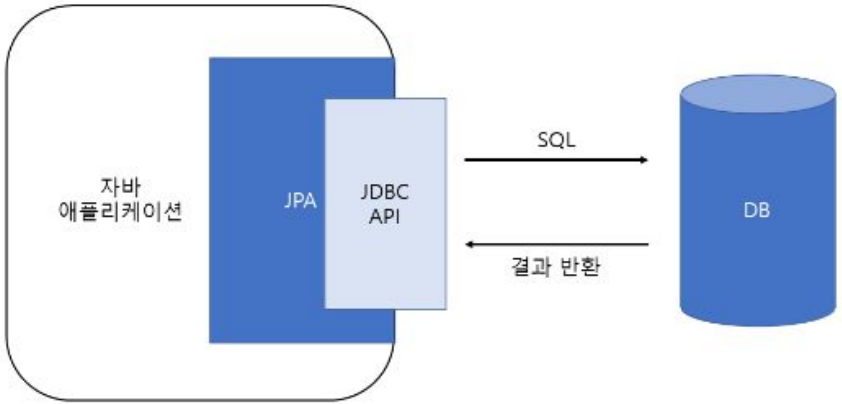
11. JPA (Oracle연동)

백성애

JPA란?

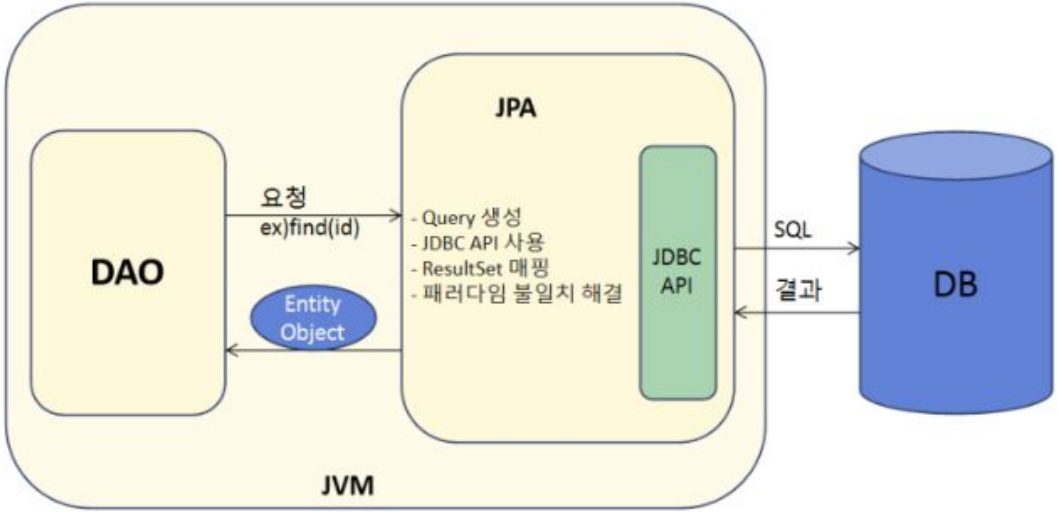
- Java Persistence API
- 자바 표준 ORM (Object Relational Mapping)
- 객체 매핑 ↔ SQLMapper
- 개발자는 OOP 프로그래밍 수행
- JPA가 관계형 데이터베이스에 맞게 SQL문을 대신 생성한 후 실행



- JPA는 인터페이스의 모음
- JPA 2.1 표준 명세를 구현한 3가지 구현체
- 하이버네이트, EclipseLink, DataNuncleus

JPA란

ORM은 개발자를 대신하여 SQL문을 생성하여 DB에 전달함으로써, 개발자는 객체지향 애플리케이션 개발에 집중할 수 있다



ORM은 Object Relational Mapping의 줄임말로 객체 관계 매핑을 의미. 객체와 RDB의 테이블을 자동으로 매핑하는 방법
ORM사용하면 데이터베이스 쿼리를 객체지향적으로 조작할 수 있다.
그에 따라 데이터베이스에 대한 종속성이 줄어든다
복잡한 서비스의 경우는 ORM만으로는 한계가 있다. 객체와 RDB관점의 불일치가 발생함

JPA와 Hibernate

- ORM도 여러 종류가 있으나, 자바에서는 JPA를 표준으로 사용한다
- JPA는 자바에서 관계형 데이터베이스를 사용하는 방식을 정의한 인터페이스
- 인터페이스이므로 실제 사용을 위해서는 ORM 프레임워크를 추가로 선택해야 하는데, 대표적으로 Hibernate를 많이 사용한다.
- 하이버네이트는 JPA인터페이스를 구현한 구현체로 내부적으로 JDBC API를 사용하고 있다.
- 하이버네이트의 목표는 자바 객체를 통해 DB종류에 상관 없이 DB를 자유자재로 사용할 수 있게 하는 데 있다.

Hibernate와 Spring Data JPA

- 스프링의 경우 하이버네이트의 기능을 더욱 편하게 사용하도록 모듈화한 Spring Data JPA를 활용한다

- Spring Data JPA는 CRUD에 필요한 인터페이스를 제공하며, 하이버네이트의 EntityManager를 직접 다루지 않고 **Repository를 정의해 사용**한다. 스프링이 적합한 쿼리를 동적으로 생성하는 방식으로 DB를 조작함

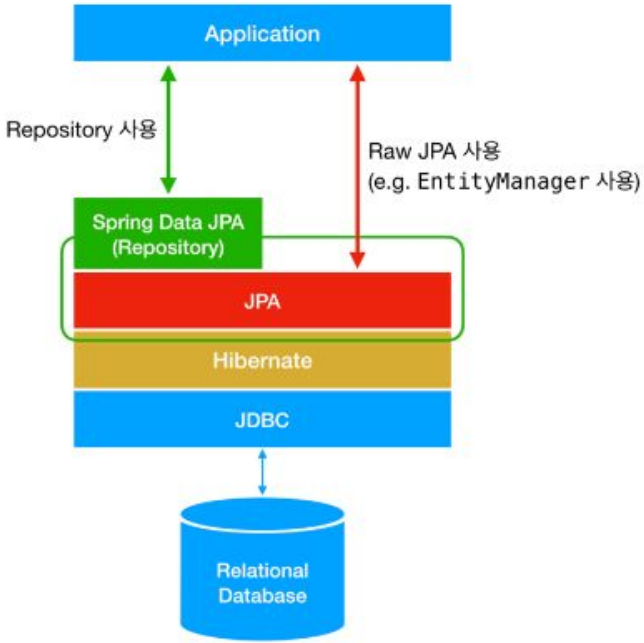


그림: https://dream-and-develop.tistory.com/390

Spring Data JPA 사용시 장점

1. 생산성 - JPA의 CRUD

- 저장 : jpa.persist(member)
- 조회 : Member member = jpa.find(memberId)
- 수정 : member.setName("변경할 이름")
- 삭제 : jpa.remove(member)

CRUD를 위한 반복적인 코드와 SQL문을 개발자가 직접 작성하지 않아도 된다

Spring Data JPA 사용시 장점

2. 유지보수 용이

- 기존 : 필드 변경(추가)시 모든 SQL 수정-SQL에 의존적인 개발

- JPA : 필드만 추가하면 됨, SQL은 JPA가 처리. 객체중심으로 개발할 수 있다.

