

Spring JPA

Test

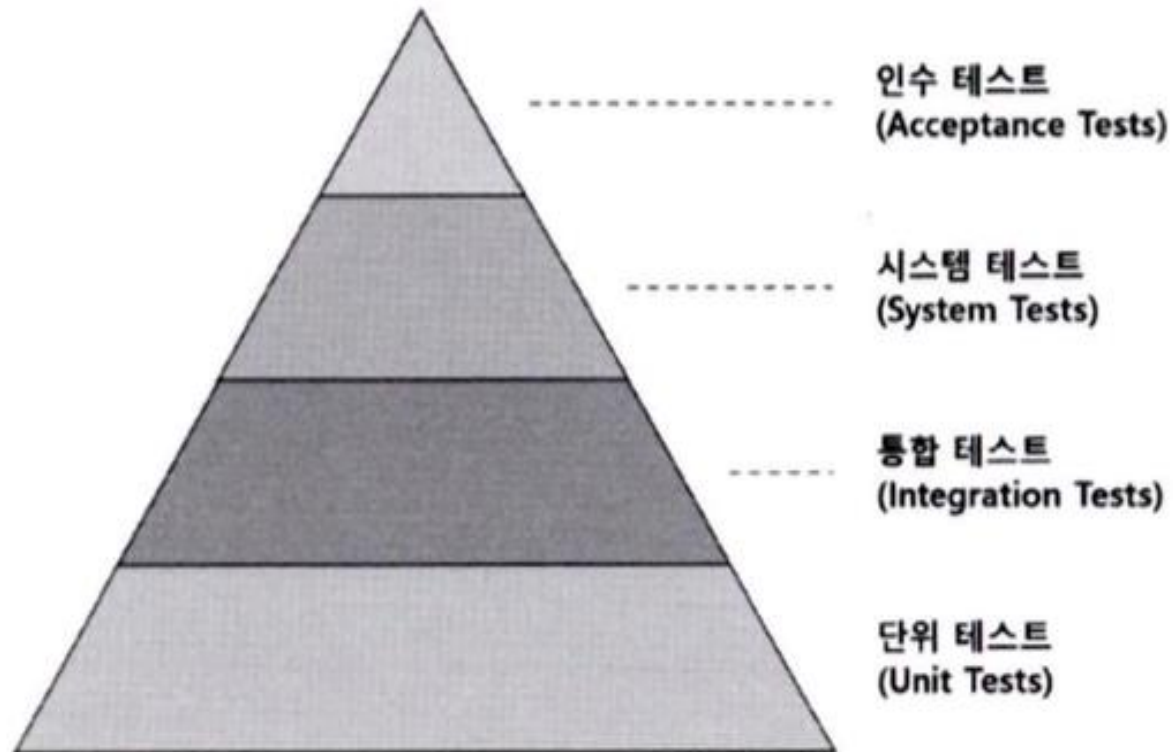
❖ Test 코드 작성 이유

- ✓ 개발 과정에서 문제를 미리 발견할 수 있음
- ✓ 리팩토링의 리스크가 줄어듦
- ✓ 애플리케이션을 가동해서 직접 테스트하는 것보다 테스트를 빠르게 진행할 수 있음
- ✓ 하나의 명세 문서로서의 기능을 수행
- ✓ 프레임워크에 맞춰 테스트 코드를 작성하면 좋은 코드를 생산할 수 있음
- ✓ 코드가 작성된 목적을 명확하게 표현할 수 있으며 불필요한 내용이 추가되는 것을 방지할 수 있음



Test

❖ 종류



Test

❖ 종류

✓ Unit Test

- 테스트 대상의 범위를 기준으로 가장 작은 단위의 테스트 방식
- 일반적으로 메서드 단위로 테스트를 수행하게 되며 메서드 호출을 통해 의도한 결과가 나오는지 확인하는 수준으로 테스트를 진행
- 테스트 비용이 적게 들기 때문에 테스트 피드백을 빠르게 받을 수 있음

✓ Integration Test

- 통합 테스트는 모듈을 통합하는 과정에서의 호환성 등을 포함해 애플리케이션이 정상적으로 동작하는지 확인하기 위해 수행하는 테스트
- 단위 테스트는 모듈을 독립적으로 테스트하는 반면 통합 테스트는 여러 모듈을 함께 테스트해서 정상적인 로직 수행이 가능한지를 확인
- 단위 테스트는 일반적으로 특정 모듈에 대한 테스트만 진행하기 때문에 데이터베이스나 네트워크 같은 외부 요인들을 제외하고 진행하는 데 비해 통합 테스트는 외부 요인들을 포함하고 테스트를 진행하므로 애플리케이션이 온전히 동작하는지를 테스트
- 테스트를 수행할 때마다 모든 컴포넌트가 동작해야 하기 때문에 테스트 비용이 커지는 단점이 있음

Test

❖ 종류

✓ System Test

- 시스템 테스트는 정보시스템이 완전히 통합되어 구축된 상태에서 정보 시스템의 기능을 총체적으로 테스트하는 것
- 통합된 각 모듈들이 원래 계획했던 대로 작동하는지 시스템의 실제 동작과 원래 의도했던 요구 사항과는 차이가 없는지 등을 판단
- 수행 시간, 파일 저장 및 처리 능력, 최대 부하, 복구 및 재시동 능력, 수작업 절차 등을 점검
- 시스템 테스트는 시스템의 내부적인 구현 방식이나 설계에 대한 지식에 관계없이 테스트를 수행하는 블랙박스 테스트의 일종으로 분류
- 시스템 테스트는 모든 통합 테스트를 통과한 통합된 소프트웨어 컴포넌트들 그리고 필요한 하드웨어들과 통합된 소프트웨어 시스템 전체를 대상으로 수행
- 통합 테스트는 서로 통합된 두 소프트웨어 혹은 하드웨어 단위 사이에 서로 잘 맞지 않는 부분이 있는지를 파악하기 위해서 수행하고 시스템 테스트는 통합된 컴포넌트들 사이에 발생하는 결함과 통합된 전체 시스템내에서 발생하는 결함 모두를 발견하기 위해서 수행

Test

❖ 종류

✓ Acceptance Test

- 프로젝트에 참여하는 사람들(기획자, 클라이언트 대표, 개발자 등)이 토의해서 시나리오를 만들고 개발자는 이에 의거해서 코드를 작성
- 개발자가 직접 시나리오를 제작할 수도 있지만 다른 의사 소통 집단으로부터 시나리오를 받아(인수) 개발한다는 의미
- 인수 테스트는 애자일 개발 방법론에서 파생했는데 익스트림 프로그래밍(XP)에서 사용하는 용어로 시나리오가 정상적으로 동작하는지를 테스트하기 때문에 통합 테스트와는 분류가 다름
- 시나리오에서 요구하는 것은 누가, 어떤 목적으로, 무엇을 하는가 인데 개발을 하다 보면 이런 기능은 API를 통해 드러나며 인수 테스트는 주로 이 API를 확인하는 방식으로 이루어짐
- 인수 테스트는 소프트웨어 인수를 목적으로 하는 테스트이기 때문에 소프트웨어를 인수하기 전에 명세한 요구사항(인수 조건)대로 잘 작동하는지 검증이 필요
- 소프트웨어를 인수할 때, 소프트웨어 내부 구조나 구현 방법을 고려하기보다는 실제 사용자 관점에서 테스트하는 경우가 많기 때문에 인수 테스트는 소프트웨어 내부 코드에 관심을 가지지 않는 블랙박스 테스트
- Java에서는 RestAssured, MockMvc 같은 도구를 활용하여 인수 테스트를 작성할 수 있음

