6장 JPA 개요

1. SQL 중심적인 개발

1) 무한반복, 지루한 코드

CRUD

INSERT INTO …

UPDATE …

SELECT …

DELETE …

자바 객체를 SQL로 …

SQL을 자바 객체로 …

SQL에 의존적인 개발을 피하기 어렵다.

2) 객체가 아닌 SQL 중심의 코드

```
public class Member {
  private String memberId;
  private String name;

  ...
}
```

```
INSERT INTO MEMBER(MEMBER_ID, NAME) VALUES
SELECT MEMBER_ID, NAME FROM MEMBER M
UPDATE MEMBER SET …
```

```
public class Member {
  private String memberId;
  private String name;
  private String tel;
  ...
}
```

```
INSERT INTO MEMBER(MEMBER_ID, NAME, TEL) VALUES
SELECT MEMBER_ID, NAME, TEL FROM MEMBER M
UPDATE MEMBER SET … TEL = ?
```

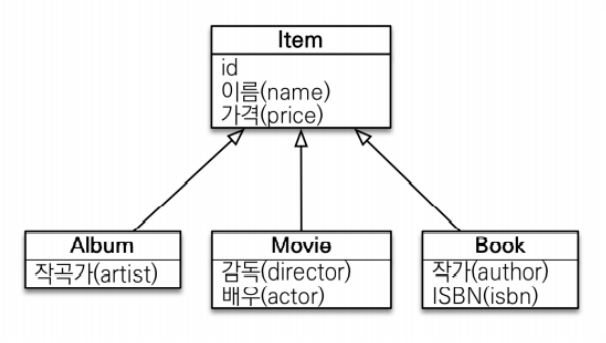
2. 객체와 RDB의 패러다임 불일치

'객체지향 프로그래밍은
추상화,캡슐화,정보은닉,상속,다형성
등 시스템의 복잡성을 제어할 수 있는
다양한 기능들을 제공한다.'

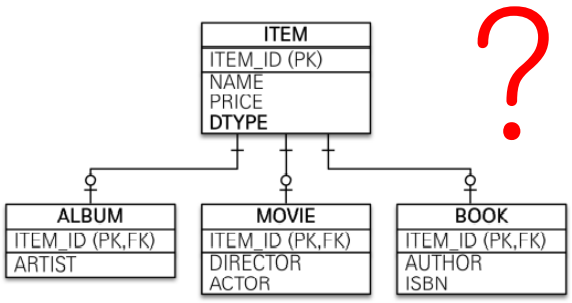
⇨

RDB 환경에서
적용하기는
거의 불가능하다.

예1> 상속



[객체 상속 관계]

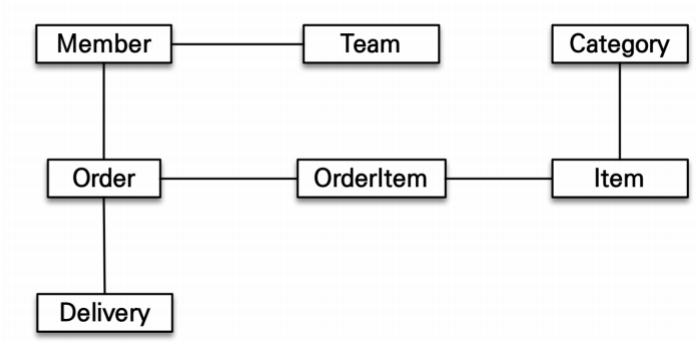


[Table 슈퍼타입 서브타입 관계]

예2> 탐색이 자유롭지 못하다.

객체 그래프 탐색

객체는 자유롭게 객체 그래프를 탐색할 수 있어야 한다.



처음 실행하는 SQL에 따라 탐색 범위 결정

```
SELECT M.*, T.*
  FROM MEMBER M
  JOIN TEAM T ON M.TEAM_ID = T.TEAM_ID

member.getTeam(); //OK
member.getOrder(); //null
```

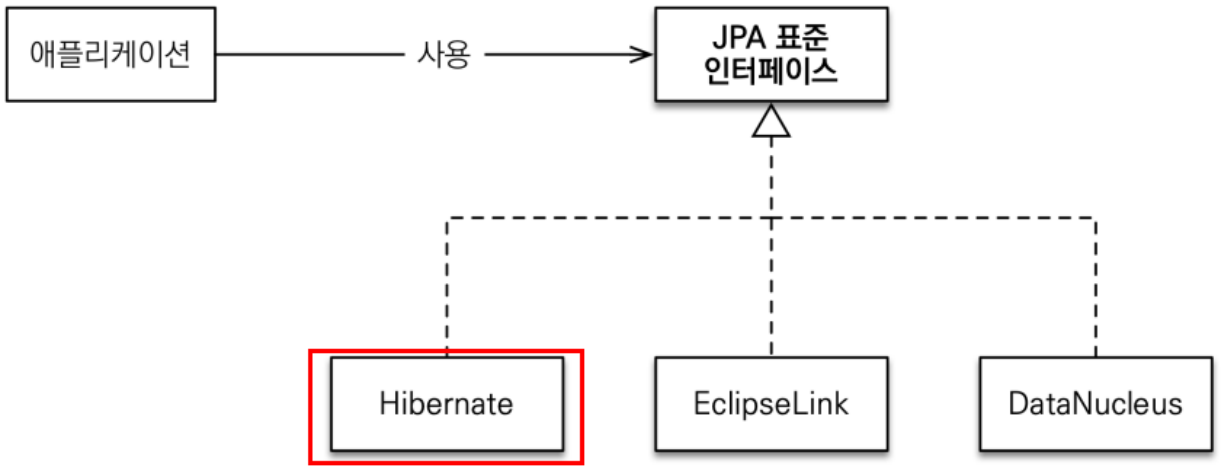
예3> …

JPA (Java Persistence API)

-자바 진영의 ORM 기술 표준 명세이다.