3. 패러다임의 불일치 해결

- OOP와 RDB는 데이터와 구조를 다루는 방식이 다르기 때문에, 이를 매핑하거나 변환할 때 여러 문제가 발생할 수 있다. JPA는 이러한 불일치를 극복하게 한다

4. 다양한 성능 최적화 기회 제공

5. 데이터 접근 추상화와 벤더 독립성-애플리케이션과 DB사이에 데이터접근 계층을 제공해서 특정 DB기술에 종속되지 않도록 함

참고-패러다임 불일치란

OOP언어와 RDB의 불일치 문제점

관계형 데이터베이스는 데이터 중심으로 구조화되어 있으며, OOP언어의 특징(추상화,캡슐화,상속성,다형성)을 지원하지 않는다.

1. **상속 구조와 테이블 매핑**
   - OOP의 상속 관계는 RDB의 테이블 구조와 직접적으로 매핑되지 않습니다. 이를 해결하기 위해 상속된 객체들을 테이블로 매핑하는 다양한 전략이 필요합니다.
2. **객체 그래프와 테이블 관계**
   - OOP에서는 객체가 다른 객체를 참조하여 객체 그래프를 형성합니다. 반면, RDB에서는 테이블 간의 외래 키 관계를 사용합니다. 이를 매핑하는 과정에서 발생하는 문제들을 해결해야 합니다.
3. **데이터 타입 불일치**
   - 객체의 필드 타입과 데이터베이스의 컬럼 타입 간의 불일치를 해결해야 합니다. 예를 들어, Java의 `Date`와 SQL의 `DATETIME` 타입 간의 변환 등이 필요합니다.
4. **상태 추적**
   - 객체의 상태 변화(예: 객체의 생성, 수정, 삭제 등)를 데이터베이스와 동기화하는 것이 필요합니다. 이는 객체의 생명 주기와 데이터베이스 트랜잭션을 관리하는 것을 의미합니다.

JPA는 이러한 패러다임의 불일치를 해결하기 위한 다양한 매핑 전략과 기능을 제공한다. 주요 기능은 다음과 같다:

엔티티 매핑

- `@Entity`, `@Table`, `@Id`, `@Column` 등의 어노테이션을 사용하여 객체와 테이블 간의 매핑을 정의합니다.

상속 매핑

- `@Inheritance` 어노테이션을 사용하여 객체 상속 구조를 테이블로 매핑하는 전략을 정의합니다. 예를 들어, 단일 테이블 전략, 조인 전략, 테이블 퍼 클래스 전략 등이 있습니다.

관계 매핑

- `@OneToOne`, `@OneToMany`, `@ManyToOne`, `@ManyToMany` 등의 어노테이션을 사용하여 객체 간의 관계를 데이터베이스의 외래 키와 매핑합니다.

JPQL

- Java Persistence Query Language를 사용하여 객체를 대상으로 쿼리를 작성할 수 있습니다. 이는 SQL과 유사하지만 객체 지향적인 문법을 사용합니다.

트랜잭션 관리

- JPA는 객체의 생명 주기와 데이터베이스 트랜잭션을 관리하여, 데이터 일관성을 유지합니다.

spring boot project 생성

- 이전 스프링부트 관련 pdf파일 참고할 것

build.gradle 설정

```
--build.gradle----------------------------
plugins {
        id 'java'
        id 'org.springframework.boot' version '3.3.4'
        id 'io.spring.dependency-management' version '1.1.6'
}

group = 'com.kosmo'
version = '0.0.1-SNAPSHOT'

java {
        toolchain {
                languageVersion = JavaLanguageVersion.of(17)
        }
}

configurations {
        compileOnly {
                extendsFrom annotationProcessor
        }
}

repositories {
        mavenCentral()
}
```

```
dependencies {
        implementation 'org.springframework.boot:spring-boot-starter-web'
        compileOnly 'org.projectlombok:lombok'
        developmentOnly 'org.springframework.boot:spring-boot-devtools'
        runtimeOnly 'com.oracle.database.jdbc:ojdbc11'
        annotationProcessor 'org.projectlombok:lombok'
        testImplementation 'org.springframework.boot:spring-boot-starter-test'
        testRuntimeOnly 'org.junit.platform:junit-platform-launcher'

//      add my module
        implementation 'org.apache.tomcat.embed:tomcat-embed-jasper' // JSP 처리 라이브러리 (Jasper)
        implementation 'jakarta.servlet.jsp.jstl:jakarta.servlet.jsp.jstl-api:3.0.0' // JSTL API (
        implementation 'org.glassfish.web:jakarta.servlet.jsp.jstl:3.0.1'
        implementation 'jakarta.servlet:jakarta.servlet-api'
        implementation 'javax.annotation:javax.annotation-api:1.3.2'
        implementation 'jakarta.inject:jakarta.inject-api:2.0.1'

//      add jpa module
        implementation 'org.springframework.boot:spring-boot-starter-data-jpa'
        // https://mvnrepository.com/artifact/com.github.gavlyukovskiy/p6spy-spring-boot-starter
        implementation 'com.github.gavlyukovskiy:p6spy-spring-boot-starter:1.9.1'
}

tasks.named('test') {
        useJUnitPlatform()
}
```

dependencies 내용

```
implementation 'org.springframework.boot:spring-boot-starter-web'

compileOnly 'org.projectlombok:lombok'

developmentOnly 'org.springframework.boot:spring-boot-devtools'

runtimeOnly 'com.oracle.database.jdbc:ojdbc11'

annotationProcessor 'org.projectlombok:lombok'

testImplementation 'org.springframework.boot:spring-boot-starter-test'

testRuntimeOnly 'org.junit.platform:junit-platform-launcher'

//add my module

implementation 'org.apache.tomcat.embed:tomcat-embed-jasper' // JSP 처리 라이브러리 (Jasper)

implementation 'jakarta.servlet.jsp.jstl:jakarta.servlet.jsp.jstl-api:3.0.0' // JSTL API (Jakarta EE 기반)

implementation 'org.glassfish.web:jakarta.servlet.jsp.jstl:3.0.1'

implementation 'jakarta.servlet:jakarta.servlet-api'
```

```
implementation 'javax.annotation:javax.annotation-api:1.3.2'
implementation 'jakarta.inject:jakarta.inject-api:2.0.1'

//add jpa module

implementation 'org.springframework.boot:spring-boot-starter-data-jpa'
implementation 'com.github.gavlyukovskiy:p6spy-spring-boot-starter:1.9.1'
```

