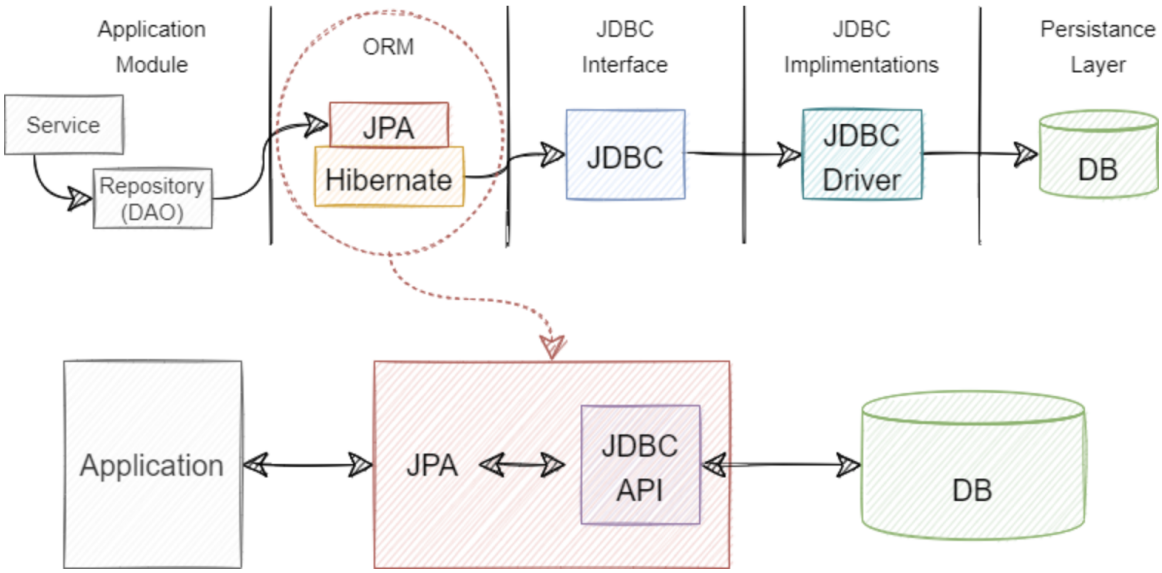
[JPA] 영속성(Persistence)

JPA에 대해서 간단히 알아보자!

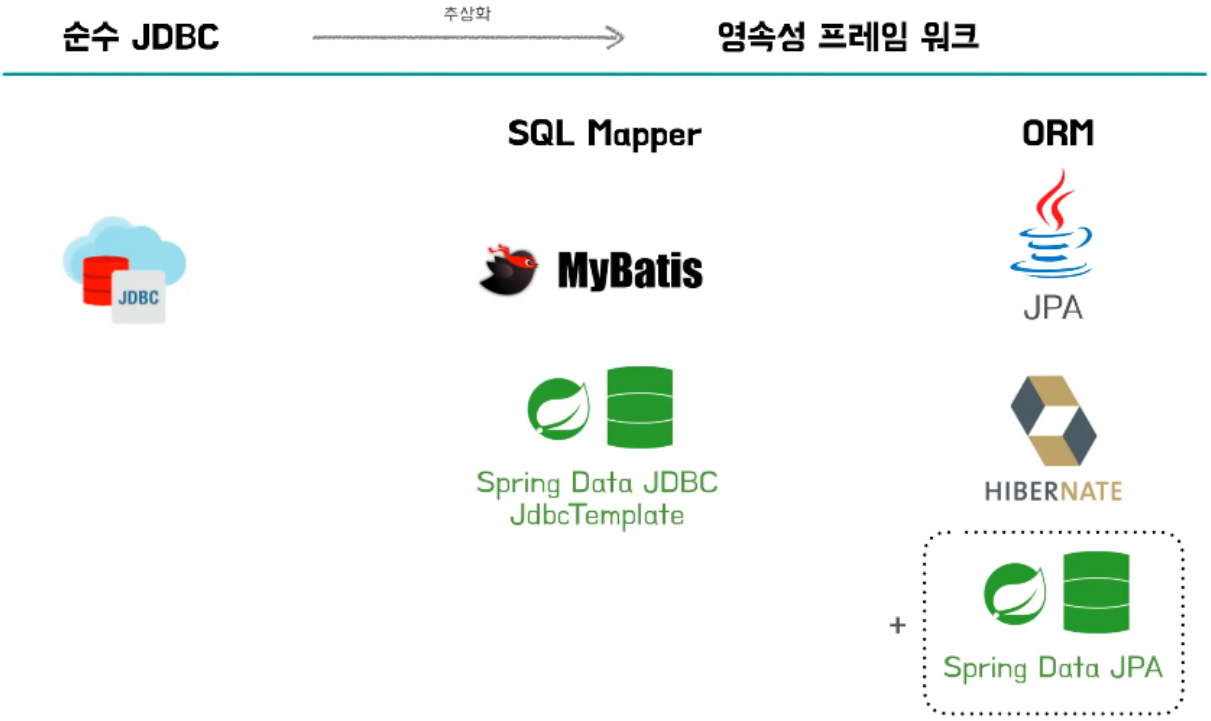
JPA란?



https://huimang2.github.io/java/jpa

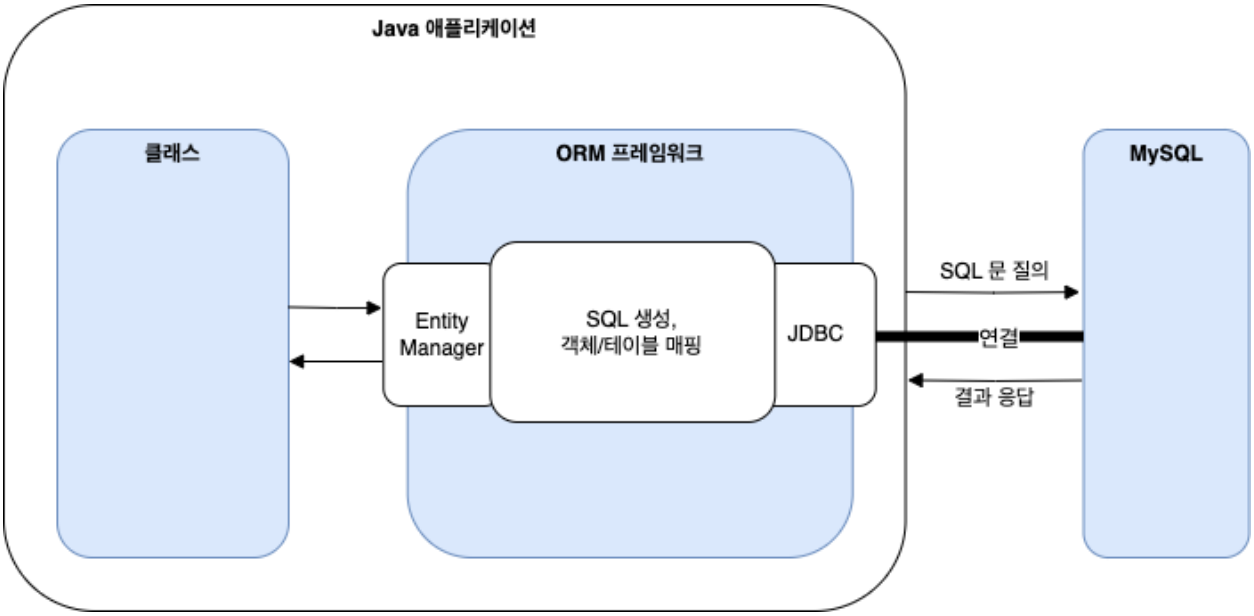
- Java Persistence API
- 자바 ORM(Object-Relational Mapping) 기술에 대한 API 표준 명세
- 객체지향 프로그래밍과 RDB 간의 매핑을 위한 API