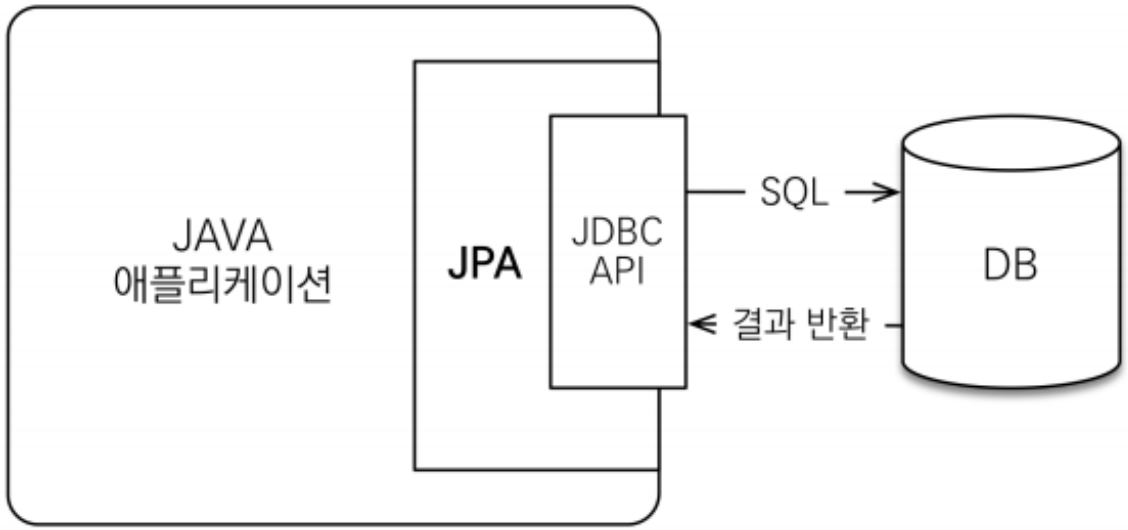
JPA는 표준 명세

-JPA는 인터페이스의 모음.
-JPA 2.1 표준 명세를 구현한 3가지 구현체

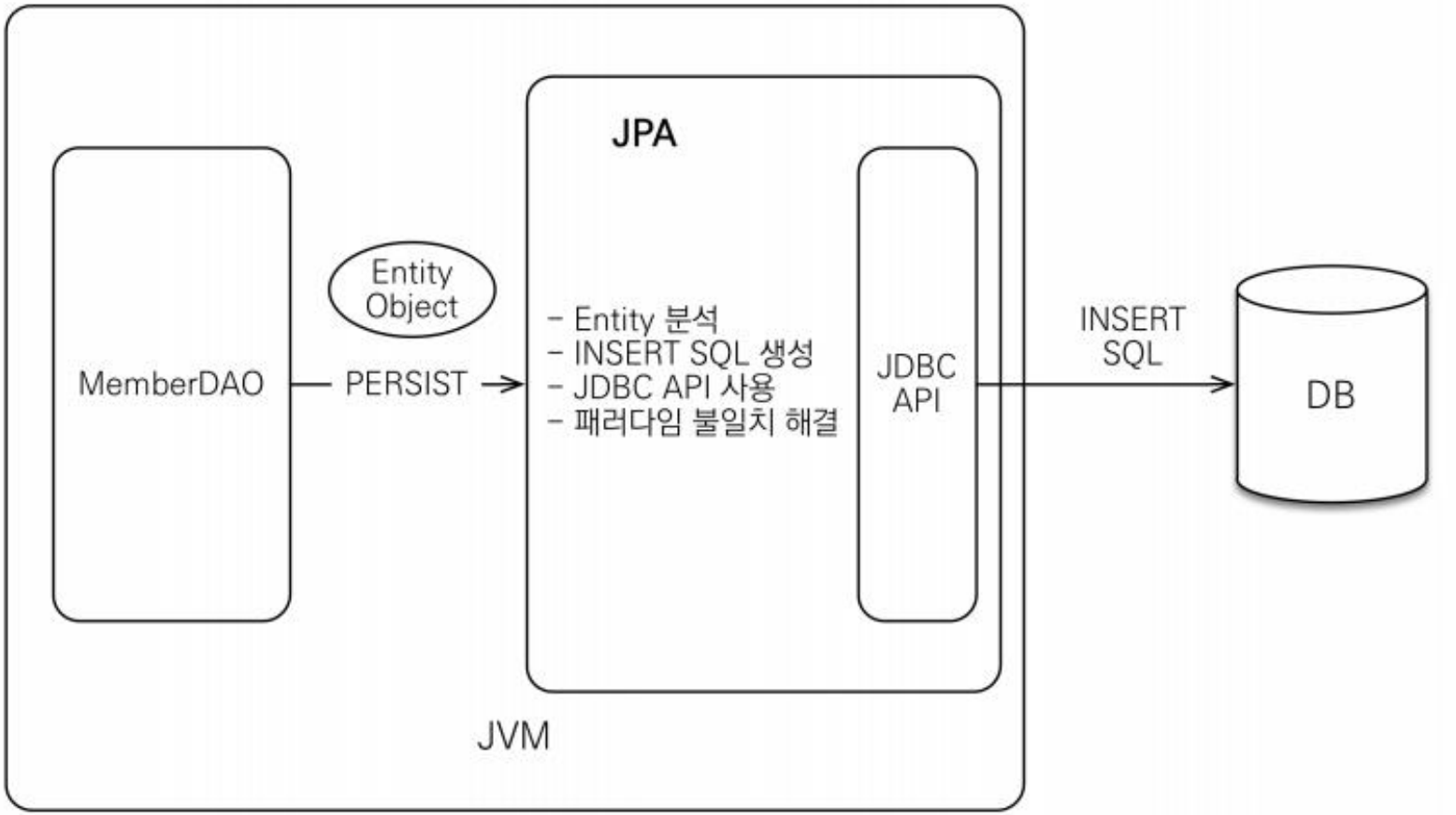


ORM?

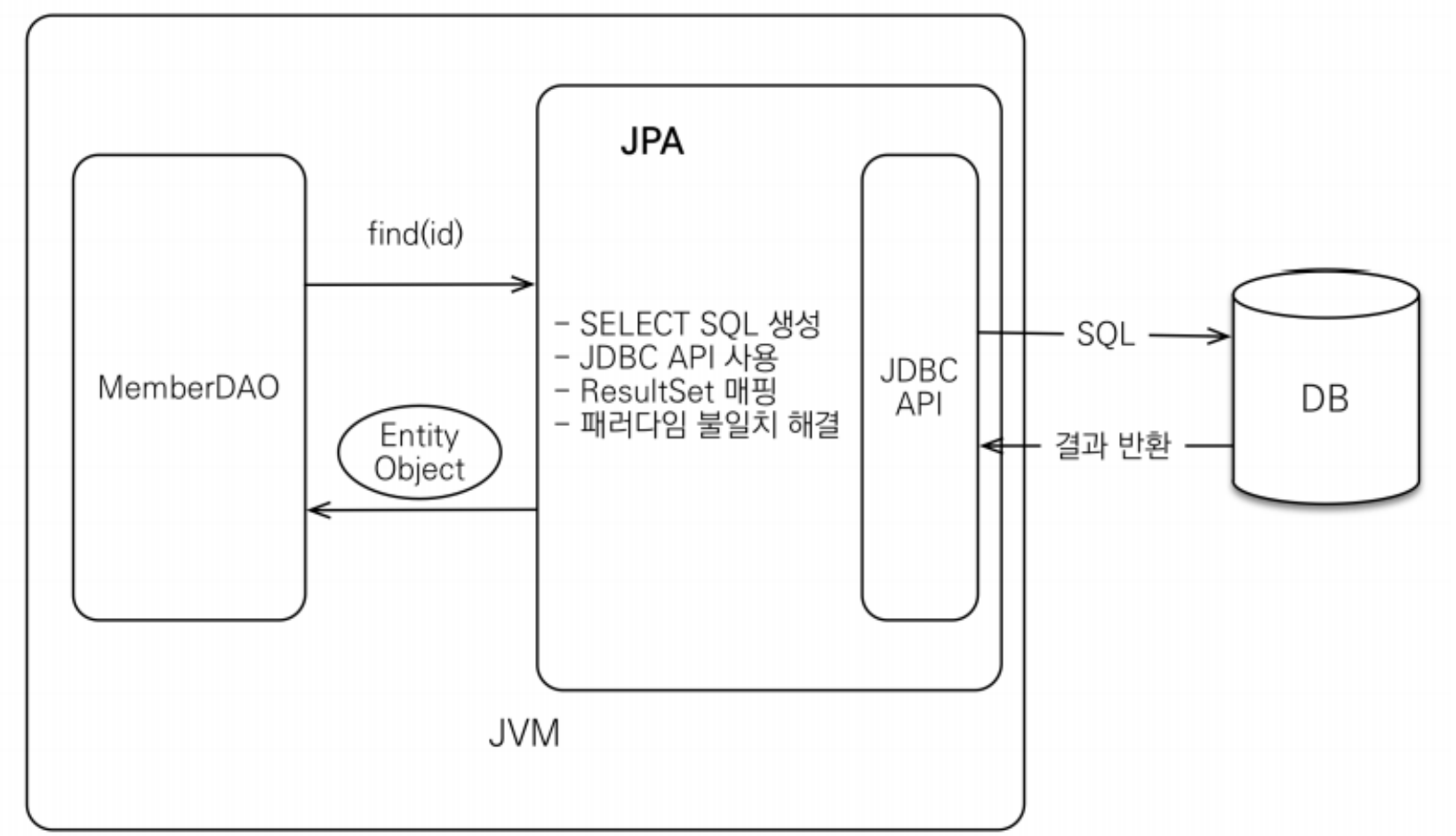
- Object-Relational Mapping (객체 관계 매핑)
- 객체는 객체대로 설계
- 관계형 데이터베이스는 관계형 데이터베이스대로 설계
- ORM 프레임워크가 중간에서 매핑



JPA 동작 - 저장


JPA
Entity
Object
MemberDAO
PERSIST
- Entity 분석
- INSERT SQL 생성
- JDBC API 사용
- 패러다임 불일치 해결
JDBC
API
INSERT
SQL
DB
JVM

JPA 동작 - 조회


JPA
find(id)
MemberDAO
- SELECT SQL 생성
- JDBC API 사용
- ResultSet 매핑
- 패러다임 불일치 해결
JDBC API
SQL
DB
Entity Object
결과 반환
JVM

6. JPA 장점

- SQL 중심적인 개발에서 객체 중심으로 개발
- 생산성
- 유지보수
- 패러다임의 불일치 해결
- 성능 (1차 캐시, 지연로딩 등)
- 데이터 접근 추상화와 벤더 독립성 등

생산성 (JPA와 CRUD)

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

성능 최적화 (1차 캐시 기능)

```
String memberId = "100";
Member m1 = jpa.find(Member.class, memberId); //SQL
Member m2 = jpa.find(Member.class, memberId); //캐시

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

SQL 1번만 실행

성능 최적화 (쓰기 지연 , batch기능)

```
transaction.begin();  // [트랜잭션] 시작

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

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

성능 최적화 (지연 로딩과 즉시 로딩)

지연 로딩 : 객체가 실제 사용될 때 로딩
즉시 로딩: Join SQL로 한번에 연관된 객체까지 미리 조회

지연 로딩

```
Member member = memberDAO.find(memberId);
Team team = member.getTeam();
String teamName = team.getName();
```

```
SELECT * FROM MEMBER

SELECT * FROM TEAM
```

즉시 로딩

```
Member member = memberDAO.find(memberId);
Team team = member.getTeam();
String teamName = team.getName();
```

```
SELECT M.*, T.*
FROM MEMBER
JOIN TEAM …
```

4. 데이터베이스 방언 (Dialect)

JPA는 특정 데이터베이스에 종속 X

각각의 데이터베이스가 제공하는 SQL문법과 함수는 조금씩 다름.

