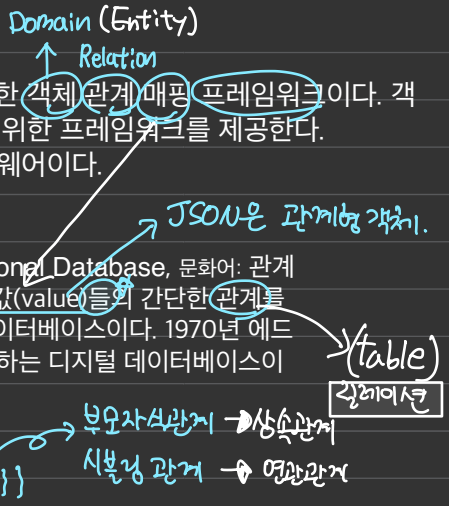


하이버네이트 ORM(Hibernate ORM)은 자바 언어를 위한 객체/관계 매핑 프레임워크이다. 객체 지향 도메인 모델을 관계형 데이터베이스로 매핑하기 위한 프레임워크를 제공한다. 하이버네이트는 GNU LGPL 2.1로 배포되는 자유 소프트웨어이다.

Key vs Index
문자 숫자
map Collection

관계형 데이터베이스(關係形 Database, Relational Database, 문화어: 관계 자료기지, 관계형자료기지, **RDB**)는 키(key)와 값(value)들의 간단한 관계를 테이블화 시킨 매우 간단한 원칙의 전산정보 데이터베이스이다. 1970년 에드거 F. 커드가 제안한 데이터 관계형 모델에 기초하는 디지털 데이터베이스이다.[1]

인덱이션 = 테이블을 또한다.
↓ 관계 X

http://databaser.net/moniwiki/wiki.php/%EC%98%A4%EB%9D%BC%ED%81%B4%EC%9D%B4%EB%9E%80%EB%AC%B4%EC%97%87%EC%9D%B8%EA%B0%80?action=body

부동 URL
↑ ↑
Rest API
대표적인 상태 전송 URL.

RES : 상태
REST : 전송
REST API : URL

2. REST 구성
쉽게 말해 REST API는 다음의 구성으로 이루어져있습니다. 자세한 내용은 밑에서 설명하도록 하겠습니다.

- 자원(RESOURCE) - URI
- 행위(Verb) - HTTP METHOD
- 표현(Representations)

3. REST 의 특징
4개로 통일 (Get/Put/post/Delete)
1) Uniform (유니폼 인터페이스)
Uniform Interface는 URI로 지정한 리소스에 대한 조작을 통일되고 한정적인 인터페이스로 수행하는 아키텍처 스타일을 말합니다.
2) Stateless (무상태성)
전송시스템이 무상태이다.
REST는 무상태성 성격을 갖습니다. 다시 말해 작업을 위한 상태정보를 따로 저장하고 관리하지 않습니다. 세션 정보나 쿠키정보를 별도로 저장하고 관리하지 않기 때문에 API 서버는 들어오는 요청만을 단순히 처리하면 됩니다. 때문에 서비스의 자유도가 높아지고 서버에서 불필요한 정보를 관리하지 않음으로써 구현이 단순해집니다.
속도↑
3) Cacheable (캐시 가능)
REST의 가장 큰 특징 중 하나는 HTTP라는 기존 웹표준을 그대로 사용하기 때문에, 웹에서 사용하는 기존 인프라를 그대로 활용이 가능합니다. 따라서 HTTP가 가진 캐싱 기능이 적용 가능합니다. HTTP 프로토콜 표준에서 사용하는 Last-Modified태그나 E-Tag를 이용하면 캐싱 구현이 가능합니다.
4) Self-descriptiveness (자체 표현 구조)
REST의 또 다른 큰 특징 중 하나는 REST API 메시지만 보고도 이를 쉽게 이해 할 수 있는 자체 표현 구조로 되어 있다는 것입니다.
5) Client - Server 구조
REST 서버는 API 제공, 클라이언트는 사용자 인증이나 컨텍스트(세션, 로그인 정보)등을 직접 관리하는 구조로 각각의 역할이 확실히 구분되기 때문에 클라이언트와 서버에서 개발해야 할 내용이 명확해지고 서로간 의존성이 줄어들게 됩니다.
6) 계층형 구조
REST 서버는 다중 계층으로 구성될 수 있으며 보안, 로드 밸런싱, 암호화 계층을 추가해 구조상의 유연성을 둘 수 있고 PROXY, 게이트웨이 같은 네트워크 기반의 중간매체를 사용할 수 있게 합니다.

전송과정에서 RES (상태)를 저장하지 마라!