ORM vs SQL Mapper vs JDBC



https://ysiksik.github.io/elegant-tekotok/2023-11-29-DOI-JDBC-SQLMapper-ORM/

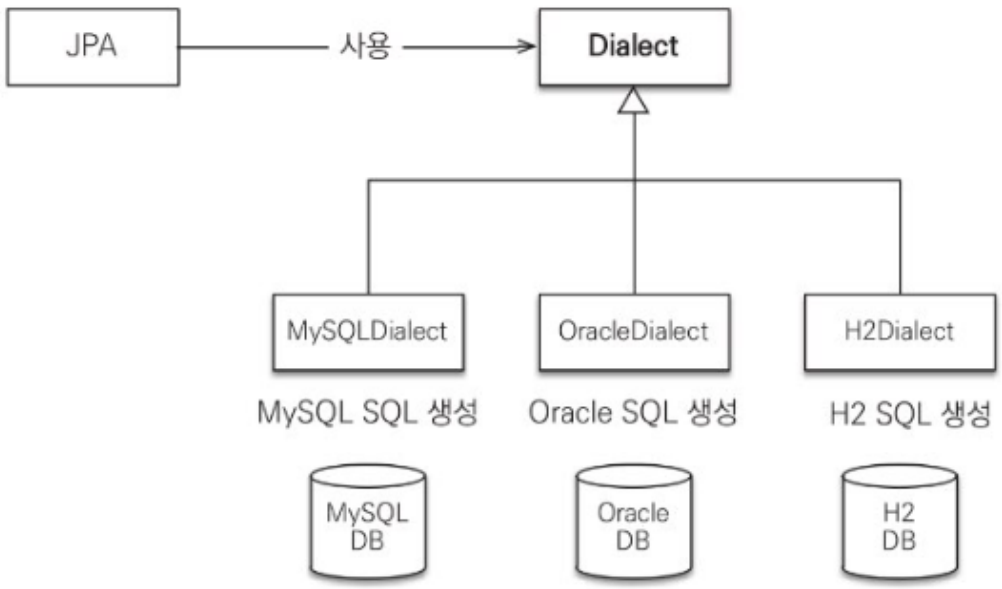
ORM



https://assu10.github.io/dev/2023/09/02/springboot-database-1/

- 객체, RDB 간의 매핑 자동화

- 대표: Hibernate, JPA
- 장점
  - 생산성 / 유지보수
    - 객체지향적으로 프로그래밍 가능
    - CRUD 간소화
      - 개발자가 직접 중복적인 CRUD SQL Query 작성할 필요 없음
    - 테이블의 구조를 변경했을 때 복잡도가 낮아서 수정이 빠름
    - 테스트 작성 편리
  - 데이터베이스 독립



출처 : 자바 ORM 표준 JPA 프로그래밍 (김영한)

- 단점
  - 복잡한 SQL문 처리 어려움
  - 학습 곡선이 높음
  - 잘못 사용하면 성능 문제 발생할 수 있음 (N+1문제)

▼ code

(** 여기서의 예시는 Spring Data JPA를 활용하여 작성)

UserEntity

```
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity
public class UserEntity {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private String email;
}
```

UserController

```
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

@RestController
@RequestMapping("/users")
public class UserController {

    @Autowired
    private UserService userService;

    @PostMapping
    public ResponseEntity<UserEntity> createUser(@RequestBody
        UserEntity savedUser = userService.saveUser(user);
```

```
        return new ResponseEntity<>(savedUser, HttpStatus.CRE
    }

    @GetMapping("/{id}")
    public ResponseEntity<UserEntity> getUser(@PathVariable L
        UserEntity user = userService.getUserById(id);
        if (user != null) {
            return new ResponseEntity<>(user, HttpStatus.OK);
        } else {
            return new ResponseEntity<>(HttpStatus.NOT_FOUND);
        }
    }

    @DeleteMapping("/{id}")
    public ResponseEntity<Void> deleteUser(@PathVariable Long
        userService.deleteUser(id);
        return new ResponseEntity<>(HttpStatus.NO_CONTENT);
    }
}
```

UserService

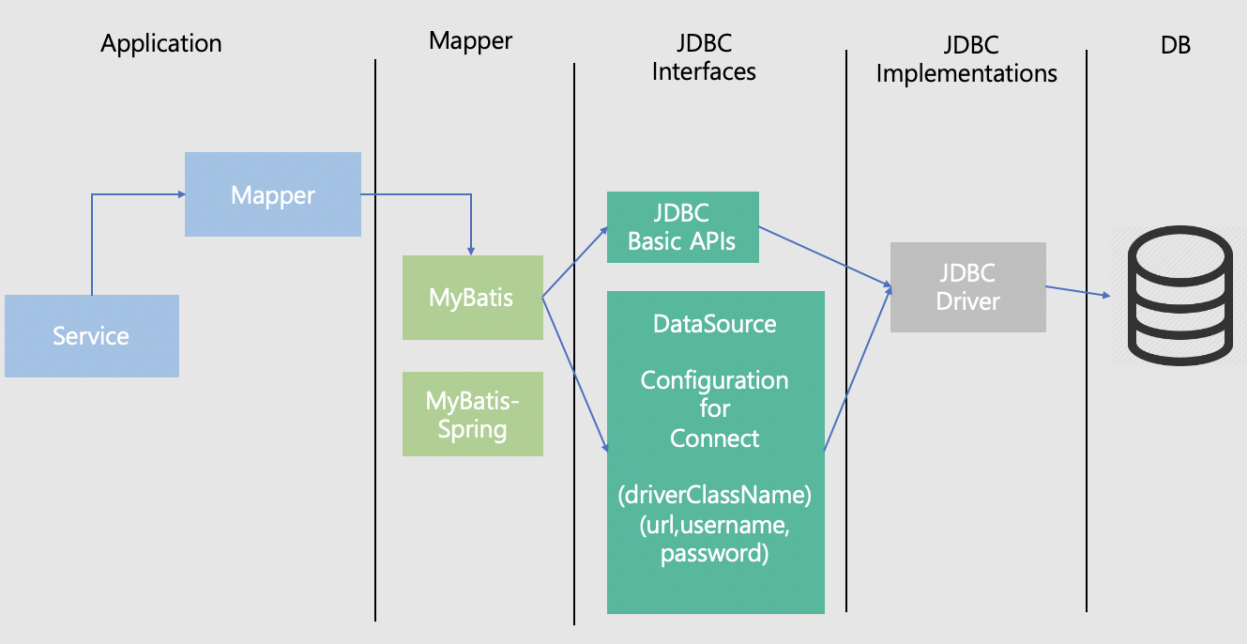
```
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.stereotype.Service;

@Service
public class UserService {

    @Autowired
    private UserRepository userRepository;

    public UserEntity saveUser(UserEntity user) {
        return userRepository.save(user);
    }
```

```
    public UserEntity getUserById(Long id) {
        return userRepository.findById(id).orElse(null);
    }

    public void deleteUser(Long id) {
        userRepository.deleteById(id);
    }
}
```

UserRepository

```
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface UserRepository extends JpaRepository<UserEnt
}
```

SQL Mapper



https://velog.io/@swjy1216/MyBatis

- SQL Query, 자바 객체 간의 매핑
- 대표: MyBatis
- 장점
  - SQL 코드 분리 ⇒ 유지보수가 편리
    (SQL 매핑 파일에 SQL Query 정의)
  - 동적 쿼리 생성에 유용

    ▼ code

    UserMapper.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.(
        "http://mybatis.org/dtd/mybatis-3-mapper.dtd">

<mapper namespace="com.example.mapper.UserMapper">

    <!-- 동적 쿼리 예시: 조건에 따라 다른 쿼리를 실행 -->
    <select id="getUserList" resultType="com.example.Us
        SELECT * FROM users
```

```
        <where>
            <if test="name != null">
                AND name = #{name}
            </if>
            <if test="email != null">
                AND email = #{email}
            </if>
        </where>
    </select>

</mapper>
```

- 단점
  - 개발자가 SQL 구문 직접 작성
  - 데이터베이스에 의존적

▼ code

UserRepository

```
import org.apache.ibatis.session.SqlSession;
import org.apache.ibatis.session.SqlSessionFactory;
import org.apache.ibatis.session.SqlSessionFactoryBuilder;

public class UserRepository {
    private SqlSessionFactory sqlSessionFactory;

    public UserRepository() {
        sqlSessionFactory = new SqlSessionFactoryBuilder().bu
    }

    public void saveUser(User user) {
        try (SqlSession sqlSession = sqlSessionFactory.openSe
            sqlSession.insert("saveUser", user);
            sqlSession.commit();
        }
```

