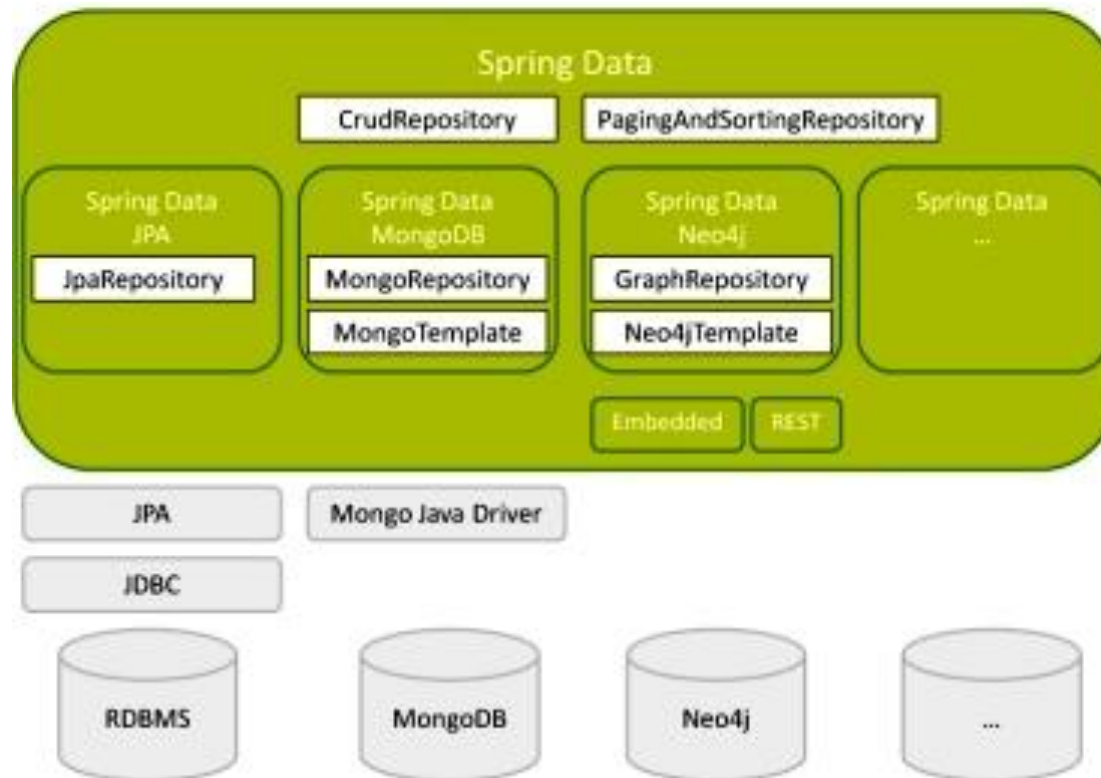


Spring Data JPA, Test

558280-1
2025년 봄학기
4/16/2025
박경신

Spring Data JPA

- Spring Data JPA 는 JPA 기반 Repository를 보다 쉽게 구현할 수 있도록 하는 Spring Data 프로젝트의 일부
 - <https://docs.spring.io/spring-data/jpa/reference/index.html>



<https://stackoverflow.com/questions/16148188/whats-the-difference-between-jpa-and-spring-data-jpa>

데이터 영속성

□ Persistence (영속성)

- 데이터들이 프로그램이 종료되어도 사라지지 않고 어떤 곳에 저장되는 개념을 영속성(Persistence)라고 함
- 물리적인 저장소를 이용해 프로그램의 상태와 상관없이 데이터를 저장하는 행위를 영속화라고 할 수 있음
- 관계형 데이터베이스에 데이터를 저장하기 위해서 SQL(Structured Query Language)을 이용해 데이터를 영속화함

□ JDBC (Java Data Base Connectivity)

- 자바에서는 데이터의 영속성을 위해 JDBC 인터페이스를 지원
- 각 데이터베이스 제조사들은 JDBC 인터페이스를 구현한 클래스를 제공하며 이를 드라이버라고 함
- JDBC는 매핑 작업을 개발자가 수행해야 하는 번거로움이 있음

OOP vs RDBMS 모델

□ 객체지향 모델 vs 관계형 데이터베이스 모델

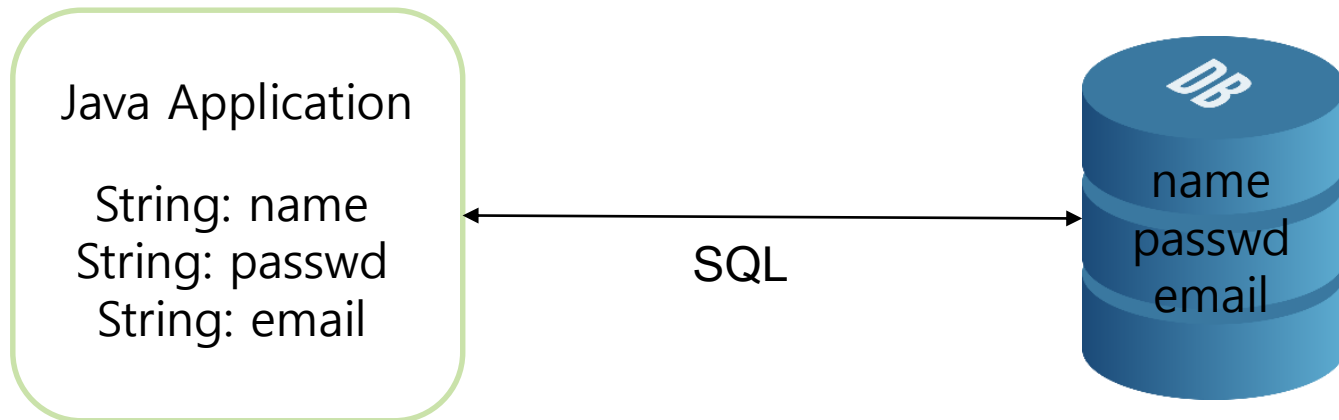
- 객체지향 모델과 테이블 중심의 관계형 데이터 베이스 모델은 패러다임이 일치하지 않으며, 이를 서로 매핑하는 것은 개발 과정에서 많은 비용을 발생하게 함
- 데이터 영속성을 위해 객체와 테이블을 서로 맞추는 과정이 필요함

객체 지향 모델	관계형 데이터베이스 모델
Object/Class	Table, Row/Record
Property	Column
Identity	Primary Key
Relationship/다른 Entity 참조	Foreign Key
Inheritance, Polymorphism	없음
Method	SQL, Logic, Trigger

Persistence Framework

□ Persistence Framework

- 자바 프로그램에서 관계형 데이터베이스 사용을 돕는 프레임워크
- Persistence Framework는 개발자가 직접 JDBC Programming을 하지 않도록 기능을 제공
- SQL Mapper vs ORM(Object Relational Mapping)
 - SQL Mapper는 자바 코드와 SQL을 분리하며 개발자가 작성한 SQL의 수행결과를 객체로 매핑
 - ORM은 객체와 관계형 데이터베이스 사이에서 매핑을 담당하며 SQL을 생성하여 패러다임의 불일치를 해결



SQL Mapper vs ORM

□ SQL Mapper

- Object와 SQL 필드를 매핑하여 데이터를 객체화 하는 기술
- 객체와 테이블 간의 관계를 매핑하는 것이 아님
- SQL 문을 직접 작성하고 쿼리 수행 결과를 어떠한 객체에 매핑할지 바인딩하는 방법
- RDBMS(Relational DataBase Management System)에 종속적
- 예시 - [JdbcTemplate](#), [MyBatis](#)

□ ORM (Object-Relational Mapping)

- Object와 DB Table을 매핑하여 데이터를 객체화 하는 기술
- 객체가 DB에 연결되기 때문에 SQL을 직접 작성하지 않음
- RDBMS에 독립적
- 복잡한 쿼리의 경우 JPQL(Java Persistence Query Language)을 사용하거나 SQL Mapper를 혼용하여 사용 가능

