JPA

JPA (Java Persistence API)

자바 `ORM` 기술에 대한 API 표준 명세 (Java에서 제공하는 API)

▼ Object-Relational Mapping : 객체 관계 매핑

- 자바 어플리케이션에서 **관계형 데이터베이스를 사용하는 방식을 정의한 인터페이스**
- 특정 기능을 하는 라이브러리가 아닌, 관계형 데이터베이스를 어떻게 사용해야 하는지 정의
- `javax.persistence` 패키지의 대부분은 `interface`, `enum`, `Exception`, 각종 `Annotation`으로 이뤄짐
- Java Class와 DB Table을 매핑

JAP 특징

1. 데이터를 **객체지향적**으로 관리
2. 자바 객체와 DB 테이블 사이의 **매핑**으로 SQL 생성
3. 객체를 통해 쿼리를 작성할 수 있는 **JPQL**(Java Persistence Query Language) 지원
4. 성능 향상을 위한 **지연 로딩**, **즉시 로딩**과 같은 기법을 제공
5. 도메인을 중심으로 Data를 가져오는 쿼리의 재사용성이 지켜짐

▼ 좀 더 깊게 들어가 보자.

[**JPA 성능 최적화**]

1. **버퍼링** 기능
2. **캐싱** 기능

1. 캐싱 기능 - 1차 캐시와 동일성(identity) 보장

```
String memberId = "100";

Member m1 = jpa.find(Member.class, memberId); // SQL

Member m2 = jpa.find(Member.class, memberId); // 캐시 (SQL 1번만 실행, m1을 가져옴)

println(m1 == m2) // true
```

⇒ 같은 트랜잭션 안에서는 같은 엔티티를 반환 (약간의 조회 성능 향상)

2. 버퍼링 기능 - 트랜잭션을 지원하는 쓰기 지연 (transactional write-behind)

[EX 1]

```
/** 1. 트랜잭션을 커밋할 때까지 INSERT SQL을 모음 */

transaction.begin();

em.persist(memberA);
em.persist(memberB);
em.persist(memberC);

// commit할 때, 데이터베이스에 INSERT SQL을 한 번에 보냄
/** 2. JDBC BATCH SQL 기능을 사용해서 한번에 SQL 전송 */

transaction.commit();
```

[EX 2]

```
/** UPDATE, DELETE로 인한 로우(ROW)락 시간 최소화 */
/** 로우(ROW)락? 트랜잭션의 순차적 진행을 보장할 수 있는 직렬화(serialization) 장치*/
transaction.begin();

changeMember(memberA);
deleteMember(memberB);

비즈니스_로직_수행(); // 비즈니스 로직 수행 동안 DB 로우 락이 걸리지 않는다.

// commit할 때, 데이터베이스에 UPDATE, DELETE SQL을 한 번에 보냄
```

```
transaction.commit();
```

- 지연 로딩 (Lazy Loading)

  객체가 실제로 사용될 때, 로딩하는 전략

  ```
  // Member 객체에 대한 SELECT 쿼리만 전달
  Member member = memberDAO.find(memberId);

  Team team = member.getTeam();

  // team 객체를 사용할 때,Team에 대한 SELECT 쿼리를 전달
  String teamName = team.getName();
  ```

  이때, Member와 Team 객체를 따로 조회하여 네트워크를 2번 타게 됨

  이럴 때는,

- 즉시 로딩 (Eager Loading)

  JOIN SQL로 한번에 연관된 객체까지 미리 조회하는 전략

  ```
  // 이때, JOIN을 통해 연관된 모든 객체를 같이 가져옴
  Member member = memverDAO.find(memberId);
  Team team = member.getTeam();
  String teamName = team.getName();
  ```

▼ 실무에서는 가급적 지연 로딩만 사용

▼ 즉시 로딩시, 예상하지 못한 SQL이 발생

JPQL에서 N+1 문제를 일으킴 (이 부분은 아직 이해를 못하겠다)

왜 사용?

