

JPA_Chapter3 - 영속성 관리

🕒 Column	@2021년 12월 12일 오후 12:31
🏷️ Tags	



개요 - 책90 ~119

매핑한 엔티티를 엔티티 매니저를 통해 어떻게 사용하는지 알아볼 것

엔티티 매니저는 엔티티를 저장, 수정, 삭제, 조회하는 등 **엔티티와 관련된 모든 일을 처리한다**

3.1 엔티티 매니저 팩토리와 엔티티 매니저

Gradle로 변경테스트

persistence.xml vs application.yml

Persistence.xml

Java - Gradle환경

SpringBoot - Gradle환경 application.yml

엔티티 매니저 팩토리

3.2 영속성 컨텍스트란?

3.3 엔티티의 생명주기

엔티티의 3가지 상태

비영속(new/transient) : 영속성 컨텍스트와 관계가 없는 상태

3.4 영속성 컨텍스트의 특징

3.4.1 엔티티 조회

영속 엔티티의 동일성 보장

3.4.2 엔티티 등록

commit을 하지 않은 상태

commit 추가

트랜잭션을 지원하는 쓰기 지연이 가능한 이유(?? 말뜻이 이해가 안됨)

3.4.3 엔티티 수정

SQL 수정 쿼리의 문제점

JPA 수정 쿼리

변경 감지

@DynamicUpdate

3.4.4 엔티티 삭제

3.5 플러시

동작 방식

flush 하는 방법

3.5.1 Flush 모드 옵션

3.6 준영속

엔티티가 준영속 상태가 되는 경우

준영속 상태의 특징

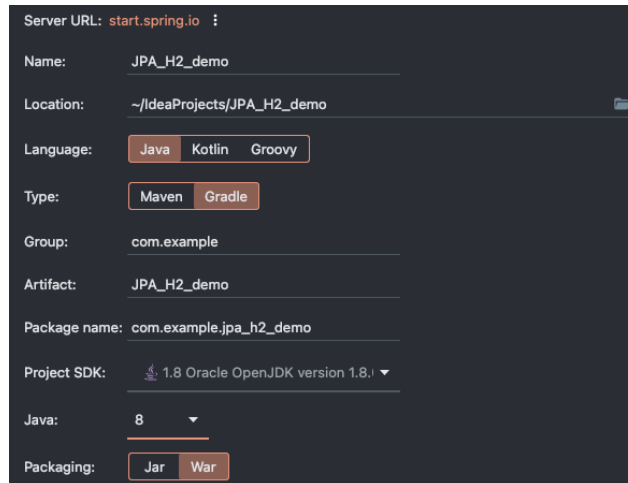
병합 : `merge()`

그래서 어디에 사용하는가?