```
    }

    public User getUserById(Long id) {
        try (SqlSession sqlSession = sqlSessionFactory.openSe
            return sqlSession.selectOne("getUserById", id);
        }
    }

    public void deleteUser(Long id) {
        try (SqlSession sqlSession = sqlSessionFactory.openSe
            sqlSession.delete("deleteUser", id);
            sqlSession.commit();
        }
    }
}
```

mybatis-config.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration PUBLIC "-//mybatis.org//DTD Config 3
        "http://mybatis.org/dtd/mybatis-3-config.dtd">

<configuration>
    <!-- 데이터베이스 연결 정보 설정 -->
    <environments default="development">
        <environment id="development">
            <transactionManager type="JDBC"/>
            <dataSource type="POOLED">
                <property name="driver" value="com.mysql.cj.j
                <property name="url" value="jdbc:mysql://loca
                <property name="username" value="username"/>
                <property name="password" value="password"/>
            </dataSource>
        </environment>
    </environments>
```

```
    <!-- SQL 매핑 파일의 위치 설정 -->
    <mappers>
        <mapper resource="com/example/mapper/UserMapper.xml"/
    </mappers>
</configuration>
```

UserMapper.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-mapper.dtd">

<!-- User 엔티티와 관련된 SQL 매핑을 정의하는 Mapper -->
<mapper namespace="com.example.mapper.UserMapper">

    <!-- saveUser -->
    <insert id="saveUser" parameterType="com.example.User">
        INSERT INTO users (name, email)
        VALUES (#{name}, #{email})
    </insert>

    <!-- getUserById -->
    <select id="getUserById" resultType="com.example.User" pa
        SELECT * FROM users WHERE id = #{id}
    </select>

    <!-- deleteUser -->
    <delete id="deleteUser" parameterType="java.lang.Long">
        DELETE FROM users WHERE id = #{id}
    </delete>

</mapper>
```

JDBC

- 자바 프로그램, DB간의 통신 API
- 개발자가 SQL 직접 작성
- 가장 저수준의 데이터베이스 액세스 기술
- 단점
  - SQL Query를 잘못 작성했을 때, 컴파일에서 확인 불가능 (런타임 시 가능)

▼ code

UserRepository

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

public class UserRepository {
    private static final String DB_URL = "jdbc:mysql://localh
    private static final String DB_USER = "username";
    private static final String DB_PASSWORD = "password";

        // SQL Query, JDBC를 활용하여 변환작업을 '직접' 해줘야함
    public void saveUser(User user) {
        try (Connection conn = DriverManager.getConnection(DB
            String sql = "INSERT INTO users (name, email) VAl
            try (PreparedStatement stmt = conn.prepareStateme
                stmt.setString(1, user.getName());
                stmt.setString(2, user.getEmail());
                stmt.executeUpdate();
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
```

```
    public User getUserById(Long id) {
        User user = null;
        try (Connection conn = DriverManager.getConnection(DB
            String sql = "SELECT * FROM users WHERE id = ?";
            try (PreparedStatement stmt = conn.prepareStateme
                stmt.setLong(1, id);
                try (ResultSet rs = stmt.executeQuery()) {
                    if (rs.next()) {
                        user = new User();
                        user.setId(rs.getLong("id"));
                        user.setName(rs.getString("name"));
                        user.setEmail(rs.getString("email"))
                    }
                }
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return user;
    }

    public void deleteUser(Long id) {
        try (Connection conn = DriverManager.getConnection(DB
            String sql = "DELETE FROM users WHERE id = ?";
            try (PreparedStatement stmt = conn.prepareStateme
                stmt.setLong(1, id);
                stmt.executeUpdate();
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

표준 JPA CRUD 예시 맛보기!

MemberEntity

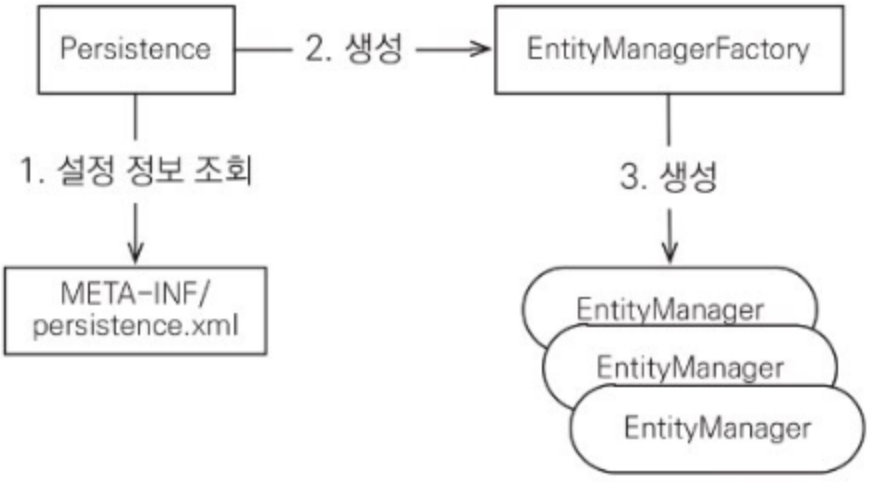
```
import javax.persistence.Entity;
import javax.persistence.Id;

@Entity
public class MemberEntity {

    @Id
    private Long id;
    private String name;
    private String email;
}
```

MemberService

```
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.transaction.Transactional;
import java.util.List;

@Transactional
public class MemberService {

    @PersistenceContext
    private EntityManager entityManager;

    public void saveMember(MemberEntity member) { // Create
        entityManager.persist(member);
    }

    public MemberEntity findMember(Long memberId) { // Read
```

```
        return entityManager.find(MemberEntity.class, memberId)
    }

    public void updateMemberName(Long memberId, String newName)
        MemberEntity member = entityManager.find(MemberEntity.cl
        if (member != null) {
            member.setName(newName);
        }
    }
    public void deleteMember(Long memberId) { // Delete
        MemberEntity member = entityManager.find(MemberEntity.cl
        if (member != null) {
            entityManager.remove(member);
        }
    }
}
```

JPA 기본 동작



출처 : 자바 ORM 표준 JPA 프로그래밍 (김영한)

1. 설정파일을 통한 JPA 설정

- `persistence.xml`
  - 데이터베이스 연결 정보, EntityManagerFactory 설정 등

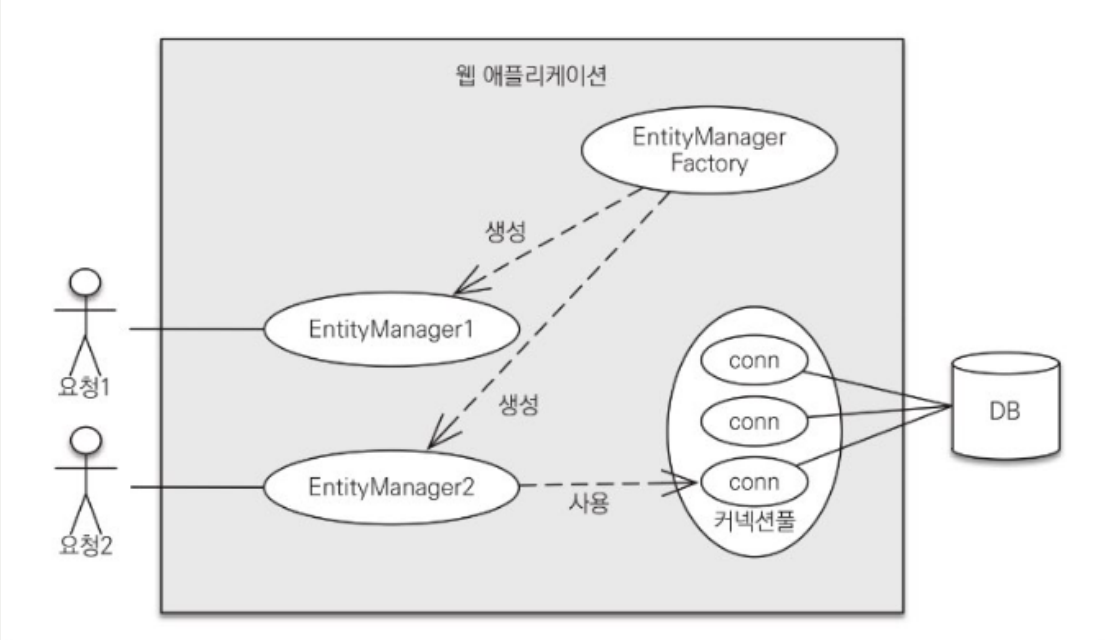
  ▼ code

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persist
             xmlns:xsi="http://www.w3.org/2001/XMLSche
             xsi:schemaLocation="http://xmlns.jcp.org/
             version="2.2">

    <persistence-unit name="myPersistenceUnit" transac
        <!-- EntityManagerFactory 설정 -->
        <provider>org.hibernate.jpa.HibernatePersisten

        <!-- 데이터베이스 연결 설정 -->
        <properties>
            <property name="javax.persistence.jdbc.dri
            <property name="javax.persistence.jdbc.url
            <property name="javax.persistence.jdbc.use
            <property name="javax.persistence.jdbc.pas

            <!-- Hibernate 설정 -->
            <property name="hibernate.dialect" value="
            <property name="hibernate.hbm2ddl.auto" va
            <property name="hibernate.show_sql" value=
        </properties>
    </persistence-unit>

</persistence>
```