1. SQL 중심적인 개발에서 객체 중심적인 개발 가능

2. 생산성 증가
3. 유지보수 용이
4. Qbject와 RDB간의 패러다임 불일치 해결

처음에는,

JDBC로 시작해서 직접 DB Connection해서 SQL을 전달

반복적인 DB Connection, SQL 재사용성의 마비를 해결하고자 'SQL Mapper' 프레임워크 사용

SQL Mapper의 Query는 주로 xml로 관리 ⇒ 유지보수 과정에서 문제 발생

그래서 나온 것이,

JPA라는 표준을 만족하는 Hibernate와 같은 프레임워크 등장

간단한 빌드와 테스트코드만으로 유지보수과정에 생기는 오류를 잡아주어 개발 생산성을 높임

SQL Mapper를 이용해 복잡한 Query를 이용한 데이터 추출이라는 강점을 가지고 있지만,

서버 개발자에게는 복잡한 데이터를 추출하는 업무보다는 비즈니스 로직을 수정하여 직접 서비스되고 있는 Query를 바꾸는 것이 주된 업무

단점 ?

1. 성능

   메서드 호출로 쿼리를 실행한다는 것은, 직접 SQL을 호출하는 것보다 성능이 떨어짐
2. 세밀함

   메서드 호출로 SQL을 실행하기 때문에 세밀함이 떨어짐

   복잡한 통계 분석 쿼리를 메서드 호출로 처리하기 힘듬

ORM vs SQL Mapper

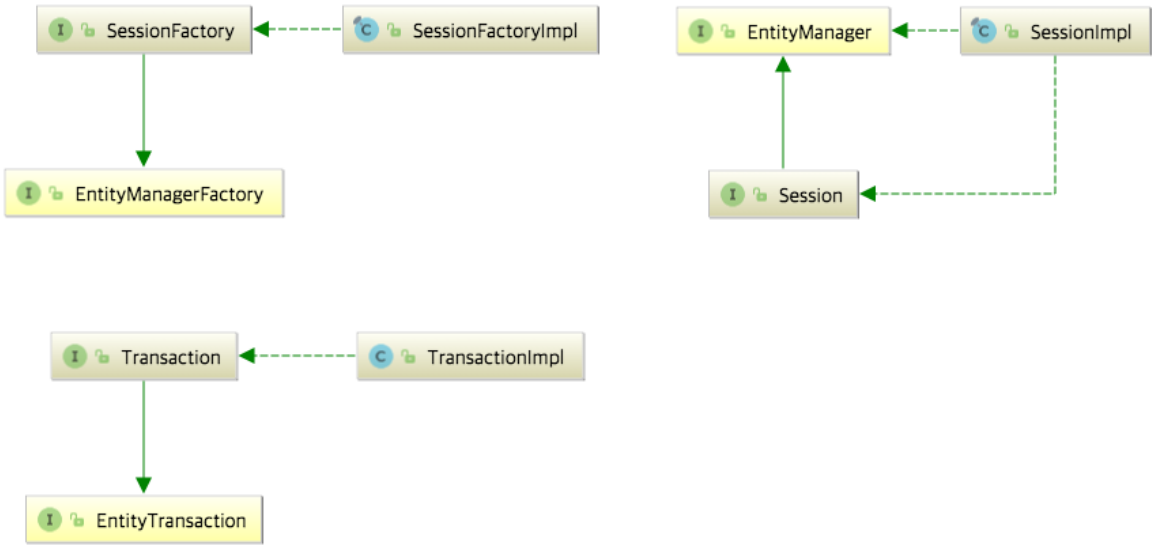
- ORM은 DB테이블을 자바 객체로 매핑하여 객체간의 관계를 바탕으로 SQL을 자동 생성, Mapper는 SQL을 명시
- ORM은 RDB의 관계를 Object에 반영하는 것이 목적이라면, Mapper는 단순히 필드를 매핑시키는 것이 목적

Hibernate ?