3.1 엔티티 매니저 팩토리와 엔티티 매니저

Gradle로 변경테스트

▼ 패키지 생성 세팅



Server URL: start.spring.io

Name: JPA_H2_demo

Location: ~/IdeaProjects/JPA_H2_demo

Language: Java Kotlin Groovy

Type: Maven Gradle

Group: com.example

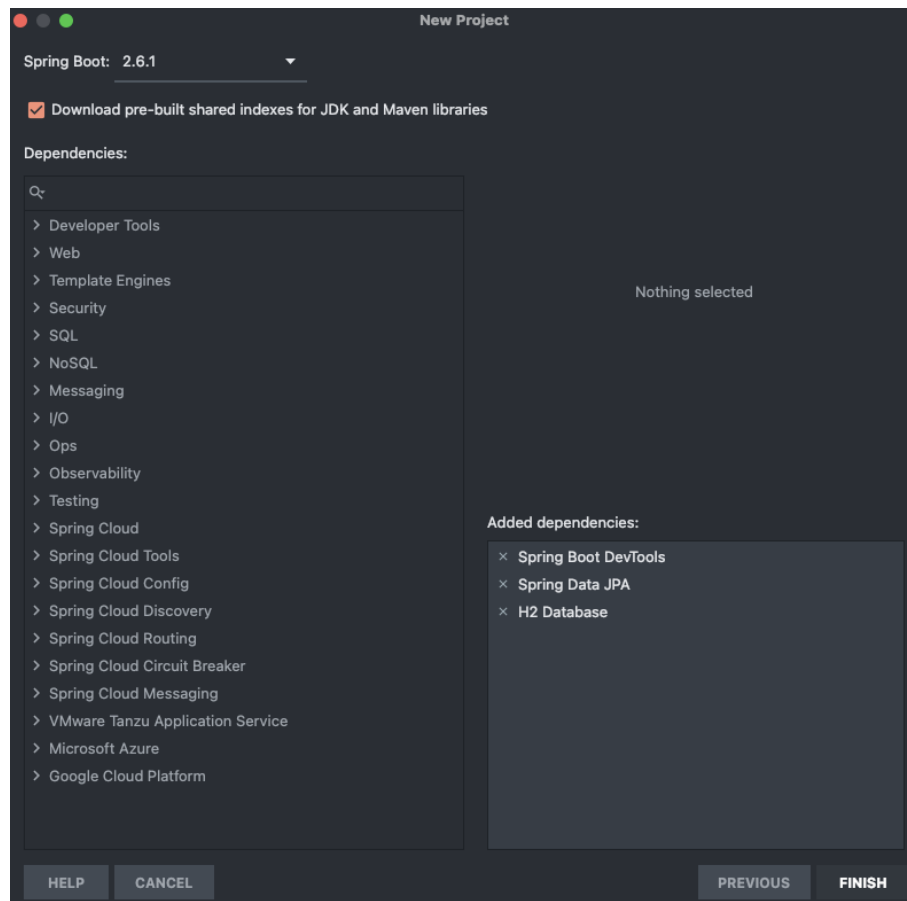
Artifact: JPA_H2_demo

Package name: com.example.jpa_h2_demo

Project SDK: 1.8 Oracle OpenJDK version 1.8.1

Java: 8

Packaging: Jar War





Spring-Boot로는 책과 같은 테스트(EntityMangerFactory를 직접 생성하기)를 할 수 없었다.

왜냐하면 Spring-Boot는 책에서 나오는 일련의 EntityMangerFactory를 생성하는 과정을 Spring-data-JPA로 관리해주기 때문이다.

Spring Data JPA Tutorial

Spring Data is a part of Spring Framework. The goal of Spring Data repository abstraction is to significantly reduce the amount of boilerplate code required to implement data access

 <https://www.javaguides.net/p/spring-data-jpa-tutorial.html>



책과 같은 테스트를 하려면 Java만 있는 상황에서 테스트를 해야 했음

Java환경 테스트 코드

https://github.com/BathwithRadio/Java_JPA_H2_Test.git

persistence.xml vs application.yml

Persistence.xml

교재 코드로는 돌아가지 않는 옵션이 있어서 조금 수정한 후 동작시키는 데 성공했다.

빨간 줄이 수정한 부분

tcp옵션으로는 도저히 되지 않아서 mem으로 수정

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence" version="2.1">

  <persistence-unit name="jpabook">

    <properties>

      <!-- 필수 속성 -->
```

```

        <property name="javax.persistence.jdbc.driver" value="org.h2.Driver"/>
        <property name="javax.persistence.jdbc.user" value="sa"/>
        <property name="javax.persistence.jdbc.password" value=""/>
        <property name="javax.persistence.jdbc.url" value="jdbc:h2:mem:testdb"/>
        <property name="hibernate.dialect" value="org.hibernate.dialect.H2Dialect" />

        <!-- 옵션 -->
        <property name="hibernate.show_sql" value="true" />
        <property name="hibernate.format_sql" value="true" />
        <property name="hibernate.use_sql_comments" value="true" />
        <property name="hibernate.id.new_generator_mappings" value="true" />

        <property name="hibernate.hbm2ddl.auto" value="create" />
    </properties>
</persistence-unit>

</persistence>

```

전체본

```

<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence" version="2.1">

    <persistence-unit name="jpabook">

        <properties>

            <!-- 필수 속성 -->
            <property name="javax.persistence.jdbc.driver" value="org.h2.Driver"/>
            <property name="javax.persistence.jdbc.user" value="sa"/>
            <property name="javax.persistence.jdbc.password" value=""/>
            <property name="javax.persistence.jdbc.url" value="jdbc:h2:tcp://localhost/~:/test"/>
            <property name="hibernate.dialect" value="org.hibernate.dialect.H2Dialect" />

            <!-- 옵션 -->
            <property name="hibernate.show_sql" value="true" />
            <property name="hibernate.format_sql" value="true" />
            <property name="hibernate.use_sql_comments" value="true" />
            <property name="hibernate.id.new_generator_mappings" value="true" />

            <!--<property name="hibernate.hbm2ddl.auto" value="create" />-->
        </properties>
    </persistence-unit>

</persistence>

```

▼ 성공 후 출력 쿼리

```

Hibernate:
    drop table MEMBER if exists
Hibernate:

```

```

        create table MEMBER (
            ID varchar(255) not null,
            age integer,
            NAME varchar(255),
            primary key (ID)
        )
        findMember=지한, age=20
        Hibernate:
            /* insert jpabook.start.Member
            */ insert
            into
                MEMBER
                (age, NAME, ID)
            values
                (?, ?, ?)
        Hibernate:
            /* update
            jpabook.start.Member */ update
            MEMBER
            set
                age=?,
                NAME=?
            where
                ID=?
        Hibernate:
            /* select
            m
            from
                Member m */ select
                member0_.ID as ID1_0_,
                member0_.age as age2_0_,
                member0_.NAME as NAME3_0_
            from
                MEMBER member0_
        members.size=1
        Hibernate:
            /* delete jpabook.start.Member */ delete
            from
                MEMBER
            where
                ID=?

```

Java - Gradle환경

Persist.xml

gradle, 오래된 버전의 entityManager를 사용해서 그런지 아래처럼 class등록이 필요했다.

아마 교재 코드의 경우 sql로 테이블을 만들어주었어서 가능했던 것 같다.