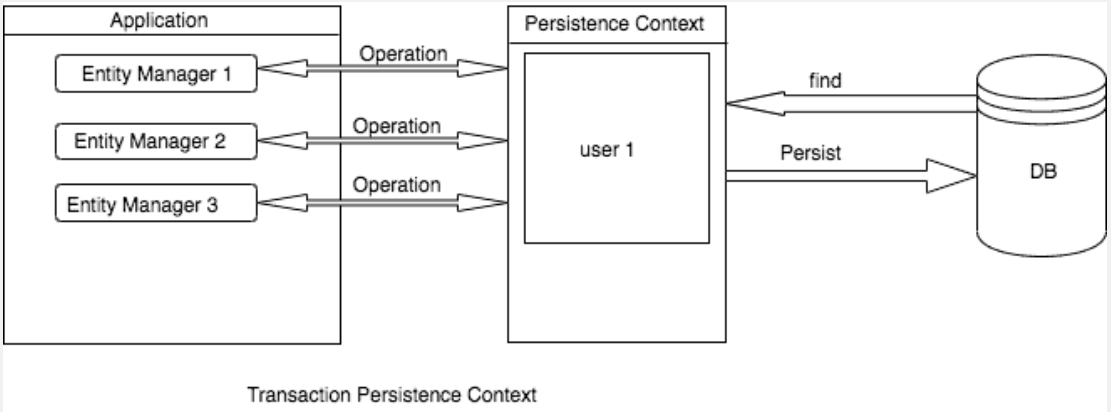
2. EntityManagerFactory 생성
3. EntityManager 생성
   - EntityManagerFactory가 EntityManager 생성

4. 영속성 컨텍스트 (Persistence Context) 생성
    - EntityManager가 영속성 컨텍스트 생성, 관리

💡 용어



출처 : 자바 ORM 표준 JPA 프로그래밍 (김영한)



https://www.baeldung.com/jpa-hibernate-persistence-context

EntityManagerFactory

- 일반적으로 애플리케이션 시작 때 한 번만 생성
  (이때, 커넥션 풀도 같이 만듦)
- 애플리케이션 전체에서 공유 (싱글톤)
- EntityManager 생성

- 여러 쓰레드 동시 접근 가능

EntityManager

- Entity 관리 (CRUD 등)
- Entity를 저장하는 가상의 데이터베이스
- 여러 쓰레드 동시 접근 하면 안됨
  → 동시성 문제 때문에
  ⇒ 한 request 당, 한 EntityManager
- DB 연결이 필요할 때 Connection 얻음
- 영속성 컨텍스트 생성, 관리

영속성 컨텍스트 (Persistence Context)

- Entity의 생명주기 관리
- 데이터베이스와 동기화되어 Entity 변경 자동 반영

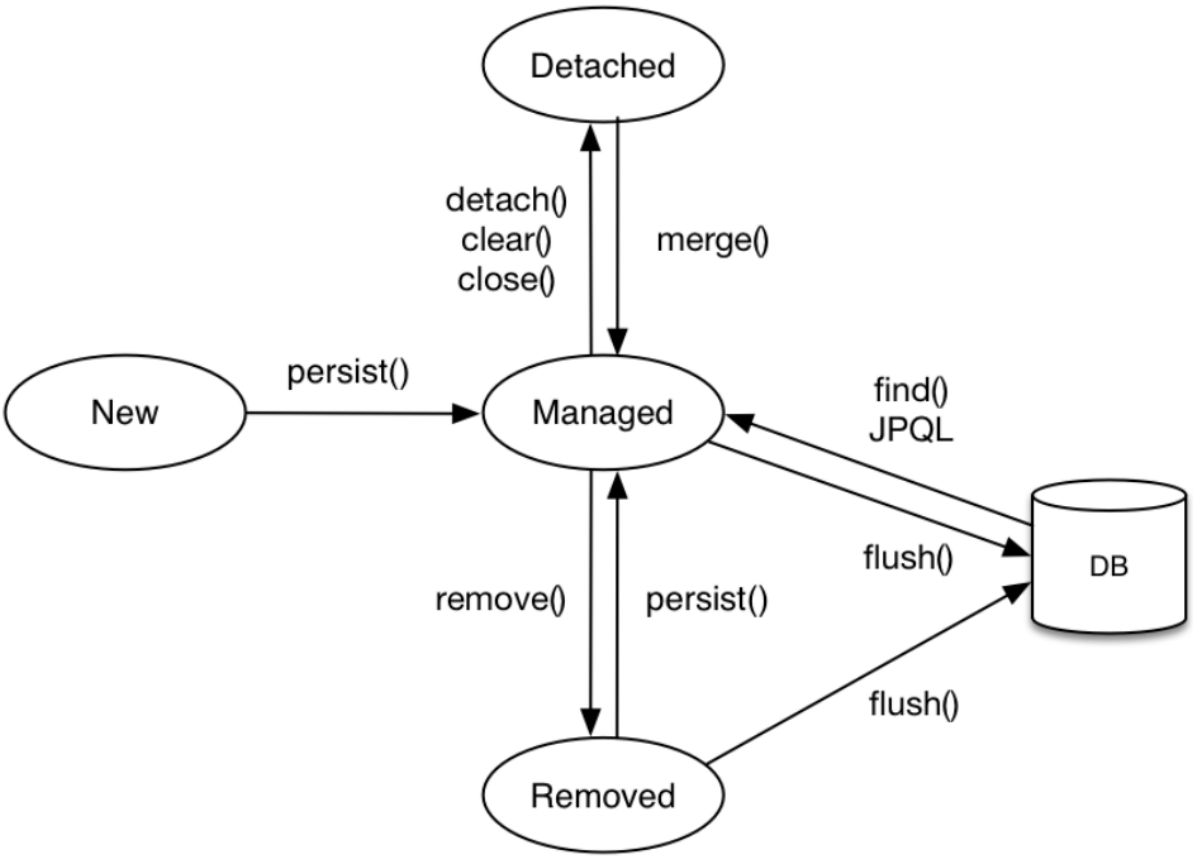
영속성 컨텍스트(Persistence Context) 란?

💡 **영속성(Persistence)의 뜻?**

- 일반적으로:
  데이터를 영구적으로 저장하고 유지하는 능력
- JPA에서:
  객체의 상태 관리

- Entity의 상태를 관리하는 환경

Entity 생명주기



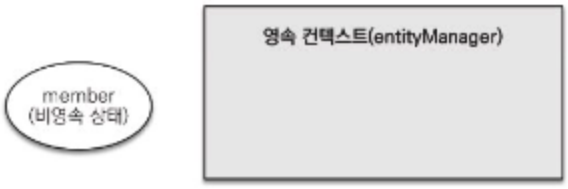
출처 : 자바 ORM 표준 JPA 프로그래밍 (김영한)

Entity의 상태

비영속 (new/transient)

- 영속성 컨텍스트와 관계 없음. 순수 객체 상태
  (영속성 컨텍스트에 존재하지 않음)

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



출처 : 자바 ORM 표준 JPA 프로그래밍 (김영한)

영속 (managed)

- 영속성 컨텍스트에 저장되어 있음

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

출처 : 자바 ORM 표준 JPA 프로그래밍 (김영한)

준영속 (detached)

- 영속성 컨텍스트에 저장되어 있다가 분리됨

삭제 (removed)