JPA는 기술 명세이다

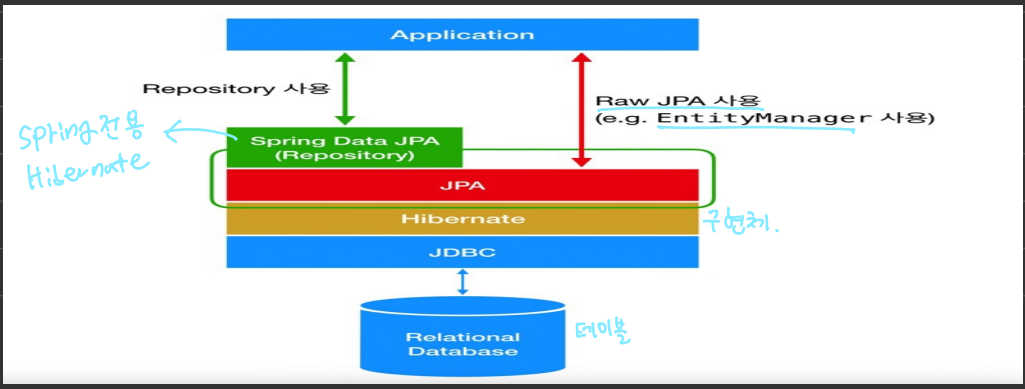
JPA는 Java Persistence API의 약자로, 자바 어플리케이션에서 관계형 데이터베이스를 사용하는 방식을 정의한 인터페이스이다. 여기서 중요하게 여겨야 할 부분은, JPA는 말 그대로 인터페이스라는 점이다. JPA는 특정 기능을 하는 라이브러리가 아니다. 마치 일반적인 백엔드 API가 클라이언트가 어떻게 서버를 사용해야 하는지를 정의한 것처럼, JPA 역시 자바 어플리케이션에서 관계형 데이터베이스를 어떻게 사용해야 하는지를 정의하는 한 방법일 뿐이다.

JPA는 단순히 명세이기 때문에 구현이 없다. JPA를 정의한 javax.persistence 패키지의 대부분은 interface, enum, Exception, 그리고 각종 Annotation으로 이루어져 있다. 예를 들어, JPA의 핵심이 되는 EntityManager는 아래와 같이 javax.persistence.EntityManager 라는 파일에 interface로 정의되어 있다. ↳ CRUD

Hibernate는 JPA의 구현체이다

Hibernate는 JPA라는 명세의 구현체이다. 즉, 위에서 언급한 javax.persistence.EntityManager와 같은 인터페이스를 직접 구현한 라이브러리이다. JPA와 Hibernate는 마치 자바의 interface와 해당 interface를 구현한 class와 같은 관계이다.

"Hibernate는 JPA의 구현체이다"로부터 도출되는 중요한 결론 중 하나는 JPA를 사용하기 위해서 반드시 Hibernate를 사용할 필요가 없다는 것이다. Hibernate의 작동 방식이 마음에 들지 않는다면 언제든지 DataNucleus, EclipseLink 등 다른 JPA 구현체를 사용해도 되고, 심지어 본인이 직접 JPA를 구현해서 사용할 수도 있다. 다만 그렇게 하지 않는 이유는 단지 Hibernate가 굉장히 성숙한 라이브러리이기 때문일 뿐이다.

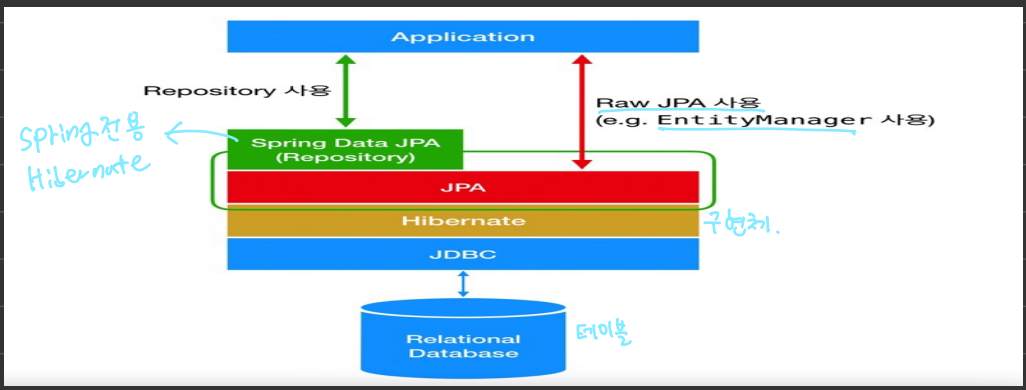


S'pring Data JPA는 JPA를 쓰기 편하게 만들어놓은 모듈이다

필자는 Spring으로 개발하면서 단 한 번도 EntityManager를 직접 다뤄본 적이 없다. DB에 접근할 필요가 있는 대부분의 상황에서는 Repository를 정의하여 사용했다. 아마 다른 분들도 다 비슷할 것이라 생각한다. 이 Repository가 바로 Spring Data JPA의 핵심이다.