src/resources/application.yml설정

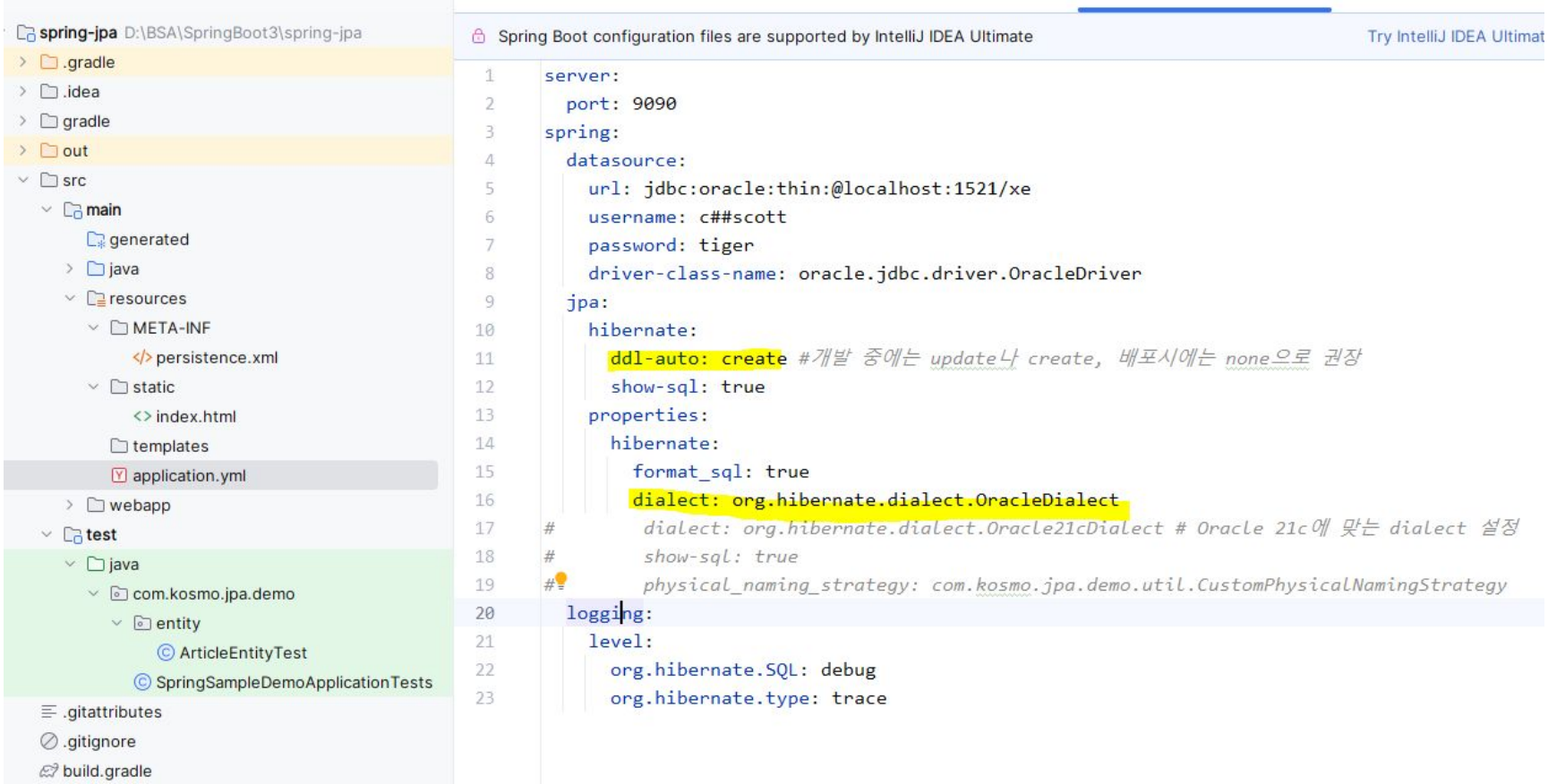
application.yml 파일은 들여쓰기와 콜론(:) 뒤에 띄어쓰기를 한 뒤 값을 기술해야 함에 주의하자.

ddl-auto: create : 애플리케이션 실행시점에 테이블을 drop하고 다시 생성한다

show-sql: true :하이버네이트 실행 SQL을 보여준다

dialect: org.hibernate.dialect.OracleDialect: 데이터베이스 방언 설정(db에 종속되는 고유기능을 해결하기 위한 설정)

resouces/application.yml



resources/META-INF/persistence.xml

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<persistence xmlns="https://jakarta.ee/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="https://jakarta.ee/xml/ns/persistence https://jakarta.ee/xml/ns/persistence/persistence_3_0.xsd"
             version="3.0">

    <persistence-unit name="entitytest">
        <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
        <class>com.kosmo.jpa.demo.entity.ArticleEntity</class>

        <properties>
            <!-- JDBC 데이터베이스 연결 설정 -->
            <property name="jakarta.persistence.jdbc.driver" value="oracle.jdbc.driver.OracleDriver"/>
            <property name="jakarta.persistence.jdbc.url" value="jdbc:oracle:thin:@localhost:1521/xe"/>
            <property name="jakarta.persistence.jdbc.user" value="c##scott"/>
            <property name="jakarta.persistence.jdbc.password" value="tiger"/>

            <!-- JPA 설정 -->
            <property name="jakarta.persistence.schema-generation.database.action" value="none"/>
            <property name="hibernate.hbm2ddl.auto" value="update"/>
            <!--<property name="hibernate.physical_naming_strategy" value="com.kosmo.jpa.demo.util.CustomPhysicalNamingStrategy"/>-->
        </properties>
    </persistence-unit>

</persistence>
```

ArticleEntity 작성 후 테스트

JPA(Java Persistence API)에서 Entity는 데이터베이스 테이블과 매핑되는 자바 클래스를 말한다.

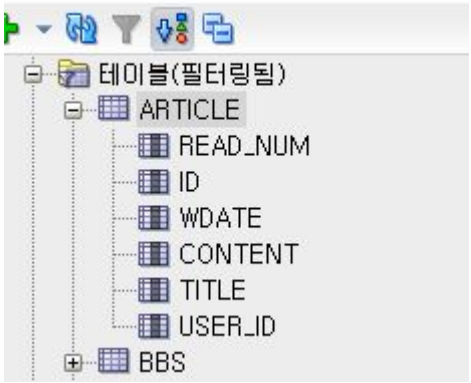
엔티티 클래스는 데이터베이스 테이블의 행(Row)에 해당하는 객체를 나타내며,

JPA를 통해 객체 지향 방식으로 데이터베이스와 상호작용할 수 있도록 한다.