ORM

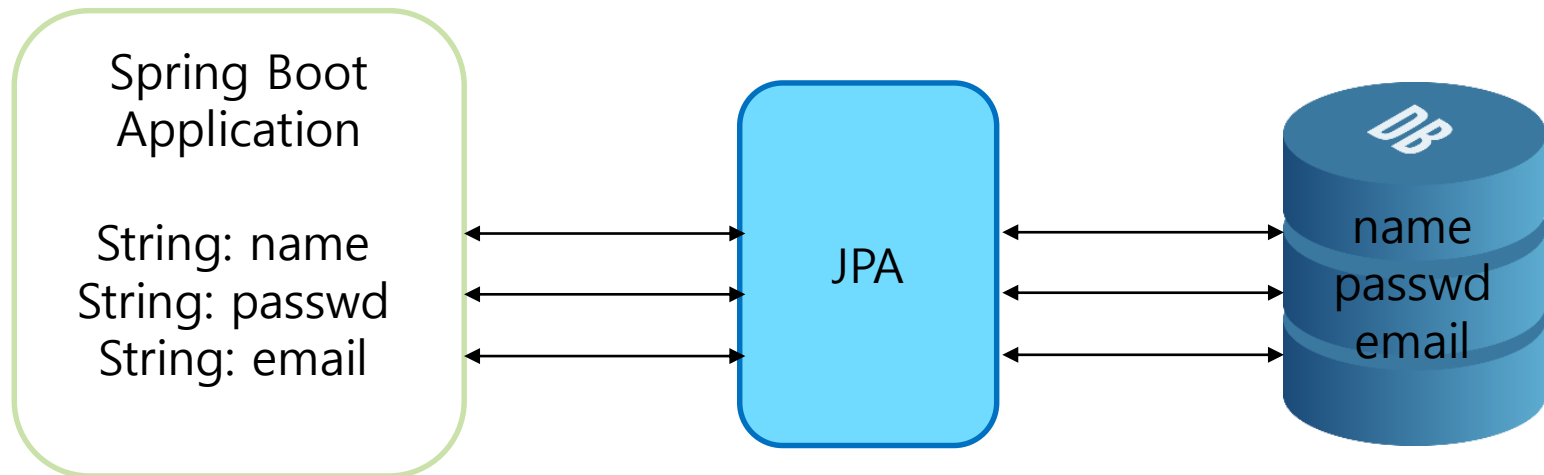
□ ORM (Object-Relational Mapping)

- ORM(Object-Relational Mapping)이란 자바의 객체(Object)와 Relation (관계형 데이터베이스)를 맵핑하는 기술
- DB의 특정 테이블이 자바의 객체로 맵핑되어 SQL문을 일일이 작성하지 않고 객체로 구현할 수 있도록 하는 프레임워크
- 관계형 데이터베이스 테이블은 객체지향적인 특성(상속, 다형성, 레퍼런스) 등이 없어서 Java와 같은 객체 지향적 언어로의 접근이 쉽지 않음
- ORM을 사용하면 보다 객체지향적으로 관계형 데이터베이스를 사용할 수 있음

JPA

□ JPA (Java Persistence API)

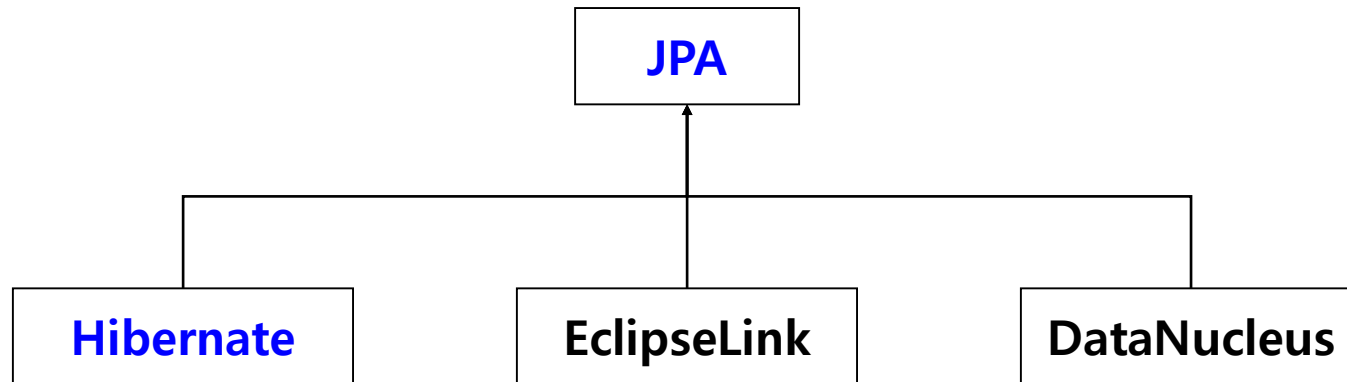
- JPA란 Java Persistence API의 약자
- 자바의 ORM(Object-Relational Mapping) 기술 표준
- CRUD(Create Read Update Delete) 메소드 기본 제공
- [Hibernate](#), Spring JPA, [EclipseLink](#) 등과 같은 구현체가 있고 이것의 표준 인터페이스가 JPA 임



Hibernate

□ Hibernate

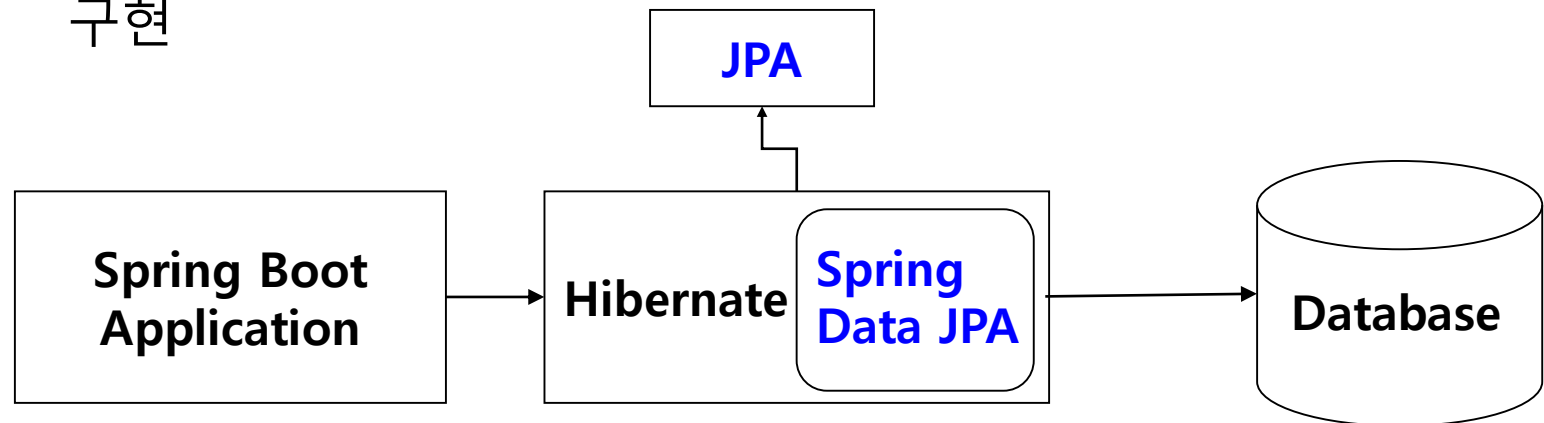
- ORM Framework 중 하나로 **JPA Provider**라고도 부름
- Hibernate는 JPA에 정의된 명세대로 **실제 기능을 구현한 구현체**
- 현재 JPA 구현체 중 가장 많이 사용됨