가) 가변문자
MySQL은 VARCHAR, Oracle은 VARCHAR2

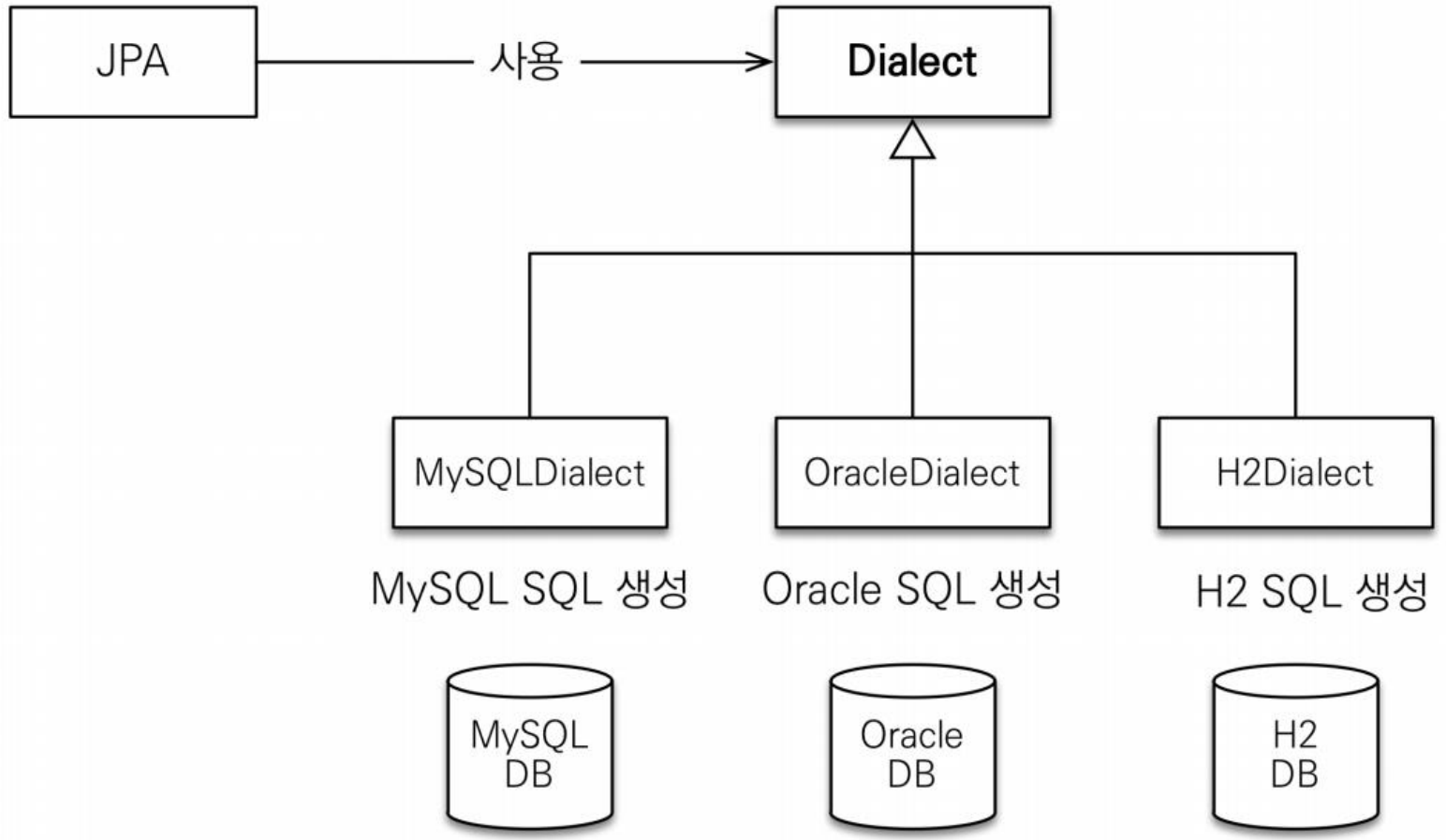
나) 함수
SQL 표준은 substring(), Oracle은 substr()

다) 페이징
MySQL 은 LIMIT, Oracle은 ROWNUM

방언(Dialect)은 SQL 표준을 지키지 않는 특정 데이터베이스만의 고유한 기능을 의미한다.

4. 데이터베이스 방언 (Dialect)

JPA에서는 특정 DB에 종속적인 방언(Dialect)을 지정하여 사용함.



4. 데이터베이스 방언 (Dialect)

hibernate.dialect 속성에 지정

- H2: org.hibernate.dialect.H2Dialect
- Oracle 10g: org.hibernate.dialect.Oracle10gDialect
- MySQL: org.hibernate.dialect.MySQL5InnoDBDialect
- PostgreSQL: `org.hibernate.dialect.PostgreSQL10Dialect`

하이버네이트는 40가지 이상의 데이터베이스 방언을 지원한다.

8장 객체와 테이블 매핑

1. 객체와 테이블 매핑 및 생성 예

@Entity : JPA가 관리할 객체

@Id : 데이터베이스 PK와 매핑

엔티티 객체

```java
package hellojpa;

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

@Entity
public class Member {

    @Id
    private Long id;
    private String name;

    //Getter, Setter …

}
```

테이블

```sql
create table Member (

    id bigint not null,
    name varchar(255),
    primary key (id)
);
```

2. @Entity (*)

@Entity 가 붙은 클래스는 JPA가 관리한다.

JPA를 사용해서 테이블과 매핑할 클래스는 반드시 @Entity가 필수이다.

@Entity에 name 속성을 사용할 수 있다.

JPA에서 사용할 엔티티 이름을 지정한다.
기본값은 클래스명이다. (권장)

주의사항

가. 기본 생성자 필수 (파라미터 없는 public|protected)
나. final 클래스, enum, interface, inner 클래스 사용은 불가
다. 저장할 필드에 final 사용 불가

@Table은 엔티티와 매핑할 테이블 정보를 지정한다.

속성	기능	기본값
name	매핑할 테이블 이름	엔티티 이름을 사용
catalog	데이터베이스 catalog 매핑	
schema	데이터베이스 schema 매핑	
uniqueConstraints (DDL)	DDL 생성 시에 유니크 제약 조건 생성	

4. 데이터베이스 스키마 자동생성

DDL을 어플리케이션 실행 시점에 자동으로 생성이 가능하다.

DB의 테이블 중심 => 객체 중심

데이터베이스 방언(Dialect)을 활용하여 데이터베이스에 맞는 적절한 DDL이 생성한다. (개발환경에서만 사용)

hibernate.hbm2ddl.auto

옵션	설명
create	기존테이블 삭제 후 다시 생성 (DROP + CREATE)
create-drop	create와 같으나 종료시점에 테이블 DROP
update	변경분만 반영(운영DB에는 사용하면 안됨)
validate	엔티티와 테이블이 정상 매핑되었는지만 확인
none	사용하지 않음

5. 필드와 컬럼 매핑 예


```java
import javax.persistence.*;
import java.time.LocalDate;
import java.time.LocalDateTime;
import java.util.Date;

@Entity
public class Member {

    @Id
    private Long id;

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

    private Integer age;

    @Enumerated(EnumType.STRING)
    private RoleType roleType;

    @Temporal(TemporalType.TIMESTAMP)
    private Date createdDate;

    @Temporal(TemporalType.TIMESTAMP)
    private Date lastModifiedDate;

    @Lob
    private String description;

    //Getter, Setter…
}
```

어노테이션	설명
@Column	컬럼 매핑
@Temporal	날짜 타입 매핑
@Enumerated	enum 타입 매핑
@Lob	BLOB, CLOB 매핑
@Transient	특정 필드를 컬럼에 매핑하지 않음(매핑 무시)

5. 필드와 컬럼 매핑

@Column

속성	설명	기본값
name	필드와 매핑할 테이블의 컬럼 이름	객체의 필드 이름
insertable, updatable	등록, 변경 가능 여부	TRUE
nullable(DDL)	null 값의 허용 여부를 설정한다. false로 설정하면 DDL 생성 시에 not null 제약조건이 붙는다.	
unique(DDL)	@Table의 uniqueConstraints와 같지만 한 컬럼에 간단히 유니크 제약조건을 걸 때 사용한다.	
columnDefinition (DDL)	데이터베이스 컬럼 정보를 직접 줄 수 있다. ex) varchar(100) default 'EMPTY'	필드의 자바 타입과 방언 정보를 사용해
length(DDL)	문자 길이 제약조건, String 타입에만 사용한다.	255
precision, scale(DDL)	BigDecimal 타입에서 사용한다(BigInteger도 사용할 수 있다). precision은 소수점을 포함한 전체 자 릿수를, scale은 소수의 자릿수다. 참고로 double, float 타입에는 적용되지 않는다. 아주 큰 숫자나 정 밀한 소수를 다루어야 할 때만 사용한다.	precision=19, scale=2

@Enumerated

속성	설명	기본값
value	• EnumType.ORDINAL: enum 순서를 데이터베이스에 저장 • EnumType.STRING: enum 이름을 데이터베이스에 저장	EnumType.ORDINAL

5. 필드와 컬럼 매핑

@Temporal

속성	설명	기본값
value	• **TemporalType.DATE**: 날짜, 데이터베이스 date 타입과 매핑 (예: 2013–10–11) • **TemporalType.TIME**: 시간, 데이터베이스 time 타입과 매핑 (예: 11:11:11) • **TemporalType.TIMESTAMP**: 날짜와 시간, 데이터베이 스 timestamp 타입과 매핑(예: 2013–10–11 11:11:11)	

@Temporal 지정은 Date 타입만 지원된 과거 버전에서 사용했던 방식으로 최신 버전에서는 LocalDate, LocalDateTime API 만 지정하면 된다.

6. 기본 키 매핑

@Id : 직접 할당인 경우에는 @Id만 사용한다. PK와 매핑됨.

@GeneratedValue : 값 자동 생성

가. GenerationType.AUTO (기본)

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

나. GenerationType.IDENTITY

dialect에 지정된 데이터베이스에 위임하여 사용됨.
특징은 commit이 아닌 persist 시점에 SQL이 실행된다.