```
package com.kosmo.jpa.demo.entity;
import jakarta.persistence.*;
import lombok.*;
import org.hibernate.annotations.CreationTimestamp;
import java.time.LocalDateTime;

@Getter
@Setter
@ToString
@NoArgsConstructor
@AllArgsConstructor
@Entity
@Table(name="article")
@SequenceGenerator(name="ARTICLE_SEQ_GEN",//시퀀스 제너레이터 이름
        sequenceName = "ARTICLE_SEQ", //시퀀스명
        initialValue = 1, //시작값
        allocationSize = 1)//시퀀스 증가 단위
public class ArticleEntity {

    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE,
    generator = "ARTICLE_SEQ_GEN" )
    private Long id;
    @Column(name="user_id", unique = true)
    private String userId;
    @Column(name="title",nullable = false)
    private String title;
    private String content;
    @Column(name="read_num")
    private int readNum;
    @CreationTimestamp
    private LocalDateTime wdate;
    /*JPA에서는 엔티티 클래스의 필드 이름을 사용하여 데이터베이스 컬럼 이름을 자동으로 생성할 때,
    기본적으로 스네이크 케이스(snake_case)로 변환된다.
    예를 들어 readNum 필드는 read_num으로 변환된다.
    * */
}
```

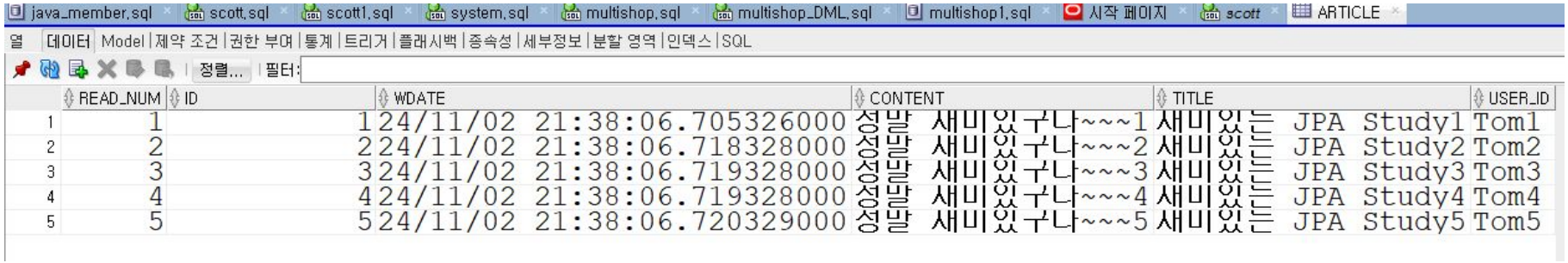
ArticleEntityTest.java

실행결과 Article테이블이 자동 생성되고 데이터가 삽입된다




```
153 import java.time.LocalDateTime;
154 import java.util.Scanner;
155
156 public class ArticleEntityTest {
157
158     public static void main(String[] args) {
159         //entitytest"는 resources/META-INF/persistence.xml 파일에 정의된 persistence unit의 이름
160         // 해당 이름에 맞는 데이터베이스 연결 정보가 사용된다
161         EntityManagerFactory factory = Persistence.createEntityManagerFactory("entitytest");
162         System.out.println("EntityManagerFactory 객체 : " + factory.getClass().getName());
163         //org.hibernate.internal.SessionFactoryImpl
164         EntityManager em=factory.createEntityManager();
165         //EntityManager는 데이터베이스와 상호 작용하는 API로, CRUD(Create, Read, Update, Delete) 작업을 수행하는 데 사용된다
166         System.out.println("EntityManager 객체 : " + em.getClass().getName());
167         //org.hibernate.internal.SessionImpl
168         ArticleEntity a1;
169         em.getTransaction().begin();
170
171         for(int i=1;i<6;i++){
172             a1=new ArticleEntity();
173             a1.setUserId("Tom"+i);
174             a1.setTitle("재미있는 JPA Study"+i);
175             a1.setContent("정말 재미있구나~~~"+i);
176             a1.setReadNum(i);
177             a1.setWdate(LocalDateTime.now());
178             em.persist(a1);
179         }
180         System.out.println("엔터키를 입력하세요...");
181         Scanner scan = new Scanner(System.in);
182         scan.nextLine();
183         scan.close();
184
185         em.getTransaction().commit();
186         em.close();
187         factory.close();
188
189     }
190 }
```

실행 결과 insert된 데이터 확인



	READ_NUM	ID	WDATE	CONTENT	TITLE	USER_ID
1	1	1	24/11/02 21:38:06.705326000	정말 재미있구나~~~1	재미있는 JPA Study1	Tom1
2	2	2	24/11/02 21:38:06.718328000	정말 재미있구나~~~2	재미있는 JPA Study2	Tom2
3	3	3	24/11/02 21:38:06.719328000	정말 재미있구나~~~3	재미있는 JPA Study3	Tom3
4	4	4	24/11/02 21:38:06.719328000	정말 재미있구나~~~4	재미있는 JPA Study4	Tom4
5	5	5	24/11/02 21:38:06.720329000	정말 재미있구나~~~5	재미있는 JPA Study5	Tom5

ArticleEntityTest2.java - JPQL을 사용하여 데이터 가져오기

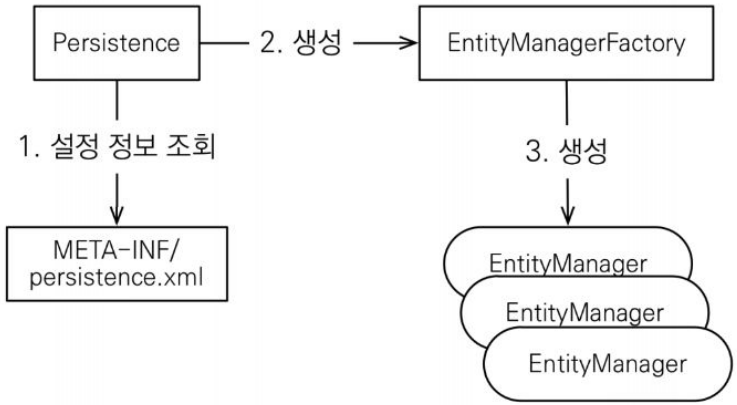
```java
package com.kosmo.jpa.demo.app;

import com.kosmo.jpa.demo.entity.ArticleEntity;
import jakarta.persistence.EntityManager;
import jakarta.persistence.EntityManagerFactory;
import jakarta.persistence.Persistence;
import jakarta.persistence.TypedQuery;

import java.util.List;

public class ArticleEntityTest2 {
    public static void main(String[] args) {
        EntityManagerFactory factory= Persistence.createEntityManagerFactory("entitytest");
        EntityManager em=factory.createEntityManager();
        TypedQuery<ArticleEntity> query
        =em.createQuery("select a from ArticleEntity a", ArticleEntity.class);
        //createQuery 메서드를 호출하여 JPQL(Java Persistence Query Language) 쿼리를 정의함.
        // 이 경우, ArticleEntity에서 모든 레코드를 선택하는 쿼리다
        List<ArticleEntity> list=query.getResultList();

        list.stream().forEach(System.out::println);
        factory.close();//DB연결을 닫고 메모리 정리
    }
}
```

실행 결과