Spring Data JPA

□ Spring Data JPA

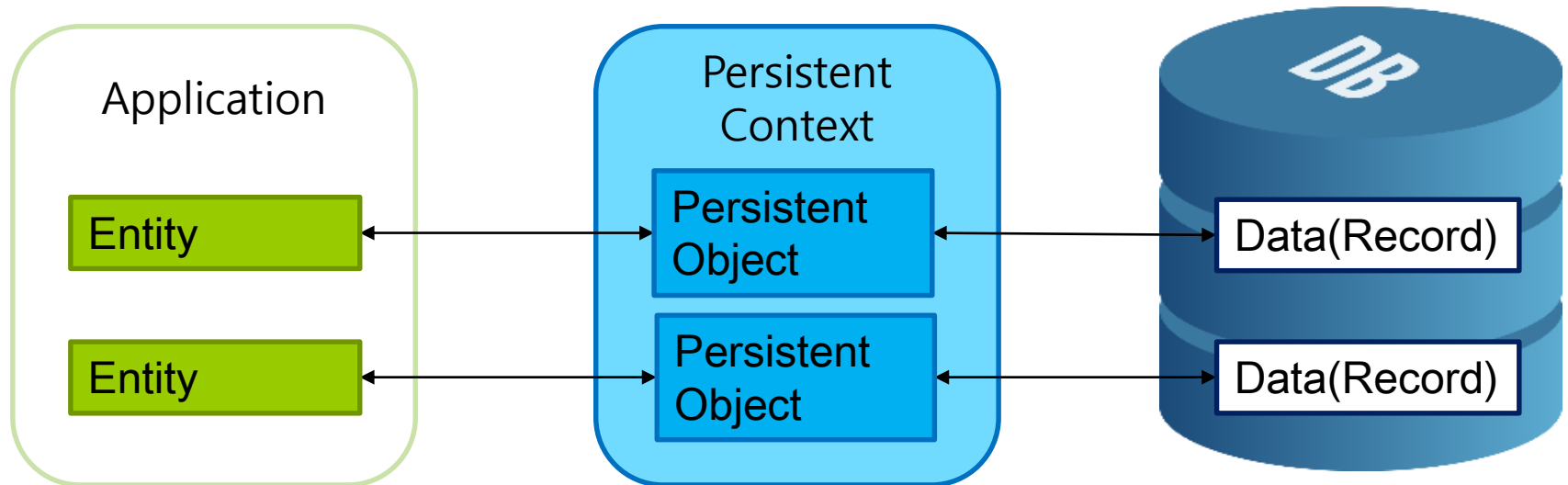
- JPA를 더 쉽게 사용하기 위한 Spring Data Framework
- JPA를 이용한 구현체를 더 추상화시켜 더 쉽고 간편하게 JPA를 이용한 프로젝트를 개발할 수 있게 해 주는 Spring 모듈
- CRUD 처리용 인터페이스 제공
- **Repository** 개발시 인터페이스만 작성하면 구현 객체를 동적으로 생성해서 주입
- 데이터 접근 계층 개발 시 인터페이스만 작성해도 됨
- Hibernate에서 자주 사용되는 기능을 조금 더 쉽게 사용할 수 있게 구현



Persistent Context

❑ Persistent Context (영속성 컨텍스트)

- Persistent Context는 Entity를 영구 저장하는 환경
- JPA가 관리하는 Entity 객체의 집합
- Entity 객체가 Persistent Context 에 들어오게 되면 JPA는 Entity 객체의 매핑 정보를 가지고 DB에 반영함
- Entity 객체가 Persistent Context에 들어오게 되어 관리 대상이 되면 그 객체를 영속 객체라고 부름



Persistent Context

- Persistent Context (영속성 컨텍스트)
 - Entity(객체)를 영구 저장하는 환경
 - Persistent Context는 Session(세션) 단위로 생명주기를 갖고 있음
 - 세션이 생성되면서 만들어지고, 세션이 종료되면 없어짐
 - Persistent Context에 접근하기 위해서 EntityManager를 사용함
 - EntityManager는 아래와 같은 방식으로 동작을 구성함
 - EntityManagerFactory를 통해 EntityManager 생성
 - EntityManager가 가지고 있는 Transaction을 시작
 - EntityManager를 통해 Persistent Context에 접근하고 객체를 CRUD 작업
 - Transaction을 Commit 하여 DB에 반영
 - EntityManager 종료

EntityManager

□ EntityManagerFactory

- EntityManager 를 만드는 공장
- 한 개만 만들어서 Application 전체에서 공유하도록 설계됨

□ EntityManager

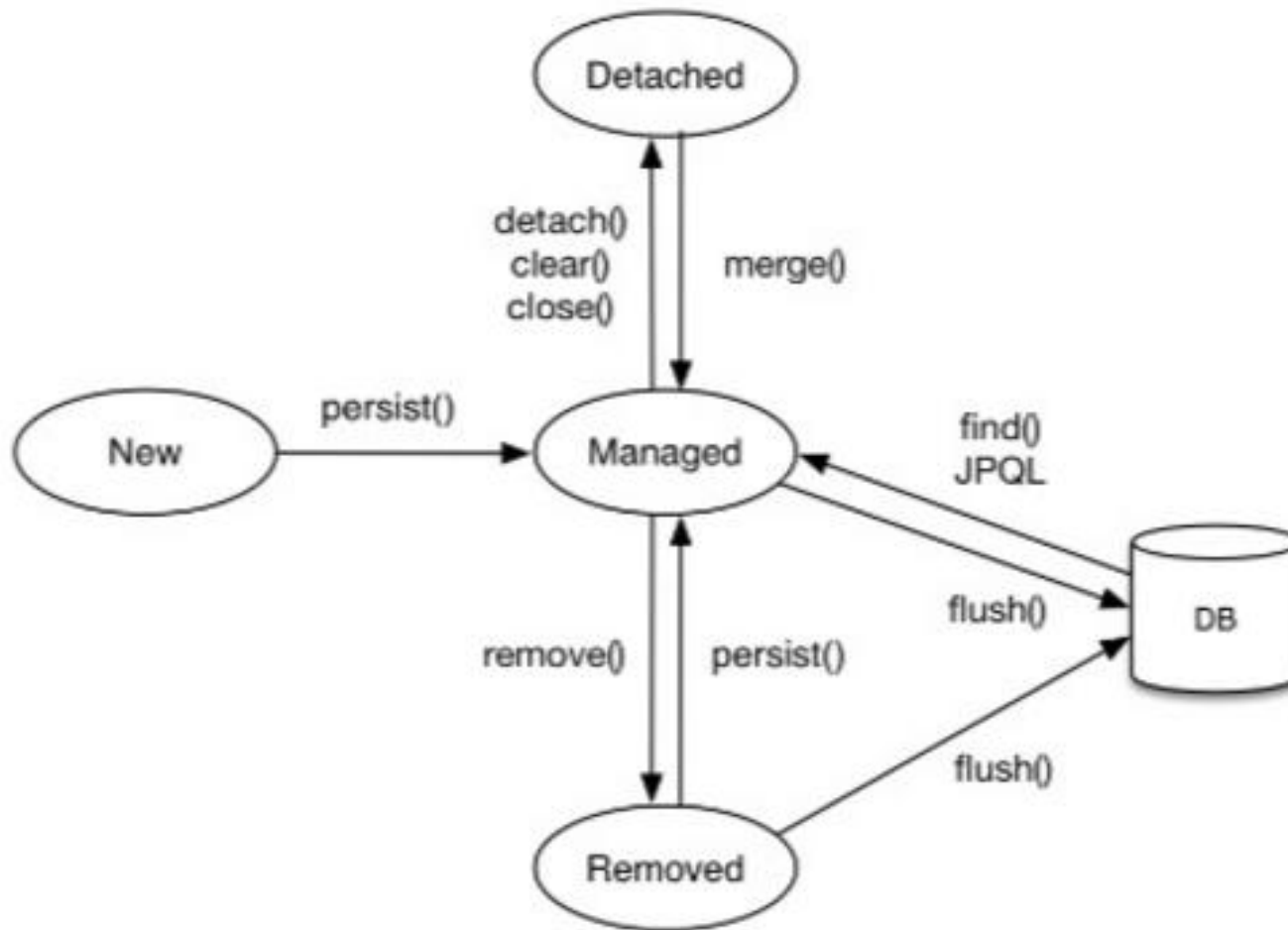
- EntityManager는 Persistent Context 내에서 Entity들을 보관 관리
- EntityManager는 JPA에서 제공하는 interface로 Spring Bean으로 등록되어 있어 **Autowired**로 사용할 수 있음
- Query Method, **Simple JPA Repository**는 직접적으로 **EntityManager**를 사용하지 않도록 한번 더 감싸준 것임
- Spring JPA에서 제공하지 않는 기능을 사용하거나 별도로 customizing을 해야 한다면 EntityManager를 직접 받아서 처리해야 함
- EntityManager는 **Entity Cache**를 갖고 있음

