1. 영속성 컨텍스트

영속성 컨텍스트(Persistence Context)는 **엔터티(Entity) 객체들의 상태를 관리하는 환경**을 말합니다. 영속성 컨텍스트는 JPA가 데이터베이스와 상호작용할 때 엔터티 객체의 라이프사이클을 관리하고, 데이터베이스에서 읽어오거나 저장할 때 엔터티 객체의 상태를 추적합니다. 영속성 컨텍스트는 논리적인 개념으로 코드에서 보이지 않습니다. 엔티티 매니저를 통해서 영속성 컨텍스트에 접근이 가능합니다.

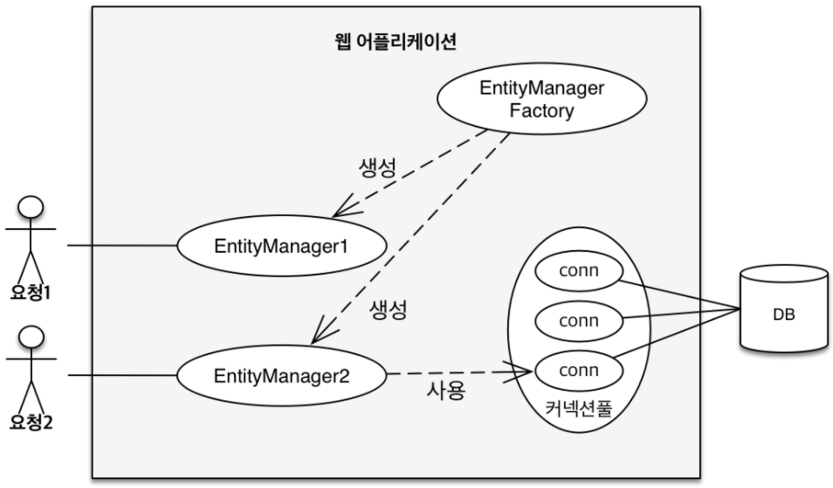
1.1 영속성 컨텍스트 특징

영속성 컨텍스트의 주요 역할은 다음과 같습니다:

1. **엔터티의 관리 (Entity Management):**
   - 영속성 컨텍스트는 엔터티 객체의 생명주기를 관리
   - 엔터티를 데이터베이스에서 읽어오거나 데이터베이스에 저장할 때 이를 관리
2. **엔터티의 캐싱 (Caching):**
   - 영속성 컨텍스트는 엔터티 객체를 캐싱하여 동일한 엔터티가 여러 번 로딩되는 것을 방지하고, 성능을 향상
3. **트랜잭션 범위의 쓰기 지연 (Transaction-scoped Write-Behind):**
   - 영속성 컨텍스트는 트랜잭션 범위에서 변경된 엔터티를 데이터베이스에 실제로 쓰기 전까지 지연
   - 이를 통해 성능을 최적화하고 데이터베이스에 효율적으로 쓰기를 수행
4. **Dirty Checking:**
   - 영속성 컨텍스트는 트랜잭션 커밋 시에 엔터티 객체의 변경사항을 감지하고, 변경된 엔터티만을 데이터베이스에 반영
5. **컨텍스트 범위의 쿼리 캐싱 (Context-scoped Query Caching):**
   - 영속성 컨텍스트는 쿼리 결과도 캐싱하여 반복적인 쿼리 수행을 최적화

영속성 컨텍스트는 주로 `EntityManager`를 통해 사용되며, 엔터티 매니저를 통해 데이터베이스와의 상호작용이 이루어집니다.
엔터티 매니저는 **영속성 컨텍스트와 관련된 트랜잭션을 시작하고 커밋하는 등의 작업을 수행**합니다.

1.2 영속성 컨텍스트 동작 원리



엔티티 매니저 팩토리와 엔티티 매니저는 웹 어플리케이션이 구동하는 시점에 `EntityManagerFactory` 를 생성하여 웹 어플리케이션이 가지고 있으며, 사용자의 요청이 있을 때 **EntityManager**를 생성하여 **커넥션 풀(Connection Pool) 을 사용해서 DB를 핸들링 하게 됩니다.**

1.3 영속성 컨텍스트 예제 1

```
EntityManager.persist(entity)
```

- 위의 엔티티 매니저 메서드인 `persist(entity)`는 `entity`를 **영속성 컨텍스트에 엔터티를 추가하는 역할**
- 이 메서드를 호출하면 전달된 엔터티 객체가 **영속성 컨텍스트에 추가**되고, 향후 **트랜잭션 커밋 시에 데이터베이스에 저장**

영속성 컨텍스트에 접근하기 위해 `EntityManager.persist(entity)`을 사용하는 것은 일반적으로 엔터티를 새로이 저장할 때 사용합니다. 즉, 해당 엔터티가 데이터베이스에 존재하지 않는 새로운 엔터티일 경우에 해당합니다.