Spring Data JPA는 Spring에서 제공하는 모듈 중 하나로, 개발자가 JPA를 더 쉽고 편하게 사용할 수 있도록 도와준다. 이는 JPA를 한 단계 추상화시킨 Repository라는 인터페이스를 제공함으로써 이루어진다. 사용자가 Repository 인터페이스에 정해진 규칙대로 메소드를 입력하면, Spring이 알아서 해당 메소드 이름에 적합한 쿼리를 날리는 구현체를 만들어서 Bean으로 등록해준다.

Spring Data JPA가 JPA를 추상화했다는 말은, Spring Data JPA의 Repository의 구현에서 JPA를 사용하고 있다는 것이다. 예를 들어, Repository 인터페이스의 기본 구현체인 SimpleJpaRepository의 코드를 보면 아래와 같이 내부적으로 EntityManager을 사용하고 있는 것을 볼 수 있다.



ORACLE 이거 사전적 의미로는 '신탁(神託)', '신의 계시', '예언' 등의 신에 관련된 뜻을 가지고 있다. ORACLE이란 이름은 아마도 DBMS 계의 신이란 뜻을 가진 것으로 풀이 할 수 있다. 이미 오라클의 뜻은 대충 나왔다. DBMS 즉, DataBase Management System 이다. 그것도 ORDBMS... 객체-관계형 데이타베이스 관리 시스템. 이것은 데이타베이스 서버를 두고 한 말이고,오라클 디벨로퍼, 디자이너 등등의 여러 제품들이 있다. 또한 회사의 이름이기도 하다. 이러한 종합적인 이미지가 오라클의 뜻을 가지고 있는 것이다. 대부분 오라클이라고하면 데이타베이스 서버를 떠올리게 된다. 여기에서는 오라클이란 것을 DBMS에 국한해서 살펴보도록 하겠다.

이론적인 배경으로 본다면 관점에 따른 분류에서 실행 데이타 모델에 속하는 것이다. 오라클 자체만을 생각한다면 논리적인 것이나 전체적으로 본다면 물리적인 것까지 포함합니다. 물론 OS에 종속적이라고 할 수 있겠다. 현대의 '관계형' 이란 이름을 달고 나온 제품들은 하나같이 '2차원 테이블구조' 라는 관계형 포멧으로 사용자에게 논리적으로 데이타(혹은 정보)를 보여주고 있다. 즉, 행과 열의 조합으로 표현한다는 것이다.

이러한 구조는 사용자들에게는 상당히 친숙하며, 또한 데이타를 표현하는데에 아주 편리하기 때문에 2차원 테이블구조를 사용하고 있으며, 관계형에는 이러한 테이블을 '릴레이션' 이라는 용어를 사용해서 2차원 테이블구조를 설명한다. 릴레이션(Relation)은 '관계' 라는 뜻을 가지고 있는데 이는 데이타 모델링을 할 때의 '관계'와는 다른 의미를 가지고 있다. 릴레이션에서의 관계는 속성들끼리의 함수적 종속에 의해서 뭉쳐있고, 테이블끼리 어떤 연결고리를 가지고 있는 것을 의미한다. 이러한 사항들은 '정규화' 라는 것을 안다면 쉽게 이해를 할 수 있으리라 생각한다. 아무런 이론적인 배경없이 오라클을 접하게 된다면 낭패를 보게 될 것입니다. 물론 공부하는 방법에 따라서 틀릴수도 있다. 그러나 별루 좋치는 않다. 적어도 오라클을 처음으로 접할때는 데이타베이스가 무엇인지 정도는 정확하게 판단하고 있어야 할 것이다.

오라클은 객체-관계형 데이타베이스 관리 시스템이라고 했다. 즉, 추상적인 데이타 타입과 메소드와 같은 객체 지향적인 구조를 지원한다는 것이다. 버전 8에서부터 지원한다. 객체-관계형이라는 말은 이러한 객체지향적인 특성을 지닌 오라클이 객체간의관계를 가질 수 있다는 것일 말한다. 또한 하나의 객체가 또 다른 객체를 포함할 수도 있는 것이다. 나중에 살펴보겠지만 이러한 객체지향적인 특성은 다중값속성등에 대한 해결을 간단히 할 수 있다.