```
21:39:54.253 [main] INFO org.hibernate.orm.connections.pooling -- HHH10001115: Connection pool size: 20 (min=1)
21:39:54.594 [main] WARN org.hibernate.dialect.Dialect -- HHH000511: The 11.2.0 version for [org.hibernate.dialect.OracleDialect] is no longer sup
 properly. The minimum supported version is 19.0.0. Check the community dialects project for available legacy versions.
21:39:55.759 [main] INFO org.hibernate.engine.transaction.jta.platform.internal.JtaPlatformInitiator -- HHH000489: No JTA platform available (set
 JTA platform integration)
ArticleEntity(id=1, userId=Tom1, title=재미있는 JPA Study1, content=정말 재미있구나~~~1, readNum=1, wdate=2024-11-02T21:38:06.705326)
ArticleEntity(id=2, userId=Tom2, title=재미있는 JPA Study2, content=정말 재미있구나~~~2, readNum=2, wdate=2024-11-02T21:38:06.718328)
ArticleEntity(id=3, userId=Tom3, title=재미있는 JPA Study3, content=정말 재미있구나~~~3, readNum=3, wdate=2024-11-02T21:38:06.719328)
ArticleEntity(id=4, userId=Tom4, title=재미있는 JPA Study4, content=정말 재미있구나~~~4, readNum=4, wdate=2024-11-02T21:38:06.719328)
ArticleEntity(id=5, userId=Tom5, title=재미있는 JPA Study5, content=정말 재미있구나~~~5, readNum=5, wdate=2024-11-02T21:38:06.720329)
21:39:56.152 [main] INFO org.hibernate.orm.connections.pooling -- HHH10001008: Cleaning up connection pool [jdbc:oracle:thin:@localhost:1521/xe]