Test

❖ 테스트 코드 작성 방법

✓ Given-When-Then 패턴

- 단계를 설정해서 각 단계의 목적에 맞게 코드를 작성
- 단계
 - ◆ Given: 테스트를 수행하기 전에 테스트에 필요한 환경을 설정하는 단계로 테스트에 필요한 변수를 정의하거나 Mock 객체를 통해 특정 상황에 대한 행동을 정의
 - ◆ When: 테스트의 목적을 보여주는 단계로 실제 테스트 코드가 포함되며 테스트를 통한 결과를 가져오게 됨
 - ◆ Then: 테스트의 결과를 검증하는 단계로 When 단계에서 나온 결과를 검증하는 작업을 수행하며 결과가 아니더라도 테스트를 통해 나온 결과에서 검증해야 하는 부분이 있다면 이 단계에 포함
- Given-When-Then 패턴은 테스트 주도 개발에서 파생된 BDD(Behavior-Driven-Development - 행위 주도 개발)를 통해 탄생한 테스트 접근 방식
- 단위 테스트보다는 비교적 많은 환경을 포함해서 테스트하는 인수 테스트에서 사용하는 것에 적합하다고 알려져 있음

Test

❖ 테스트 코드 작성 방법

✓ 좋은 테스트를 위한 5가지 속성 - FIRST

- 빠르게(Fast): 테스트는 빠르게 수행되어야 함
- 고립된(Isolated), 독립적(Elated): 하나의 테스트 코드는 목적으로 여기는 하나의 대상에 대해서만 수행되어야 하는데 하나의 테스트가 다른 테스트 코드와 상호 작용하거나 관리할 수 없는 외부 소스를 사용하게 되면 외부 요인으로 인해 테스트가 수행되지 않을 수 있음
- 반복 가능한(Repeatable): 테스트는 어떤 환경에서도 반복 가능하도록 작성해야 하는데 이 의미는 앞의 Isolated 규칙과 비슷한 의미를 갖고 있는데 테스트는 개발 환경의 변화나 네트워크의 연결 여부와 상관없이 수행되어야 함
- 자가 검증(Self-Validating): 테스트는 그 자체만으로도 테스트의 검증이 완료되어야 하는데 테스트가 성공했는지 실패했는지 확인할 수 있는 코드를 함께 작성해서 결과와 기대치를 비교하는 작업을 코드가 아니라 개발자가 직접하고 있다면 좋지 못한 테스트 코드
- 적시에(Timely): 테스트 코드는 테스트하려는 애플리케이션 코드를 구현하기 전에 완성되어야 하는데 너무 늦게 작성된 테스트 코드는 정상적인 역할을 수행하기 어려울 수 있으며 테스트 코드로 인해 발견된 문제를 해결하기 위해 소모되는 개발 비용도 커지기 쉬운데 다만 이 개념은 테스트 주도 개발의 원칙을 따르는 테스트 작성 규칙으로 테스트 주도 개발을 기반으로 애플리케이션을 개발하는 것이 아니라면 이 규칙은 제외하고 진행하기도 함

Test

❖ spring-boot-starter-test

- ✓ Spring Boot는 테스트에 필요한 여러 도구를 쉽게 사용할 수 있도록 starter를 제공
- ✓ 빌드 파일에 추가하면 다음과 같은 테스트 라이브러리가 자동으로 추가됨
 - Spring Boot Test
 - JsonPath
 - JUnit5
 - AssertJ
 - Mckito
 - JSONassert
 - Spring Test

Test

❖ JUnit

✓ JUnit

- JUnit은 자바 언어에서 사용되는 대표적인 테스트 프레임워크로서 단위 테스트 뿐 만 아니라 통합 테스트를 할 수 있는 기능도 제공
- JUnit의 가장 큰 특징은 어노테이션 기반의 테스트 방식을 지원한다는 것으로 JUnit을 사용하면 몇 개의 어노테이션만으로 간편하게 테스트 코드를 작성할 수 있음
- JUnit을 활용하면 단정문(assert)을 통해 테스트 케이스의 기댓값이 정상적으로 도출됐는지 검토할 수 있다는 장점이 있음

✓ JUnit Life Cycle

- @Test: 테스트 코드를 포함한 메서드를 정의
- @BeforeAll: 테스트를 시작하기 전에 호출되는 메서드를 정의
- @BeforeEach: 각 테스트 메서드가 실행되기 전에 동작하는 메서드를 정의
- @AfterAll: 테스트를 종료하면서 호출되는 메서드를 정의
- @AfterEach: 각 테스트 메서드가 종료되면서 호출되는 메서드를 정의

Test

❖ 가짜 객체 만들기

- ✓ 클래스를 테스트 할 때 주입받아야 하는 객체가 존재하는 경우 실제 테스트할 객체가 아니므로 직접 생성해서 주입하는 것은 불필요

- @MockBean 이용

- @MockBean 클래스 변수명;

- mock 함수 이용

- @BeforeEach void setUp() {

- 변수명 = mock(클래스);

- }

Test

❖ Controller 테스트

✓ 내장 컨테이너를 이용해서 테스트

```
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT) // <1>
@AutoConfigureWebTestClient // <2>
public class LoadingWebSiteIntegrationTest {

    @Autowired WebTestClient client; // <3>

    @Test // <4>
    void test() {
        client.get().uri("/").exchange() //
                                                .expectStatus().isOk() //
                                                .expectHeader().contentType(TEXT_HTML)

//
                                                .expectBody(String.class) //
                                                .consumeWith(exchangeResult -> {

            assertThat(exchangeResult.getResponseBody()).contains("<a href=#{}/add");
            });
    }
}
```

Test

❖ Controller 테스트

✓ 내장 컨테이너를 이용해서 테스트

- @SpringBootTest는 Spring Boot가 실제 애플리케이션을 구동하는데 @SpringBootTest 이 붙은 클래스를 찾아서 내장 컨테이너를 실행하며 WebEnvironment.RANDOM_PORT는 테스트할 때 임의의 포트에 내장 컨테이너를 바인딩
- @AutoConfigureWebTestClient는 애플리케이션에 요청을 날리는 WebTestClient 인스턴스를 생성
- @Autowired를 사용해서 WebTestClient 인스턴스를 테스트 케이스에 주입
- 실제 테스트 메서드는 WebTestClient를 사용해서 홈 컨트롤러의 루트 경로를 호출하고 WebTestClient에는 단언 기능이 포함되어 있어서 이 경우 HTTP 응답 코드, Content-Type 헤더를 검증하고 자바 8의 Consumer를 사용해서 응답 본문(response body)에 값 검증을 수행

Spring JPA

❖ ORM(Object Relational Mapping)

- ✓ 객체 지향 패러다임을 관계형 데이터베이스에 보존하는 기술
- ✓ 객체와 관계형 데이터베이스의 테이블을 매핑해서 사용하는 방법
- ✓ 관계형 데이터베이스에서는 Table(Table)을 설계하는데 새로운 Table에는 칼럼을 정의하고 칼럼에 맞는 데이터 타입을 지정해서 데이터를 보관하는 틀을 만든다는 의미에서 Class와 상당히 유사
- ✓ 클래스는 데이터베이스의 테이블과 매핑하기 위해 만들어진 것이 아니기 때문에 RDB 테이블과 어쩔 수 없는 불일치가 존재하는데 ORM은 이 둘의 불일치와 제약사항을 해결하는 역할을 담당

Member
String id String pw String name

member
uid varchar2(50) upw varchar2(50) uname varchar2(100)