```

<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence" version="2.1">

  <persistence-unit name="Java_JPA_H2_Test">
    <class>start.Member</class>
    <properties>

      <!-- 필수 속성 -->
      <property name="javax.persistence.jdbc.driver" value="org.h2.Driver"/>
      <property name="javax.persistence.jdbc.user" value="sa"/>
      <property name="javax.persistence.jdbc.password" value=""/>
      <property name="javax.persistence.jdbc.url" value="jdbc:h2:mem:testdb"/>
      <property name="hibernate.dialect" value="org.hibernate.dialect.H2Dialect" />

      <!-- 옵션 -->
      <property name="hibernate.show_sql" value="true" />
      <property name="hibernate.format_sql" value="true" />
      <property name="hibernate.use_sql_comments" value="true" />
      <property name="hibernate.id.new_generator_mappings" value="true" />

      <property name="hibernate.hbm2ddl.auto" value="create" />
    </properties>
  </persistence-unit>
</persistence>

```

Gradle설정

```

plugins {
    id 'java'
}

group 'org.example'
version '1.0-SNAPSHOT'

repositories {
    mavenCentral()
}

dependencies {
    implementation 'org.hibernate:hibernate-entitymanager:4.3.10.Final'
    implementation 'com.h2database:h2:1.4.187'
    testImplementation 'org.junit.jupiter:junit-jupiter-api:5.7.2'
    testRuntimeOnly 'org.junit.jupiter:junit-jupiter-engine:5.7.2'
}

test {

```

```
useJUnitPlatform()  
}
```

SpringBoot - Gradle환경 application.yml

```
spring:  
  application:  
    name: JPA_H2_demo  
  profiles:  
    active: default  
  datasource:  
#    hikari:  
#      jdbc-url: jdbc:h2:mem://localhost/~test  
#      url: jdbc:h2:tcp://localhost/~test -> localhost" [90067-200] 발생  
      url: jdbc:h2:mem:testdb  
      driverClassName: org.h2.Driver  
      username: sa  
      password:  
  jpa:  
    hibernate:  
      ddl-auto: create-drop  
    properties:  
      hibernate:  
        show_sql: true  
        format_sql: true  
        use_sql_comment: true  
      id:  
        new_generator_mappings: true
```

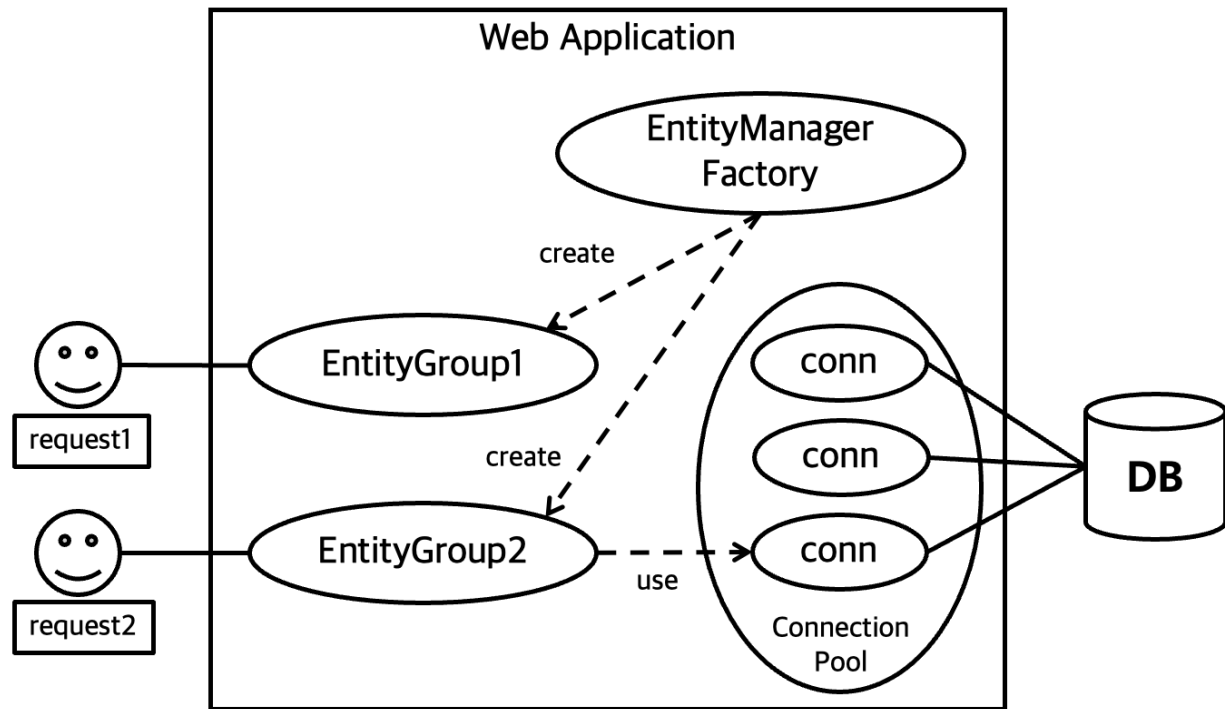
엔티티 매니저 팩토리

엔티티 매니저를 만드는 공장 - 만드는 비용이 크므로 하나만 만들어서 애플리케이션 전체에 공유할 것

엔티티 매니저 팩토리는 여러 스레드가 동시에 접근해도 안전하다

반면에 엔티티 매니저는 여러 스레드가 동시에 접근하면 동시성 문제가 발생

⇒ 스레드 간에 절대 공유하면 안된다.



일반적인 웹 어플리케이션

엔티티 매니저는 데이터베이스 연결이 **꼭 필요한 시점까지 커넥션을 얻지 않는다.**

ex) 트랜잭션을 시작할 때

3.2 영속성 컨텍스트란?

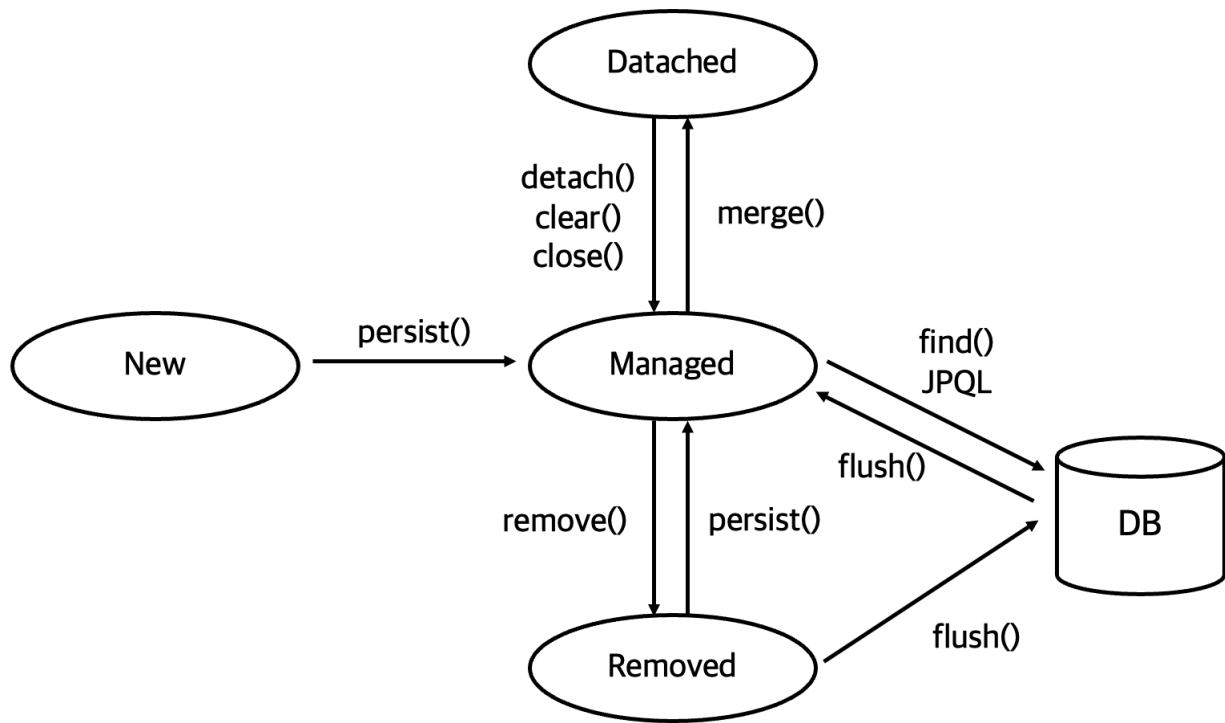
엔티티를 영구 저장하는 환경

`persist()` 메소드는 엔티티 매니저를 사용해서 회원 엔티티를 영속성 컨텍스트에 저장한다.

3.3 엔티티의 생명주기

엔티티의 3가지 상태

비영속(new/transient) : 영속성 컨텍스트와 관계가 없는 상태



엔티티 객체를 생성했지만 아직 저장하지 않은 상태

