소의 특수한 특성을 유지하면서 데이터 접근을 위한 친숙하고 일관된 Spring 기반의 프로그래밍 모델을 제공하는 프로젝트이다.
Spring Data는 데이터 접근 기술로서 relational and non-relational database, map-reduce 프레임워크, 클라우드 기반의 서비스를 쉽게 사용할 수 있도록 해준다.
Spring Data는 데이터베이스와 관련된 많은 하위 프로젝트(Spring Data JPA, Spring Data REST, …)를 포함하는 포괄적인 프로젝트이다.

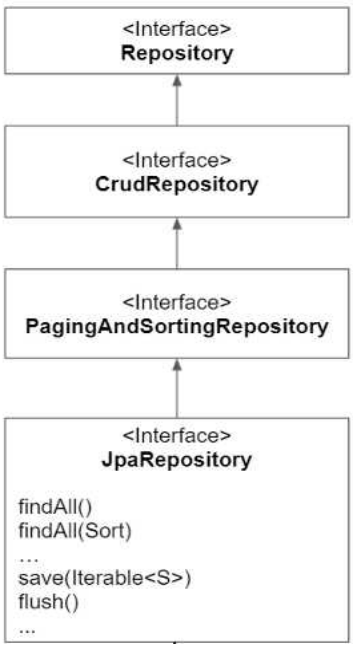


Spring Data JPA

Spring Data JPA 는 Spring Framework에서 JPA를 편리하게 사용할 수 있도록 지원하는 프로젝트로서 CRUD 처리를 위한 공통 인터페이스를 제공한다.

Repository 개발 시 인터페이스만 작성해도 실행 시점에 Spring Data JPA가 구현(자식) 객체를 동적으로 생성해서 주입시키므로 데이터 접근 계층을 개발할 때 구현 클래스 없이 인터페이스만 작성해도 개발을 완료할 수 있도록 지원한다.

Spring Data JPA를 사용하기 위해 'JpaRepository<T, ID>' 인터페이스를 상속한 Repository 인터페이스를 정의한다. 단지 인터페이스를 상속했을 뿐인데, 기본적인 메서드를 이미 정의한 상태이며 정의한 인터페이스를 구현할 필요도 없다. Spring FW가 알아서 해준다.



쿼리 메서드

위의 메서드들은 모든 엔티티에 대해 공통으로 쓰일 수 있는 메서드를 제공하지만, 사실 비즈니스 로직을 다루는 것은 그리 간단하지 않다. 조건을 지정하여 조회하거나, 제거하거나 저장할 수 있는 기능들을 커스터마이징 해야 한다.

Spring Data JPA는 Repository를 커스터마이징 하기 위해 쿼리 메서드 기능을 제공한다.
1. 메서드 이름으로 쿼리 생성 -> 간단한 쿼리 처리 시 좋음
2. @Query 안에 JPQL 정의 -> 복잡한 쿼리 처리 시 좋음
3. 메서드 이름으로 JPA NamedQuery 호출 (잘 안쓰임)

일반적으로 많이 사용하는 쿼리 메소드

키워드	예	생성되는 SQL
And	findByLastnameAndFirstname	where x.lastname=?1 and x.firstname=?2
Or	findByLastnameOrFirstname	where x.lastname=?1 or x.firstname=?2
Between	findByStartDateBetween	where x.startDate between ?1 and ?2
LessThan	findByAgeLessThan	where x.age < ?1
LessThanEqual	findByAgeLessThanEqual	where x.age <= ?1
GreaterThan	findByAgeGreaterThan	where x.age > ?1
GreaterThanEqual	findByAgeGreaterThanEqual	where x.age >= ?1
After	findByStartDateAfter	where x.startDate > ?1
Before	findByStartDateBefore	where x.startDate < ?1
IsNull	findByAgeIsNull	where x.age is null
IsNotNull, NotNull	findByAgeIsNotNull	where x.age is not null
Like	findByFirstnameLike	where x.firstname like ?1
Not Like	findByFirstnameNotLike	where x.firstname not like ?1
StartingWith	findByFirstnameStartingWith	where x.firstname like ?1||'%'
EndingWith	findByFirstnameEndingWith	where x.firstname like '%'||?1
Containing	findByFirstnameContaining	where x.firstname like '%'||?1||'%'
OrderBy	findByAgeOrderByLastnameDesc	where x.age=?1 order by x.lastname desc
Not	findByLastnameNot	where x.lastname <> ?1
In	findByAgeIn(Collection<Age> ages)	where x.age in ?1

Spring Data JPA가 제공하는 JpaRepository 인터페이스를 상속받으면 상속하는 것만으로도 기본적인 CRUD 기능을 사용할 수 있다. 그런데 이름으로 검색하거나, 가격으로 검색하는 기능은 공통으로 제공할 수 있는 기능이 아니다. 따라서 이런 경우에는 **쿼리 메서드** 기능을 사용하거나 **@Query** 를 사용해서 직접 쿼리를 실행한다.

쿼리 메서드를 이용하는 방법과 @Query로 JPQL을 사용하는 방법 모두 복잡한 동적 쿼리에는 좀 부족한 편이다. 복잡한 동적 쿼리는 QueryDSL을 사용하는 것이 좋다.

Spring Boot Application 에 설정된 어노테이션 설명

@SpringBootApplication 어노테이션은 스프링 부트의 가장 기본적인 설정을 선언한다.
내부적으로 @ComponentScan과 @EnableAutoConfiguration을 설정한다.

@ComponentScan 어노테이션은 스프링 3.1부터 도입된 어노테이션이며 스캔 위치를 설정한다.
(프로젝트 생성시 기본으로 만들어지는 패키지의 하위에 클래스들을 두는 경우에는 생략 가능)

@EnableJpaRepositories 어노테이션은 JPA Repository들을 활성화하기 위한 어노테이션이다.

@EntityScan 어노테이션은 엔티티 클래스를 스캔할 곳을 지정하는 데 사용한다.