JPA의 구현체

- javax.persistence.EntityManager와 같은 Interface를 구현한 라이브러리
- 자바의 Interface와 해당 Interface를 구현한 class와 같은 관계

[**JPA**와 **Hibernate**의 상속 및 구현 관계]



▼ 반드시 Hibernate를 사용할 필요는 없음! 직접 구현하거나 DataNucleus, EclipseLink 등 다른 JPA 구현체를 사용해도 됨

▼ 다만, Hibernate가 굉장히 성숙한 라이브러리라 잘 사용

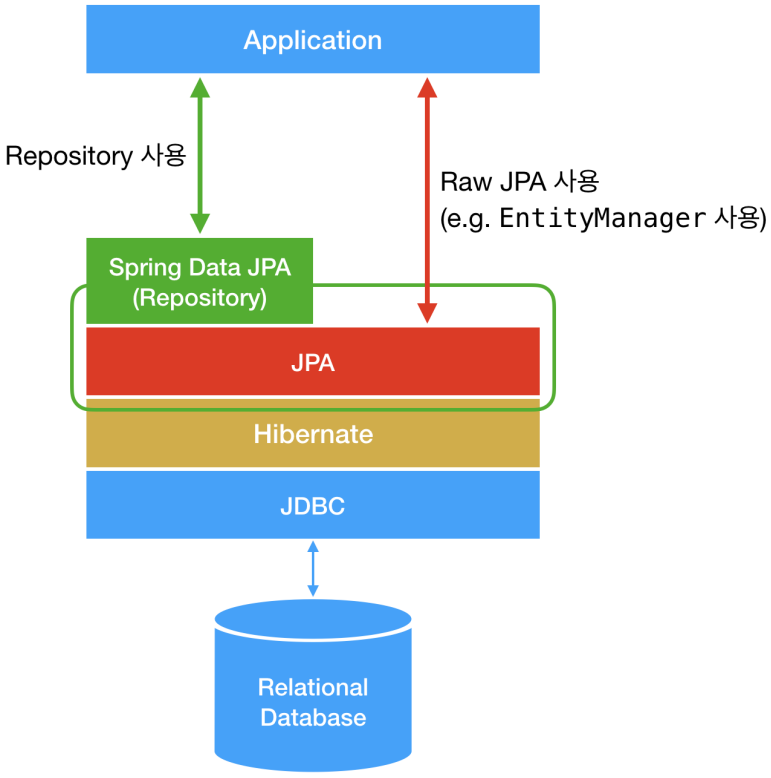
▼ JDBC API를 사용하지 않는다는 것은 아님! 메서드 내부에서 JDBC API가 동작, 개발자가 직접 SQL을 작성하지 않을 뿐.

Spring Data JPA ?

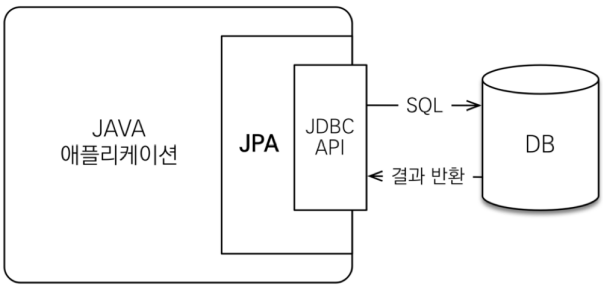
JPA를 쓰기 편하게 만들어놓은 모듈

`Repository` : Spring Data JPA의 핵심

- Spring에서 제공하는 모듈 중 하나
- JPA를 한 단계 추상화시킨 Repository라는 인터페이스를 제공함
- 해당 메소드 이름에 적합한 쿼리를 날리는 구현체를 만들어서 Bean으로 등록



[JPA 동작 과정]