EntityManager

❑ CRUD(Create Read Update Delete)

- **Create** – `em.persist(member);` Entity를 Persistent Context에 저장
- **Read** – `Member member = em.find(memberId);` 1차 캐시에서 Entity 조회 (1차 캐시에 Entity가 없을 경우 DB를 조회해 Entity 생성)
- **Update** – `member.setName("newname"); em.merge(member);`
- `merge()` – 준영속 상태의 Entity를 다시 영속 상태로 변경
- **Delete** – `em.remove(member);` Persistent Context에서 삭제
- `query`
- `flush()` – Persistent Context 변경 내용을 DB에 반영 (호출시 변경 필드에 의해 자동으로 update/insert 됨)
- `clear()` – Persistent Context를 완전히 초기화
- `close()` – Persistent Context를 종료
- `detach()` – 특정 Entity를 준영속 상태로 전환

Entity Life Cycle



Entity Life Cycle

□ 비영속(New/Transient) 상태

- Persistent Context와 전혀 관계가 없는 상태, 순수한 Java 객체, Entity Manager가 관리하지 않는 상태
- Entity 객체를 생성하였지만 아직 Persistent Context에 저장하지 않은 상태를 의미

□ 영속(Managed) 상태

- Persistent Context에 저장된 상태
- Entity가 Persistent Context에 의해 관리됨을 의미
- PK(Primary Key)을 통해 필요한 Entity 객체를 꺼내 사용할 수 있음
- 영속 상태가 되었다고 바로 DB에 값이 저장되는 것이 아니고 Transaction Commit 시점에서 Persistent Context에 있는 정보들을 DB에 Query로 보냄

@Autowired

```
private EntityManager em; // Autowired로 EntityManager추가  
Member member = new Member(); //객체만 생성한 비영속상태  
em.persist(member); // 객체를 저장한 영속상태
```

Entity Life Cycle

□ 준영속(Detached) 상태

- Persistent Context에 저장되었다가 분리된 상태
- Entity 를 준영속 상태로 만들려면 `detach()` 를 호출
- 준영속 상태에서는 1차 캐시, 쓰기 지연, 변경 감지, 지연 로딩을 포함한 Persistent Context가 제공하는 어떠한 기능도 동작하지 않음
- 식별자 값을 가지고 있음

□ 삭제(Removed) 상태

- Persistent Context 와 DB에서 해당 Entity를 삭제한 상태
- 삭제된 객체는 Persistent Context에 존재하지 않음

`em.detach(member);` // detach하면 특정 Entity를 영속 -> 준영속 상태

`em.clear();` // clear하면 관리되고 있던 Entity가 준영속 상태가 됨

`em.close();` // close해도 관리되던 Entity는 준영속 상태가 됨

`em.merge(member);` // detach된 Entity를 merge하면 준영속 -> 영속 상태

`em.remove(member);` // 영속성 컨텍스트와 DB에서 삭제 상태

Persistent Context 특징

□ Persistent Context 특징

- Persistent Context는 Entity를 식별자 값(@Id로 테이블의 기본 키와 매핑한 값)으로 구분함
- 영속 상태는 반드시 식별자 값이 있어야 함 (없으면 예외 발생)
- Persistent Context에 저장된 Entity 는 Transaction을 Commit하는 순간, Persistent Context에 새로 저장된 Entity를 DB에 반영함

Persistent Context 사용 이점

□ Persistent Context 사용 이점

- 1차 캐시 – 1차 캐시(Persistent Context 내부 캐시)에 key=@Id: value=Entity 로 저장
- 동일성 보장 – DB에서 조회한 데이터를 기반으로 새로운 Entity를 생성하는 것이 아닌 1차 캐시에서 삽입/조회해서 동일성 보장
- Transaction을 지원하는 쓰기 지연 – persist()를 호출했을 때, 바로 insert 쿼리를 보내는 것이 아니라, Persistent Context에 집어넣고 Transaction Commit 하기 직전까지 "쓰기 지연 SQL 저장소"에 insert 쿼리문을 쌓아둠
- 변경 감지 (Dirty checking) – Entity의 변경 사항을 DB에 자동으로 반영
- 지연 로딩(Lazy Loading) – 연관 관계 매핑되어 있는 Entity 조회시 Proxy를 반환해서 쿼리를 필요할 때 보내는 기능

Persistent Context 사용 예시

```
// EntityManagerFactory
EntityManagerFactory emf = Persistence.createEntityManagerFactory("basicjpa"); //persistent.xml
// EntityManager
EntityManager em = emf.createEntityManager();
// EntityTransaction 모든 DB 로직은 transaction 내부에서 실행되어야 함
EntityTransaction tx = em.getTransaction();
try {
    tx.begin();
    // Entity
    Person entity = new Person("Kang", 20, 60.0, 176.4, Gender.FEMALE);
    // Persistent Object
    em.persist(entity);
    tx.commit(); // commit 이후 DB에 적용
} catch (Exception e) {
    e.printStackTrace();
    tx.rollback(); // 에러 발생시 rollback
} finally {
    em.close();
}
emf.close();
```

H2 Database

□ H2 Database

- 자바 기반의 오픈소스 관계형 데이터베이스 관리 시스템 (RDBMS)으로 테스트 단계 또는 작은 규모의 프로젝트에서 사용됨

□ 3가지 모드를 지원

■ In-Memory Mode

- 애플리케이션(WAS) 구동 시 H2 DB 데이터를 메모리에 올려서 관리하는 방식으로 애플리케이션이 종료되면 메모리에 올라가 있던 모든 데이터는 사라짐. 즉, 휘발성이기 때문에 간단한 테스트에 사용하기 좋음

■ Embedded Mode

- 애플리케이션(WAS) 구동 시 H2 DB 데이터를 PC에 저장해서 관리하는 방식으로 In-Memory와는 달리 데이터가 사라지지 않는 비휘발성 모드임

■ Server Mode

H2 Database

- ▣ Gradle 또는 Maven에 의존성만 추가해 주면 쉽고 빠르게 H2 DB를 이용할 수 있음

```
<!-- Spring Data JPA -->
```

```
<dependency>
```

```
    <groupId>org.springframework.boot</groupId>
```

```
    <artifactId>spring-boot-starter-data-jpa</artifactId>
```

```
</dependency>
```

```
<!-- H2 in-memory database -->
```

```
<dependency>
```

```
    <groupId>com.h2database</groupId>
```

```
    <artifactId>h2</artifactId>
```

```
</dependency>
```