Spring JPA

❖ ORM(Object Relational Mapping)

- ✓ Class 와 Table이 유사하듯이 Instance 와 Row(Record, Tuple)도 유사하며 객체 지향에서는 Class를 이용해서 객체(Instance)를 생성해서 데이터를 보관하는데 관계형 데이터베이스에서는 Table에 Row를 이용해서 데이터를 저장하며 차이는 객체 라는 단어가 데이터 + 행위(메서드)라는 의미라면 Row는 데이터 만을 의미
- ✓ 관계(relation) 와 참조(reference) 라는 의미도 유사한데 관계형 데이터베이스는 Table 사이의 관계를 통해서 구조적인 데이터를 표현한다면 객체 지향에서는 참조를 통해서 어떤 객체가 다른 객체들과 어떤 관계를 맺고 있는지를 표현
- ✓ 객체 지향과 관계형 데이터베이스는 유사한 특징을 가지고 있어서 객체 지향을 자동으로 관계형 데이터베이스에 맞게 처리해 주는 기법에 대해서 아이디어를 내기 시작했고 그것이 ORM의 시작
- ✓ ORM은 완전히 새로운 패러다임을 주장하는 것이 아니라 객체 지향 과 관계형 사이의 변환 기법을 의미하는 것으로 특정 언어에 국한되는 개념이 아니고 관계형 패러다임을 가지고 있다면 데이터베이스의 종류를 구분하지 않음
- ✓ 대다수의 객체 지향 언어에는 ORM을 위한 여러 프레임워크들이 존재

Spring JPA

❖ ORM(Object Relational Mapping)

✓ 장점

- 특정 데이터베이스에 종속되지 않음
- 객체 지향적 프로그래밍
- 생산성 향상

✓ 단점

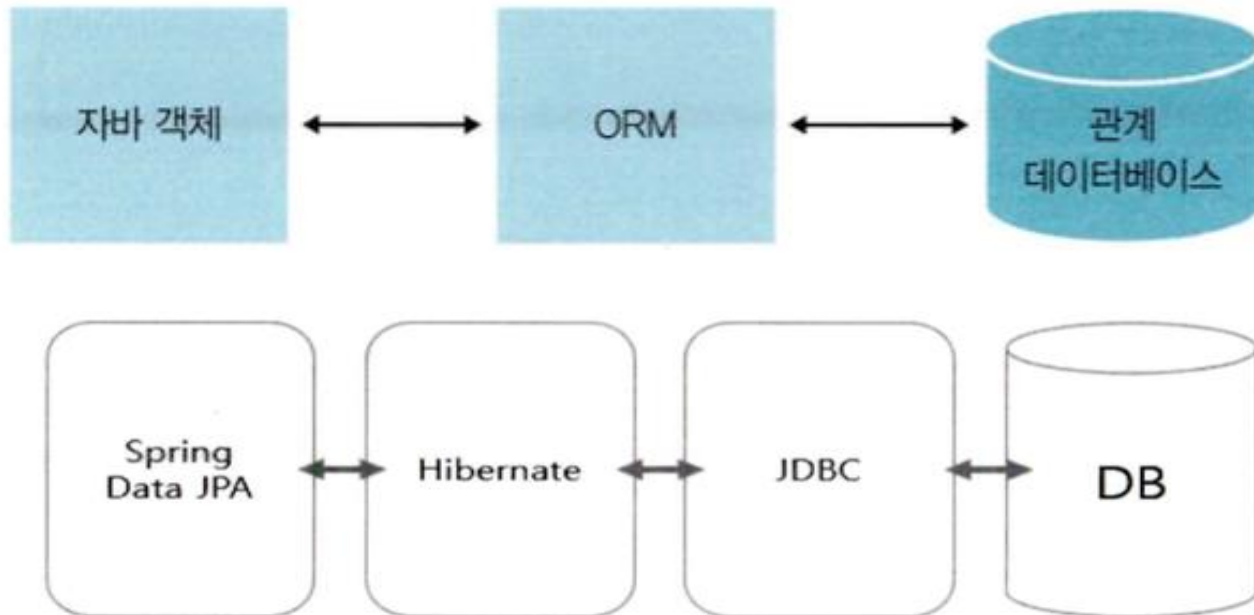
- 복잡한 쿼리 처리
- 성능 저하 위험
- 학습 시간



Spring JPA

❖ JPA(Java Persistence API)

- ✓ 자바 ORM 기술에 대한 API 표준
- ✓ JPA는 인터페이스이고 이를 구현한 대표적인 구현체로 Hibernate, EclipseLink, DataNucleus, OpenJpa, TopLink 등이 있음
- ✓ JPA 인터페이스를 구현한 가장 대표적인 오픈 소스가 Hibernate 이고 Spring Data JPA는 Hibernate를 쉽게 사용할 수 있는 추가적인 API를 제공



Context & Entity

❖ JPA(Java Persistence API)

✓ 구성 요소

● Entity Manager

- ◆ 영속성 컨텍스트에 접근하여 Entity에 대한 데이터베이스 작업을 제공
- ◆ 내부적으로 데이터베이스 커넥션을 사용해서 데이터베이스에 접근
- ◆ 메서드
 - find(): 영속성 컨텍스트에서 Entity를 검색하고 영속성 컨텍스트에 없을 경우 데이터베이스에서 데이터를 찾아 영속성 컨텍스트에 저장
 - persist(): Entity를 영속성 컨텍스트에 저장
 - remove(): Entity Class를 영속성 컨텍스트에서 삭제
 - flush(): 영속성 컨텍스트에 저장된 내용을 데이터베이스에 반영

● Persistence Context

- ◆ Entity를 영구 저장하는 환경으로 Entity Manager를 통해 영속성 컨텍스트에 접근

Context & Entity

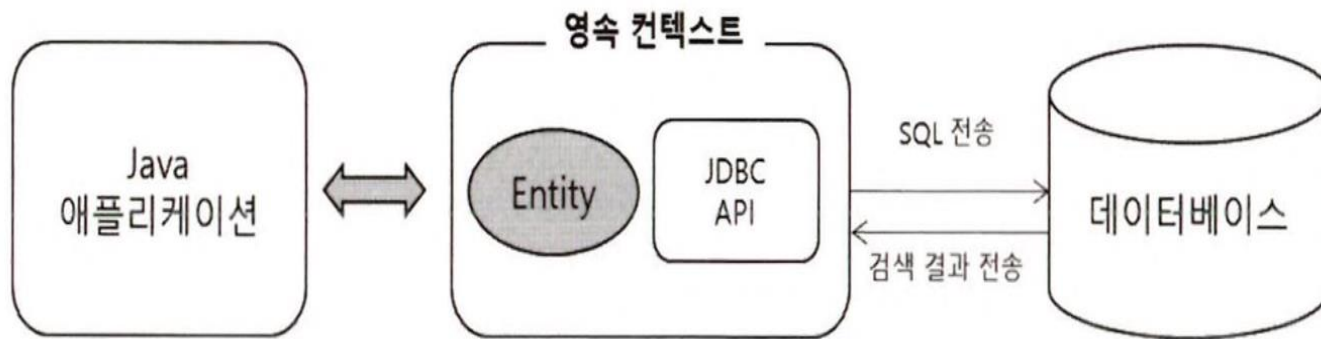
❖ JPA 와 영속 컨텍스트

✓ 동작 방식

- JPA의 EntityManager를 이용하면 비즈니스 메서드의 CRUD 기능은 자바 코드 몇 줄 만으로도 쉽게 구현할 수 있지만 EntityManager의 메서드가 호출될 때 JPA 내부적으로 Entity 객체와 데이터베이스의 테이블을 매핑하는 일련의 과정은 단순하지 않음
- 등록 기능을 구현하기 위해 EntityManager의 persist 메서드를 호출하면 EntityManager 와 연결된 Persistence Context 에 Entity 객체가 등록되고 Persistence Context는 등록된 Entity를 분석하여 INSERT 구문을 만들고 데이터베이스에 INSERT 구문을 전송
- JPA가 Entity를 기반으로 데이터를 처리하는 데 있어서 핵심 역할을 하는 것이 Persistence Context 이며 JPA의 Persistence Context 가 애플리케이션과 실제 데이터베이스를 연결해주는 버퍼 기능을 제공
- Persistence Context에 등록된 Entity는 다양한 상태로 존재할 수 있는데 Entity의 상태는 EntityManager를 통해 변경