- 삭제됨

영속 컨텍스트의 특징

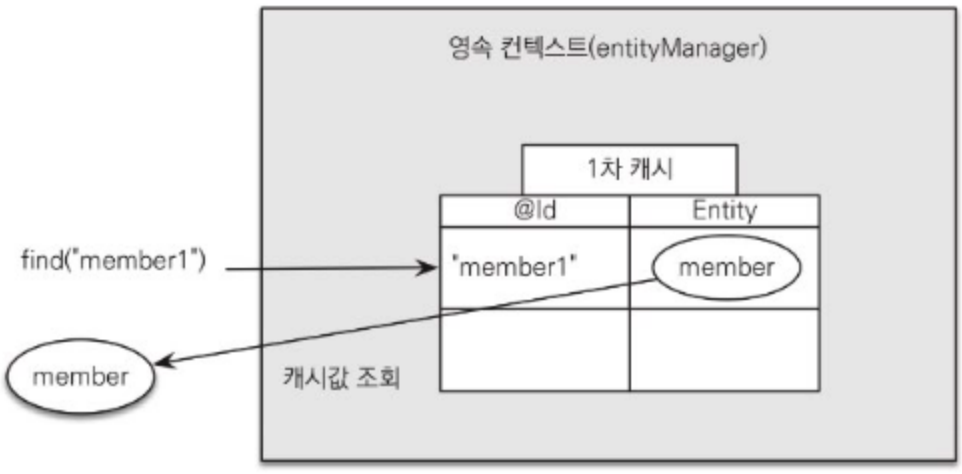
- Entity를 식별자 값(`@Id`)로 구분
- 영속된 Entity가 데이터베이스에 동기화(저장, 반영)되는 시점: `flush` (트렌젝션 commit하는 순간)
- 장점

- 1차 캐시
- 영속 객체 자동 변경 감지
- 객체 동일성 보장
- 트랜잭션을 지원하는 쓰기 지연
- 지연 로딩
- 일반적으로 쓰래드 단위

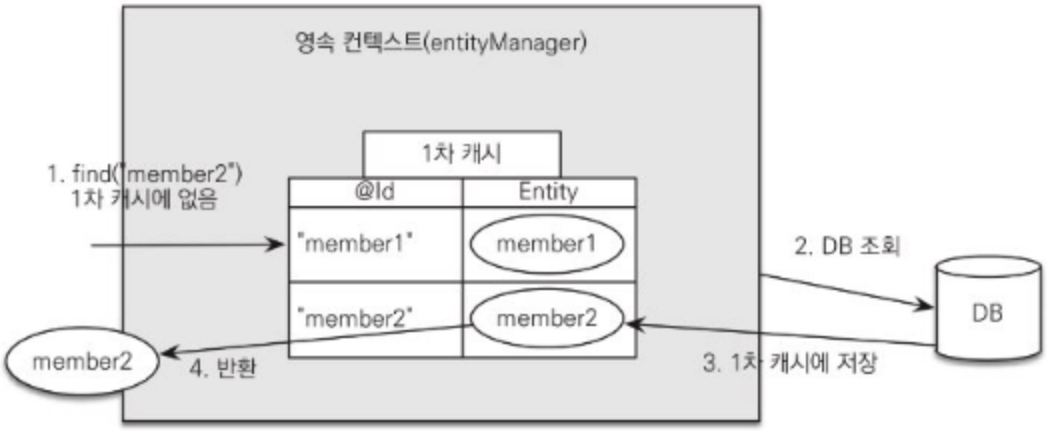
1차 캐시

- 영속성 컨텍스트 내부에 존재하는 캐시 (Map 형태)
- 영속 상태의 Entity는 모두 이곳에 저장
- 식별자 값으로 객체(A,B,C) 조회 시, 영속성 컨텍스트에
  - 있는 경우: 영속성 컨텍스트에서 찾아서 반환 (A)
  - 없는 경우: DB 쿼리를 통해, 영속성 컨텍스트에 영속화한 후 반환 (B, C)
- 애플리케이션 단계에서 Repeatable Read 등급의 격리 수준 제공 가능

▼ 예시



출처 : 자바 ORM 표준 JPA 프로그래밍 (김영한)



출처 : 자바 ORM 표준 JPA 프로그래밍 (김영한)

```
EntityManager em = emf.createEntityManager();

// member1 생성, 영속화
Member member = new Member();
member.setId("member1");
member.setName("m1");
member.setEmail("m1@member.com");
em.persist(member);

// member1 find
Member member1 = em.find(Member.class, "member1");

// member2 find
Member member2 = em.find(Member.class, "member2");
```

객체 동일성 보장

- 식별자 값을 통해 같은 객체임을 인식, 관리

▼ 예시

```
// member 생성, 영속화
Member member = new Member();
member.setId("member1");
member.setName("m1");
member.setEmail("m1@member.com");
em.persist(member);

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

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

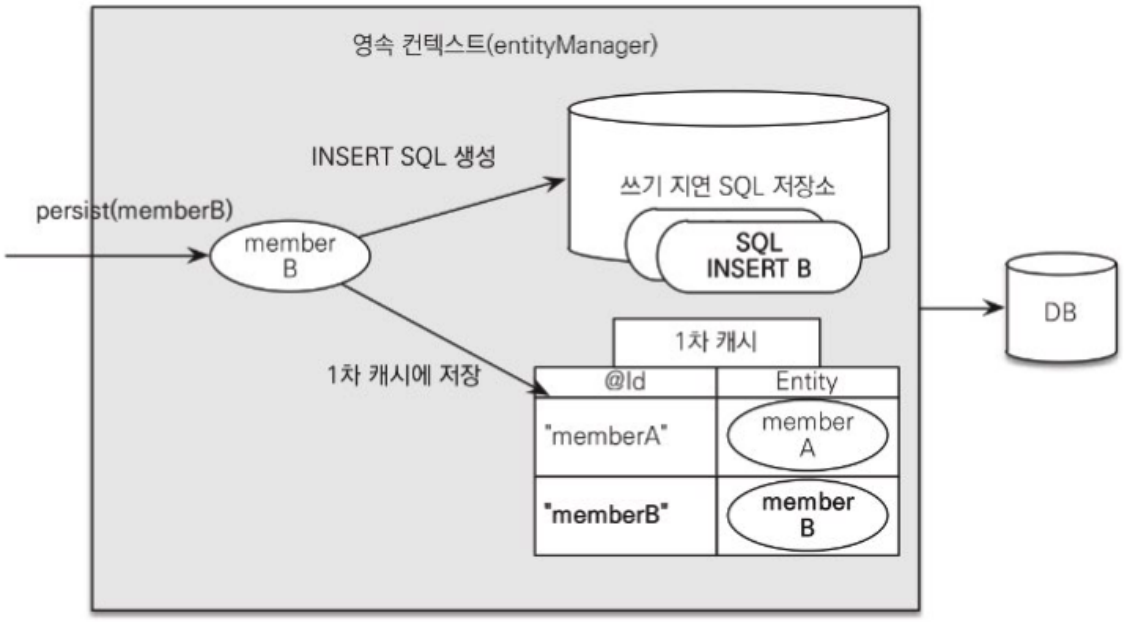
💡 **동일성 vs 동등성**

- 동일성(identity): 실제 인스턴스가 같음 (`==`)
- 동등성(equality): 실제 인스턴스는 다를 수 있으나, 인스턴스의 값이 같음 (`equal()`)

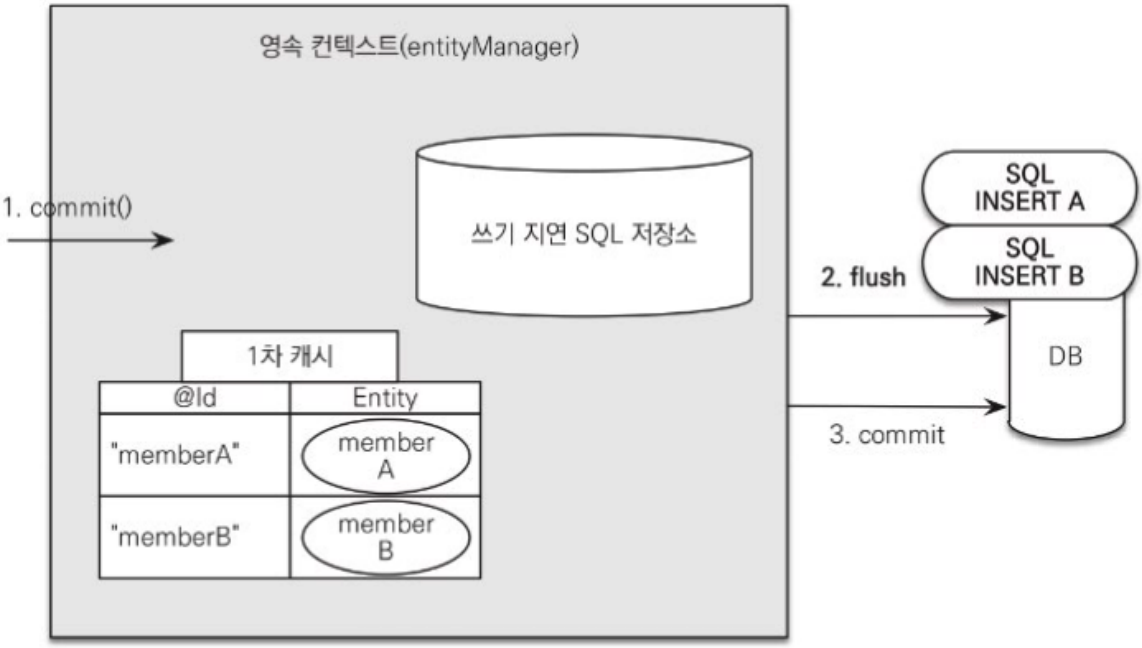
트랜잭션을 지원하는 쓰기 지연 (transactional write-behind)

- DB요청 최소화, 성능 향상
- 트랜잭션 범위 내에서 Entity의 변경을 모아,
  트랜젝션이 commit되기 전에(
  `flush`) 한꺼번에 DB에 반영

▼ 예시



출처 : 자바 ORM 표준 JPA 프로그래밍 (김영한)



출처 : 자바 ORM 표준 JPA 프로그래밍 (김영한)