```

Member member = new Member();
member.setId(id);
member.setUsername("지한");
  
```

⇒ 영속성 컨텍스트와 데이터베이스와 관련이 없는 상태를 비영속 상태라 한다.

- 영속(managed) : 영속성 컨텍스트에 저장된 상태

엔티티 매니저를 통해 엔티티를 영속성 컨텍스트에 저장, 관리

⇒ 영속성 컨텍스트에 의해 관리되는 상태

```

em.persist(member);
  
```

- 준영속(detached): 영속성 컨텍스트에 저장되었다가 분리된 상태

영속성 컨텍스트가 관리하다가 관리하지 않는 상태

```
//아래 세가지의 경우  
em.detach(member);  
em.close();  
em.clear();
```

- 삭제(removed) : 삭제된 상태

엔티티를 데이터베이스와 영속성 컨텍스트에서 삭제한 상태

```
em.remove(member);
```

3.4 영속성 컨텍스트의 특징

- 식별자 값

영속성 컨텍스트는 엔티티를 식별자 값(@Id로 테이블의 기본 키와 매핑한 값)으로 구분한다

```
@Id  
@Column(name = "ID")  
private String id;
```

⇒ 영속 상태는 식별자 값이 반드시 있어야 한다. - 없으면 예외 발생

- 데이터 베이스 저장

JPA는 보통 트랜잭션을 커밋하는 순간 영속성 컨텍스트에 새로 저장된 엔티티를 데이터 베이스에 반영한다 → 이를 flush라 한다.

- 장점
 - 1차 캐시
 - 동일성 보장
 - 트랜잭션을 지원하는 쓰기 지연
 - 변경 감지
 - 지연 로딩

3.4.1 엔티티 조회

영속성 컨텍스트는 내부에 캐시를 가지고 있다 → 1차 캐시

영속성 상태의 엔티티는 모두 이곳에 저장된다.

→ Map인데 Key는 @Id로 매핑한 식별자, Value는 엔티티 인스턴스

```
public static void logic(EntityManager em) {  
  
    Member member = new Member();  
    member.setId("member1");  
    member.setUsername("회원1");  
  
    //등록  
    em.persist(member);  
  
    //여기까지 진행해도 아직 엔티티는 데이터베이스에 저장되지 않는다.  
  