다. GenerationType.SEQUENCE

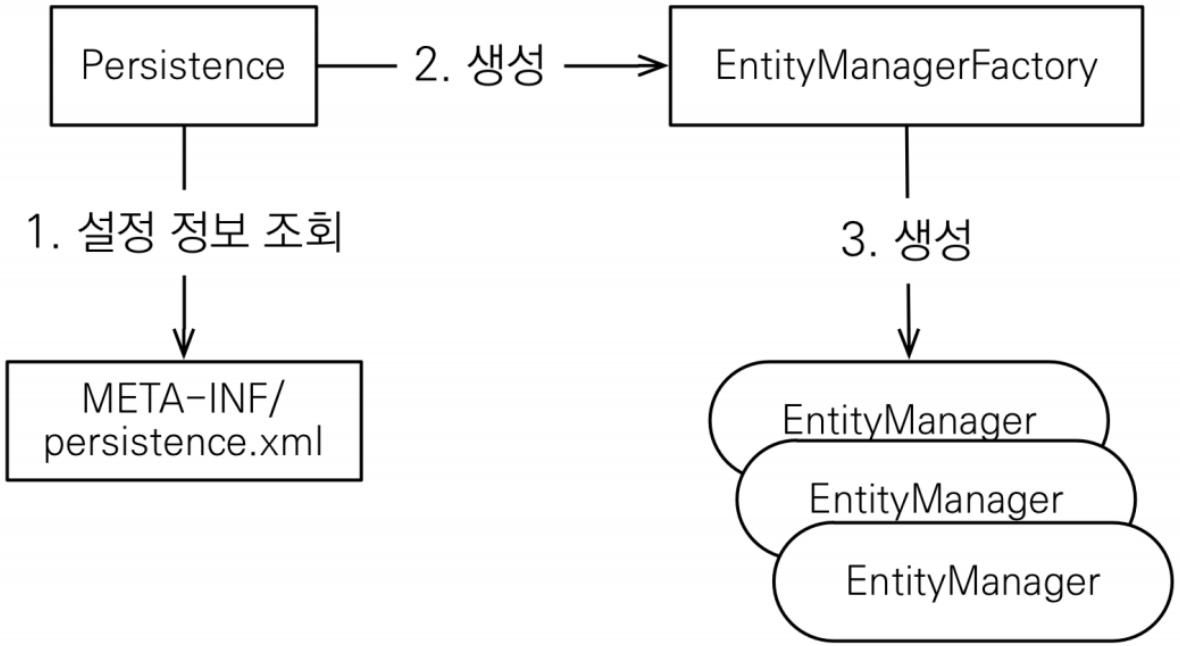
시퀀스 객체 사용하고 Oracle,PostgreSQL,H2 DB에서 사용됨.
@SequenceGenerator 필요

라. GenerationType.TABLE

키 생성하는 전담 테이블을 사용하는 방식으로 모든 DB에서 사용 가능.
@TableGenerator 필요

9장 영속성 관리

1. EntityManagerFactory 와 EntityManager

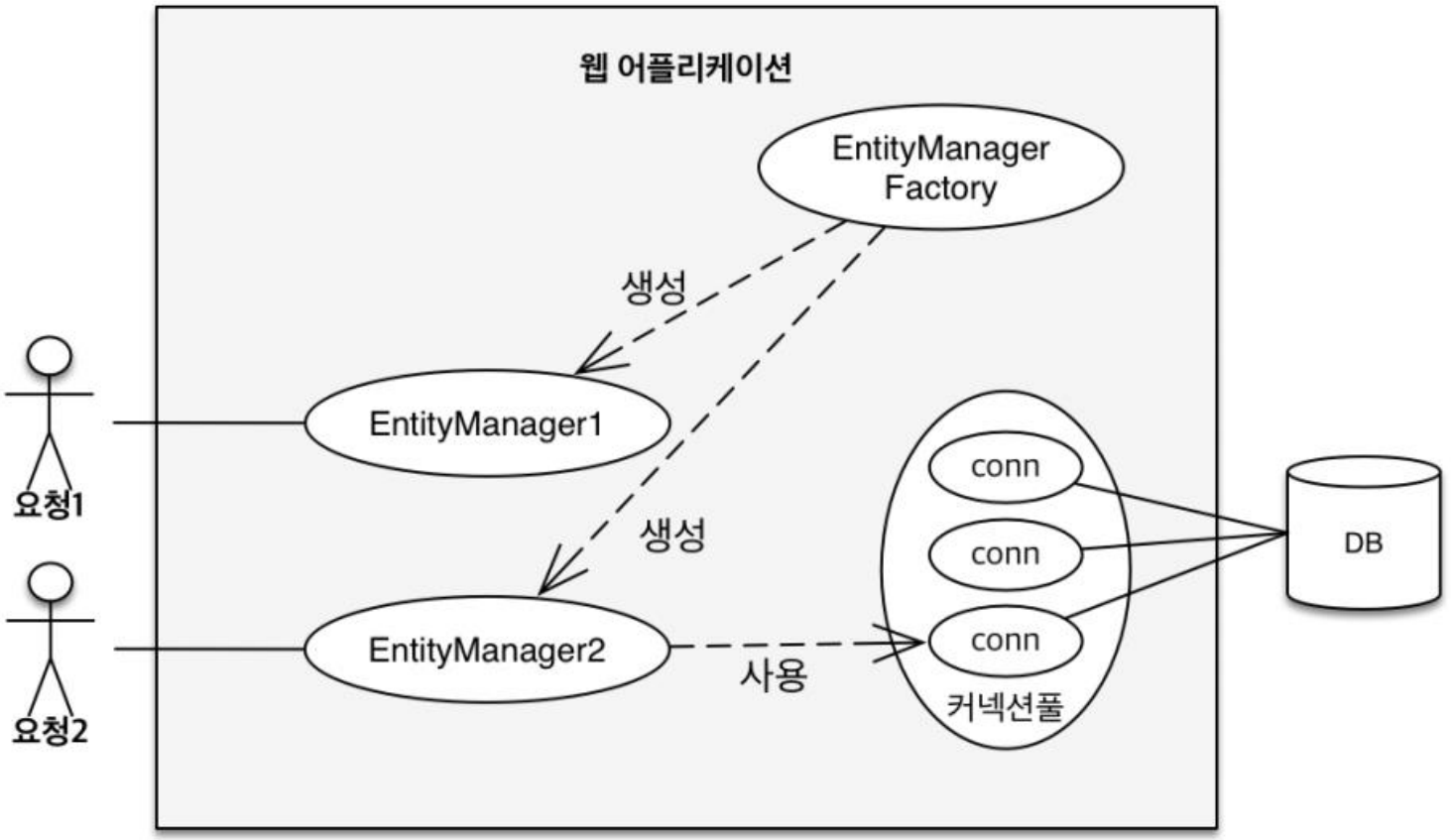


EntityManagerFactory는 하나만 생성해서 어플리케이션 전체에서 공유해서 사용한다.

EntityManager는 쓰레드간에 공유하면 안되고 사용하고 제거해야 된다.

JPA의 모든 데이터 변경은 반드시 트랜잭션 안에서 실행되어야 한다.

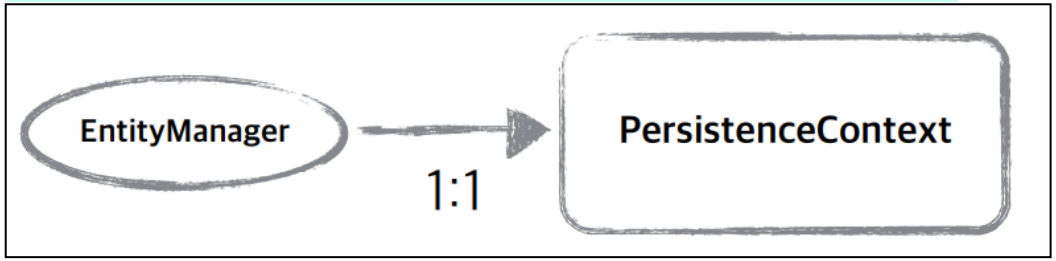
1. EntityManagerFactory 와 EntityManager



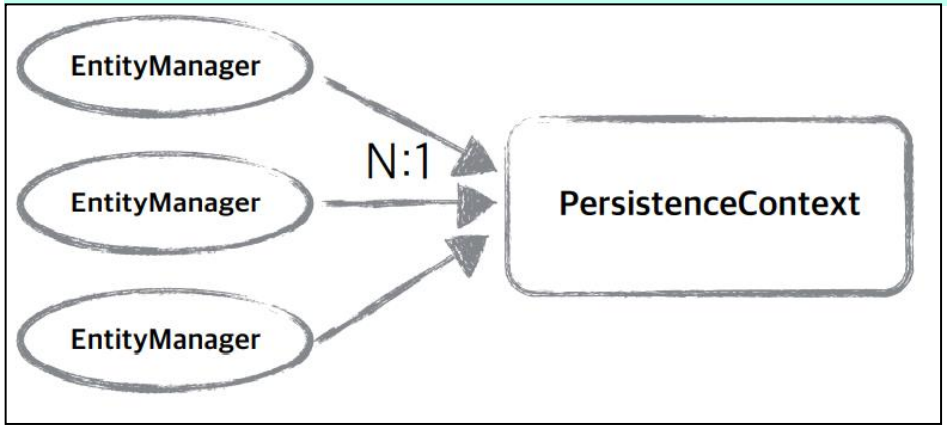
2. 영속성 컨텍스트 (*)

영속성 컨텍스트는 논리적인 개념으로서 EntityManager를 통해서 영속성 컨텍스트에 접근할 수 있다.
영속성 컨텍스트내에는 JPA가 관리하는 영속성 객체(Entity)가 있다.