```
// EntityManager 생성
EntityManager em = emf.createEntityManager();
```

```
// 트랜잭션 시작 (EntityManager는 데이터 변경 시 트랜잭션 시작해야함)
EntityTransaction tx = em.getTransaction();
tx.begin();

try {
    //  멤버 생성 및 영속화
    Member memberA = new Member();
        memberA.setId("memberA");
    memberA.setName("mA");
    memberA.setEmail("mA@member.com");
    em.persist(memberA);

    Member memberB = new Member();
        memberB.setId("memberB");
    memberB.setName("mB");
    memberB.setEmail("mB@member.com");
    em.persist(memberB);

    // 트랜잭션 commit => 쓰기 지연 발생
    tx.commit();
} catch (Exception e) {
    tx.rollback();
    e.printStackTrace();
} finally {
    em.close();
}
```

💡 **등록/업데이트/삭제 동일하게 작동**

- 등록/업데이트/삭제 쿼리를 쓰기 지연 SQL 저장소에 모아놨다가,
  트랜잭션 commit 직전에(
  `flush`) 한꺼번에 DB에 쿼리 전달
- 단, 영속된 Entity자체는
  `em.persist(memberA)` / `memberA.setName("test")` / `em.remove(memeberA)` 호출 시,
  바로 영속성 컨텍스트에서 생성/업데이트/제거

자동 변경 감지 (dirty checking)

- `flush` 때 스냅샷, Entity 비교해서 변경된 Entity 찾아 자동 업데이트
  (스냅샷: Entity를 영속성 컨텍스트에 보관할 때, 최초 상태 복사, 저장)
- 변경 감지는 영속 상태의 Entity만 적용
- 필드 업데이트
  - 기본적으로는, 모든 필드 업데이트
    - 장점: 수정 쿼리가 항상 동일 (재사용)
  - 설정 시 동적 업데이트 가능
    - 하이버네이트 확장 기능
    - 변경된 필드만 업데이트
    - 컬럼이 대략 30개 이상인 경우

    ▼ code

```
import javax.persistence.*;

@Entity
@Table(name = "users")
@DynamicUpdate // 동적 업데이트 설정 활성화
public class User {

    @Id
```

```
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

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

    @Column(name = "email")
    private String email;
}
```

💡 **동적 삽입/업데이트**

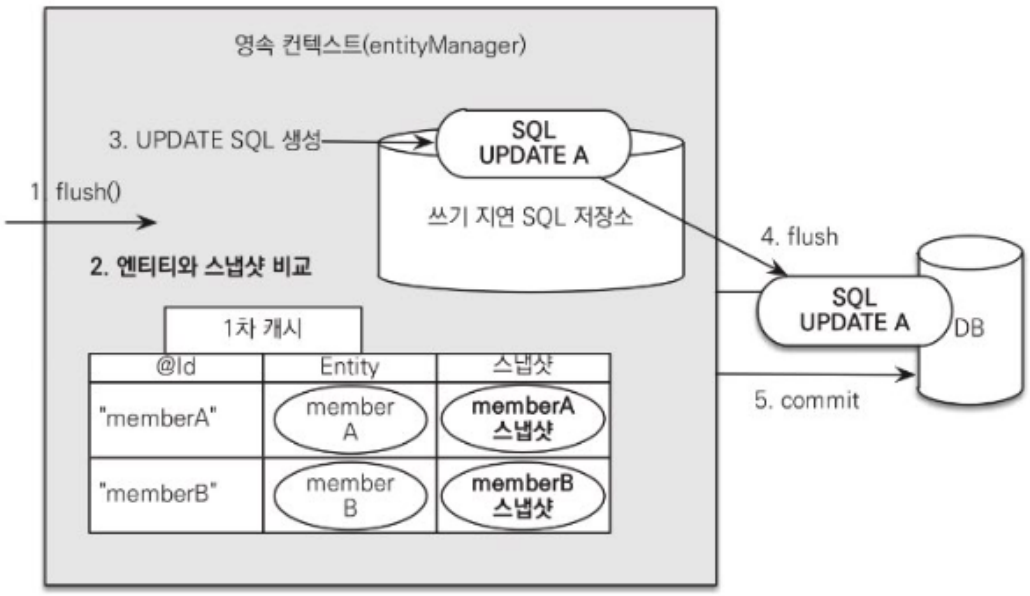
동적 삽입

- 데이터가 존재하는 필드만 생성
- `@DynamicInsert`

동적 업데이트

- 변경된 필드만 업데이트
- `@DynamicUpdate`

▼ 예시




```
// EntityManager 생성
EntityManager em = emf.createEntityManager();

// 트랜잭션 시작 (EntityManager는 데이터 변경 시 트랜잭션 시작해야함)
EntityTransaction tx = em.getTransaction();
tx.begin();

try {
    // 멤버 조회
        Member memberA = em.find(Member.class, "memberA");

    // 멤버 수정
    memberA.setName("mAAA");
    memberA.setEmail("mAAA@member.com");

        // em.update(member) -> 이런 메소드는 존재하지 않는다!!

    // 트랜잭션 commit
    tx.commit();
} catch (Exception e) {
```

```
    tx.rollback();
    e.printStackTrace();
} finally {
    em.close();
}
```

지연 로딩

- 연관된 Entity 실제로 필요할 때까지 로딩 지연

▼ code

```
@Entity
public class Parent {
    @Id
    private Long id;

    @OneToMany(mappedBy = "parent", fetch = FetchType.LAZY) /
    private List<Child> children;
}

@Entity
public class Child {
    @Id
    private Long id;

    @ManyToOne
    @JoinColumn(name = "parent_id")
    private Parent parent;
}
```

```
// Parent 엔티티 조회 (연관된 Child 엔티티는 로딩되지 않음)
Parent parent = em.find(Parent.class, 1L);
```

```
// 연관된 Child 엔티티 접근 시점에 Lazy Loading을 통해 로딩됨
List<Child> children = parent.getChildren();
for (Child child : children) {
    System.out.println("Child name: " + child.getName());
}
```

일반적으로 쓰래드 단위

- 일반적으로는 한 쓰레드 당, 한 영속성 컨텍스트
  (영속성 컨텍스트 공유하는 경우도 있음)

플러시(flush)

- 영속성 컨텍스트의 변경 내용을 데이터베이스에 반영
- 동작
  1. 모든 Entity 스냅샷과 비교하여, 수정된 Entity 찾고, 수정 쿼리를 `쓰기 지연 SQL 저장소`에 저장
  2. `쓰기 지연 SQL 저장소`의 쿼리들(등록,업데이트,삭제)을 데이터베이스에 전송
- 호출 방법
  - 수동 호출: `em.flush()` 직접 호출
  - 자동 호출
    - 트랜잭션 commit 시
    - JPQL 쿼리 호출 시

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

// JPQL 실행 -> flush 발생
```

```
query = em.cresteQuery("select m from Member m", Member
List <Member> members = query.getResultList(); // membe
```

- 모드 옵션
  - `FlushModeType.AUTO` : (기본값) commit, 쿼리 실행 시 플러시
  - `FlushModeType.COMMIT` : 커밋할 때만 플러시 (성능 최적화를 위해)