    //한 건 조회  
    Member findMember = em.find(Member.class, "member1");  
    System.out.println("findMember=" + findMember.getUsername());  
  
}
```

결과

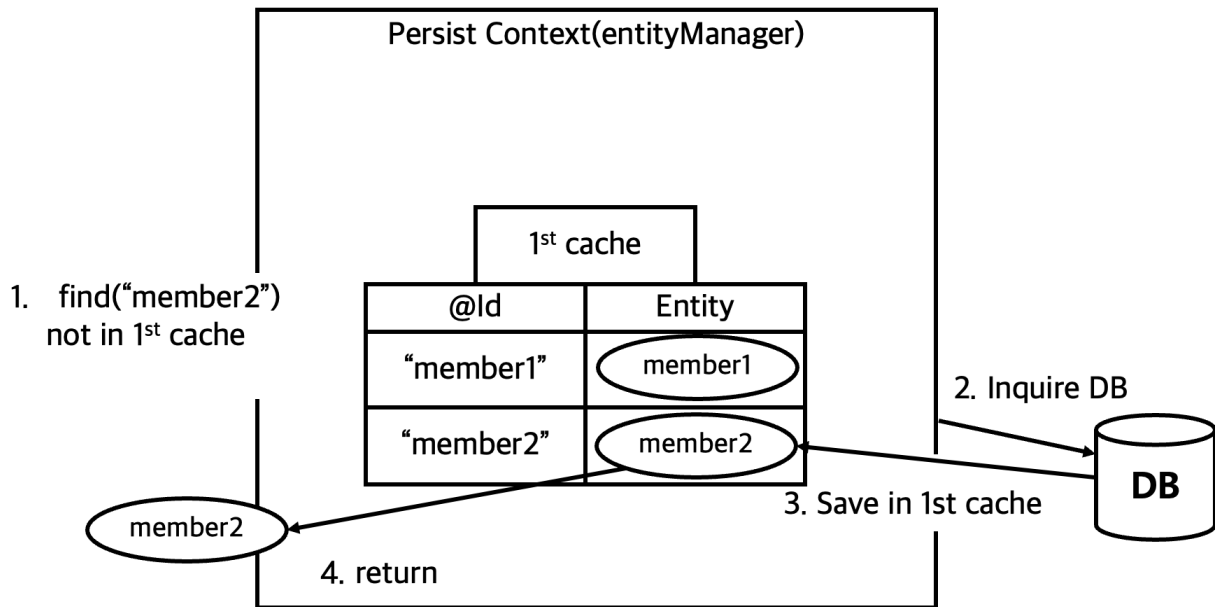
```
Hibernate:
  create table MEMBER (
    ID varchar(255) not null,
    NAME varchar(255),
    primary key (ID)
  )
findMember=회원1
```

insert , select문이 나오지 않았는데 검색은 되는 상황



em.persist(member)로 저장해도 DB에 저장되는 것이 아니다
그럼 em.find로 검색되는 값이 무엇이나 라고 생각할 것인데,
find는 호출되는 값은 1차 캐시에서 식별자 값으로 엔티티를 찾는다.
그 때 찾는 엔티티가 있으면 데이터베이스를 조회하지 않고 메모리
에 있는 1차 캐시에서 엔티티를 조회하는 것이다.

만약 1차 캐시에 데이터가 없으면 엔티티 매니저는 데이터베이스를
조회해서 엔티티를 생성한다. 그리고 1차 캐시에 저장한 후에 영속
상태의 엔티티를 반환한다.



1차 캐시에 없어 데이터베이스 조회

영속 엔티티의 동일성 보장

동일성 : 실제 인스턴스가 같다

동등성 : 인스턴스는 다르지만 가지고 있는 값은 같다.

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

System.out.print("a and b are same :: ");
System.out.println(a == b);
```

결과

```
a and b are same :: true
```

반복해서 호출해도 영속성 컨텍스트는 1차 캐시에 있는 같은 엔티티 인스턴스를 반환한다.

⇒ 영속성 컨텍스트는 성능상 이점과 엔티티의 동일성을 보장한다.

3.4.2 엔티티 등록

git 12.12 branch [3.4.2 엔티티 등록 - 쓰기 지연 확인](#) 참조

엔티티 매니저는 트랜잭션을 커밋하기 직전까지 데이터베이스에 엔티티를 저장하지 않고 내부 쿼리 저장소에 Insert SQL을 쌓아놓는다.