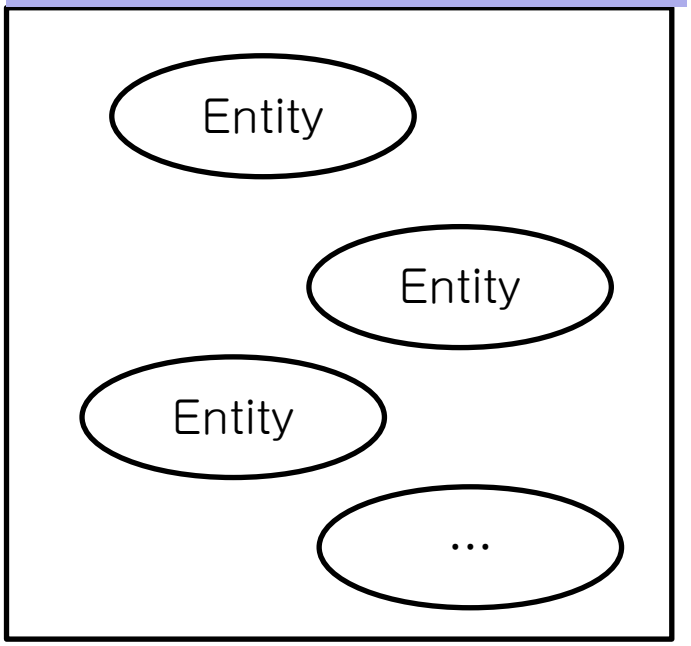
J2SE 환경



J2EE의 스프링 프레임워크 환경



영속성 컨텍스트



3. Entity 생명주기

가. 비영속 (new/transient)

영속성 컨텍스트와 전혀 관계가 없는 새로운 상태

나. 영속 (managed)

트랜잭션 범위의 영속성 컨텍스트에서 관리되는 상태

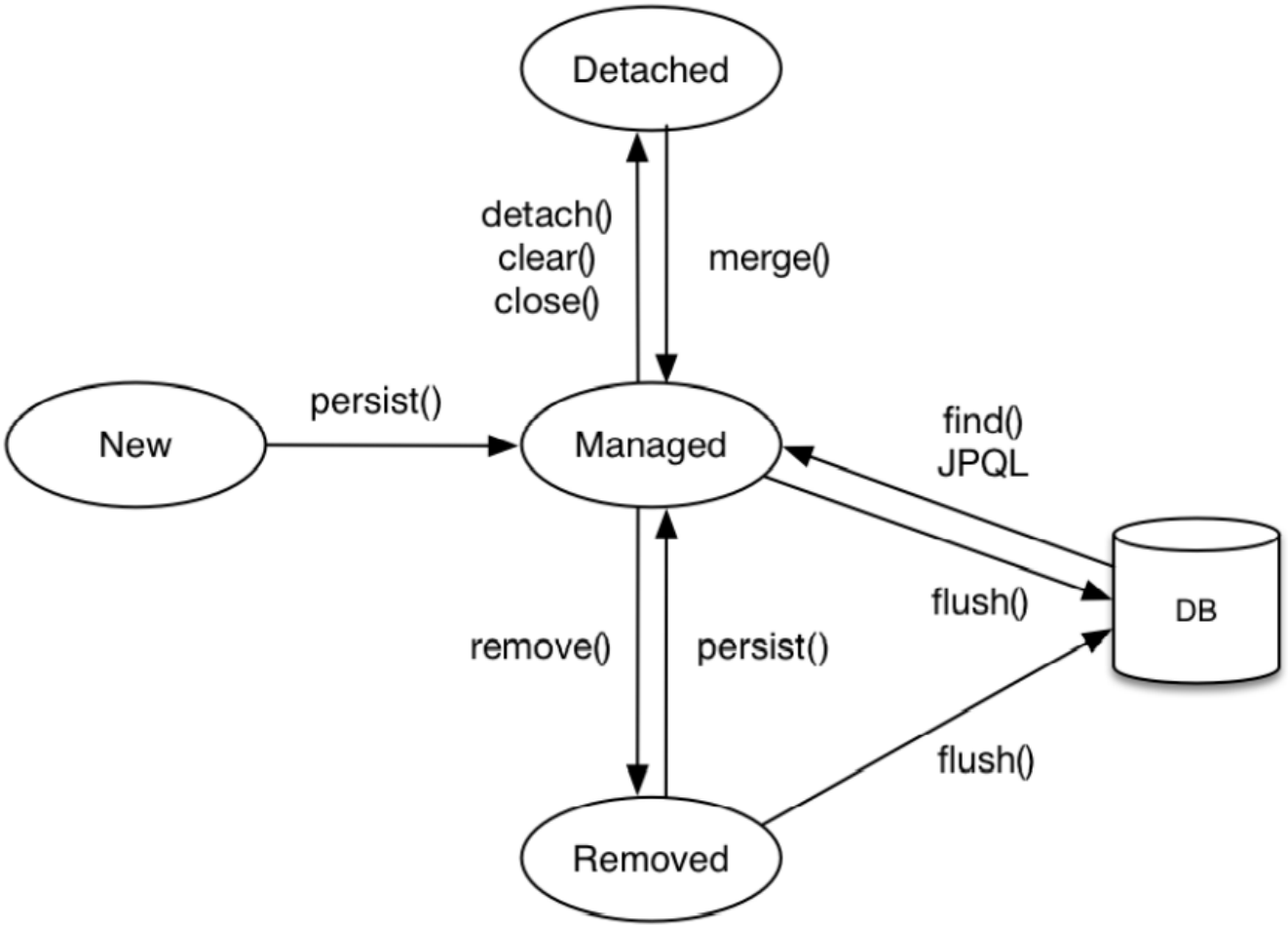
다. 준영속 (detached)

영속성 컨텍스트에 저장되었다가 분리된 상태
더 이상 엔티티 객체는 관리가 안됨

라. 삭제 (removed)

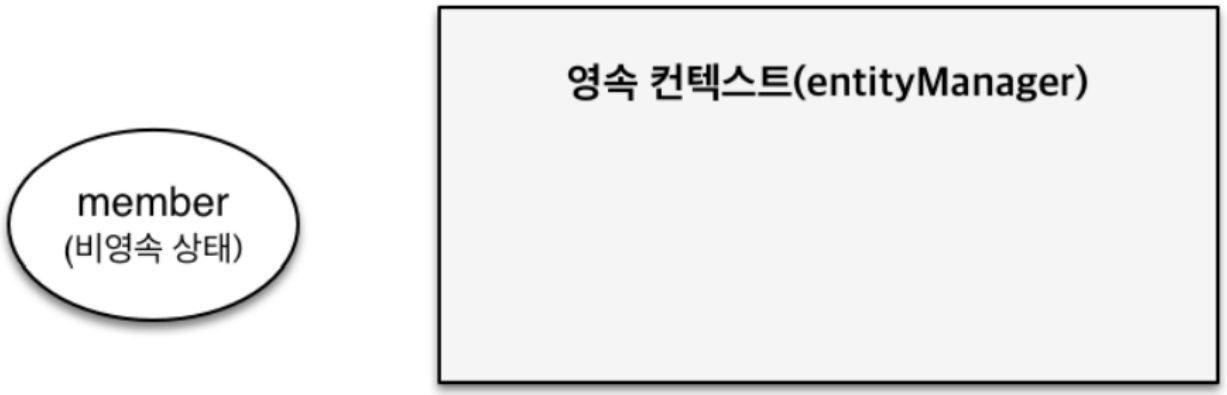
삭제된 상태

3. Entity 생명주기



가. 비영속 (new/transient)

영속성 컨텍스트와 전혀 관계가 없는 새로운 상태




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

나. 영속 (managed)

영속성 컨텍스트에 관리되는 상태 (트랜잭션 범위)


```
//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);
```

다. 준영속 (detached) 및 삭제

영속성 컨텍스트에 저장되었다가 분리 및 삭제된 상태로서 더 이상 엔티티는 관리가 안된다.

```
//회원 엔티티를 영속성 컨텍스트에서 분리, 준영속 상태
em.detach(member);
```

```
//객체를 삭제한 상태(삭제)
em.remove(member);
```

4. 영속성 컨텍스트의 장점

1) 1차 캐시

2) 동일성(identity) 보장

3) 트랜잭션을 지원하는 쓰기 지연
(transactional write-behind)

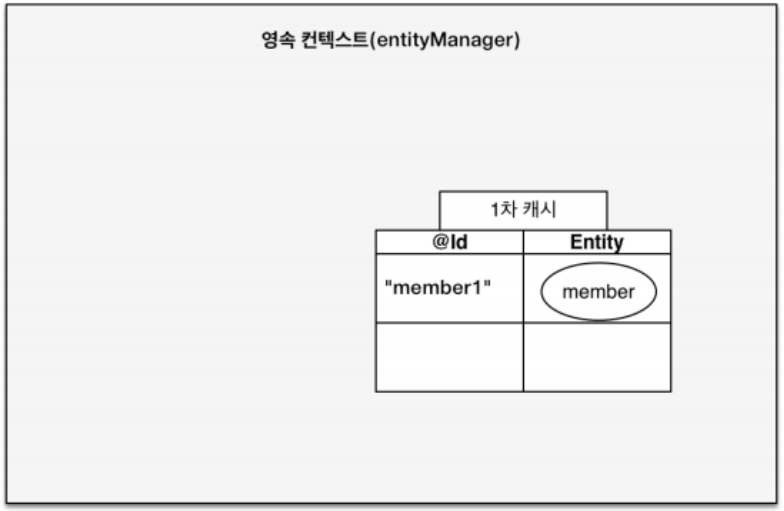
4) 변경 감지 (Dirty Checking)

5) 지연 로딩 (Lazy Loading)