Process finished with exit code 0
```

JPA 동작 방식

- 가장 먼저 META-INF/persistence.xml의 파일에 설정되어 있는 DB, user, pwd 등의 정보를 조회한다.
- 그 후 **EntityManagerFactory**를 생성한다. EntityManagerFactory는 DB당 한개씩만 가지도록 한다.
- 사용자 의 요청이 들어올 때 마다 공유하지 않고 각각 **EntityManager**를 생성하여 다 수행하면 버리는 방식으로 사용 한다.
- 모든 데이터의 변경은 트랜잭션 안에서 실행하도록 해야한다.



JPA 동작 방식

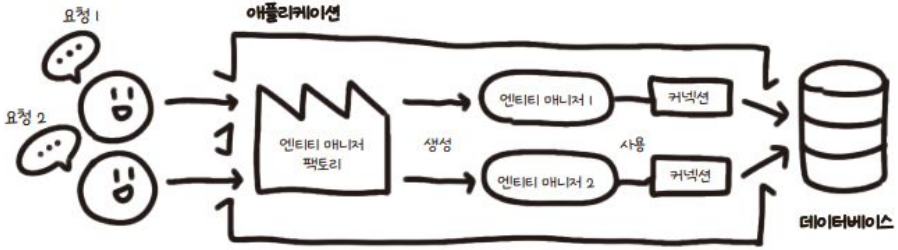


그림: https://wikidocs.net/237213

- EntityManagerFactory
  - 엔티티 매니저 인스턴스를 관리하는 주체. 애플리케이션 실행시 1개만 만들어지며, 엔티티 매니저를 생성한다.

- EntityManager
  - EntityManager는 JPA(Java Persistence API)에서 데이터베이스 작업을 수행하는 핵심 인터페이스다.
  - 이는 엔티티의 생명 주기를 관리하고, 데이터베이스와 상호작용하여 데이터를 생성, 읽기, 수정, 삭제(CRUD)할 수 있도록 한다

EntityManager의 주요 역할

1. 엔티티 생명주기 관리
2. 쿼리 실행
3. 트랜잭션 관리

[주요 생명주기 메서드]

persist(Object entity): 엔티티를 영속 컨텍스트에 저장하여 영속 상태로 만듭니다.
merge(Object entity): 분리된 상태의 엔티티를 영속 컨텍스트에 병합합니다.
remove(Object entity): 영속 상태의 엔티티를 제거하여 삭제된 상태로 만듭니다.
find(Class<T> entityClass, Object primaryKey): 기본 키로 엔티티를 조회하여 영속 상태로 만듭니다.
detach(Object entity): 엔티티를 영속 컨텍스트에서 분리하여 분리된 상태로 만듭니다.

EntityManager의 주요 역할

쿼리 실행

EntityManager는 JPQL(Java Persistence Query Language)이나 네이티브 SQL을 사용하여 데이터베이스 쿼리를 실행할 수 있습니다.
쿼리 메서드:
createQuery(String qlString): JPQL 쿼리를 생성합니다.
createNamedQuery(String name): 이름이 지정된 쿼리를 생성합니다.
createNativeQuery(String sqlString): 네이티브 SQL 쿼리를 생성합니다.

EntityManager의 주요 역할

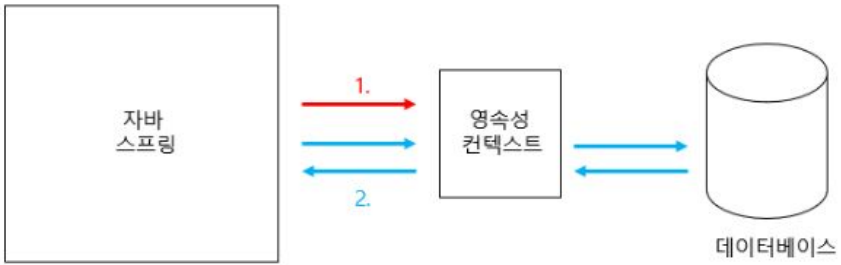
트랜잭션 관리

EntityManager는 데이터베이스 트랜잭션을 시작, 커밋, 롤백하는 기능을 제공합니다.
일반적으로 트랜잭션 관리는 EntityTransaction 인터페이스를 통해 수행됩니다.
트랜잭션 메서드:
getTransaction(): 현재 트랜잭션을 반환합니다.
joinTransaction(): 현재 활성화된 트랜잭션에 참여합니다.
영속 컨텍스트 관리

EntityManager는 영속 컨텍스트를 관리하여 엔티티의 상태를 동기화하고, 변경 사항을 데이터베이스에 반영합니다.
영속 컨텍스트 메서드:
flush(): 영속 컨텍스트의 변경 내용을 데이터베이스에 반영합니다.
clear(): 영속 컨텍스트를 초기화합니다.
close(): EntityManager를 닫고, 관련된 자원을 해제합니다.
isOpen(): EntityManager가 열려 있는지 여부를 확인합니다.

영속성 컨텍스트-Persistence Context

- **PersistenceContext**
- 영속성 컨테이너는 엔티티를 영구 저장하는 환경이다
- JPA를 이해하기 위해서는 영속성 컨테이너를 이해하는 것이 중요하다
- 애플리케이션과 데이터베이스 사이에 엔티티와 레코드의 괴리를 해소하는 기능과 객체를 보관하는 기능을 수행한다.
- 엔티티 객체가 영속성 컨텍스트에 들어와 JPA의 관리대상이 되는 시점부터 해당 객체를 영속 객체라 한다.
-
- DB접근 세션이 생성되면
  - 영속성 컨텍스트가 만들어지고
  - 세션이 종료되면 영속성 컨텍
  - 스트도 없어진다.
  - EntityManager는 영속성
  - 컨텍스트를 접근하기 위한
  - 수단으로 사용된다.



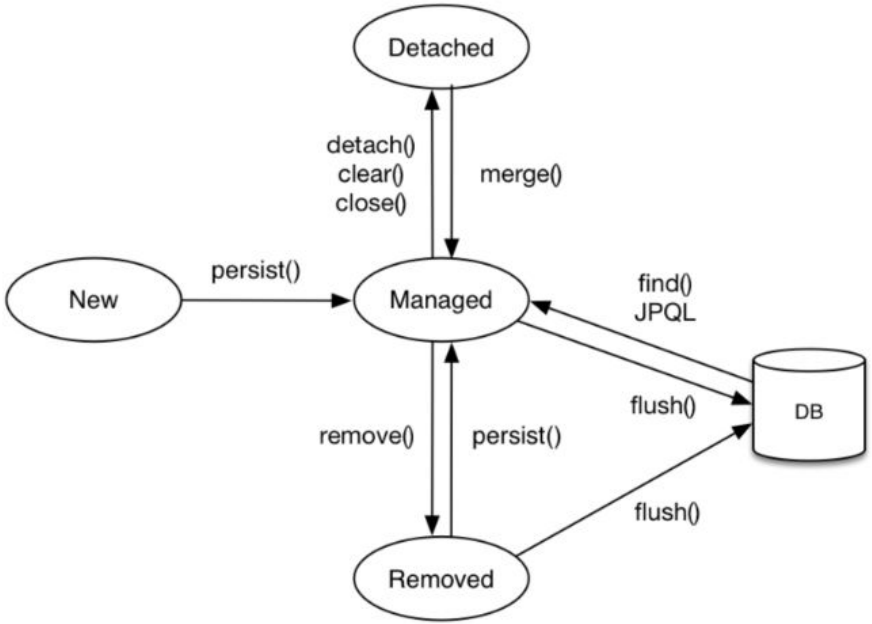
엔티티 생명주기

[1] 비영속 (new)

[2] 영속 (managed)

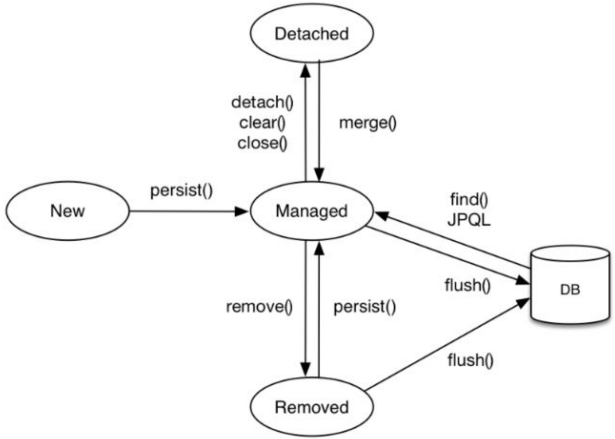
[3] 준영속 상태(detached)

[4] 삭제 상태(removed)

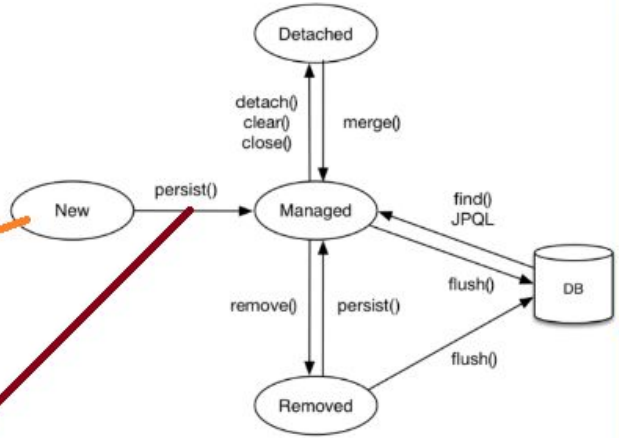


엔티티 생명주기

- **비영속(New)** : 엔티티를 생성만 한 상태. 영속성 컨텍스트와는 관련이 없는 상태다
- **영속상태(Managed)**: em.persist()를 통해 영속성 컨텍스트에 엔티티를 넣은 상태. DB에 저장되지는 않으며, 트랜잭션 커밋 시점에 데이터베이스에 반영된다
- **준영속(Detached)** : em.detach()를 통해 영속성 컨텍스트에 엔티티가 저장되었다가 분리된 상태
- **삭제(Removed)** : em.remove()를 통해 DB에 저장된 엔티티까지 삭제시킨 상태



```java
package com.jpa.data_jpa;

import com.jpa.data_jpa.entity.Member;
import jakarta.persistence.EntityManager;
import jakarta.persistence.EntityManagerFactory;
import jakarta.persistence.EntityTransaction;
import jakarta.persistence.Persistence;

public class TestJPA {
    public static void main(String[] args) {
        EntityManagerFactory emf= Persistence.createEntityManagerFactory("demojpa");

        EntityManager em=emf.createEntityManager();

        EntityTransaction tx=em.getTransaction();

        try{
            tx.begin();
            Member u1=new Member("아무개23","hi");
            em.persist(u1);
            tx.commit();
            System.out.println(u1.getId());
        }catch (Exception e){
            tx.rollback();
        }finally {
            em.close();
        }
        emf.close();
```



회원 객체가 영속성 컨텍스트에 저장된 상태.
여기까지는 DB에 Insert SQL을 보내지 않은 단계다.

트랜잭션을 DB에 반영한다. 이때 영속성 컨텍스트에 저장된 회원정보가 DB에 insert되면서 반영된다

PersistenceContext의 기능

JPA는 영속성 컨텍스트라는 중간 계층을 만들어 애플리케이션과 DB사이에 사용하고 있다. 중간계층을 만들면 버퍼링, 캐싱 등을 할 수 있는 장점이 있다

영속성 컨테스트는 1차 캐시 구조를 가진다.

em.find()를 통해서 엔티티를 조회할 경우, 우선적으로 1차 캐시부터 먼저 탐색한다.
즉 DB까지 조회하지 않고 엔티티를 조회할 수 있다.

1차 캐시에서 탐색 후, 없는경우에만 직접 DB로 쿼리를 날려 해당되는 엔티티를 가져와서 1차캐시에 저장하고 나서, 반환해준다.

1차 캐시는 Map<Key,Value>으로 저장된다.

동일성 보장

- 하나의 트랜잭션에서 같은 키값으로 영속성 컨텍스트에 저장된 엔티티 조회시 같은 엔티티 조회를 보장한다
- 바로 1차 캐시에 저장된 엔티티를 조회하기 때문에 가능하다

- 쉽게 말해서 아래와 같은 코드를 실행하면 동일하다고 true가 출력된다.