예를 들어:

```
EntityManager entityManager = entityManagerFactory.createEntityManager();
EntityTransaction transaction = entityManager.getTransaction();

try {
    transaction.begin();

    // 새로운 엔터티를 생성하고 영속성 컨텍스트에 추가
    MyEntity entity = new MyEntity();
    entityManager.persist(entity);

    transaction.commit();
} catch (Exception e) {
    if (transaction != null && transaction.isActive()) {
        transaction.rollback();
    }
    e.printStackTrace();
} finally {
    entityManager.close();
}
```

위의 예제에서 `entityManager.persist(entity)`를 호출하여 새로운 `MyEntity` 객체를 영속성 컨텍스트에 추가하고, 트랜잭션 커밋 시에 데이터베이스에 저장하게 됩니다.

반면에 이미 데이터베이스에 존재하는 엔터티를 가져와서 수정하고자 할 때는 `EntityManager.find()` 등의 메서드를 사용하여 영속성 컨텍스트에서 엔터티를 가져온 후, 해당 엔터티를 수정하고 트랜잭션 커밋 시에 변경사항이 반영됩니다.

2. 엔티티 매니저 팩토리

엔터티 매니저 팩토리(EntityManagerFactory)는 **JPA에서 엔터티 매니저(EntityManager) 인스턴스를 생성하는데 사용되는 팩토리**입니다. 엔터티 매니저는 **영속성 컨텍스트를 관리**하고, **데이터베이스와의 상호작용을 처리하는 핵심 객체 중 하나**입니다.

여러 엔터티 매니저를 생성하고 관리하기 위해 엔터티 매니저 팩토리가 사용됩니다. 엔터티 매니저 팩토리는 어플리케이션의 라이프사이클 동안 단 하나만 생성되어야 합니다. 주로 어플리케이션의 시작 시점에 생성되고, 어플리케이션 종료 시에 닫히게 됩니다.

2.1 엔티티 매니저 팩토리 예제 1

일반적으로 엔터티 매니저 팩토리를 생성하는 코드는 다음과 같습니다:

```
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

public class JpaExample {
    public static void main(String[] args) {
        // "persistence-unit-name"은 persistence.xml 파일에 정의된 영속성 유닛의 이름
입니다.
        EntityManagerFactory entityManagerFactory =
                Persistence.createEntityManagerFactory("persistence-unit-name");

        // EntityManagerFactory를 사용하여 EntityManager를 생성할 수 있습니다.
        // 생성된 EntityManager는 데이터베이스와의 트랜잭션 및 엔터티 관리를 담당합니다.
        EntityManager entityManager = entityManagerFactory.createEntityManager();

        // 어플리케이션 로직 수행...

        // 사용이 끝난 EntityManager 및 EntityManagerFactory는 반드시 닫아주어야 합니
다.
        entityManager.close();
        entityManagerFactory.close();
    }
}
```

위의 코드에서 "persistence-unit-name"은 persistence.xml 파일에서 정의한 **영속성 유닛의 이름**을 나타냅니다. persistence.xml 파일은 JPA 설정을 담고 있는 파일로, 데이터베이스 연결 정보, 엔터티 클래스의 위치, 트랜잭션 관리 방식 등을 설정할 수 있습니다.

엔터티 매니저 팩토리는 **어플리케이션 전체에서 공유되어야 하므로, 주로 싱글톤 패턴이나 의존성 주입(Dependency Injection)을 통해 관리됩니다.**

3. 엔티티 매니저

엔터티 매니저(EntityManager)는 Java Persistence API (JPA)에서 **영속성 컨텍스트를 관리**하고, **엔터티 객체와 데이터베이스 간의 상호작용을 담당하는 인터페이스**입니다. 엔터티 매니저는 **주로 데이터베이스와의 트랜잭션을 관리하고, 엔터티의 영속성을 제어**합니다.

3.1 엔티티 매니저 특징

여러 가지 중요한 기능을 수행합니다:

1. **영속성 컨텍스트 관리:**
    - 엔터티 매니저는 영속성 컨텍스트를 생성하고 관리
    - 영속성 컨텍스트는 엔터티 객체의 생명주기를 추적하고, 엔터티와 데이터베이스 간의 상태를 동기화
2. **트랜잭션 관리:**