▼ JPA 내부에서 JDBC API를 사용하여, SQL을 호출해 DB와 통신

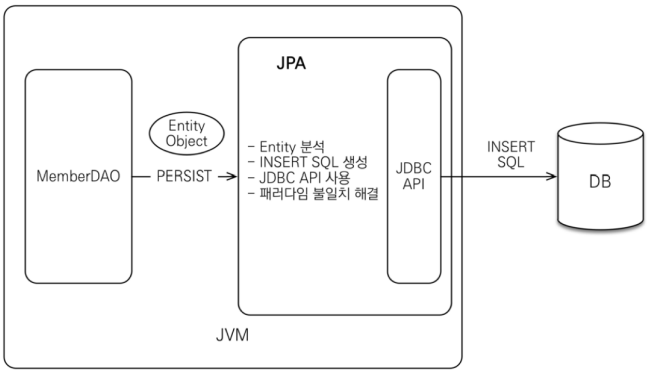
▼ 개발자가 직접 JDBC API를 사용할 필요가 없음!

insert

MemberDAO에서 객체를 저장하고 싶을때 JPA에 Member 객체를 넘김

JPA는,

1. Member 엔티티 분석
2. INSERT SQL 생성
3. JDBC API를 사용하여 SQL을 DB에 전달



[EX 1 - Insert]

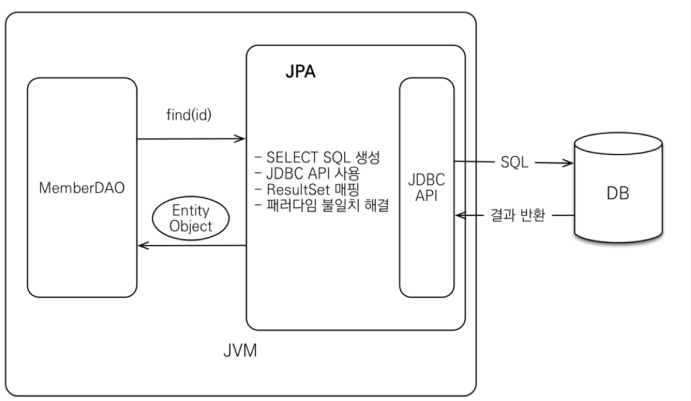
```
SQL Mapper
INSERT INTO USER(name, phone_num) VALUES('man', '01011112222')

JPA
// user 객체에 데이터를 저장 후,
em.persist(user)
```

find

Member의 PK 값을 JPA에 넘김

JPA는,

1. 엔티티의 매핑 정보를 바탕으로 적절한 SELECT SQL을 생성
2. JDBC API를 사용하여 SQL을 DB에 전달
3. DB로 부터 결과를 받아 옴
4. 결과(ResultSet)를 객체에 매핑



[EX 1 - Update]

```
SQL Mapper
UPDATE USER SET name='김싸피' WHERE user_id=143

JPA
// user 객체를 찾아서,
User user = em.find(User.class, 143);
user.setName('김싸피'); // Dirty Checking
```

▼ Dirty Checking (더티 체킹)

Transaction 안에서 엔티티의 변경이 일어나면, 변경 내용을 자동으로 데이터베이스에 반영하는 JPA의 특징

- 데이터베이스에 변경 데이터를 저장하는 시점
  1. Transactio commit 시점
  2. Entitymanager flush 시점
  3. JPQL 사용 시점

[EX 2 - Delete]

```
SQL Mapper
DELETE FROM USER WHERE user_id=143;

JPA
User user = em.find(User.class, 143); em.remove(user);
```

참고자료

JPA 정의

- https://suhwan.dev/2019/02/24/jpa-vs-hibernate-vs-spring-data-jpa/
- https://velog.io/@adam2/JPA는-도데체-뭘까-orm-영속성-hibernate-spring-data-jpa
- https://interconnection.tistory.com/107
- https://victorydntmd.tistory.com/195