Context & Entity

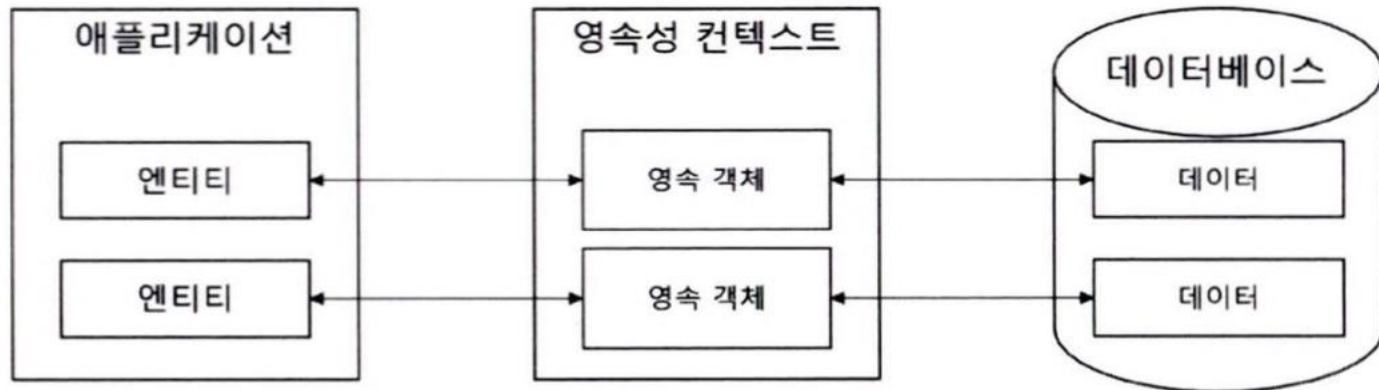
- ❖ JPA 와 영속 컨텍스트
 - ✓ 동작 방식



Context & Entity

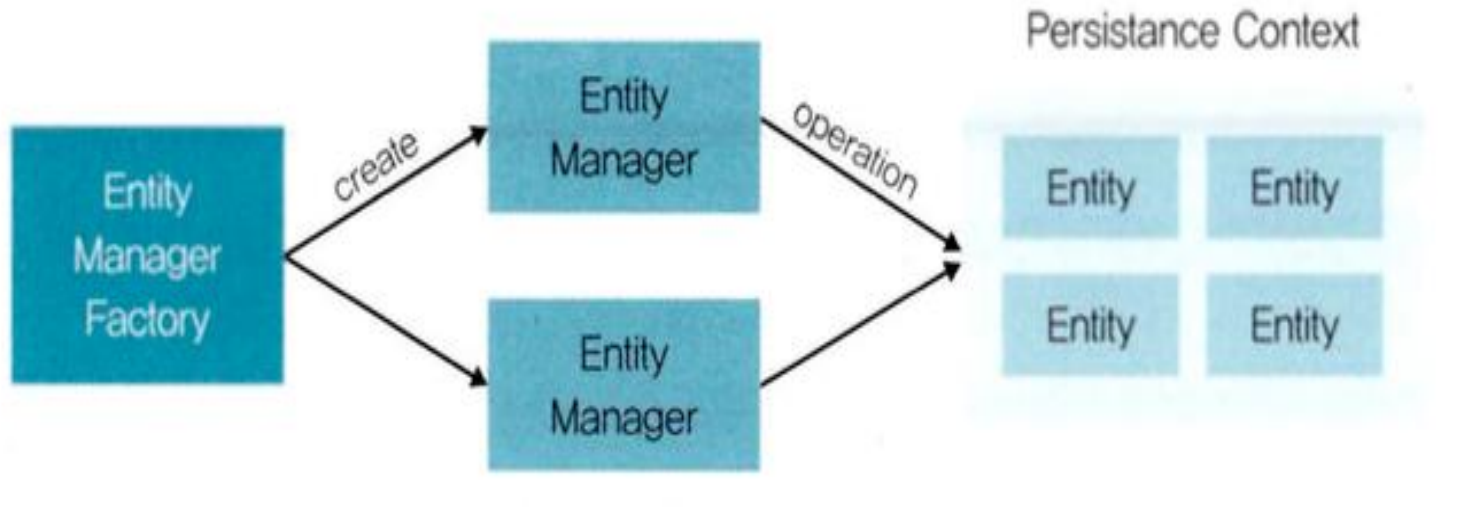
❖ 영속성 컨텍스트(Persistence Context)

- ✓ 애플리케이션과 데이터베이스 사이에서 Entity와 레코드의 괴리를 해소하는 기능과 객체를 보관하는 기능을 수행
- ✓ Entity 객체가 영속성 컨텍스트에 들어오면 JPA는 Entity 객체의 매핑 정보를 데이터베이스에 반영하는 작업을 수행
- ✓ Entity 객체가 영속성 컨텍스트에 들어와 JPA의 관리 대상이 되는 시점부터는 해당 객체를 영속 객체(Persistence Object) 라고 부름
- ✓ 영속성 컨텍스트는 세션 단위의 생명주기를 가지는데 데이터베이스에 접근하기 위한 세션이 생성되면 영속성 컨텍스트가 만들어지고 세션이 종료되면 영속성 컨텍스트도 없어짐