준영속 상태

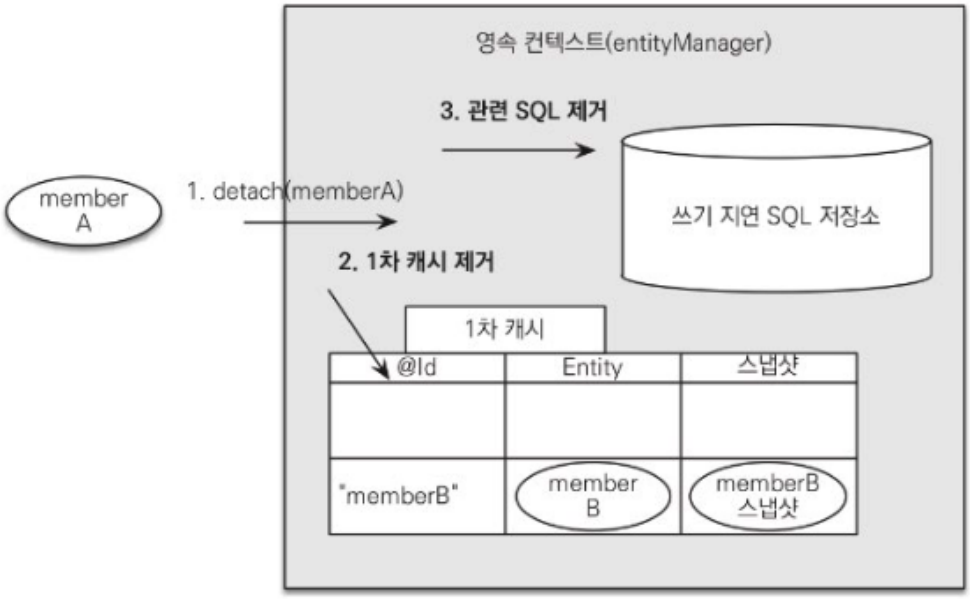
- 영속 상태의 Entity가 영속 컨텍스트로부터 분리된 것

특징

- 영속성 컨텍스트가 제공하는 기능 전부 사용 불가능
- 식별자 값을 가지고 있다 (영속 상태 였음으로)

준영속 상태로 만드는 방법

- `em.detach(entity)` : 특정 엔티티만 준영속 상태로 전환



출처 : 자바 ORM 표준 JPA 프로그래밍 (김영한)

- `em.clear()` : 영속성 컨텍스트 완전히 초기화

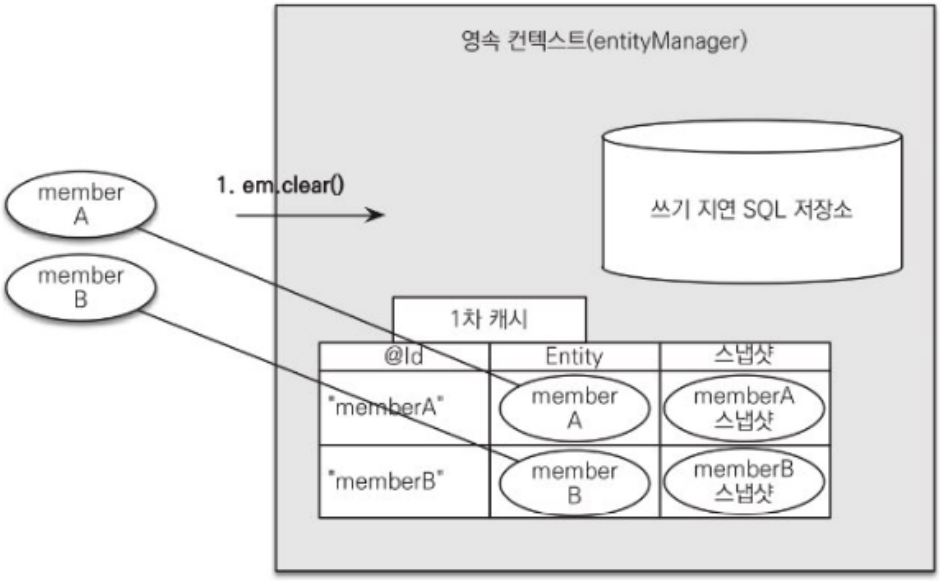


그림 3.14 영속성 컨텍스트 초기화 전

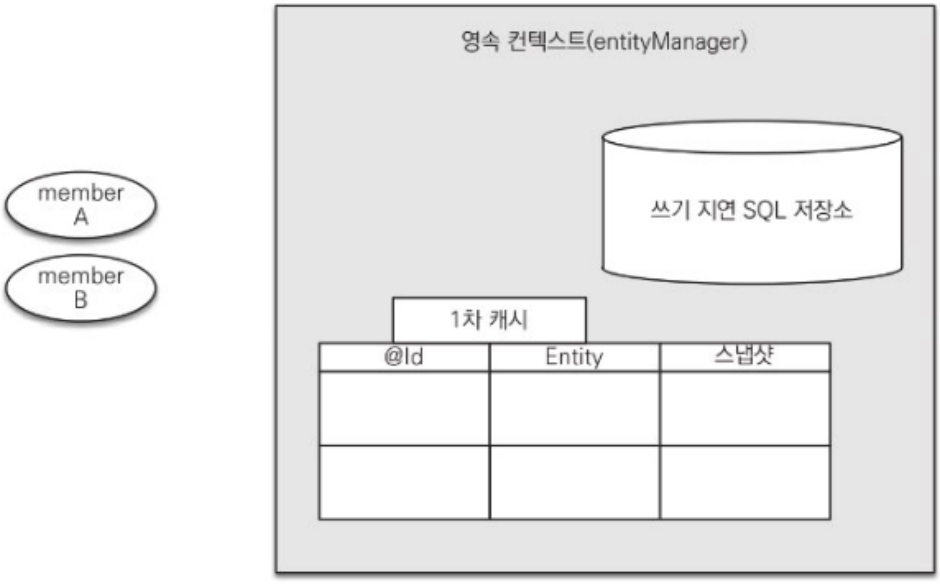
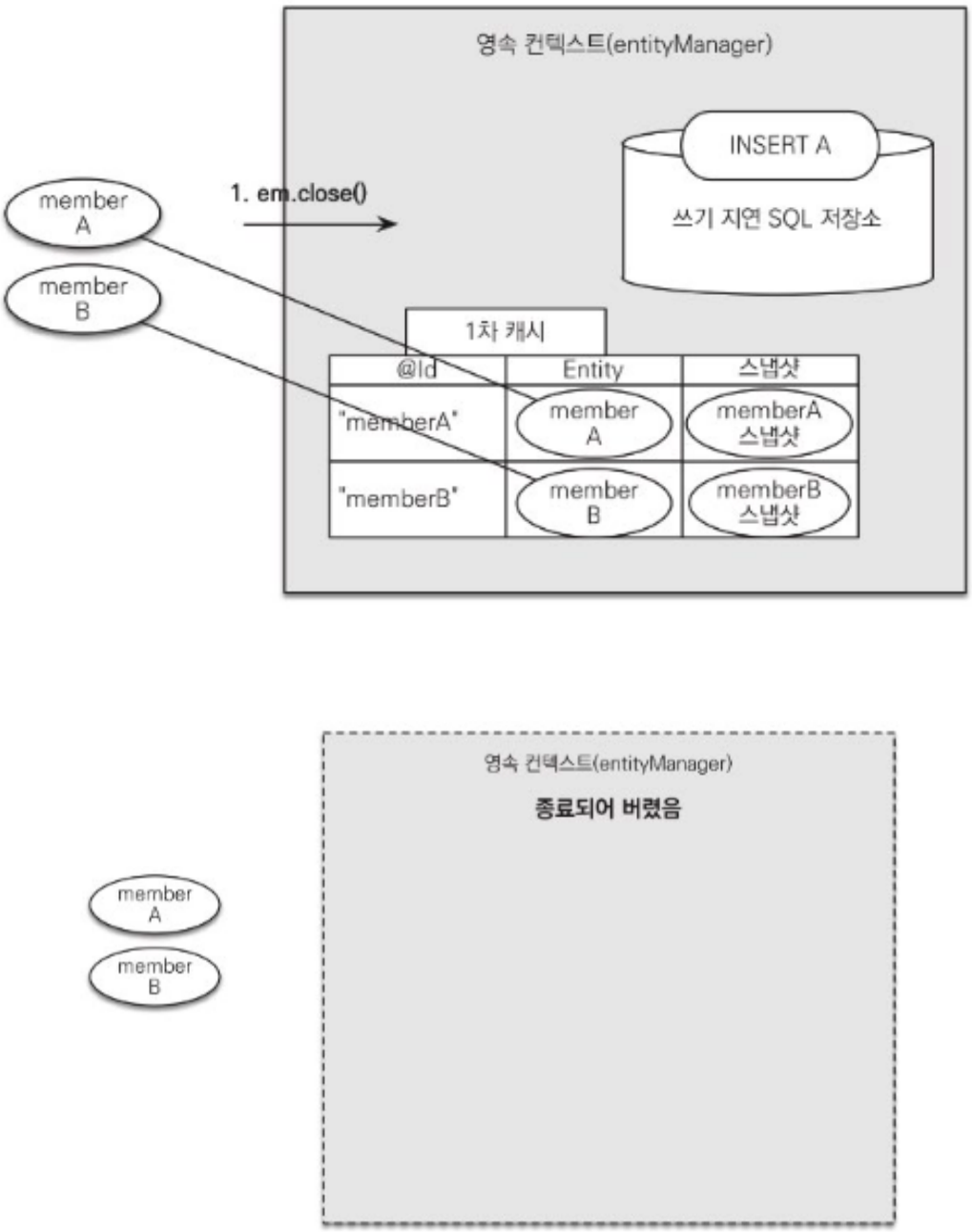