그리고 트랜잭션을 커밋할 때 모아둔 쿼리를 데이터베이스에 보내는데 이것을 **트랜잭션을 지원하는 쓰기 지연(transactional write-behind)** 이라 한다.

commit을 하지 않은 상태

```
public class Entity_Transaction {
    //
    public static void main(String[] args) {
        //엔티티 매니저 팩토리 생성
        EntityManagerFactory emf = Persistence.createEntityManagerFactory("Java_JPA_H2_Test");
        EntityManager em = emf.createEntityManager();
        EntityTransaction transaction = em.getTransaction();

        //엔티티 매니저는 데이터 변경시 트랜잭션을 시작해야 한다.
        transaction.begin(); // transaction start

        Member memberA = new Member();
        memberA.setId("memberA");
        memberA.setUsername("회원A");

        Member memberB = new Member();
        memberB.setId("memberB");
        memberB.setUsername("회원B");

        em.persist(memberA);
        em.persist(memberB);
        //여기까지 Insert SQL을 데이터에 보내지 않는다.
    }
}
```

결과

테이블 생성 쿼리는 출력되어도 Insert쿼리는 출력되지 않는다.

```
Hibernate:
    create table MEMBER (
        ID varchar(255) not null,
        NAME varchar(255),
        primary key (ID)
    )
```

⇒ 현재 Insert 쿼리는 쓰기 지연 SQL저장소에 보관된 상태이다.

commit 추가

```
public class Entity_Transaction {
    //
    public static void main(String[] args) {
        //엔티티 매니저 팩토리 생성
        EntityManagerFactory emf = Persistence.createEntityManagerFactory("Java_JPA_H2_Test");
        EntityManager em = emf.createEntityManager();
        EntityTransaction transaction = em.getTransaction();

        //엔티티 매니저는 데이터 변경시 트랜잭션을 시작해야 한다.
        transaction.begin(); // transaction start

        Member memberA = new Member();
        memberA.setId("memberA");
        memberA.setUsername("회원A");

        Member memberB = new Member();
        memberB.setId("memberB");
        memberB.setUsername("회원B");

        em.persist(memberA);
        em.persist(memberB);
        //여기까지 Insert SQL을 데이터에 보내지 않는다.

        transaction.commit();
    }
}
```

결과

비로소 Insert문이 생성된 것을 확인 할 수 있다.


```

Hibernate:
    create table MEMBER (
        ID varchar(255) not null,
        NAME varchar(255),
        primary key (ID)
    )

Hibernate:
    /* insert start.Member
    */ insert
    into
        MEMBER
        (NAME, ID)
    values
        (?, ?)

Hibernate:
    /* insert start.Member
    */ insert
    into
        MEMBER
        (NAME, ID)
    values
        (?, ?)

```



트랜잭션을 커밋하면 엔티티 매니저는 영속성 컨텍스트를 플러시 한다.
 플러시는 영속성 컨텍스트의 변경 내용을 데이터베이스에 동기화 하는 작업이다.
 쓰기 지연 SQL저장소에 모인 쿼리를 데이터 베이스에 보내는 것.

트랜잭션을 지원하는 쓰기 지연이 가능한 이유(?? 말뜻이 이해가 안됨)

```

public class Entity_Transaction {
    //
    public static void main(String[] args) {
        //엔티티 매니저 팩토리 생성
        EntityManagerFactory emf = Persistence.createEntityManagerFactory("hibernate");
        EntityManager em = emf.createEntityManager();
        EntityTransaction transaction = em.getTransaction();

        //엔티티 매니저는 데이터 변경시 트랜잭션을 시작해야 한다.
        transaction.begin(); // transaction start begin

        Member memberA = new Member();
        memberA.setId("memberA");
        memberA.setUsername("회원A");

        Member memberB = new Member();
        memberB.setId("memberB");
        memberB.setUsername("회원B");

        em.persist(memberA);
        em.persist(memberB); // save

        //여기까지 Insert SQL을 데이터에 보내지 않는다.

        transaction.commit(); // commit
    }
}

```

위의 코드를 두가지 경우로 생각해보자

1. 데이터 저장 즉시 등록 쿼리를 데이터 베이스에 전송

persist를 호출 할 때마다 즉시 데이터 베이스에 등록 쿼리를 보냄

마지막에 트랜잭션 커밋

2. 등록 쿼리를 메모리에 모아둔다

트랜잭션을 커밋할 때 모아둔 등록 쿼리를 데이터베이스에 보낸 후 커밋

⇒ 둘의 결과는 같다 - 결국 트랜잭션을 커밋하지 않으면 DB에 저장되지 않기 때문

3.4.3 엔티티 수정

SQL 수정 쿼리의 문제점

수정 쿼리는 개발 중 수정할 사항이 늘어날 수록 이전 쿼리를 수정하던가, 새로운 수정용 쿼리를 추가해야 한다. ⇒ 이렇게 되면 수정 쿼리가 많아지는 것은 물론 비즈니스 로직을 분석하기 위해 SQL을 계속 확인해야 한다. ⇒ **직간접적으로 비즈니스로직이 SQL에 의존하게 된다.**

JPA 수정 쿼리

git 12.12 branch **3.4.3 엔티티 수정 - 수정 확인** 참조

JPA로 엔티티를 수정할 때는 엔티티를 단순 조회해서 데이터만 변경하면 된다.

주석 처리한 update같은 메소드는 존재하지 않는다.