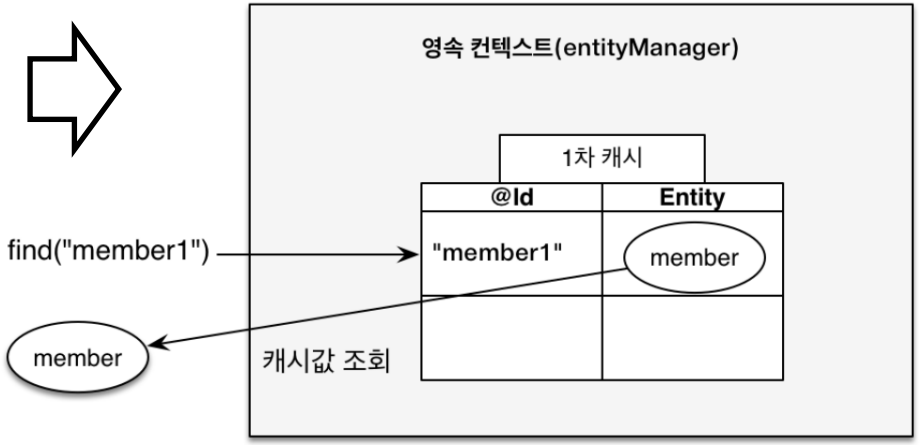
4. 영속성 컨텍스트의 장점

1) 1 차 캐시에서 조회

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

//엔티티를 영속
em.persist(member);
```

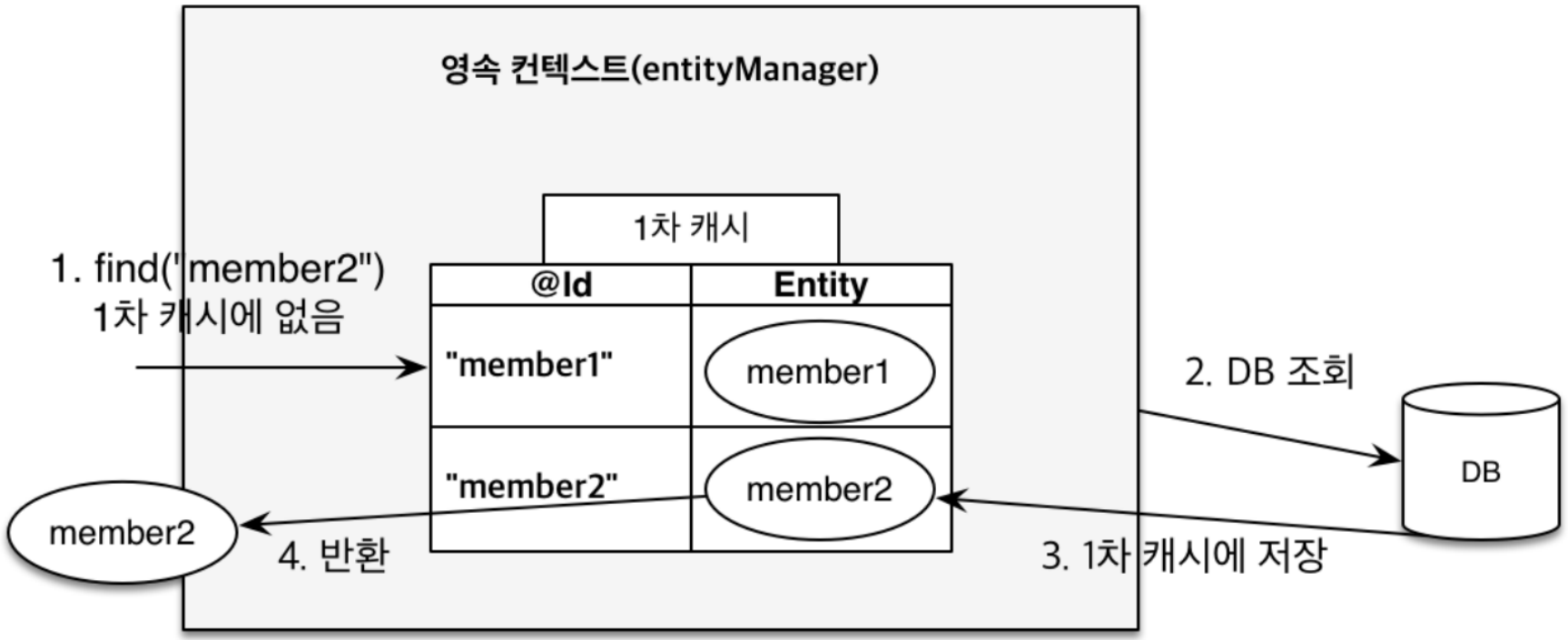



```
//1차 캐시에서 조회
Member findMember = em.find(Member.class, "member1");
```



DB에서 조회

```
Member findMember2 = em.find(Member.class, "member2");
```



4. 영속성 컨텍스트의 장점

2) 영속 엔티티의 동일성 보장

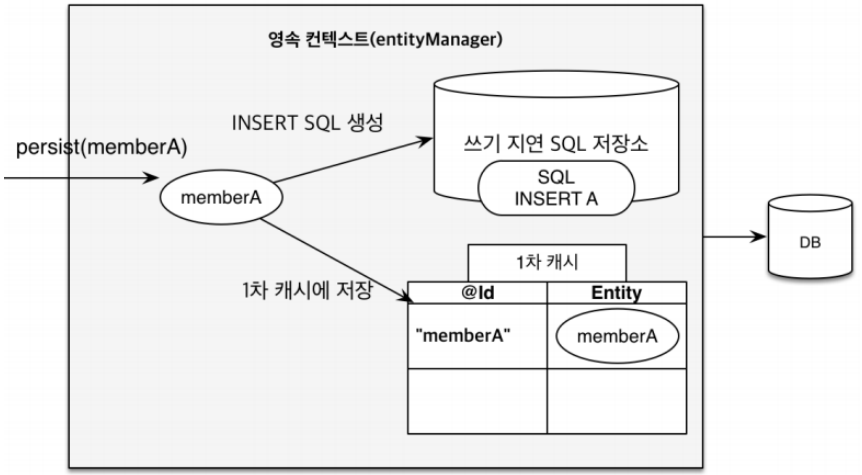
```
Member a = em.find(Member.class, "member1");
Member b = em.find(Member.class, "member1");

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

3) 트랜잭션을 지원하는 쓰기 지연

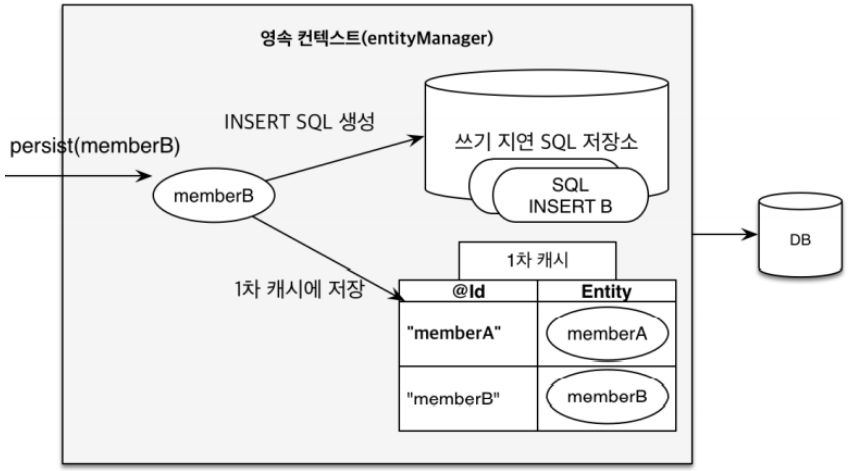
```
EntityManager em = emf.createEntityManager();
EntityTransaction transaction = em.getTransaction();
//엔티티 매니저는 데이터 변경시 트랜잭션을 시작해야 한다.
transaction.begin();  // [트랜잭션] 시작

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

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

4. 영속성 컨텍스트의 장점

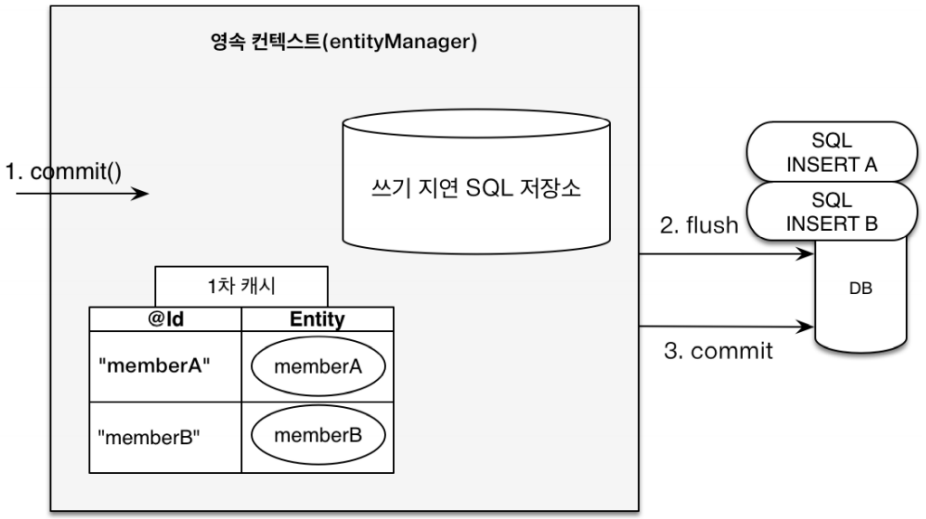
```
em.persist(memberA);
```




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

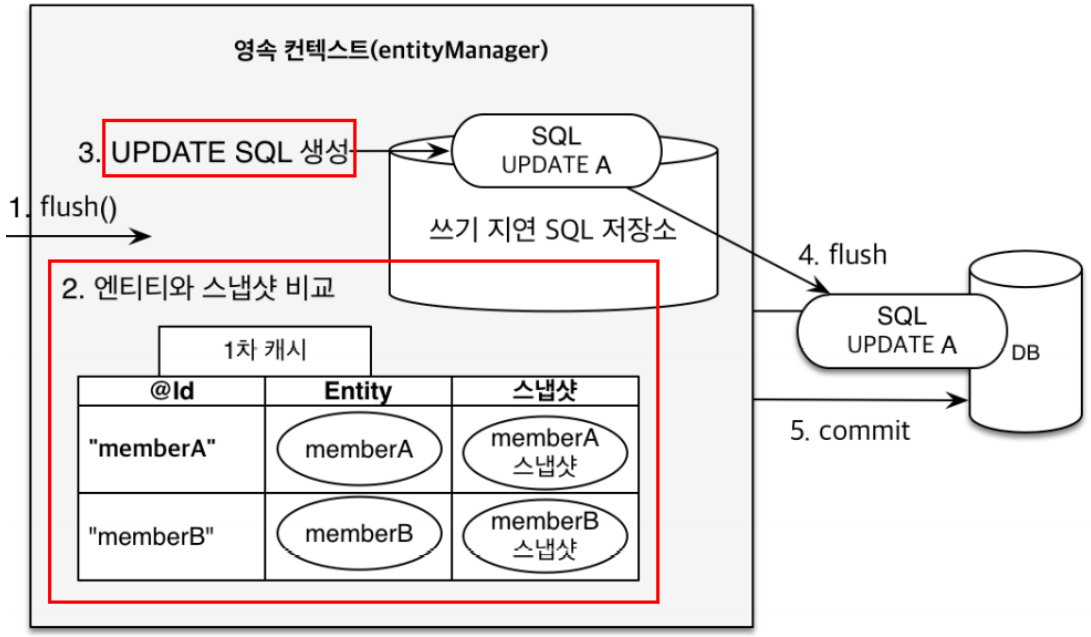