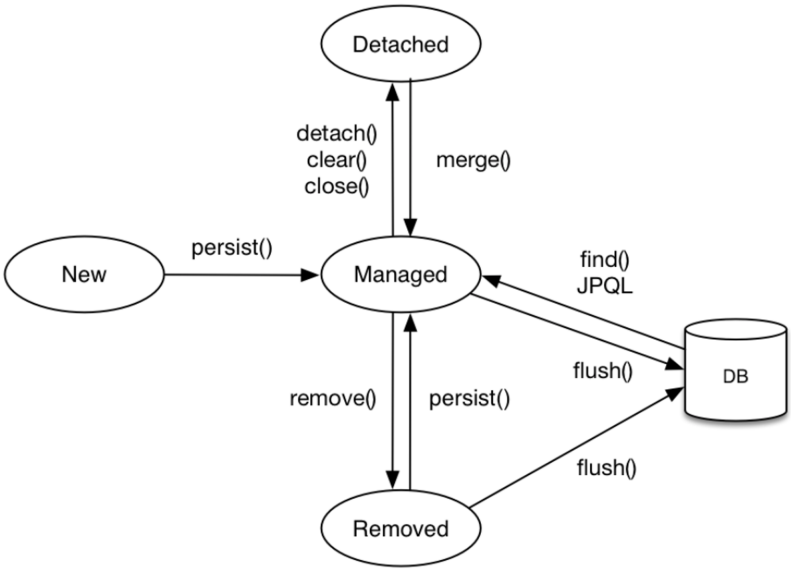
- 엔터티 매니저는 트랜잭션을 시작하고 커밋하거나 롤백하는 등의 트랜잭션 관리를 수행
- 이를 통해 여러 엔터티에 대한 변경 사항을 원자적으로 처리

3. **데이터베이스와의 상호작용:**
   - 엔터티 매니저는 데이터베이스와의 상호작용을 담당
   - 엔터티를 데이터베이스에 저장하거나 조회하고, 쿼리를 실행하여 데이터를 가져 옴
4. **캐싱:**
   - 엔터티 매니저는 영속성 컨텍스트 내에서 엔터티를 캐싱하여 성능을 향상
   - 이를 통해 동일한 엔터티를 반복적으로 로딩하는 것을 방지하고, 애플리케이션의 성능을 최적화
5. **엔터티 조회 및 수정:**
   - 엔터티 매니저는 데이터베이스에서 엔터티를 조회하거나 수정하는 작업을 수행
   - JPQL (Java Persistence Query Language)을 사용하여 데이터베이스에 대한 쿼리를 작성하고 실행이 가능
6. **Dirty Checking:**
   - 트랜잭션 커밋 시에 엔터티 매니저는 영속성 컨텍스트 내의 엔터티 객체의 변경사항을 감지
   - 변경된 엔터티만을 데이터베이스에 반영
7. **쿼리 실행 및 결과 처리:**
   - 엔터티 매니저는 JPQL이나 Criteria API를 사용하여 데이터베이스에 대한 쿼리를 실행하고, 그 결과를 처리

엔터티 매니저는 주로 엔터티 매니저 팩토리(`EntityManagerFactory`)를 통해 생성되며, 트랜잭션 범위 내에서 사용됩니다.
사용이 끝나면 엔터티 매니저를 닫아야 합니다.

4. 엔터티의 생명주기

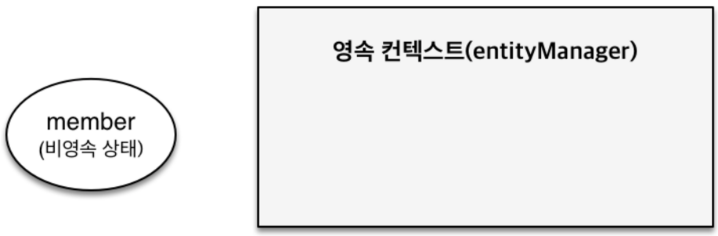
엔터티(Entity)의 생명주기는 **해당 엔터티가 영속성 컨텍스트 내에서 어떻게 관리되고 변화하는지를 나타내는 개념**입니다.
JPA에서 엔터티는 다음과 같은 생명주기를 가지고 있습니다:



4.1 New/Transient

- 새로운 엔터티 (New/Transient)는 엔터티 객체가 생성되고, 이 객체는 영속성 컨텍스트와 관련이 없습니다.(비영속 상태)
- 데이터베이스에 저장되지 않은 상태입니다.
- 즉, 새로운 인스턴스가 생성된 상태를 의미합니다.

```
MyEntity entity = new MyEntity(); // 새로운 엔터티 생성
```




```
//객체를 생성한 상태(비영속 상태)
Member member = new Member();
member.setId("member1");
member.setUsername("회원1");
```