스프링 4대원칙

POJO / DI / ORM / AOP

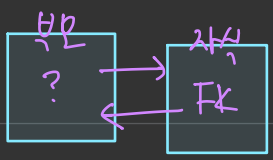
어노테이션 Maven gradle / JPA / 트랜잭션/보안

JPA에서 가장 중요한 것 (순수 JPA 세계관)

JPA에서 가장 중요한 것을 뽑자면, "객체와 관계형 데이터베이스 테이블이 어떻게 매핑되는지를 이해하는 것"이라고 생각합니다. 🥇
왜냐하면 JPA의 목적인 "객체 지향 프로그래밍과 데이터베이스 사이의 패러다임 불일치를 해결"이라는 것과 가장 직접적으로 연관되어 있기 때문입니다.
Hibernate Document(2. Domain Model)에서도 객체와 테이블 매핑이 전체 스크롤 중, 조금 과장해서 표현하면 절반 수준의 비중인 것을 봐도 객체와 테이블 매핑의 중요성을 짐작할 수 있습니다.
그러나 객체와 테이블 매핑에 대한 내용을 조금 더 구체적으로 나누면 컬럼, 타입, 테이블, ... 등에 대한 1차원적인 매핑과 테이블 간의 연관 관계 매핑으로 나눌 수 있습니다.
(개인 의견)

FK 가지고 있는 것이 자식.

출처: https://jeong-pro.tistory.com/231 [기본기를 쌓는 정아마추어 코딩블로그]



단방향, 양방향
데이터베이스 테이블은 외래 키 하나로 양 쪽 테이블 조인이 가능합니다.
따라서 데이터베이스는 단방향이니 양방향이니 나눌 필요가 없습니다.
그러나 객체는 참조용 필드가 있는 객체만 다른 객체를 참조하는 것이 가능합니다.
그렇기 때문에 두 객체 사이에 하나의 객체만 참조용 필드를 갖고 참조하면 단방향 관계, 두 객체 모두가 각각 참조용 필드를 갖고 참조하면 양방향 관계라고 합니다.
엄밀하게는 양방향 관계↔는 없고 두 객체가 단방향 참조를 각각 가져서 양방향 관계처럼 사용하고 말하는 것입니다. ⬅️ ➡️
JPA를 사용하여 데이터베이스와 패러다임을 맞추기 위해서 객체는 단방향 연관 관계를 가질지, 양방향 연관 관계를 가질지 선택해야합니다.
선택은 비즈니스 로직에서 두 객체가 참조가 필요한지 여부를 고민해보면 됩니다.

- Board.getPost()처럼 참조가 필요하면 Board→Post 단방향참조
  - 만약 참조가 굳이 필요없으면 참조를 안하면 됨
- post.getBoard()처럼 참조가 필요하면 Post→Board 단방향참조
  - 만약 참조가 굳이 필요없으면 참조를 안하면 됨

이렇게 비즈니스 로직에 맞게 선택했는데 두 객체가 서로 단방향 참조를 했다면 양방향 연관 관계가 되는 것입니다.
단방향 연관 관계와 양방향 연관 관계를 구분하는 방법은 이렇게 이해하면 됩니다.

출처: https://jeong-pro.tistory.com/231 [기본기를 쌓는 정아마추어 코딩블로그]

복잡성↑ 부모 자식 관계를 명확하게 해야한다 → 양방향 매핑을 해야한다.

연관관계의 주인?

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

주인: 실제 값을 변경 할수 있는 "객체"

- 주인은 mappedBy를 사용하지 않는다.
- 주인이 아니면 mappedBy를 사용하여 주인을 지정해야 한다.

엔티티를 **단방향으로 매핑하면 참조를 하나**만 사용하므로 이 참조로 외래키를 관리하면된다.
그런데 엔티티를 양방향으로 매핑하면 **회원 → 팀**, **팀 → 회원** 두곳에서 서로를 참조한다. 따라서 객체의 연관관계를 관리하는 포인트가 2곳으로 늘어난다.
엔티티를 양방향 연관관계로 설정하면 객체의 참조는 둘인데 외래 키는 하나다. 따라서 둘 사이에 차이가 발생한다. (어디서 외래 키를 관리해야 할까?)
이런 차이로 인해 JPA에서는 두 객체 연관관계 중 하나를 정해서 테이블의 외래키를 관리해야 하는데 이것을 연관관계의 **주인(Owner)**라고 한다.

Pagination.

싱글턴 패턴이란

static

전역 변수를 사용하지 않고 객체를 하나만 생성 하도록 하며, 생성된 객체를 어디에서든지 참조할 수 있도록 하는 패턴
https://gmlwjd9405.github.io/2018/07/06/singleton-pattern.html