Configuration

□ **application.properties** 에서 h2 database in-memory 설정

- 사용자 이름 sa 와 빈(empty) password

spring.datasource.url=jdbc:h2:mem:test

spring.datasource.driverClassName=org.h2.Driver

#spring.datasource.username=sa

#spring.datasource.password=password

□ JPA 설정

spring.jpa.database-platform=org.hibernate.dialect.H2Dialect

spring.jpa.hibernate.ddl-auto=create

spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.H2Dialect

spring.jpa.properties.hibernate.format-sql=true

spring.jpa.show-sql=true

□ H2 Console 설정

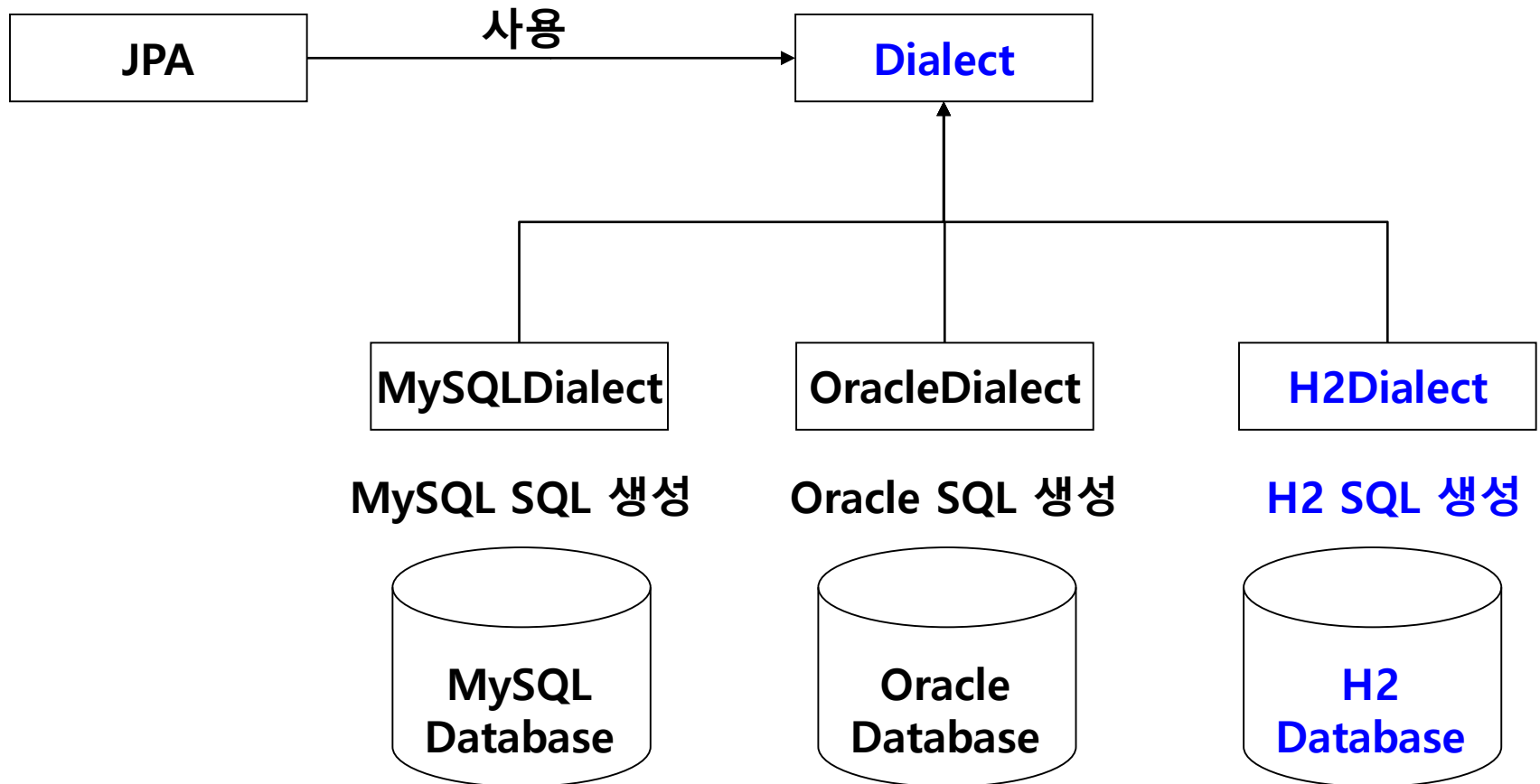
spring.h2.console.enabled=true

spring.h2.console.path=/h2-console

Configuration

spring: application.yml
datasource:
 driver-class-name: org.h2.Driver
 url: 'jdbc:h2:mem:test' # H2 DB 연결 주소 (In-Memory Mode)
h2:
 console: # H2 DB를 웹에서 관리할 수 있는 기능
 enabled: true # H2 Console 사용 (http://localhost:8080/h2-console)
 path: /h2-console # H2 Console 접속 주소
jpa:
 database-platform: org.hibernate.dialect.H2Dialect
 hibernate:
 ddl-auto: create # DB 초기화 전략 (none, create, create-drop, update, validate)
properties:
 hibernate:
 dialect: org.hibernate.dialect.H2Dialect
 format-sql: true # 쿼리 로그 포맷 (정렬)
 show-sql: true # 실행된 쿼리 로그 출력

org.hibernate.dialect.H2Dialect



Hibernate.ddl-auto

□ Hibernate의 ddl-auto 설정

- Entity 만 등록해놓으면 DDL(Data Definition Language)을 자동으로 작성하여 Table 생성 또는 수정

- **ddl-auto** 속성의 종류

- **create** - create는 Entity로 등록된 클래스와 매핑되는 테이블을 자동으로 생성(create). 이 과정에서 기존에 해당 클래스와 매핑되는 테이블이 존재한다면 기존 테이블을 삭제(drop)하고 테이블을 생성
- **create-drop** - create와 비슷하게 Entity 로 등록된 클래스와 매핑되는 테이블이 존재한다면 기존 테이블을 삭제하고 자동으로 테이블을 생성해주는 것은 같지만, 애플리케이션이 종료될 때 테이블을 삭제
- **update** - Entity로 등록된 클래스와 매핑되는 테이블이 없으면 새로 생성하는 것은 create와 동일하지만 기존 테이블이 존재한다면 위의 두 경우와 달리 테이블의 컬럼을 변경
- **validate** - DDL을 작성하여 테이블을 생성하거나 수정하지 않고, Entity 클래스와 테이블이 정상적으로 매핑되는지 검사. 만약 테이블이 아예 존재하지 않거나, 테이블에 Entity 필드에 매핑되는 컬럼이 존재하지 않으면 예외를 발생시키면서 애플리케이션을 종료
- **none** (default)

Entity

@Entity

@Getter

@NoArgsConstructor // for JPA only

@AllArgsConstructor

@Table(name="PERSON")

```
public class Person {
```

```
    @Id // Primary Key
```

```
    @GeneratedValue(strategy=GenerationType.IDENTITY)
```

```
    @Column(name="ID")
```

```
    private Long id;
```

```
    @Column(name="NAME")
```

```
    private String name;
```

```
    @Column(name="AGE")
```

```
    private int age;
```

```
    ...
```

```
}
```

Entity

□ <http://localhost:8080/h2-console/>

The screenshot displays the H2 Console web interface. The left sidebar contains a 'Login' section with the following fields:

- Saved Settings: Generic H2 (Embedded)
- Setting Name: Generic H2 (Embedded) [Save] [Remove]
- Driver Class: org.h2.Driver
- JDBC URL: jdbc:h2:mem:test
- User Name: sa
- Password: [Empty field]
- [Connect] [Test Connection]

The main console area on the right shows the database structure for 'jdbc:h2:mem:test':

- PERSON
 - AGE
 - GENDER
 - HEIGHT
 - WEIGHT
 - ID
 - NAME
- Indexes
- INFORMATION_SCHEMA
- Users

At the bottom of the console area, it indicates 'H2 2.2.224 (2023-09-17)'. The top right of the console area features a toolbar with buttons: Run, Run Selected, Auto complete, Clear, and SQL state. Below the toolbar, the SQL query 'SELECT * FROM PERSON' is entered in the text area.

Entity

□ Entity 클래스

- **@Entity [Class]** – 해당 클래스가 JPA Entity 클래스라고 정의
- **@Table [Class]** – 해당 클래스가 DB의 어느 테이블에 매핑되는지 정의
- **@Id** – DB 테이블의 Primary Key Column과 매핑
- **@GeneratedValue** – Primary Key 생성 규칙을 나타냄. 스프링부트 2.0에서는 GenerationType.IDENTITY 옵션을 추가해야 auto_increment 가 됨
- **@Column** – 매핑할 DB의 Column 이름과 필드 변수의 이름이 다를 경우 매핑하기 위해 사용. 테이블의 칼럼을 나타내며 굳이 선언하지 않더라도 해당 클래스의 필드는 모두 칼럼이 됨. 문자열의 경우 VARCHAR(255) 가 기본값인데, 사이즈를 500 으로 늘리고 싶거나 타입을 TEXT로 변경하고 싶거나 등의 경우에 사용함.