```
Member a = em.find(Member.class, "member1");
Member b = em.find(Member.class, "member1");
System.out.println(a == b);
```

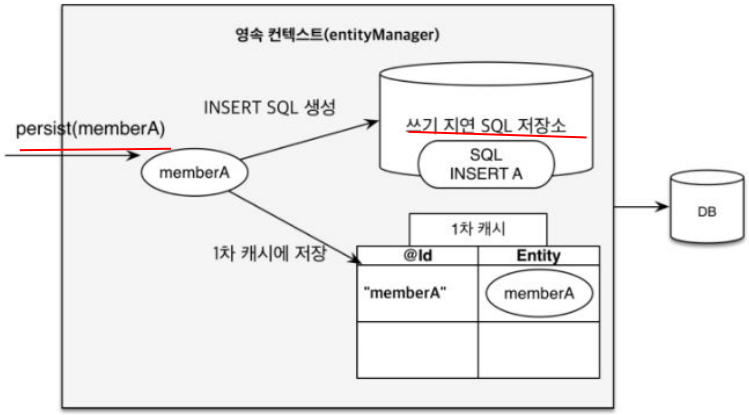
- 메모리 주소 연산을 하듯이, 동일한 1차캐시에 저장되어 있기에 위와같은 결과가 도출된다.

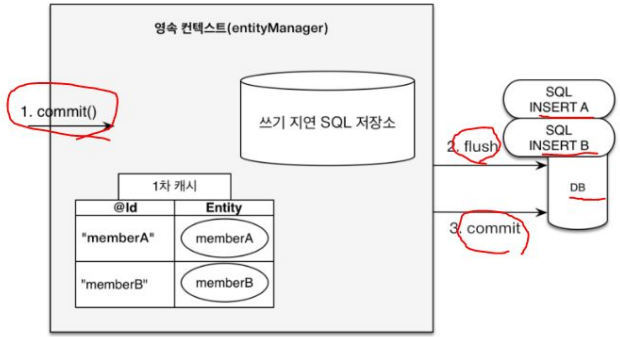
트랜잭션을 지원하는 쓰기 지연(Lazy)

영속성 컨텍스트에는 쓰기지연 SQL저장소가 존재한다

em.persist()를 호출하면 1차 캐시에 저장되는 것과 동시에 쓰기 지연 SQL저장소에 SQL문이 저장된다.

이렇게 SQL문을 쌓아두고 트랜잭션을 커밋하는 시점에 저장된 SQL문들이 flush되면서 데이터베이스에 반영된다.



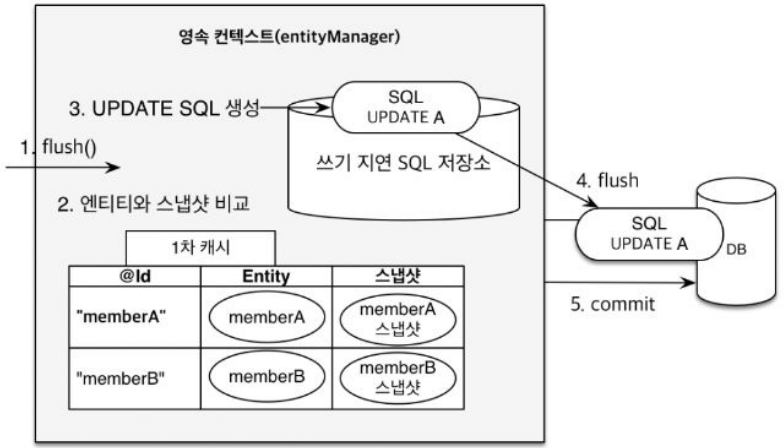


변경 감지

- JPA는 1차 캐시에 DB에서 처음 불러온 엔티티의 스냅샷을 갖고 있다.
- 그리고 1차 캐시에 저장된 엔티티와 스냅샷을 비교후 변경 내용이 있다면 UPDATE SQL문을 쓰기 지연 SQL저장소에 담아둔다.
- 그리고DB에 커밋 시점에 변경 내용을 자동으로 반영한다. 따라서 따로 UPDATE문을 호출할 필요가 없다.
- Dirty-Checking이라고도 불리는 해당 기능은 엔티티에 대한 속성이 변경하고자 할때, 해당 엔티티에 대해서 다시 em.persist()를 하지 않아도 자동으로 DB에 update문을 날려주는 기능이라고 보면 된다.

```java
EntityTransaction tx=em.getTransaction();

try{
    tx.begin();
    //Member u1=new Member("아무개23","hi");
    //em.persist(u1);

    Member tmp=em.find(Member.class, o: 1);
    System.out.printf("%s   %d  %s\n",tmp.getName(), tmp.getId(), tmp.getUserid());
    //데이터 수정
    tmp.setName("김아무개");
    tmp.setUserid("Kim");

    System.out.printf("%s   %d  %s\n",tmp.getName(), tmp.getId(), tmp.getUserid());
    tx.commit();

}catch (Exception e){
    tx.rollback();
}finally {
```

```java
package com.jpa.data_jpa.entity;

import jakarta.persistence.*;
import lombok.Getter;
import lombok.Setter;
import lombok.ToString;

@Entity
@Getter
@Setter
@Table(name="member")
@ToString(of={"id","name","userid"})
public class Member {
    @Id
    @GeneratedValue
    private Long id;
    private String name;
    private String userid;

    protected Member(){}
    public Member(String name, String userid){
        this.name=name;
        this.userid=userid;
    }
}
```



지연 로딩

JPA의 지연로딩(lazy loading)은 쿼리로 요청한 데이터를 애플리케이션에 바로 로딩하는 것이 아니라 필요할 때 쿼리를 날려 데이터를 조회하는 것.

이러한 특징들이 갖는 공통점은 데이터베이스의 접근을 최소화하여 성능을 높일 수 있다는 점이다.

자주 쓰지 않게 하거나 변화를 자동 감지해서 미리 준비하거나 하는 등의 방법을 사용한다.

회원관리 Repository작성 - MemberJpaRepository

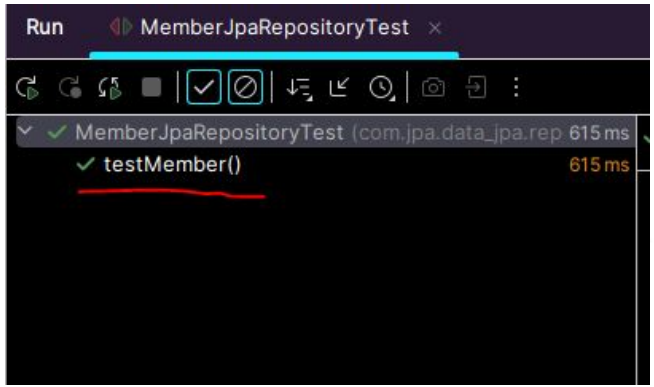
JPA를 이용한 MemberJpaRepository 작성하여 단위 테스트를 해보자.