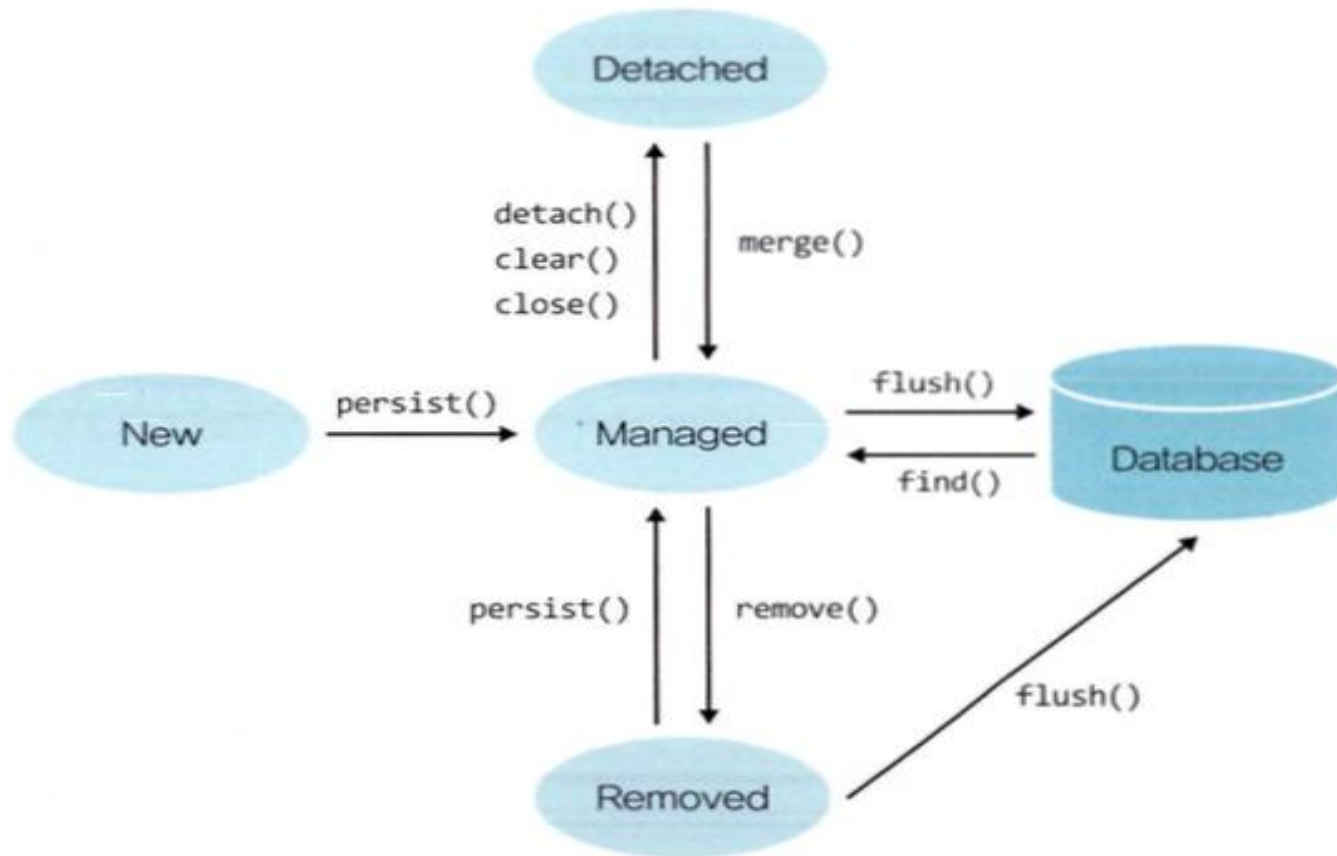
Context & Entity

❖ 동작 방식



Context & Entity

- ❖ JPA(Java Persistence API)
 - ✓ Entity 수명 주기



Context & Entity

❖ JPA(Java Persistence API)

✓ Entity 수명 주기

- 비영속(new): new 키워드를 통해 생성된 상태로 영속성 컨텍스트와 관련이 없는 상태
- 영속(managed)
 - ◆ Entity가 영속성 컨텍스트에 저장된 상태로 영속성 컨텍스트에 의해 관리되는 상태
 - ◆ 영속 상태에서 데이터베이스에 저장되지 않으며 트랜잭션 Commit 시점에 데이터베이스에 반영
- 준영속(detached): 영속성 컨텍스트에 Entity가 저장되었다가 분리된 상태
- 삭제(removed): 영속성 컨텍스트와 데이터베이스에서 삭제된 상태

Context & Entity

❖ JPA(Java Persistence API)

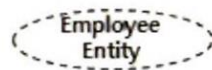
✓ Entity 수명 주기

● 비영속(new)

```
Employee employee = new Employee();
```

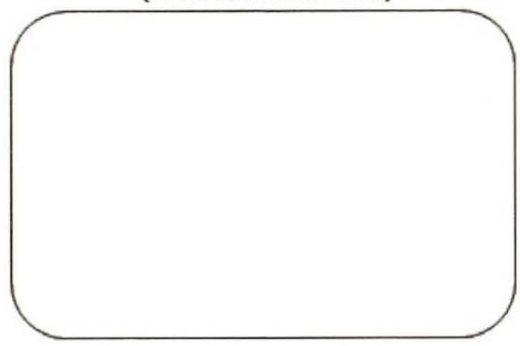
```
Employee.setName("아담");
```

◆ 현재 상태



Employee
Entity

A dashed oval containing the text "Employee Entity".



영속 컨테이너
(Persistence Context)

A large rounded rectangle representing the Persistence Context.

Context & Entity

❖ JPA(Java Persistence API)

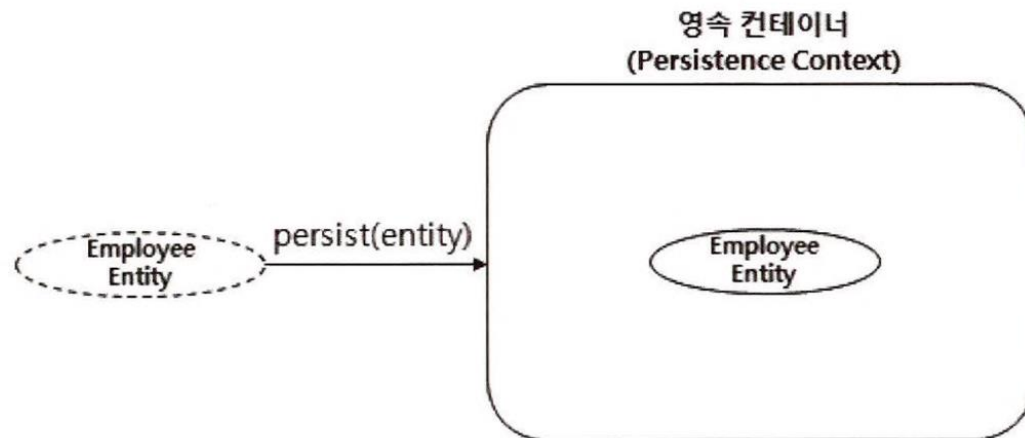
✓ Entity 수명 주기

● 영속(managed)

- ◆ Entity 가 save를 호출하거나 Repository가 find 메서드를 호출해서 데이터를 찾아오면 이 상태에 진입

`employee.save();`

- ◆ 현재 상태



Context & Entity

❖ JPA(Java Persistence API)

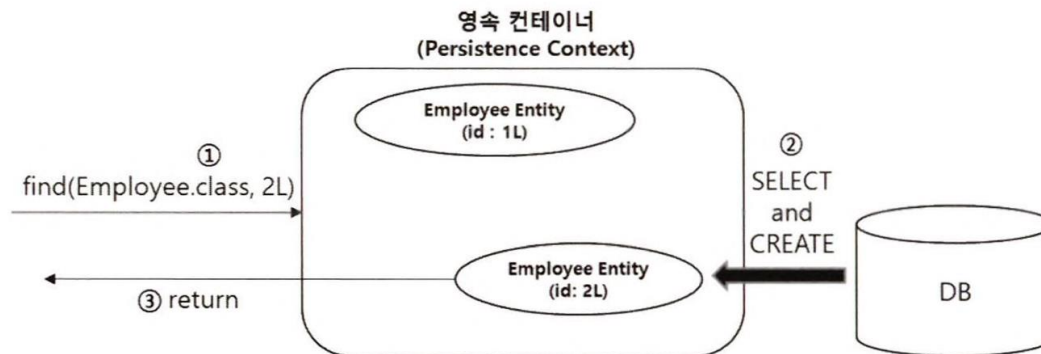
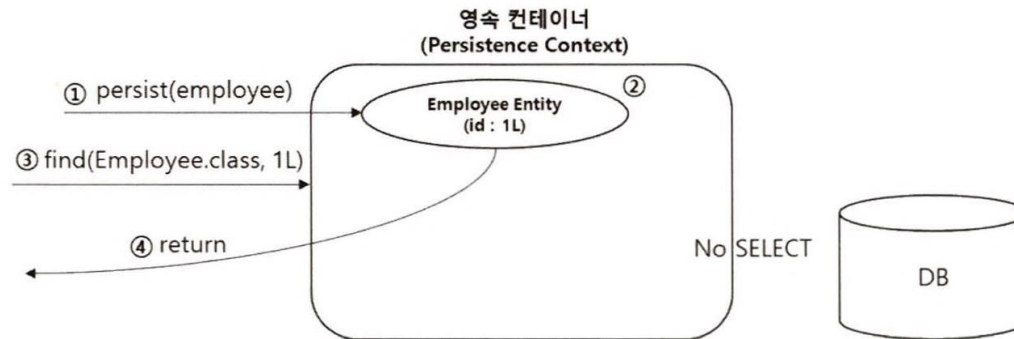
✓ Entity 수명 주기