4.2 managed

- `EntityManager.persist(entity)` 메서드를 통해 새로운 엔터티를 영속성 컨텍스트에 등록하면 영속 상태가 됩니다.
- 영속 상태 (Managed)의 엔터티는 영속성 컨텍스트가 관리하며, 이 상태에서는 데이터베이스와 동기화되어 변경사항이 추적됩니다.

```
EntityManager entityManager = entityManagerFactory.createEntityManager();
EntityTransaction transaction = entityManager.getTransaction();

try {
    transaction.begin();

    // 영속성 컨텍스트에 등록되어 영속 상태가 됨
    entityManager.persist(entity);

    transaction.commit();
} catch (Exception e) {
    if (transaction != null && transaction.isActive()) {
        transaction.rollback();
    }
    e.printStackTrace();
} finally {
    entityManager.close();
}
```


```
//객체를 생성한 상태(비영속)
Member member = new Member();
member.setId("member1");
member.setUsername("회원1");

EntityManager em = emf.createEntityManager();
em.getTransaction().begin();

//객체를 저장한 상태(영속)
em.persist(member);
```

4.3 detached

- 준영속 상태 (Detached)는 영속 상태의 엔터티가 영속성 컨텍스트에서 분리되면 준영속 상태가 됩니다.
- 이 상태에서는 영속성 컨텍스트가 더 이상 해당 엔터티를 관리하지 않습니다.
- 주로 트랜잭션이 종료되거나 `EntityManager.detach(entity)` 메서드를 호출할 때 발생합니다.

```
entityManager.detach(entity); // entity를 준영속 상태로 변경
//회원 엔티티를 영속성 컨텍스트에서 분리, 준영속 상태
em.detach(member);
```

4.4 removed

- 삭제 상태 (Removed)는 `EntityManager.remove(entity)` 메서드를 호출하여 영속 상태의 엔터티를 삭제하면 삭제 상태가 됩니다.
- 이후 트랜잭션 커밋 시에 데이터베이스에서 해당 엔터티가 삭제됩니다.

```
entityManager.remove(entity); // 엔터티를 삭제 상태로 변경
//객체를 삭제한 상태(삭제)
em.remove(member);
```

이러한 생명주기는 엔터티 매니저를 통해 제어되며, 영속성 컨텍스트 내에서 엔터티의 상태가 변화합니다.
엔터티의 생명주기 관리를 통해 JPA는 데이터베이스와 자바 객체 간의 일관성을 유지하고, 효과적으로 엔터티를 관리할 수 있습니다.

5. 영속성 컨텍스트의 이점

영속성 컨텍스트는 주로 `EntityManager`를 통해 사용되며, 엔터티 매니저를 통해 데이터베이스와의 상호작용이 이루어집니다.
엔터티 매니저는 **영속성 컨텍스트와 관련된 트랜잭션을 시작하고 커밋하는 등의 작업을 수행**합니다.

영속성 컨텍스트의 이점은 다음과 같습니다:

1. **1차 캐시**
2. **동일성(identity) 보장**
3. **트랜잭션을 지원하는 쓰기 지연(transactional write-behid)**
4. **변경 감지(Dirty Checking)**
5. **지연 로딩(Lazy Loading)**

6. 1차 캐시

1차 캐시(First-Level Cache)는 **영속성 컨텍스트(Persistence Context) 내에서 엔터티(Entity) 객체를 저장하고 있는 캐시**를 말합니다. 영속성 컨텍스트는 엔터티 매니저(EntityManager)를 통해 데이터베이스와 상호 작용하는 환경이며, 이 환경 내에서 엔터티 객체들은 1차 캐시에 저장됩니다.

6.1 1차 캐시 특징

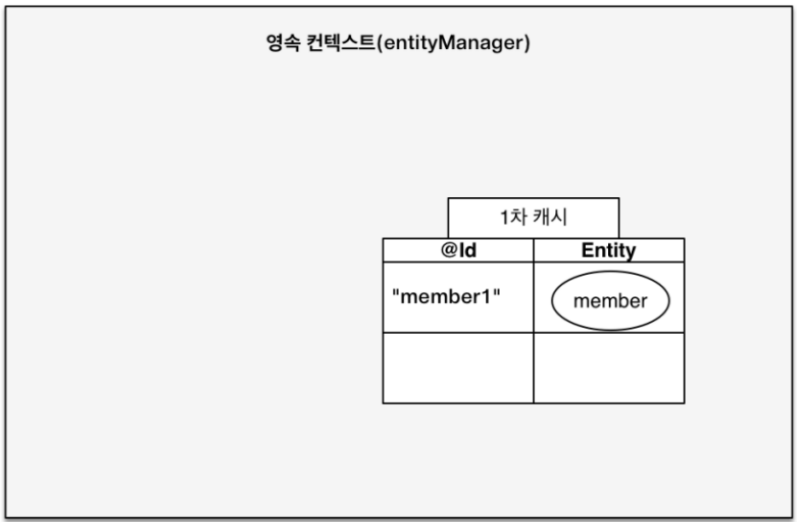
1차 캐시의 주요 특징은 다음과 같습니다:

1. **엔터티 객체 저장:**
   - 영속성 컨텍스트는 데이터베이스에서 읽어온 엔터티 객체를 메모리에 저장
   - 이 메모리에 저장된 엔터티 객체의 집합이 1차 캐시
2. **동일성 보장:**
   - 같은 식별자(primary key)를 가진 엔터티 객체는 1차 캐시에서 고유한 식별자를 통해 관리됨
   - 따라서 동일한 엔터티를 여러 번 조회할 경우 1차 캐시에서 해당 엔터티를 가져오므로 동일성이 보장함
3. **쿼리 수행 최적화:**
   - 동일한 트랜잭션 내에서는 1차 캐시에서 엔터티를 찾음
   - 따라서 데이터베이스에 중복 쿼리를 보내는 것을 방지하고 성능을 최적화함
4. **Dirty Checking 및 트랜잭션 지원:**
   - 1차 캐시는 트랜잭션 내에서 엔터티 객체의 변경을 추적하고,
   - 트랜잭션이 커밋될 때 변경된 엔터티를 데이터베이스에 반영
5. **트랜잭션 범위 캐시:**
   - 1차 캐시는 트랜잭션 범위 내에서만 유효하며, 트랜잭션이 커밋되거나 롤백되면 1차 캐시도 함께 초기화됨

1차 캐시는 JPA에서 영속성 컨텍스트의 핵심 기능 중 하나로, 성능 향상과 일관성 유지에 기여

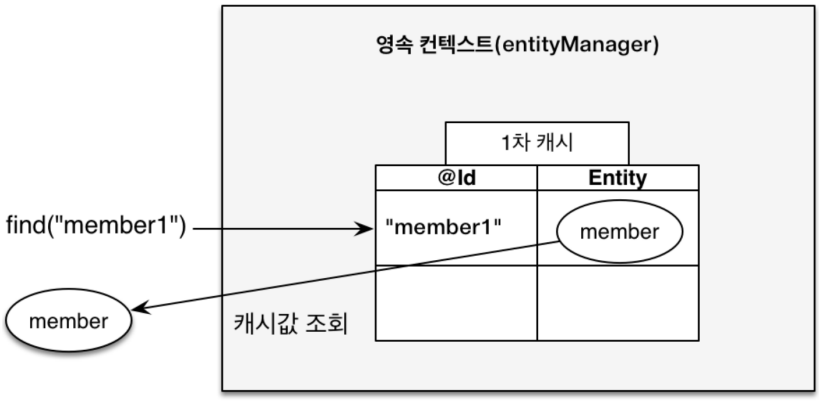
6.2 1차 캐시 예제 1

1. 생성된 엔티티를 `em.persist(member)` **수행할때 1차 캐시에 저장**됩니다.



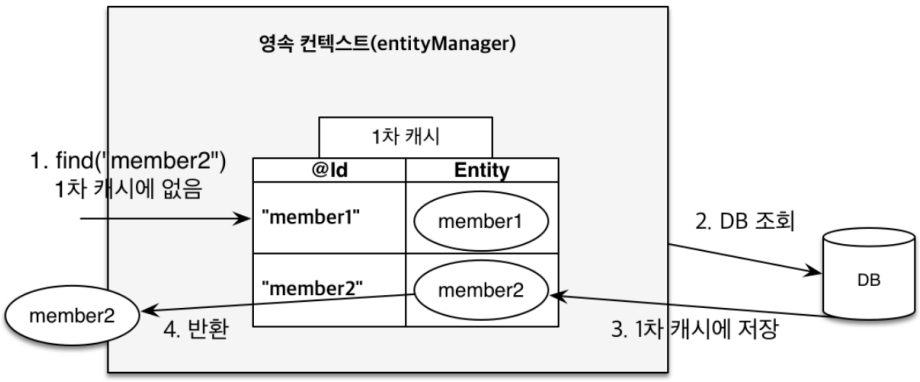

```
//엔티티를 생성한 상태(비영속)
Member member = new Member();
member.setId("member1");
member.setUsername("회원1");

//엔티티를 영속
em.persist(member);
```

1. `em.find(Member.class, "member1");` 메서드가 수행될때 1차 캐시에서 먼저 같은 id 값을 갖는 엔티티 객체가 있는지 찾습니다.
   엔티티 객체가 1차 캐시에 존재하면 데이터베이스에 쿼리를 보내지 않고 해당 객체를 바로 반환합니다.




```
//엔티티를 생성한 상태(비영속)
Member member = new Member();
member.setId("member1");
member.setUsername("회원1");

//엔티티를 영속 = 1차 캐시에 저장됩니다.
em.persist(member);

//1차 캐시에서 조회
Member findMember = em.find(Member.class, "member1");
```