엔티티의 변경사항을 자동으로 감지해서 반영하는 기능을 **변경 감지(dirty checking)**이라 한다.

```
public class JPA_update_logic {
    //
    public static void main(String[] args) {
        //엔티티 매니저 팩토리 호출
        EntityManagerFactoryemf = MemberDataInsert.memberInsert();
        EntityManagerem = emf.createEntityManager();
        EntityTransactiontransaction = em.getTransaction();
        transaction.begin();

        //영속 엔티티 데이터 조회
        Member foundMemberA = em.find(Member.class, "memberA");
        //조회 확인
        System.out.println("호출된 MemberA 이름 :: " + foundMemberA.getUsername());

        //영속 엔티티 데이터 수정
        foundMemberA.setUsername("modified nameA");

        //em.update(); - 없는 메소드

        transaction.commit();
    }
}
```

결과

자동으로 update문이 실행된다.

```
호출된 MemberA 이름 :: 회원A
Hibernate:
    /* update
       start.Member */ update
       MEMBER
    set
       NAME=?
    where
       ID=?
```

변경 감지

JPA는 엔티티를 영속성 컨텍스트에 보관할 때, 최초의 상태를 복사해서 저장해 두는데 이것을 **스냅샷**이라고 한다. 그리고 flush시점에 스냅샷과 엔티티를 비교해서 변경된 엔티티를 찾는다.



변경 감지는 영속성 컨텍스트가 관리하는 영속 상태의 엔티티에만 적용된다.

비영속, 준영속처럼 영속성 컨텍스트의 관리를 받지 못하는 엔티티는 값을 변경해도 데이터베이스에 반영되지 않는다.

또한, 변경을 감지한 후 update할 때 **JPA는 기본적으로 엔티티의 모든 필드를 업데이트 한다.**

```
호출된 MemberA 이름 ::: 회원A
Hibernate:
    /* update
      start.Member */ update
      MEMBER
    set
      age=?,
      NAME=?
    where
      ID=?
```

age field를 추가하고 name만 업데이트 한 경우

모든 필드를 업데이트하면 데이터 전송량이 증가하지만 아래와 같은 이점이 있다.

- 수정 쿼리가 항상 같다.(바인딩 데이터만 다름) → 애플리케이션 로딩 시점에 쿼리를 미리 생성하고 재사용할 수 있다.
- 동일한 쿼리를 보내면 DB는 이전에 파싱된 쿼리를 재사용 할 수 있다.

@DynamicUpdate

필드가 많거나 저장되는 내용이 클 때 동적으로 쿼리를 생성하는 확장기능

@DynamicInsert 도 있다.

Member 엔티티

```
@Entity
@DynamicUpdate
@Table(name="MEMBER")
public class Member {

    @Id
    @Column(name = "ID")
    private String id;

    @Column(name = "NAME")
    private String username;

    private Integer age;
```

logic

username만 수정한다

```
public class JPA_update_logic {
    //
    public static void main(String[] args) {
        //엔티티 매니저 팩토리 호출
        EntityManagerFactory emf = MemberDataInsert.memberInsert();
        EntityManager em = emf.createEntityManager();
        EntityTransaction transaction = em.getTransaction();
        transaction.begin();

        //영속 엔티티 데이터 조회
        Member foundMemberA = em.find(Member.class, "memberA");
        //조회 확인
        System.out.println("호출된 MemberA 이름 ::: " + foundMemberA.getUsername());

        //영속 엔티티 데이터 수정
        foundMemberA.setUsername("modified nameA");

        //em.update(); - 없는 메소드

        transaction.commit();
    }
}
```

사용 전 후 비교 → 사용 후에는 변경되는 이름만 쿼리를 생성한다.

```
호출된 MemberA 이름 ::: 회원A
Hibernate:
    /* update
      start.Member */ update
      MEMBER
    set
      age=?,
      NAME=?
    where
      ID=?
```

전

```
호출된 MemberA 이름 ::: 회원A
Hibernate:
    /* update
      start.Member */ update
      MEMBER
    set
      NAME=?
    where
      ID=?
```

후

3.4.4 엔티티 삭제

```
public static void memberDelete(EntityManager em){
    //
    //삭제
    Member memberA = em.find(Member.class, "memberA");
    em.remove(memberA);
    //삭제 후 조회
    List<Member> members = em.createQuery("select m from Member m", Member.class).getResultList();
    for (Member member : members){
        System.out.println("member name after deleted::: " + member.getUsername());
    }
}
```

```

member name ::: 회원A
member name ::: 회원B
members.size=2
Hibernate:
/* delete start.Member */ delete
from
    MEMBER
where
    ID=?
Hibernate:
/* select
    m
from
    Member m */ select
    member0_.ID as ID1_0_,
    member0_.age as age2_0_,
    member0_.NAME as NAME3_0_
from
    MEMBER member0_
member name after deleted::: 회원B