● 영속(managed)

- ◆ Entity 가 save를 호출하면 영속성 컨텍스트는 persist 메서드를 호출해서 INSERT를 처리
- ◆ Entity를 수정한 후 바로 save를 호출하면 이 경우에도 persist 메서드를 호출해서 Primary Key를 확인한 후 삽입을 할 것 인지 수정을 할 것인지를 판단해서 필요한 SQL 구문을 수행
- ◆ find를 호출하는 경우에는 조회하고자 하는 Entity가 영속성 컨텍스트에 존재하면 Entity를 반환하고 그렇지 않으면 데이터베이스에서 조회해서 새로운 Entity를 생성한 후 영속성 컨텍스트에 등록 한 후 리턴

Context & Entity

- ❖ JPA(Java Persistence API)
 - ✓ Entity 수명 주기
 - 영속(managed)



Context & Entity

❖ JPA(Java Persistence API)

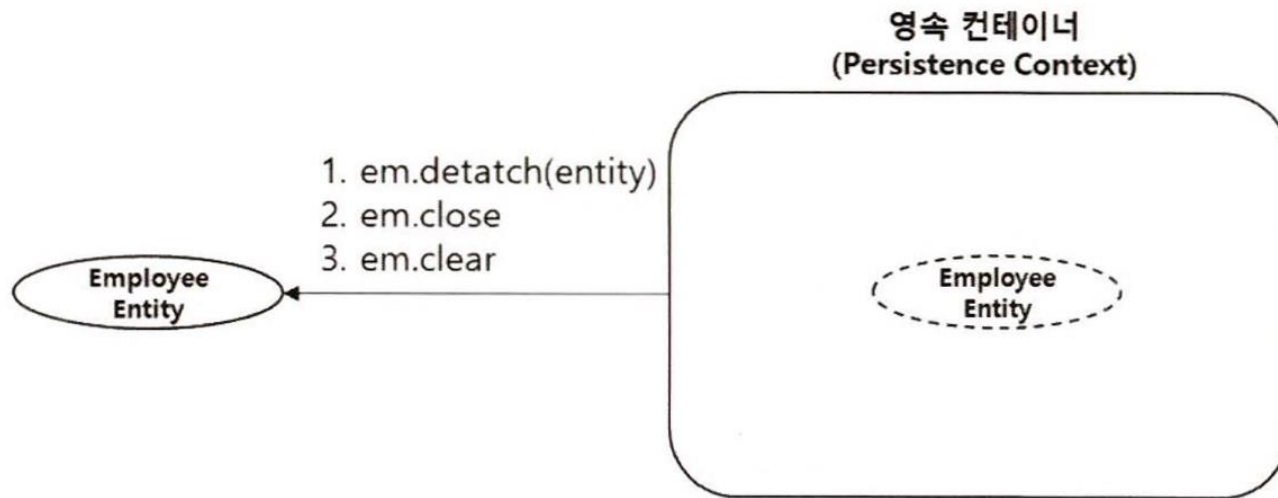
✓ Entity 수명 주기

● 준영속(detached)

- ◆ 영속성 컨테이너에 있던 Entity가 특정 작업에 의해 영속성 컨테이너에서 벗어난 상태
- ◆ 분리 상태의 Entity는 영속성 컨테이너의 통제에서 벗어났기 때문에 영속성 컨테이너가 Entity의 상태를 관리하지 못하기 때문에 Entity를 수정해도 데이터베이스에는 아무런 영향을 미치지 못함
- ◆ 영속성 컨테이너를 벗어난 상태라는 점에서 분리 상태를 생성 상태와 비슷하다고 생각할 수 있지만 두 상태를 구분하는 가장 중요한 기준은 식별자 값의 유무라고 생각하면 되는데 생성 상태의 Entity는 한 번도 관리 상태로 전환된 적이 없으므로 식별자 값을 가지지 않지만 분리 상태는 한 번 관리 상태로 전환된 적이 있다가 영속성 컨테이너를 벗어났기 때문에 반드시 식별자 값을 가지고 있음

Context & Entity

- ❖ JPA(Java Persistence API)
 - ✓ Entity 수명 주기
 - 준영속(detached)



Context & Entity

- ❖ JPA(Java Persistence API)
 - ✓ Entity 수명 주기
 - 삭제(removed)
 - ◆ 영속성 컨테이너에서 제거된 상태를 의미

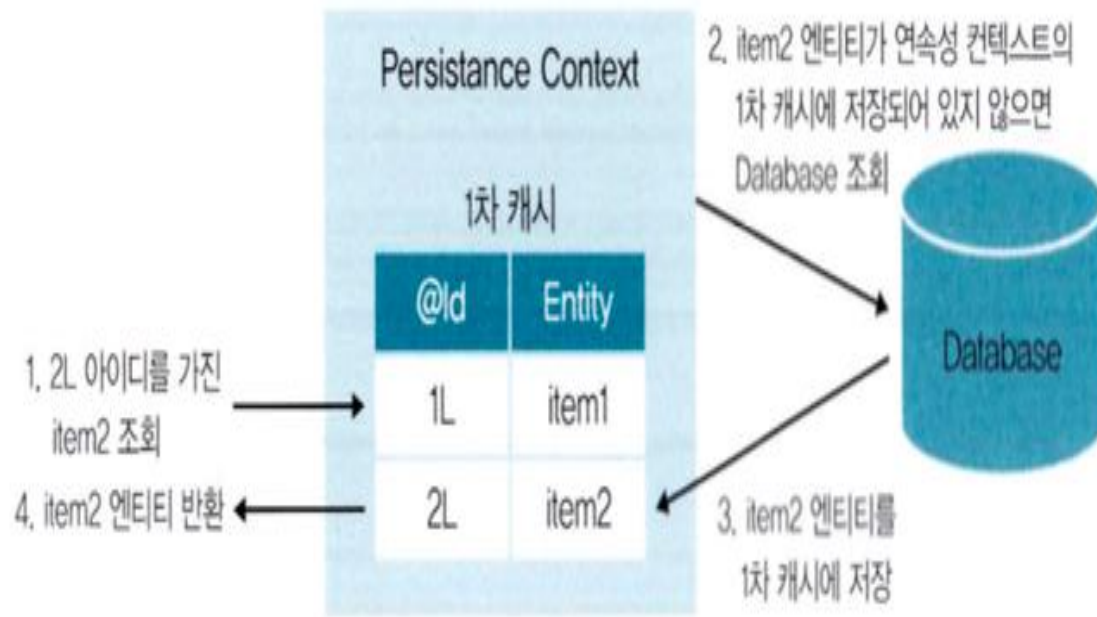


Context & Entity

❖ JPA(Java Persistence API)