JpaRepository

```
public interface PeopleRepository extends JpaRepository<Person, Long> {  
    // SELECT p FROM Person p WHERE p.name = ?1  
    Optional<Person> findByName(String name);  
    // Custom Query  
    // @Query("SELECT p FROM Person p WHERE p.age > :age")  
    List<Person> findByAgeGreaterThan(@Param("age") int age);  
    // Custom Query  
    // @Query("SELECT p FROM Person p WHERE p.weight > ?1 and p.weight <  
    ?2")  
    List<Person> findByWeightBetween(double min, double max);  
    // Custom Query  
    // @Query("SELECT p FROM Person p WHERE p.height > :min and p.height  
    < :max")  
    List<Person> findByHeightBetween(double min, double max);  
}
```

JpaRepository

□ JpaRepository

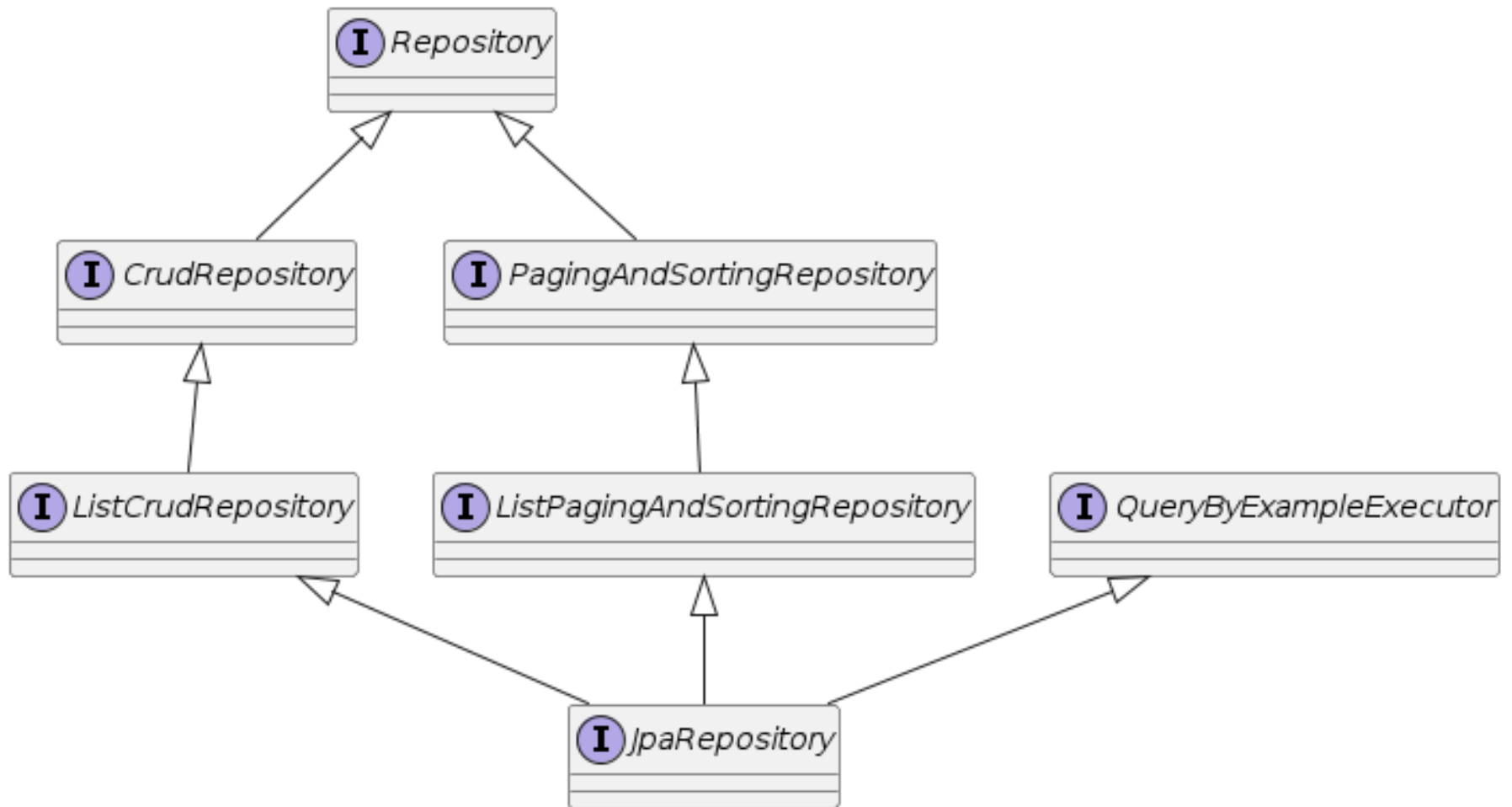
- Entity 클래스로 DB에 접근하게 해줌.
- **JPA 에선 Repository 라고 부르며 인터페이스로 생성됨.** 반면 ibatis 나 MyBatis 등에서 DAO(Data Access Object) 라고 불림.
- **JpaRepository<Entity, PK Type>** 을 상속하면 이미 만들어진 save(), delete() 등 **기본적인 CRUD 메소드가 자동으로 생성됨.**
- Spring Data JPA 프레임워크는 JpaRepository 에 정의된 메서드들을 내부에서 미리 구현했으며, 구현 클래스는 **SimpleJpaRepository** 임.
- 이 SimpleJpaRepository 클래스는 Hibernate 프레임워크에서 제공하는 클래스와 메서드를 사용하여 CRUD 메서드들을 제공함.

JpaRepository

□ JpaRepository 주요 메서드

- save() - 새로운 Entity는 저장하고 이미 있는 Entity는 수정함.
- delete() - Entity 하나를 삭제. 내부에서 EntityManager.remove()를 호출함.
- findOne() - Entity 하나를 조회. 내부에서 EntityManager.find()를 호출함.
- getOne() - Entity를 Proxy로 조회. 내부에서 EntityManager.getReference()를 호출함.
- findAll() - 모든 Entity를 조회. 정렬이나 페이징 조건을 파라미터로 전달할 수 있음.
- findBy() - Entity 하나를 조회. Optional 타입의 객체를 반환함.

JpaRepository



JpaRepository

@NoRepositoryBean

```
public interface JpaRepository<T, ID> extends ListCrudRepository<T, ID>,
ListPagingAndSortingRepository<T, ID>, QueryByExampleExecutor<T> { ...
}
```

@NoRepositoryBean

```
public interface ListCrudRepository<T, ID> extends CrudRepository<T, ID> {...
}
```

@NoRepositoryBean

```
public interface CrudRepository<T, ID> extends Repository<T, ID> {...
}
```

@NoRepositoryBean

```
public interface ListPagingAndSortingRepository<T, ID> extends
PagingAndSortingRepository<T, ID> {...
}
```

@NoRepositoryBean

```
public interface PagingAndSortingRepository<T, ID> extends Repository<T,
ID> {...
}
```

<https://docs.spring.io/spring->

[data/jpa/docs/current/api/org/springframework/data/jpa/repository/JpaRepository.html](https://docs.spring.io/spring-)

JpaRepository

- Repository 인터페이스
 - Repository 인터페이스는 실제 구현이 없음
- CrudRepository 인터페이스
 - 기본적인 CRUD(Create Read Update Delete) 메소드 제공
 - save(), findAll(), findById(), existsById(), count(), deleteById(), delete(), deleteAll(), saveAll()
- PagingAndSortingRepository 인터페이스
 - Paging과 Sorting 추상화를 사용하여 Entity를 검색하는 메서드 제공
 - findAll(Pageable), findAll(Sort)
- QueryByExampleExecutor
 - Example 쿼리를 수행할 수 있는 더 다양한 CRUD 메소드 제공
 - findOne(), findAll(), findBy(), count(), exists()

SimpleJpaRepository

□ SimpleJpaRepository

- SimpleJpaRepository은 Spring Data JPA에서 제공하는 것으로 CrudRepository interface의 기본적인 구현체
- 우리가 프로젝트를 구현하면서 생성한 Repository Interface에 JpaRepository를 상속받으면 사용할 수 있던 기본적인 findAll()이나 findById()같은 메소드들이 구현된 클래스
- <https://docs.spring.io/spring-data/data-jpa/docs/current/api/org/springframework/data/jpa/repository/support/SimpleJpaRepository.html>

SimpleJpaRepository

□ SimpleJpaRepository

- JPA의 Entity Manager는 트랜잭션이 있어야 동작
- Spring Data JPA는 Persistent Context를 사용하기 위해 모든 메소드에 기본적으로 트랜잭션을 생성