1. `em.find(Member.class, "member1");` 메서드 수행에서 1차 캐시에 찾는 엔티티가 없으면 데이터베이스를 조회합니다.
   조회한 데이터를 1차 캐시에 저장합니다.




```
// 1차 캐시에 없으면 데이터베이스에서 조회
Member findMember2 = em.find(Member.class, "member2");
```

7. 동일성 보장

영속 엔티티의 동일성(identity) 보장합니다. 1차 캐시로 반복 가능한 읽기(REPEATABLE READ) 등급의 트랜잭션 격리 수준을 데이터베이스가 아닌 애플리케이션 차원에서 제공합니다. 즉, 1차 캐시 안에 엔티티의 동일성을 보장합니다.

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

System.out.println(a == b); //동일성 비교 true
```

멤버 b는 1차 캐시에서 데이터를 가져오므로 a와 b는 같은 객체입니다.
id 값이 같은 두 객체가 동일한 객체라는 것은 자바의 철학과 유사합니다.

8. 트랜잭션을 지원하는 쓰기 지연

트랜잭션을 지원하는 쓰기 지연(transactional write-behid)은 SQL 쿼리를 commit 시점에 flush 할 때 한번에 날린다는 의미입니다.

구체적으로, 트랜잭션을 지원하는 쓰기 지연 기능을 사용할 때, **영속성 컨텍스트 내에서 모든 변경된 엔터티의 데이터베이스에 대한 변경사항을 한 번에 데이터베이스에 반영한다는 의미**입니다.

일반적으로 JPA에서는 트랜잭션 내에서 엔터티 객체를 수정하면 해당 변경사항이 즉시 데이터베이스에 반영되는 것이 아니라, **트랜잭션이 커밋될 때까지 지연**됩니다. **이를 트랜잭션을 지원하는 쓰기 지연(Lazy Writing)라고 합니다.**

8.1 flush

트랜잭션을 커밋할 때 flush가 발생하면, 영속성 컨텍스트 내에 있는 변경된 엔터티들의 상태를 확인하고, 해당 변경사항을 데이터베이스에 적용합니다. 이때 flush는 **데이터베이스에 변경사항을 반영하기 위해 SQL 쿼리를 생성하고 실행하는 작업을 의미**합니다.

- JPA에서의 "flush"는 데이터베이스와의 상호작용을 위한 SQL 쿼리를 생성하고 실행하는 과정을 의미합니다.
- 이때 발생하는 SQL 쿼리는 `INSERT`, `UPDATE`, `DELETE` 등이 포함될 수 있습니다.

따라서 "flush가 발생한다"는 말은 영속성 컨텍스트에서의 변경사항을 실제 데이터베이스에 적용하기 위해 SQL 쿼리를 실행한다는 의미입니다. 이는 주로 트랜잭션을 커밋할 때 발생하며, 때로는 명시적으로 `flush()` 메서드를 호출하여 강제로 수행할 수도 있습니다.

8.2 flush 방법

JPA에서의 flush는 영속성 컨텍스트의 상태를 데이터베이스에 동기화하는 작업을 의미합니다.
Flush 작업은 다음과 같은 상황에서 자동으로 발생할 수 있습니다:

1. 트랜잭션 커밋 시:
    - 트랜잭션을 커밋할 때, 영속성 컨텍스트의 변경사항을 데이터베이스에 반영하기 위해 flush가 자동으로 발생합니다.
2. JPQL 쿼리 실행 시:
    - JPQL(Java Persistence Query Language)을 사용하여 직접 쿼리를 실행할 때, flush가 발생합니다.
    - 이때 쿼리 실행 이전에 영속성 컨텍스트의 변경사항을 데이터베이스에 동기화합니다.
3. 명시적인 flush 호출 시:
    - `EntityManager` 인터페이스에서 제공하는 `flush()` 메서드를 호출하여 명시적으로 flush를 수행할 수 있습니다.