```

`em.remove` 에 삭제 대상 엔티티를 넘겨주면 엔티티를 삭제함

삭제 쿼리를 쓰기 지연 SQL저장소에 등록하고 flush후 실제 DB에 삭제 쿼리 전달

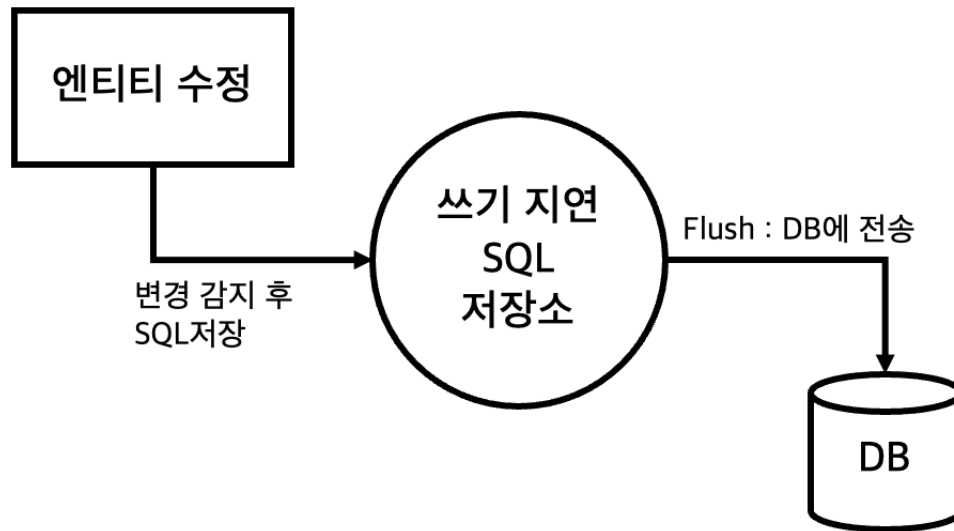
`em.remove` 를 호출하는 순간 대상이 되는 객체는 영속성 컨텍스트에서 제거된다.

⇒ 다시 사용하지 말고 가비지 컬렉션의 대상이 되도록 내버려 둘 것

3.5 플러시

플러시(flush)는 영속성 컨텍스트의 변경 내용을 데이터베이스에 반영한다. - 동기화

동작 방식



1. 변경 감지가 동작, 영속성 컨텍스트에 있는 모든 엔티티를 스냅샷과 비교해 수정된 엔티티를 찾는다. 수정된 엔티티는 수정 쿼리를 만들어 쓰기 지연 SQL 저장소에 등록한다.
2. 쓰기 지연 SQL 저장소의 쿼리를 데이터베이스에 전송한다. (등록, 수정, 삭제 쿼리)

flush 하는 방법

1. `em.flush()` 직접 호출

test와 타 프레임워크와 JPA를 함께 사용할 때를 제외하고 거의 사용하지 않는다.

2. 트랜잭션 커밋 시 자동 호출

DB에 변경 내용을 SQL로 전달하지 않고 트랜잭션만 커밋하면 DB에 반영되지 않는다.

트랜잭션 커밋전에 반드시 플러시를 호출해서 db에 반영해야 한다

3. JPQL 쿼리 실행 시 자동 호출

사용자가 쓰기 지연 SQL 저장소에 등록하고 플러시를 안 하고 JPQL을 실행한 경우 쓰기 지연 SQL 저장소 안의 쿼리들도 실행 한 후 JPQL이 실행되는 결과를 사용자에게 보여주어야 하므로 flush가 호출된다.

3.5.1 Flush 모드 옵션

```
em.setFlushMode(FlushModeType.COMMIT); // 플러시모드 직접 설정
em.setFlushMode(FlushModeType.AUTO); // 디폴트, 커밋할 때만 플러시
```

3.6 준영속

영속 상태의 엔티티가 영속성 컨텍스트에서 분리(detached)된 것을 준영속 상태라 한다.

⇒ 준영속 상태의 엔티티는 영속성 컨텍스트가 제공하는 기능을 사용할 수 없다.

엔티티가 준영속 상태가 되는 경우

1. `detach()` 로 직접 준영속화 한 경우
2. `clear()` 로 영속성 컨텍스트 초기화 한 경우
3. `close()` 로 영속성 컨텍스트를 종료한 경우

준영속 상태의 특징

- 거의 비영속 상태에 가깝다
- 식별자 값을 가지고 있다.

이미 한 번 영속상태였으므로 반드시 식별자 값을 가지고 있다.

- 지연로딩을 할 수 없다.

병합 : `merge()`


`merge()` 메소드는 준영속 상태의 엔티티를 받아서 새로운 영속 상태의 엔티티를 반환한다.

또한 비영속 엔티티도 영속 상태로 만들 수 있다.

그래서 어디에 사용하는가?

Detached Entity Objects

Detached entity objects are objects in a special state in which they are not managed by any but still represent objects in the database. Compared to managed entity objects, detached objects are limited in functionality: Many JPA methods do not accept detached objects (e.g.).

 <https://www.objectdb.com/java/jpa/persistence/detach>

Detached objects are useful in situations in which an `EntityManager` is not available and for transferring objects between different `EntityManager` instances.

⇒ 엔티티 매니저를 사용할 수 없는 경우, 서로 다른 엔티티 매니저 인스턴스 간에 객체를 전달하는데 유용하다고 함

```
# 한 일
* lecture : 모바일아카데미 관련 개발 및 테스트

# 할 일
* lecture : 모바일아카데미 관련 테스트
* notie : 테스트 코드 작성
```