```java
package com.jpa.data_jpa.entity;

import jakarta.persistence.*;
import lombok.Getter;
import lombok.Setter;
import lombok.ToString;

@Entity
@Getter
@Setter
@Table(name="member")
@ToString(of={"id","name","userid"})
public class Member {
    @Id
    @GeneratedValue
    private Long id;
    private String name;
    private String userid;

    protected Member(){}
    public Member(String name, String userid){
        this.name=name;
        this.userid=userid;
    }
}
```

```java
package com.jpa.data_jpa.repository;

import jakarta.persistence.EntityManager;
import jakarta.persistence.PersistenceContext;
import com.jpa.data_jpa.entity.Member;
import org.springframework.stereotype.Repository;

@Repository
public class MemberJpaRepository {

    @PersistenceContext //스프링 컨테이너가 자동 주입함
    private EntityManager entityManager;

    public Member save(Member user){
        entityManager.persist(user);
        return user;
    }

    public Member find(Long id){
        return entityManager.find(Member.class, id);
    }
}
```

JUnit5 단위테스트

JPA를 이용




```java
package com.jpa.data_jpa.repository;

import com.jpa.data_jpa.entity.Member;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.annotation.Rollback;
import org.springframework.transaction.annotation.Transactional;
import static org.assertj.core.api.Assertions.assertThat;

//@RunWith(SpringRunner.class)//===> JUnit5에서는 사용하지 않아도 됨
@SpringBootTest
@Transactional //org.springframework 안의 Transactional을 선택해주자.
@Rollback(false)
class MemberJpaRepositoryTest {

    @Autowired
    MemberJpaRepository memberJpaRepository;

    @Test
    public void testMember(){
        Member user=new Member("홍길동","hong");
        Member createUser=memberJpaRepository.save(user);

        Member findUser=memberJpaRepository.find(createUser.getId());
        //assertThat()은 주어진 조건이 참이면 테스트 통과. 거짓이면 실패로 테스트 결과를 보고 확인하면 된다
        assertThat(findUser.getId()).isEqualTo(user.getId());
        assertThat(findUser.getName()).isEqualTo(user.getName());
        assertThat(findUser.getUserid()).isEqualTo(user.getUserid());
        //테이블만 생성되고 데이터가 등록되어있지 않는다. @Transactional 을 붙이면
        //스프링 테스트는 마치고 rollback을 시켜버린다. 따라서 DB에서 데이터를 확인하기 어렵다
        //===> 데이터를 보고 싶으면 @Rollback(false) 붙이자

        assertThat(findUser).isEqualTo(user);
    }
```

Spring Data JPA

앞서 EntityManager를 통해 엔티티를 저장하고 조회를 해 보았다. 이는 JPA가 엔티티를 어떻게 관리하는지 학습하기 위함이고,

Spring Data JPA에서는 엔티티 매니저를 직접 이용해 코드를 작성하지 않아도 된다.

대신 DAO(Data Access Object) 역할을 하는 Repository 인터페이스를 설계한 후 사용한다.

스프링 데이터 JPA는 스프링 데이터 공통 기능에서 JPA의 유용한 기술이 추가된 기술이다. JPA를 더 편리하게 사용하는 메서드를 제공한다. JPARepository 인터페이스를 상속받고 제너릭에 관리할 <엔티티명, 엔티티기본키 타입>을 입력하면 기본 CRUD 메서드를 사용할 수 있다.

JpaRepository를 상속받는 인터페이스 구현 및 단위테스트

- JPA
  - Repository(인터페이스)
    - JpaRepository<Entity클래스,PK타입> 상속
      - CRUD 메소드 자동 생성

```
//# 이번엔 JpaRepository를 상속받는 인터페이스 MemberRepository를 작성후 단위테스트 해본다
----------------------
package com.jpa.data_jpa.repository;

import com.jpa.data_jpa.entity.Member;
import org.springframework.data.jpa.repository.JpaRepository;

public interface MemberRepository extends JpaRepository<Member,Long> {
//인터페이스만 정의하면 스프링jpa에서 구현체를 만들어준다
}
```

단위 테스트

JPA에서 Optional을 사용하면
데이터베이스에서 조회한 결과가 존재하지 않는 경우를 명확히 처리할 수 있게 해준다.
NullPointerException을 방지해줌

2. Optional 메서드 사용

`Optional` 클래스는 다양한 유용한 메서드를 제공하여 NPE를 방지하고 더 나은 코드를 작성할 수 있도록 돕습니다.

- `isPresent()`: 값이 존재하는지 확인
- `ifPresent(Consumer<? super T> action)`: 값이 존재하는 경우 주어진 액션을 수행
- `orElse(T other)`: 값이 존재하면 그 값을 반환하고, 그렇지 않으면 다른 값을 반환
- `orElseGet(Supplier<? extends T> other)`: 값이 존재하면 그 값을 반환하고, 그렇지 않으면 다른 값을 제공하는 Supplier를 호출하여 반환
- `orElseThrow(Supplier<? extends X> exceptionSupplier)`: 값이 존재하면 그 값을 반환하고, 그렇지 않으면 주어진 예외를 던짐

```
package com.jpa.data_jpa.repository;

import com.jpa.data_jpa.entity.Member;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.annotation.Rollback;
import org.springframework.transaction.annotation.Transactional;

import java.util.Optional;

import static org.assertj.core.api.Assertions.assertThat;


@SpringBootTest
@Transactional
@Rollback(false)
class MemberRepositoryTest {

    @Autowired
    private MemberRepository memberRepository;

    @Test
    public void testMember(){
        Member member=new Member("이찬성","Lee");
        Member createMember=memberRepository.save(member);

        Optional<Member> opt=memberRepository.findById(createMember.getId());
        Member findMember=opt.get();

        assertThat(findMember.getId()).isEqualTo(member.getId());
        assertThat(findMember.getName()).isEqualTo(member.getName());
        assertThat(findMember.getUserid()).isEqualTo(member.getUserid());
        assertThat(findMember).isEqualTo(member);

    }
}
```

참고: 자바 ORM 표준 JPA 프로그래밍(김영한)

api document

https://javadoc.io/doc/org.springframework.data/spring-data-jpa/latest/spring.data.jpa/module-summary.html

참고사이트

https://dream-and-develop.tistory.com/390