```
EntityManager entityManager = entityManagerFactory.createEntityManager();
EntityTransaction transaction = entityManager.getTransaction();

try {
    transaction.begin();

    // 엔터티 수정 등의 작업 수행

    // 명시적으로 flush 호출
    entityManager.flush();

    transaction.commit();
} catch (Exception e) {
    if (transaction != null && transaction.isActive()) {
        transaction.rollback();
    }
    e.printStackTrace();
} finally {
    entityManager.close();
}
```

Flush 작업은 영속성 컨텍스트의 상태와 데이터베이스의 상태를 일치시키는 역할을 합니다. 따라서 트랜잭션이 커밋되기 전에 변경사항을 데이터베이스에 적용하여 일관성을 유지하고, 데이터베이스에 반영되지 않은 변경사항을 미리 확인할 수 있습니다.

Flush는 성능 상의 이유로 자주 발생하는 것이 아니며, 보통은 트랜잭션 커밋 시에 알아서 처리됩니다. 하지만 특정 시점에 명시적으로 flush를 호출해야 하는 경우도 있습니다. 참고로, **플러시를 해도 영속성 컨텍스트의 엔티티가 지워지지 않고, 1차 캐시 정보가 유지**됩니다. 1차 캐시의 엔티티를 지우려면 `em.clear()` 를 호출해야 합니다.

8.3 flush 예제 1

예를 들어:

```
EntityManager entityManager = entityManagerFactory.createEntityManager();
EntityTransaction transaction = entityManager.getTransaction();

try {
    transaction.begin();

    // 엔터티 수정 등의 작업 수행

    // 트랜잭션 커밋 시에 flush가 발생
    transaction.commit();
} catch (Exception e) {
    if (transaction != null && transaction.isActive()) {
        transaction.rollback();
    }
    e.printStackTrace();
} finally {
    entityManager.close();
}
```

위의 코드에서 `transaction.commit()`이 호출될 때, 영속성 컨텍스트의 변경사항이 데이터베이스에 반영되기 위해 flush가 발생합니다.

여러 개의 엔터티를 수정하더라도 트랜잭션이 커밋되기 전까지는 데이터베이스에 대한 실제 변경이 발생하지 않습니다.
이는 데이터베이스와의 효율적인 상호작용과 성능 향상에 기여합니다.

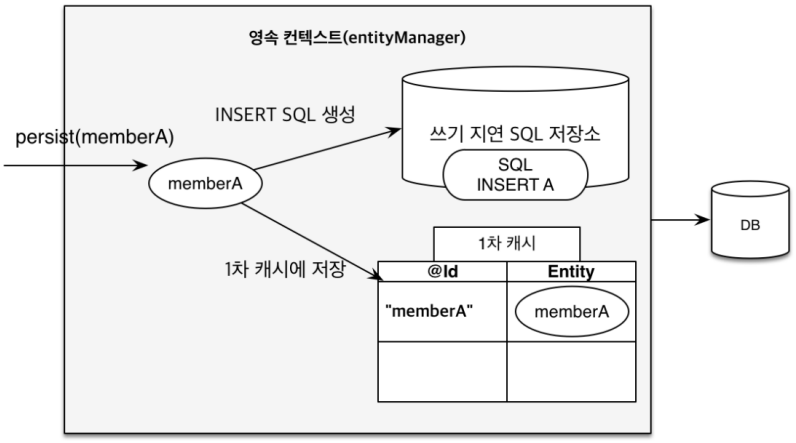
8.4 flush 예제 2

1. 아래 코드에서 `em.persist(memberA);, em.persist(memberB);`를 수행하면서 생성되는 SQL 쿼리 INSERT 2개는 commit 시점에 한 번에 쓰기 지연 SQL 저장소로 전송됩니다.

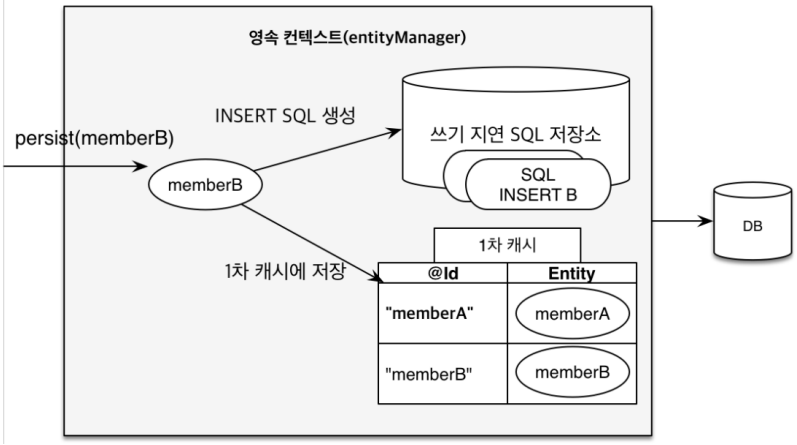
```
EntityManager em = emf.createEntityManager();
EntityTransaction transaction = em.getTransaction();
//엔티티 매니저는 데이터 변경시 트랜잭션을 시작해야 한다.
transaction.begin(); // [트랜잭션] 시작

em.persist(memberA);
em.persist(memberB);
//여기까지 INSERT SQL을 데이터베이스에 보내지 않는다.

//커밋하는 순간 데이터베이스에 INSERT SQL을 보낸다.
transaction.commit(); // [트랜잭션] 커밋
```

1. 생성되는 SQL 쿼리는 쓰기 지연 SQL 저장소에 저장되어 있습니다.