```
transaction.commit();
```



4) 변경 감지 (Dirty Checking) – 엔티티 수정

```
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);

transaction.commit(); // [트랜잭션] 커밋
```

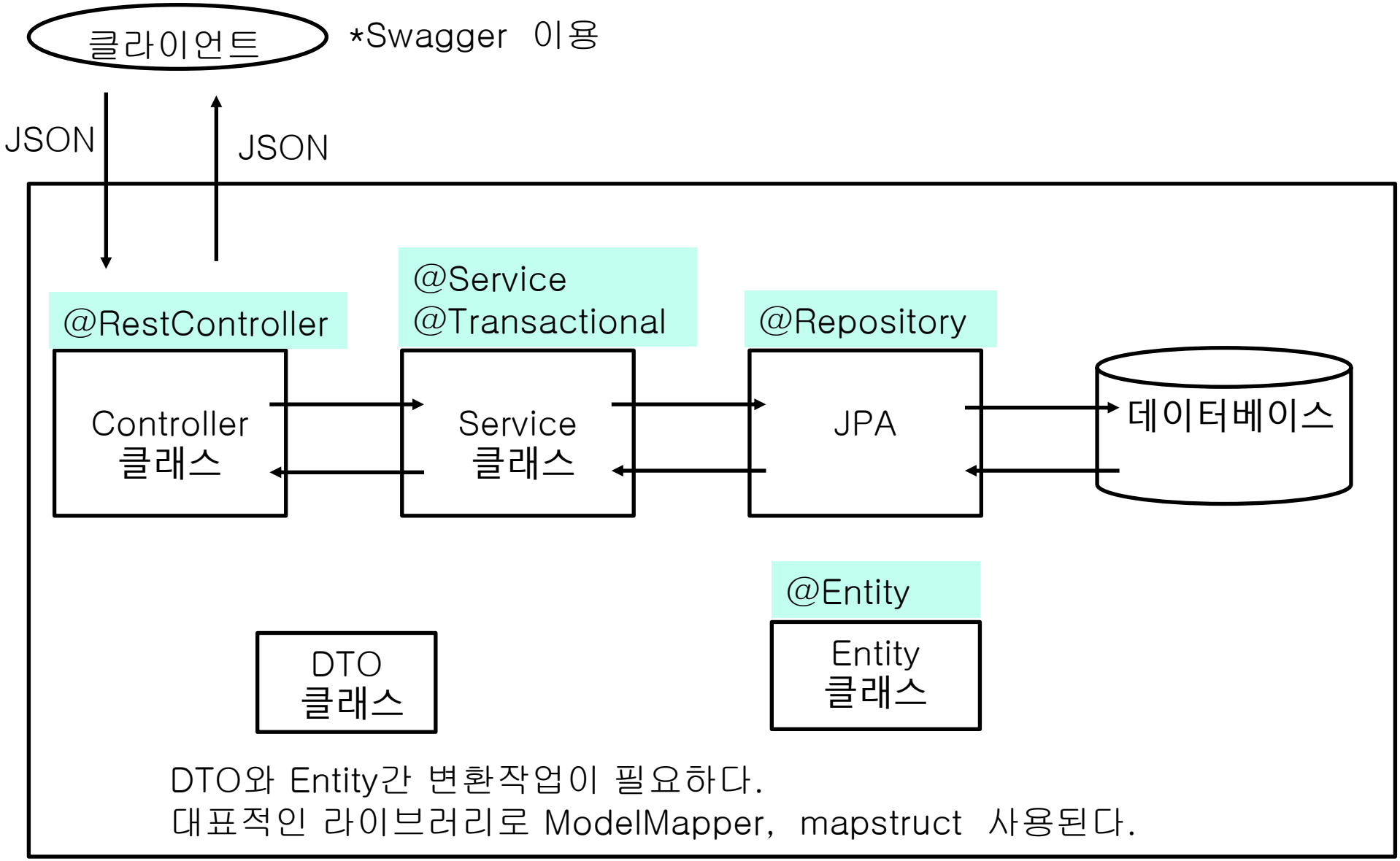


5) 엔티티 삭제

```
//삭제 대상 엔티티 조회
Member memberA = em.find(Member.class, "memberA");

em.remove(memberA); //엔티티 삭제
```

13장 Spring Data JPA

* JPA 연동 아키텍쳐


클라이언트
*Swagger 이용
JSON
JSON
@RestController
@Service
@Transactional
@Repository
Controller
클래스
Service
클래스
JPA
데이터베이스
@Entity
DTO
클래스
Entity
클래스
DTO와 Entity간 변환작업이 필요하다.
대표적인 라이브러리로 ModelMapper, mapstruct 사용된다.

1. Spring Data JPA 개요

SpringBoot와 JPA만 사용해도 개발 생산성이 많이 증가하지만,
Spring Data JPA를 사용하면 훨씬 더 획기적으로 개발 생산성이 증가된다.

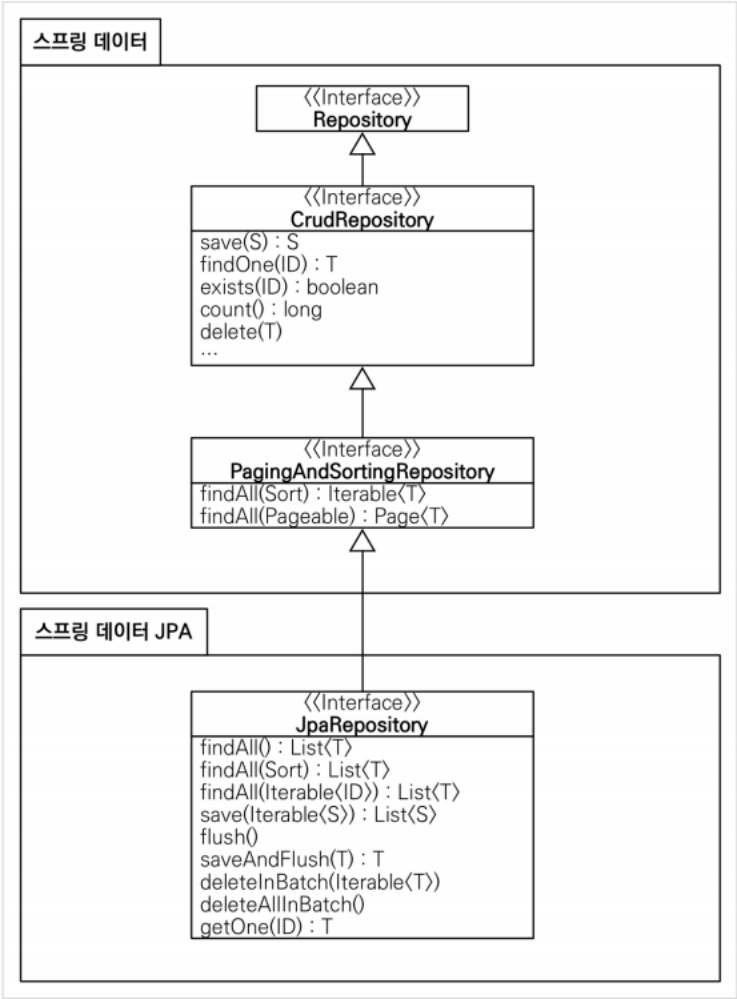
Spring Data JPA는 JPA를 매우 편리하게 사용하도록 도와주는 기술이다.

1) 의존성 설정

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
</dependency>
<dependency>
    <groupId>org.bgee.log4jdbc-log4j2</groupId>
    <artifactId>log4jdbc-log4j2-jdbc4</artifactId>
    <version>1.16</version>
</dependency>
<dependency>
    <groupId>org.modelmapper</groupId>
    <artifactId>modelmapper</artifactId>
    <version>2.4.1</version>
</dependency>
```