@Repository

@Transactional(readOnly = true) // Read를 위한 변경 감지는 트랜잭션
성능낭비라 readOnly 속성을 사용함 flush() 생략

```
public class SimpleJpaRepository<T, ID> implements  
JpaRepositoryImplementation<T, ID> {
```

```
    @Override
```

```
    public List<T> findAll() {  
        return getQuery(null, Sort.unsorted()).getResultList();  
    }
```

```
    // 다양한 CRUD 메서드 ...
```

```
}
```

SimpleJpaRepository

□ SimpleJpaRepository 의 save() 메소드

- entityInformation에서 새로운 Entity이면 persist(), 아니면 merge()를 호출함
- merge는 한번 persist 상태였다가 detached 된 상태에서 그 다음 persist 상태가 될 때, merge 됨

@Transactional // Write에 대한 메소드들은 @Transactional을 명시
@Override

```
public <S extends T> S save(S entity) {  
    if (this.entityInformation.isNew(entity)) {  
        this.em.persist(entity);  
        return entity; // insert  
    } else {  
        return this.em.merge(entity); // update  
    }  
}
```

SimpleJpaRepository

- SimpleJpaRepository 의 delete() 메소드
 - Entity가 null인지 확인하고 EntityManager를 통해 삭제

@Transactional // Write에 대한 메소드들은 @Transactional을 명시
@Override

```
public void delete(T entity) {  
    if (this.entityInformation.isNew(entity)) {  
        return entity;  
    }  
    Class<?> type = ProxyUtils.getUserClass(entity);  
    T existing = (T) em.find(type, entityInformation.getId(entity));  
    if (existing == null) {  
        return;  
    }  
    em.remove(em.contains(entity) ? entity : em.merge(entity));  
}
```

@Transactional

□ @Transactional [Class|Method]

- 스프링 프레임워크는 @Transactional 붙어 있는 클래스나 메소드에 Transaction 처리를 위해 선언적 트랜잭션을 사용함.
- 클래스에 선언하게 되면, 해당 클래스에 속하는 모든 메서드에 공통적으로 적용됨.
- 메서드에 선언하게 되면, 해당 메소드에만 적용됨.
- 외부에서 이 클래스의 메소드를 호출할 때 Transaction을 begin하고 메소드를 종료할 때 Transaction을 commit한다. 만약 예외가 발생하면 Transaction을 rollback함.

메소드 이름으로 Query 생성

- 메소드 이름으로 쿼리를 생성하는 기능
 - Spring Data JPA가 제공하는 기능
 - 메소드 이름만으로 쿼리를 생성하는 기능이 있는데 인터페이스에 메소드만 선언해주면 해당 메소드 이름으로의 적절한 JPQL(Java Persistence Query Language) 쿼리를 생성.
 - 메소드 이름으로 JPA **NamedQuery** 호출

```
// SELECT m FROM Member m WHERE m.name = :name  
Optional<Member> findByName(String name);
```

메소드 이름으로 Query 생성

Keyword	Sample	JPQL
And	findByLastnameAndFirstname	... where x.lastname = ?1 and x.firstname = ?2
Is, Equals	findByName,findByNames,findByNameEquals	... where x.name = ?1
Between	findByStartDateBetween	... where x.startDate between ?1 and ?2
GreaterThan	findByAgeGreaterThan	... where x.age > ?1
After	findByStartDateAfter	... where x.startDate > ?1
IsNull, Null	findByAge(Is)Null	... where x.age is null
StartingWith	findByNameStartingWith	... where x.name like ?1
OrderBy	findByAgeOrderByNameDesc	... where x.age = ?1 order by x.name desc
True	findByActiveTrue()	... where x.active = true

@Query 작성

- 메소드에 Query 를 직접 써서 만드는 방법
 - JPA에서 메서드 Parameter를 통해서 @Query에 그 parameter가 들어지는 방식
 - 물음표(?) 경우엔 parameter의 위치에 따른 숫자를 넣어주면 됨
?1 -> parameter 첫번째 자리에 있는 것을 넣겠다는 뜻
 - 콜론(:) 경우엔 :name -> 파라미터의 이름으로 검색하는 경우

// Custom Query

```
@Query("SELECT m FROM Member m WHERE m.weight > ?1 and  
m.weight < ?2")
```

```
List<Member> findByWeightBetween(double min, double max);
```

```
@Query("SELECT m FROM Member m WHERE m.height > :min and  
m.height < :max")
```

```
List<Member> findByHeightBetween(double min, double max);
```

```
@Query("SELECT m FROM Member m WHERE m.age > :age")
```

```
List<Member> findByAgeGreaterThan(@Param("age") int age);
```

JPQL(Java Persistence Query Language)

□ JPQL(Java Persistence Query Language)

- JPQL은 테이블이 아닌 Entity 객체를 조회하는 객체지향 Query
- SQL과 비슷한 문법을 가지며, JPQL은 결국 SQL로 변환됨
- SQL을 추상화 했기 때문에 특정 벤더에 종속적이지 않음
- JPA는 JPQL을 분석하여 SQL을 생성한 후 DB에서 조회
- JPA에서 제공하는 메소드 호출 만으로 복잡한 Query 작성이 어렵다는 문제에서 JPQL 탄생
- 기본문법은 대소문자 구분함
 - Entity의 이름인 Member는 대문자 @Entity(name="Member")로 설정 가능
 - Entity의 속성 name은 소문자
 - 반면 SELECT, FROM, AS 키워드는 대소문자 구분하지 않음

```
String jpql = "select m from Member m where m.name = :name";
```

TypedQuery vs Query

- Query 객체로 TypedQuery와 Query가 있음
 - EntityManager 객체에서 `createQuery()` 메소드를 호출하면 쿼리가 생성
 - 반환할 타입을 명확하게 지정할 수 있으면 TypedQuery 객체를 사용

```
String jpql = "select m from Member m";
```

```
TypedQuery<Member> query = em.createQuery(jpql, Member.class);
```

```
List<Member> list = query.getResultList();
```

- 반환할 타입을 명확하게 지정할 수 없으면 Query 객체를 사용

```
String jpql = "select m.name, m.age from Member m";
```

```
Query query = em.createQuery(jpql);
```

```
List<Object> list = query.getResultList();
```

Parameter Binding

- 이름 기준 Parameter Binding은 콜론(:)을 사용

```
public Optional<Member> findByName(String name) {  
    String jpql = "select m from Member m where m.name = :name";  
    List<Member> result = em.createQuery(jpql, Member.class)  
        .setParameter("name", name)  
        .getResultList();  
    return result.stream().findAny();  
}
```

- 위치 기준 Parameter Binding은 물음표(?)를 사용

```
public Optional<Member> findByName(String name) {  
    String jpql = "select m from Member m where m.name = ?1";  
    List<Member> result = em.createQuery(jpql, Member.class)  
        .setParameter(1, name)  
        .getResultList();  
    return result.stream().findAny();  
}
```

Service

@Service

```
public class PeopleService {  
    @Autowired  
    private PeopleRepository repository;  
    public List<Person> findAll() {  
        return repository.findAll();  
    }  
    public Optional<Person> findById(Long id) {  
        return repository.findById(id);  
    }  
    public Optional<Person> findByName(String name) {  
        return repository.findByName(name);  
    }  
    public Person save(Person person) {  
        return repository.save(person);  
    } ...  
}
```

Controller

@RestController

@RequestMapping("/people")

public class PeopleController {

 @Autowired

 private PeopleService service;

 @GetMapping("/list")

 public List<Person> getAll() {

 return service.findAll();

 }

 @GetMapping("/name/{name}")

 public Person getPerson(@PathVariable("name") String name) {

 return service.findByName(name).orElse(null);

 }

 @GetMapping("/id/{id}")

 public Person getPerson(@PathVariable("id") long id) {

 return service.findById(id).orElse(null);

 } ...