```
em.persist(memberA);
```

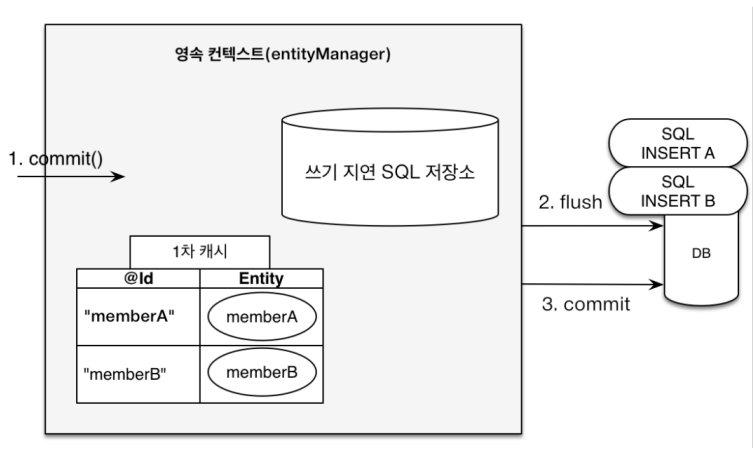



```
em.persist(memberB);
```



1. transaction을 커밋할 때 flush가 호출되어서 한 번에 쿼리를 날립니다.
   INSERT뿐만 아니라 UPDATE, DELETE도 커밋 시점에 한 번에 쿼리가 나갑니다.

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



9. 변경 감지

변경 감지(Dirty Checking)는 JPA의 영속성 컨텍스트에 등록된 엔티티의 데이터를 수정하면 `update`를 호출하지 않아도 자동으로 UPDATE 쿼리를 전송합니다.

```
EntityManager em = emf.createEntityManager();
EntityTransaction transaction = em.getTransaction();
transaction.begin(); // [트랜잭션] 시작

// 영속 엔티티 조회
Member memberA = em.find(Member.class, "memberA");

// 영속 엔티티 데이터 수정
memberA.setUsername("hi");
memberA.setAge(10);

//--- em.update(member) 이런 코드가 있어야 하지 않을까? ----- 없어도 됩니다!
transaction.commit(); // [트랜잭션] 커밋
```

9.1 변경 감지 예제 1

```
Member member = new Member();
//member.setId("memberA");
member.setUsername("회원1");
member.setAge(20);

em.persist(member);

Member memberA = em.find(Member.class, "memberA");
memberA.setUsername("hi");
memberA.setAge(10);
```

SELECT * FROM MEMBER;

ID	AGE	NAME
memberA	10	hi

(1 row, 3 ms)

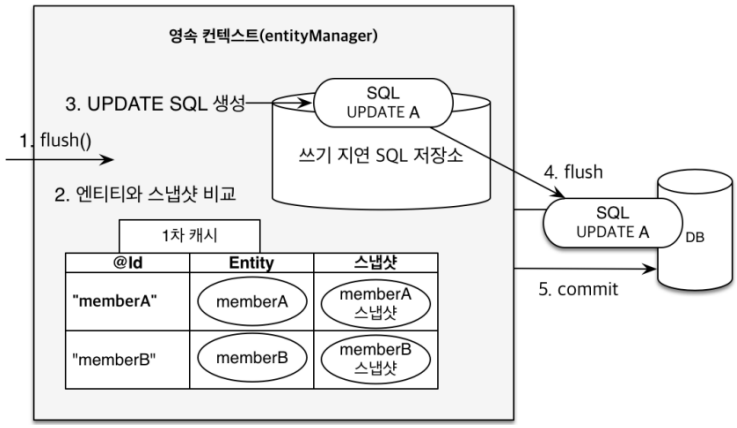
나이는 10, 이름은 hi로 잘 변경된 것을 확인할 수 있습니다.
이렇게 엔티티의 변경사항을 데이터베이스에 자동으로 반영하는 기능을 변경 감지(dirty checking)이라고 합니다.

9.2 변경 감지 동작 원리

변경 감지 기능의 과정은 다음과 같습니다.

변경 감지

(Dirty Checking)



1. 트랜잭션을 커밋하면 엔티티 매니저 내부에서 먼저 플러시(`flush()`)가 호출됩니다.
2. 엔티티와 스냅샷을 비교해서 변경된 엔티티를 찾습니다.
3. 변경된 엔티티가 있으면 수정 쿼리를 생성해서 쓰기 지연 SQL 저장소에 보냅니다.
4. 쓰기 지연 저장소의 SQL을 DB에 보냅니다.
5. 데이터베이스 트랜잭션을 커밋합니다.