2) Repository 인터페이스 작성

```
@Repository
public interface SpringDataJpaCustomerRepository extends JpaRepository<Customer, Long> {

    List<Customer> findByName(String name);

    @Query(value = "select c from Customer c where c.name = :name'
    List<Customer> findCustomerListByName(String name);

    @Query(value = "select c.name from Customer c")
    List<String> findStringList();

    @Query("select c from Customer c WHERE c.id = ?1 and c.name =
    Optional<Customer> findCustomerByIdAndName(Long id, String nam

    @Query(value = "select c.name from Customer c")
    List<String> findStringSortList(Sort sort);

    @Query(value = "select c from Customer c ORDER BY id")
    Page<Customer> findAllCustomersWithPagination(Pageable pageabl

    @Query(value = "select c from Customer c where c.name = :name
    Page<Customer> findByNameCustomersWithPagination(String name,
```



- 인터페이스를 통한 기본적인 CRUD 기능 제공
- findByName(), findByEmail()처럼 메서드 이름만으로 조회 기능을 제공한다.
- 필요시 Named Query를 직접 지정할 수 있다.
- 정렬 및 페이징 기능

3) Service 작성

```java
public interface CustomerService {


    CustomerDTO save(CustomerDTO customer);
    Optional<CustomerDTO> findById(String id);
    List<CustomerDTO> findByName(String name);
    List<CustomerDTO> findAll();

    String update(CustomerDTO customer);
    String remove(String id);

    //JPQL
    List<CustomerDTO> findCustomerListByName(String name);
    List<String> findStringList();
    Optional<CustomerDTO> findCustomerByIdAndName(String id, String name);

    List<String> findStringSortList(String v);

    Page<CustomerDTO> findAllCustomersWithPagination(int offse
    Page<CustomerDTO> findByNameCustomersWithPagination(String
}
```

```java
@Service
@Transactional
public class CustomerServiceImpl implements CustomerService {

    private SpringDataJpaCustomerRepository repository;

    public CustomerServiceImpl(SpringDataJpaCustomerRepository repository) {
        this.repository = repository;
    }

    @Override
    public CustomerDTO save(CustomerDTO customer) {
//////////////////////////////////////////////////////////////
        ModelMapper mapper = new ModelMapper();
        Customer entity = mapper.map(customer, Customer.class);
//////////////////////////////////////////////////////////////
        repository.save(entity);

        return customer;
    }
```

4) Controller 작성

```java
@RestController
public class CustomerController {

    @Autowired
    CustomerService service;

    @GetMapping("/cust")
    public List<CustomerDTO> retriveCustList() {
        List<CustomerDTO> result = service.findAll();
        return result;
    }
    @GetMapping("/cust/{id}")
    public Optional<CustomerDTO> retriveCustById(@PathVariab
        Optional<CustomerDTO> dto = service.findById(id);
        return dto;
    }
    @GetMapping("/cust/v1/{name}")
    public List<CustomerDTO> retriveCustByName(@PathVariable
        return service.findByName(name);
    }
    @PostMapping("/cust")
    public CustomerDTO createCust(@RequestBody CustomerDTO
        CustomerDTO dto = service.save(customer);
        return dto;
    }
```

```java
//JPQL
@GetMapping("/cust/jpql/{name}")
public List<CustomerDTO> findCustomerListByName(String name) {
    return service.findCustomerListByName(name);
}
@GetMapping("/cust/jpql")
public List<String> findStringList() {
    return service.findStringList();
}
@GetMapping("/cust/jpql/sort")
public List<String> findStringSortList() {
    return service.findStringSortList("name");
}
@GetMapping("/cust/jpql/id/{id}/name/{name}")
public Optional<CustomerDTO> findCustomerByIdAndName(@PathVariable String id, String name) {
    return service.findCustomerByIdAndName(id, name);
}
@GetMapping("/cust/page/offset/{offset}/siz/{size}")
public List<CustomerDTO> findAllCustomersWithPagination(@PathVariable int offset,
                                                        @PathVariable int size){
    Page<CustomerDTO> xx = service.findAllCustomersWithPagination(offset, size);
    System.out.println(xx.getContent());
    return xx.getContent();
}
@GetMapping("/cust/page/name/{name}/offset/{offset}/siz/{size}")
public List<CustomerDTO> findByNameCustomersWithPagination(@PathVariable String name,
        @PathVariable int offset,
        @PathVariable int size){
    Page<CustomerDTO> xx = service.findByNameCustomersWithPagination(name, offset, size);
    System.out.println(xx.getContent());
    return xx.getContent();
}
```

5) Entity 작성

```java
@Entity
public class Customer {

    @Id
    @GeneratedValue
    private Long id;

    private String name;

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }
```

6) DTO 작성

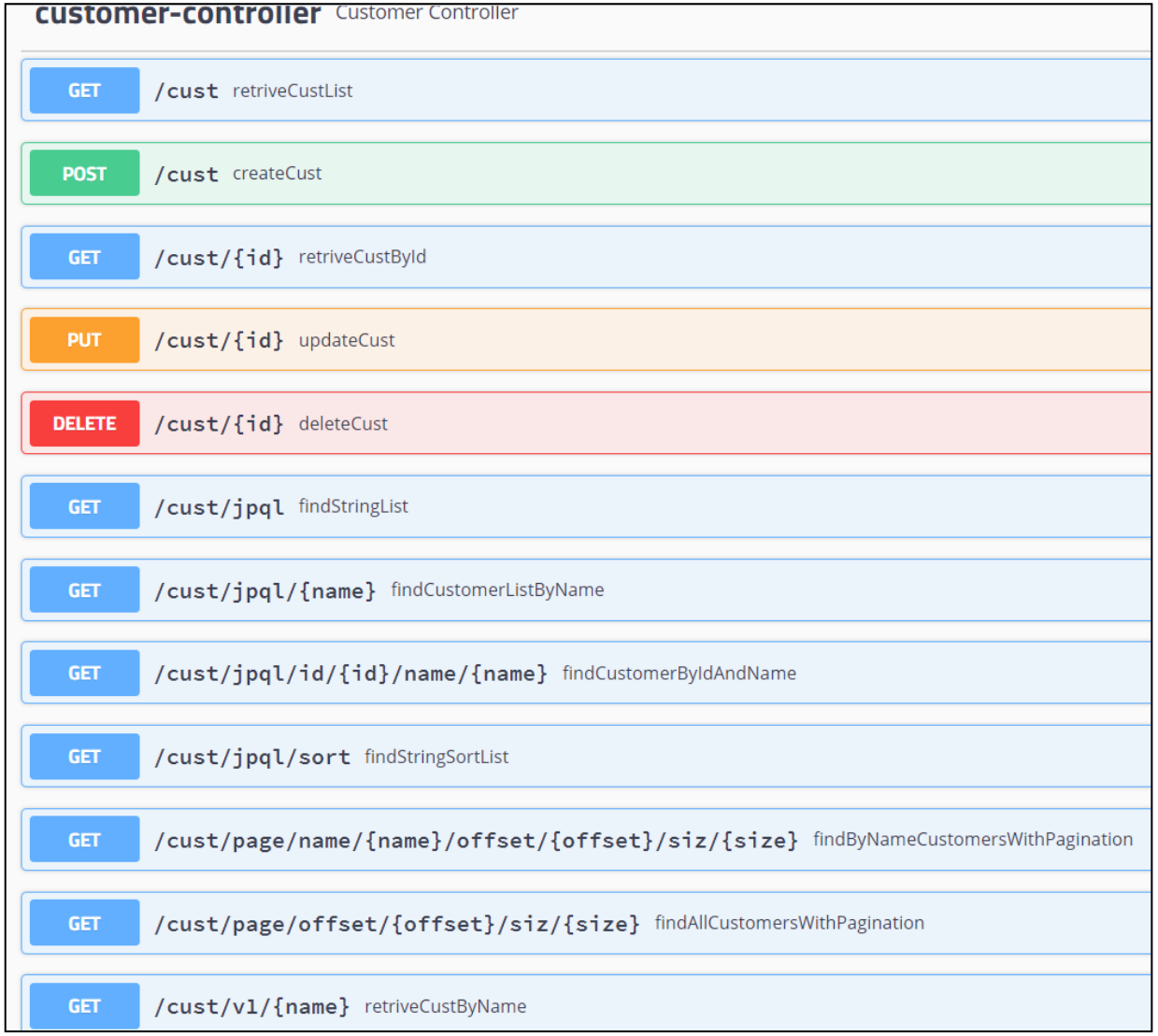
```java
@Data
public class CustomerDTO {

    private String id;
    private String name;

}
```

Entity와 DTO(VO)간의 변경은
ModelMapper API 활용한다.

7) properties 작성

```
server.port=8090

spring.datasource.url=jdbc:h2:tcp://localhost/~/jpa
spring.datasource.driver-class-name=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=sa
spring.jpa.show-sql=true
spring.jpa.hibernate.ddl-auto=none
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
```

http://localhost:8090/swagger-ui.html



Thank you