Controller

```
...  
@ResponseStatus(HttpStatus.CREATED) // 201  
@PostMapping("/new") // create a person (POST)  
public Person create(@RequestBody Person person) {  
    return service.save(person);  
}  
@PutMapping("/update") // update a person (PUT)  
public Person update(@RequestBody Person person) {  
    return service.save(person);  
}  
@ResponseStatus(HttpStatus.NO_CONTENT) // 204  
@DeleteMapping("/{id}") // delete a person (DELETE)  
public void deleteById(@PathVariable long id) {  
    service.deleteById(id);  
}  
}
```

resource에 schema.sql & data.sql 사용

- 스프링부트 resource에 sql 파일 생성
 - application configuration 수정

spring.jpa.hibernate.ddl-auto=none # schema.sql 사용시 none으로
수정

spring.jpa.defer-datasource-initialization=true # data.sql 사용시 추가
(Spring Boot 2.5 이상 버전)

spring.sql.init.mode=always # data.sql 사용시 추가 always 모든
데이터베이스에 sql 스크립트를 동작

schema.sql

```
drop table if exists PERSON;  
create table PERSON (  
    ID bigint generated by default as identity,  
    NAME varchar(255) not null,  
    AGE integer not null,  
    WEIGHT float(53) not null,  
    HEIGHT float(53) not null,  
    GENDER tinyint not null check (gender between 0 and 1),  
    primary key (ID)  
);
```

data.sql

```
INSERT INTO PERSON(NAME,AGE,WEIGHT,HEIGHT,GENDER)
VALUES ('Dooly',1000,40.0,126.4,1),
       ('Douner',1986,60.0,150.0,0),
       ('Ddochi',30,55.0,165.0,1),
       ('Heedong',3,36.00,100.6,1),
       ('Michol',25,71.8,176.4,1),
       ('Gildong',40,68.2,169.4,1),
       ('Younghi',10,47.2,152.4,0),
       ('Cheolsoo',10,48.2,155.4,1);
```

Self-configured Tests

- spring-boot-starter-test 는 jsonpath, assertj, hamcrest, junit, mockito 를 모두 포함함
- pom.xml에 spring-boot-starter-test 의존성(dependency) 추가

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
    </dependency>
</dependencies>
```

Self-configured Tests

- **@SpringBootTest**는 통합(integration) 테스트를 제공하는 기본적인 스프링 부트 테스트 애노테이션
 - @SpringBootTest는 스프링부트 애플리케이션 테스트에 필요한 모든 빈을 등록하여 테스트 진행 (큰 규모의 통합 테스트)
 - 실제 운영 환경에서 사용될 클래스들을 통합하여 테스트
 - 단위 테스트와 같이 기능 검증을 위한 것이 아닌 스프링 프레임워크에서 전체적으로 플로우가 제대로 동작하는지 검증하기 위해 사용됨
 - **@SpringBootTest** 사용시 JUnit 버전에 따라 유의사항이 존재함
 - JUnit4 : @RunWith(SpringRunner.class)와 함께 사용
 - JUnit5 : 따로 명시할 필요 없음

Self-configured Tests

□ Mock

- 테스트 시 실제 구현 대신 사용되는 가짜 객체로 Mock 객체는 원하는 방식으로 동작하도록 프로그래밍되어 있으며, 실제 의존성을 가지지 않고도 테스트를 수행할 수 있게 해줌
- WebApplicationContext를 불러오며 내장된 서블릿 컨테이너가 아닌 Mock 서블릿을 제공함
- @SpringBootTest 의 webEnvironment 옵션의 기본값
 - **@SpringBootTest(webEnvironment=SpringBootTest.WebEnvironment.MOCK)**
- @AutoConfigureMockMvc 을 함께 사용하여 MockMvc를 사용한 테스트 진행 가능함
 - **MockMvc**는 HTTP 요청과 응답을 의미하는 객체로, 서블릿 컨테이너를 실행하지 않고도 Controller 테스트를 용이하게 해줌
 - @AutoConfigureMockMvc 은 Mock 테스트 시 필요한 의존성을 제공함

@Autowired

MockMvc mockMvc;

Unit Tests

□ JUnit 단위 테스트

- 자바에서 독립된 단위테스트(Unit Test)를 지원하는 프레임워크
- asset (검증)을 이용해서 테스트 케이스 수행결과를 판별
- JUnit2 이후부터 `@Test` `@Before` `@After` 등 제공
- `@Test` 호출할 때마다 새로운 인스턴스를 생성하며 독립적인 테스트가 이루어짐

Unit Tests

□ Unit Tests

■ @JsonTest

- @JsonTest 을 사용하면 JSON 직렬화를 테스트하는데 필요한 스프링 빈만을 사용하여 Spring Test Context를 자동으로 구성할 수 있음

■ @WebMvcTest

- 웹 상에서 요청과 응답에 대한 테스트
- @WebMvcTest은 @Controller, @ControllerAdvice, @JsonComponent, @JsonFilter, WebMvcConfigure, HandlerMethodArgumentResolver만 로드되기 때문에 전체 테스트보다 가벼움

■ @WebFluxTest

- Spring WebFlux 컨트롤러가 예상대로 작동하는지 테스트

■ @DataJpaTest

- @DataJpaTest를 이용한 데이터베이스 데이터 저장 테스트

■ @RestClientTest

- Rest 통신의 JSON 형식이 예상대로 응답을 반환하는지 등을 테스트함

@WebMvcTest

```
@WebMvcTest(SpringBootTestingController.class)
class SpringBootTestingApplicationTests {
    @Autowired
    private MockMvc mockMvc;
    @Test
    public void testHello() throws Exception {
        mockMvc.perform(get("/hello"))
            .andExpect(status().isOk())
            .andExpect(content().string("Hello AJ!!"));
    }
}

@RestController
public class SpringBootTestingController {
    @GetMapping("/hello")
    public String hello() {
        return "Hello AJ!!";
    }
}
```

@DataJpaTest

@DataJpaTest

```
class SpringBootTestingApplicationTests {  
    @Autowired  
    private PeopleRepository peopleRepository;  
    @Test  
    public void testSave() throws Exception {  
        Person ace = new Person("Ace", 24, 70.0, 180.0, Gender.MALE);  
        peopleRepository.save(ace);  
        long savedID = ace.getId();  
        Person person = peopleRepository.findById(savedID).orElseThrow();  
        assertEquals(savedID, person.getId());  
        assertEquals("Ace", person.getName());  
        assertEquals(24, person.getAge());  
        assertEquals(70.0, person.getWeight());  
        assertEquals(180.0, person.getHeight());  
        assertEquals(Gender.MALE, person.getGender());  
    }  
}
```

@SpringBootTest

```
@SpringBootTest(webEnvironment=SpringBootTest.WebEnvironment.RANDOM_PORT)
```

```
class SpringBootTestApplicationTests {  
    @LocalServerPort  
    int randomServerPort; // need for POST, PUT, DELETE  
    @Autowired  
    private TestRestTemplate restTemplate;  
    @Autowired  
    private PeopleRepository peopleRepository;  
    @Test  
    public void deleteByIdTest() {  
        Person person = peopleRepository.findById(9L).orElseThrow();  
        Long id = person.getId();  
        // test DELETE by id  
        ResponseEntity<Void> response = restTemplate.exchange(  
            "http://localhost:" + randomServerPort + "/people/" + id,  
            HttpMethod.DELETE, null, Void.class);  
        // test 204  
        assertEquals(HttpStatus.NO_CONTENT, response.getStatusCode());  
    }  
}
```

...

@SpringBootTest

```
@Test
public void createTest() {
    Person dis = new Person("Dis", 20, 70.0, 180.0, Gender.MALE);
    HttpHeaders headers = new HttpHeaders();
    headers.add("Content-Type", "application/json");
    HttpEntity<Person> entity = new HttpEntity<>(dis, headers);
    // test POST
    ResponseEntity<Person> responseEntity =
restTemplate.postForEntity("http://localhost:" + randomServerPort +
"/people/new", entity, Person.class);
    assertEquals(HttpStatus.CREATED, responseEntity.getStatusCode());
    // find Person dis
    Person person = peopleRepository.findByName("Dis").orElseThrow();
    assertEquals("Dis", person.getName());
    assertEquals(20, person.getAge());
    ...
}
```