그림 3.15 영속성 컨텍스트 초기화 후

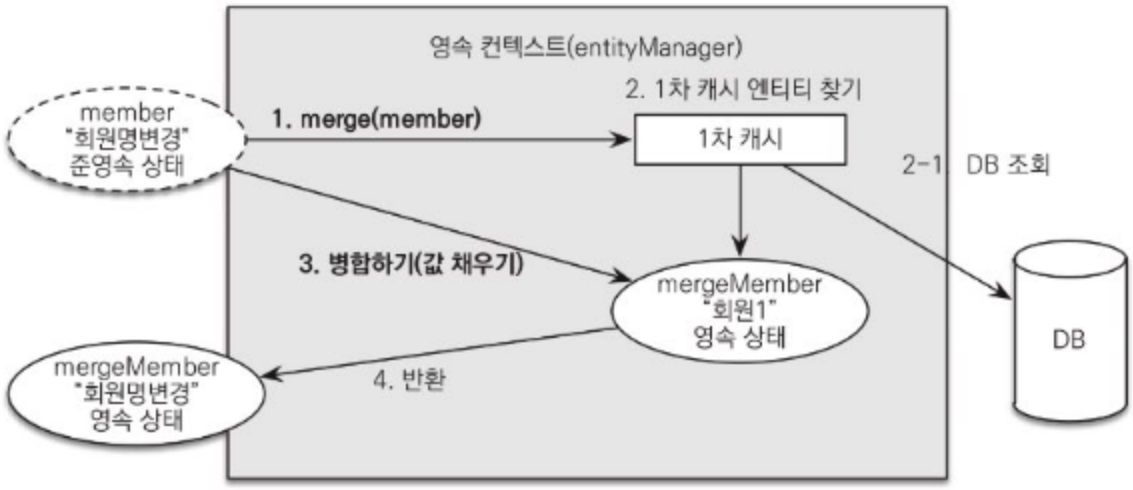
출처 : 자바 ORM 표준 JPA 프로그래밍 (김영한)

- `em.close()` : 영속성 컨텍스트 종료



출처 : 자바 ORM 표준 JPA 프로그래밍 (김영한)

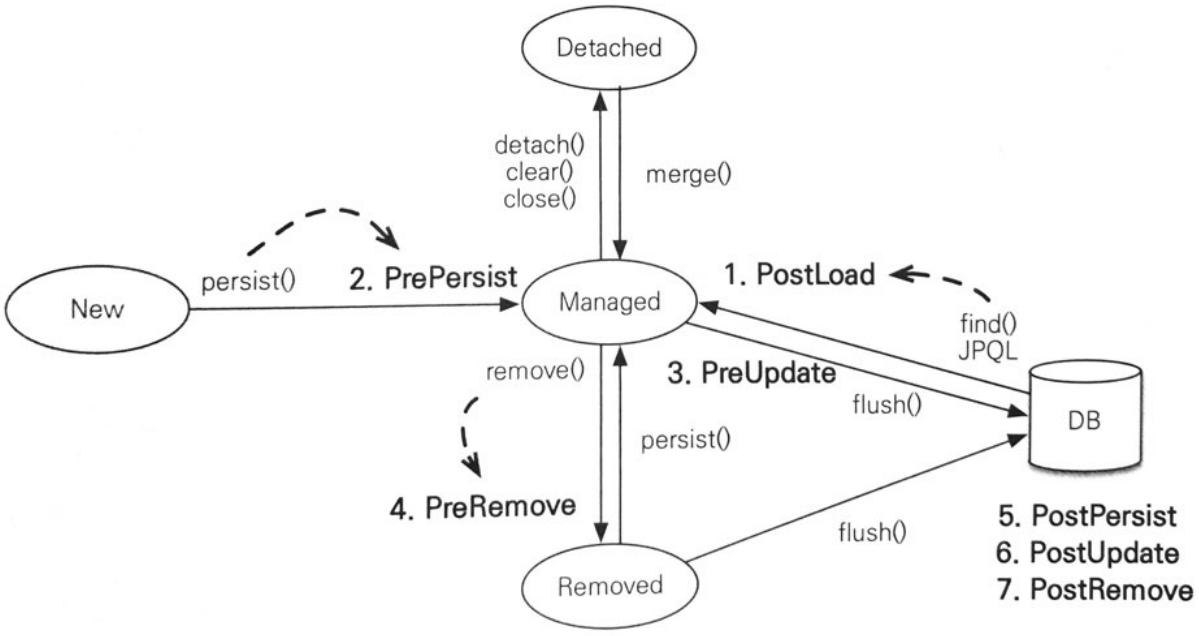
병합(merge)



출처 : 자바 ORM 표준 JPA 프로그래밍 (김영한)

- 준영속/비영속 상태의 엔티티를 영속 상태로 변경 (save / update)
- 동작
  1. 식별자로 기존 Entity 찾기
     - 1차 캐시에 존재하는가? → 1차 캐시에서 조회
     - DB에 존재하는가? → DB에서 조회 후 1차 캐시에 저장
     - 어느 곳에도 존재하지 않는가? → 1차 캐시에 새로 생성
  2. 현재 Entity 값으로 영속된 Entity 값 채우기
  3. 결과값 반환하기

JPA Entity 생명주기에 따른 이벤트



https://devbksheen.tistory.com/entry/JPA-리스너를-이용한-로그-수집

- `@PostLoad`
  - Entity가 영속성 컨텍스트에 로드된 후 (Entity를 찾은 후)
    - 1차 캐시에서 찾거나, 데이터베이스에서 찾아오거나
- `@PrePersist` / `@PreUpdate` / `@PreRemove`
  - Entity가 영속성 컨텍스트에 반영되기 전에 호출
- `@PostPersist` / `@PostUpdate` / `@PostRemove`
  - Entity가 데이터베이스에서 반영된 후 호출

Q&A

1. 서버가 여러대로 구성되어 있으면 영속성 컨텍스트에서 동시성 문제는 없는가?

- 기본적으로는 쓰레드 → EntityManager → 영속성 컨텍스트가 1:1:1이라서 동시성 문제가 없을 것같다
- 2차 캐시가 동시성을 해결해주는 줄 알았는데 2차 캐시는 성능에 상관되어 있는 듯하다

2. 1차 캐시를 비활성화 할 수 있는가?

- JPA의 핵심 기능, 영속성 컨텍스트의 핵심 요소이기 때문에, 일반적으로 비활성화하는 것은 JPA에서 지원하지 않는다

3. 1차 캐시에서 애플리케이션 단계에서 Repeatable Read 등급의 격리 수준 제공 가능하다는 건,
   트렌젝션 격리단계와 관련이 되는 것인가?

- 1차 캐시는 영속성 컨텍스트 수명과 관련 있고, 트랜잭션의 격리 수준과는 직접적인 관련이 없다
- 영속성 컨텍스트에서 여러 트랜잭션이 동시에 동일한 Entity에 접근할 때 데이터의 일관성을 유지해줄 수 있다로 이해할 수 있을 것같다