✓ Persistency Context 사용 이유

- 영속성 컨텍스트는 애플리케이션과 데이터베이스 사이의 중간 계층
- 중간 계층을 만들면 버퍼링, 캐싱 등 을 활용할 수 있는 장점이 있음



Context & Entity

❖ JPA(Java Persistence API)

✓ Persistency Context 사용 이유

● 1차 캐시

- ◆ 영속성 컨텍스트에는 1차 캐시가 존재하며 Map<KEY, VALUE>을 이용해서 저장
- ◆ entityManager.find() 호출 시 영속성 컨텍스트의 1차 캐시를 조회해서 Entity가 존재할 경우 해당 Entity를 반환하고 Entity가 없으면 데이터베이스에서 조회 후 1차 캐시에 저장 및 반환

● 동일성 보장

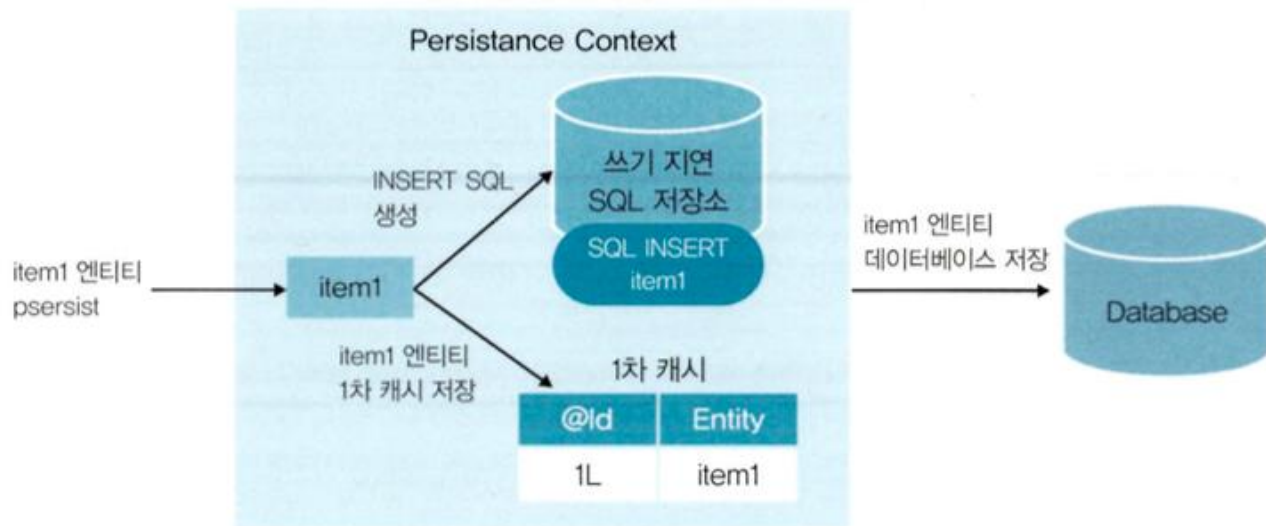
- ◆ 하나의 트랜잭션에서 같은 키 값으로 영속성 컨텍스트에 저장된 Entity 조회 시 같은 Entity 조회를 보장
- ◆ 1차 캐시에 저장된 Entity를 조회하기 때문에 가능

Context & Entity

❖ JPA(Java Persistence API)

✓ Persistency Context 사용 이유

- 트랜잭션을 지원하는 Class 와 Table이 유사하듯이 쓰기 지연
 - ◆ 영속성 컨텍스트에는 쓰기 지연 SQL 저장소가 존재
 - ◆ entityManager.persist()를 호출하면 1차 캐시에 저장되는 것과 동시에 쓰기 지연 SQL 저장소에 SQL문이 저장되는데 이렇게 SQL을 쌓아두고 트랜잭션을 Commit하는 시점에 저장된 SQL문들이 flush되면서 데이터베이스에 반영됨
 - ◆ 모아서 보내기 때문에 성능에서 이점을 볼 수 있음



Context & Entity

❖ JPA(Java Persistence API)

✓ Persistency Context 사용 이유

● 변경 감지

- ◆ JPA는 1차 캐시에 데이터베이스에서 처음 불러온 Entity의 스냅샷 값을 갖고 있으며 1차 캐시에 저장된 Entity와 스냅샷을 비교 후 변경 내용이 있다면 UPDATE SQL문을 쓰기 지연 SQL 저장소에 담아두고 데이터베이스에 Commit 시점에 변경 내용을 자동으로 반영
- ◆ update문을 호출할 필요가 없음

Spring JPA

- ❖ JPA(Java Persistence API)를 사용
 - ✓ gradle을 사용하는 프로젝트 생성

New Spring Starter Project

Service URL:

Name:

☒ Use default location

Location:

Type: Packaging:

Java Version: Language:

Group:

Artifact:

Version:

Description:

Package:

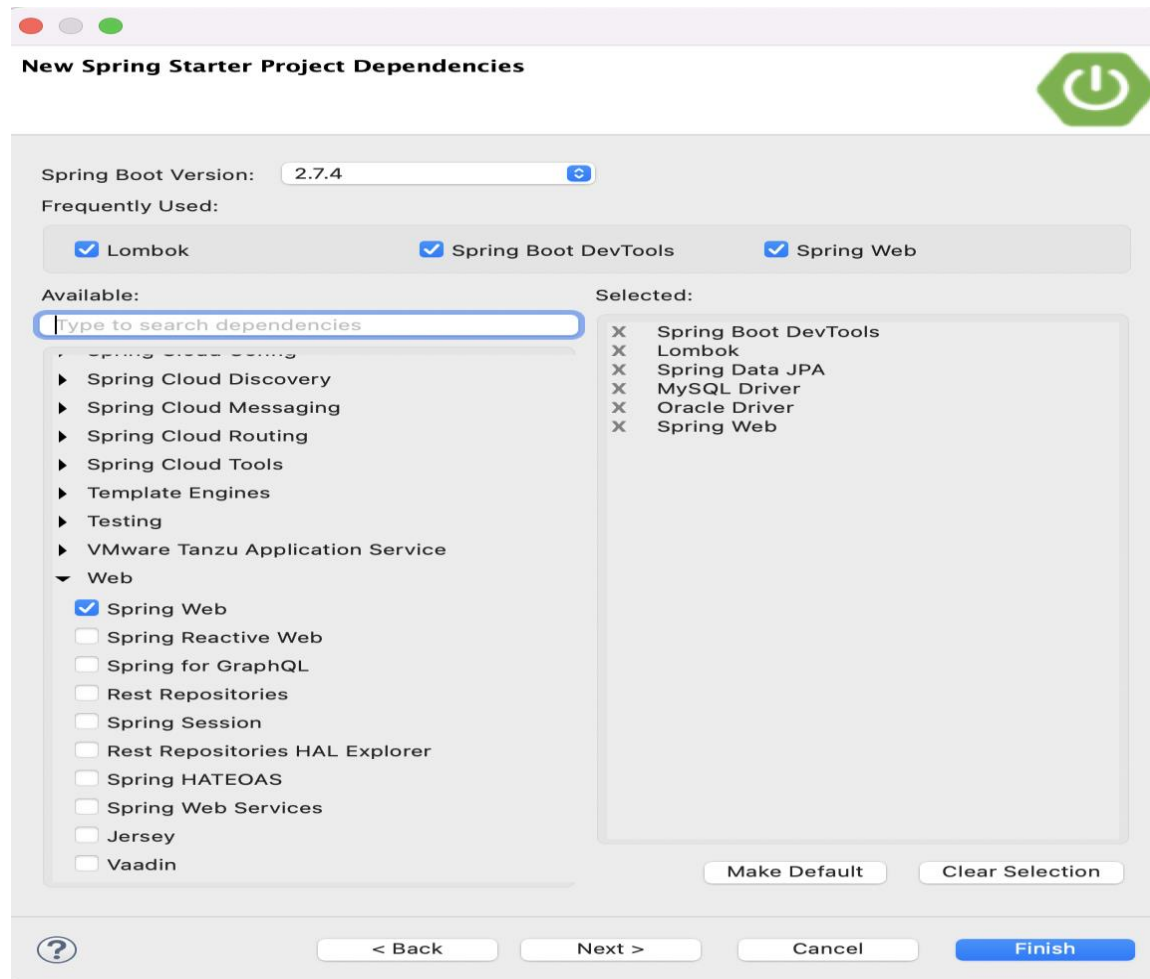
Working sets

☐ Add project to working sets

Working sets:

Spring JPA

- ❖ JPA(Java Persistence API)를 사용
 - ✓ 의존성 설정



Spring JPA

❖ JPA(Java Persistence API)를 사용

- ✓ 실행하면 에러: JPA 라이브러리가 추가되면 사용할 데이터베이스에 대한 설정이 포함되어야 하는데 이 정보가 없어서 에러
- ✓ application.yml 파일을 생성하고 데이터베이스 접속 정보 설정

#웹 서버 실행 포트 설정

server:

port: 80

#데이터베이스 접속 정보

spring:

datasource:

url: jdbc:mariadb://localhost:3306/adam

driver-class-name: org.mariadb.jdbc.Driver

username: root

password: wnddkd

Spring JPA

❖ JPA(Java Persistence API)를 사용

- ✓ 실행 로그를 확인해보면 HikariPool...로 시작하는 부분이 기록되는 것을 볼 수 있는데 이는 Spring Boot가 기본적으로 이용하는 커넥션 풀(Connection Pool)이 HikariCP 라이브러리를 이용하기 때문이며 org.hibernate로 시작하는 하이버네이트 설정 로그가 보임

```
2022-10-16 17:19:58.675 INFO 11084 --- [ restartedMain] o.a.c.c.C.[Tomcat].[localhost].[/]  
2022-10-16 17:19:58.675 INFO 11084 --- [ restartedMain] w.s.c.ServletWebServerApplicationContext  
2022-10-16 17:19:58.813 INFO 11084 --- [ restartedMain] com.zaxxer.hikari.HikariDataSource  
2022-10-16 17:19:59.212 INFO 11084 --- [ restartedMain] com.zaxxer.hikari.HikariDataSource  
2022-10-16 17:19:59.258 INFO 11084 --- [ restartedMain] o.hibernate.jpa.internal.util.LogHelper  
2022-10-16 17:19:59.294 INFO 11084 --- [ restartedMain] org.hibernate.Version  
2022-10-16 17:19:59.487 INFO 11084 --- [ restartedMain] o.hibernate.annotations.common.Version  
2022-10-16 17:19:59.578 INFO 11084 --- [ restartedMain] org.hibernate.dialect.Dialect  
2022-10-16 17:19:59.725 INFO 11084 --- [ restartedMain] o.h.e.t.j.p.i.JtaPlatformInitiator  
2022-10-16 17:19:59.732 INFO 11084 --- [ restartedMain] j.LocalContainerEntityManagerFactoryBean  
2022-10-16 17:19:59.767 WARN 11084 --- [ restartedMain] JpaBaseConfiguration$JpaWebConfiguration  
2022-10-16 17:20:00.071 INFO 11084 --- [ restartedMain] o.s.b.d.a.OptionalLiveReloadServer  
2022-10-16 17:20:00.115 INFO 11084 --- [ restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer  
2022-10-16 17:20:00.124 INFO 11084 --- [ restartedMain] com.adamsoft.ipa.JpaApplication
```

Spring JPA

❖ JUnit

✓ 테스트 코드 실행

- src/test/java 디렉토리에 Test를 위한 클래스를 만들고 마우스 오른쪽을 눌러서 [Run As] – [JUnit Test]를 실행

```
public class TestLifecycle {
```

```
    @BeforeAll
```

```
    static void beforeAll() {
```

```
        System.out.println("## BeforeAll Annotation 호출 ##");
```

```
        System.out.println();
```

```
    }
```

```
    @AfterAll
```

```
    static void afterAll() {
```

```
        System.out.println("## afterAll Annotation 호출 ##");
```

```
        System.out.println();
```

```
    }
```

Spring JPA

❖ JUnit

✓ 테스트 코드 실행

- src/test/java 디렉토리에 Test를 위한 클래스를 만들고 마우스 오른쪽을 눌러서 [Run As] – [JUnit Test]를 실행

```
@BeforeEach
void beforeEach() {
    System.out.println("## beforeEach Annotation 호출 ##");
    System.out.println();
}
```

```
@AfterEach
void afterEach() {
    System.out.println("## afterEach Annotation 호출 ##");
    System.out.println();
}
```

Spring JPA

❖ JUnit

✓ 테스트 코드 실행

- src/test/java 디렉토리에 Test를 위한 클래스를 만들고 마우스 오른쪽을 눌러서 [Run As] – [JUnit Test]를 실행

```
@Test
void test1() {
    System.out.println("## test1 시작 ##");
    System.out.println();
}
@Test
@DisplayName("Test Case 2!!!")
void test2() {
    System.out.println("## test2 시작 ##");
    System.out.println();
}
// Disabled Annotation : 테스트를 실행하지 않게 설정하는 어노테이션
@Test
@Disabled
void test3() {
    System.out.println("## test3 시작 ##");
    System.out.println();
}
}
```

Spring JPA

❖ JUnit

✓ 테스트 코드 실행

● 결과

BeforeAll Annotation 호출

beforeEach Annotation 호출

test1 시작

afterEach Annotation 호출

beforeEach Annotation 호출

test2 시작

afterEach Annotation 호출

afterAll Annotation 호출

Spring JPA

❖ Entity Class 와 JpaRepository

✓ Spring Data JPA 개발에 필요한 코드

- JPA를 통해서 관리하게 되는 객체 – Entity Class 와 Entity Object
- Entity 객체들을 처리하는 기능을 가진 Repository
 - ◆ Repository는 Spring Data JPA에서 제공하는 인터페이스로 설계하는데 스프링 내부에서 자동으로 객체를 생성하고 실행하는 구조라 개발자 입장에서는 단순히 인터페이스를 하나 정의하는 작업만으로도 충분
 - ◆ 기존의 방식에서는 모든 코드를 직접 구현하고 트랜잭션 처리가 필요했지만 Spring Data JPA에서는 자동으로 생성되는 코드를 이용하므로 단순 CRUD나 페이지 처리 등을 개발할 때 직접 코드를 작성하지 않아도 됨

Spring JPA

❖ Entity Class 와 JpaRepository

✓ Entity 관련 annotation

● @Entity:

- ◆ Entity Class는 Spring Data JPA에서는 반드시 @Entity라는 annotation을 추가해야만 함
- ◆ @Entity는 해당 Class가 Entity를 위한 Class이며 해당 Class의 Instance들이 JPA로 관리되는 Entity 객체라는 것을 의미
- ◆ @Entity가 붙은 Class는 옵션에 따라서 자동으로 Table을 생성할 수 있는데 이 경우 @Entity가 있는 Class의 속성에 따라서 자동으로 칼럼들이 생성됨
- ◆ 클래스를 Entity로 선언하는데 name 속성을 이용해서 Entity 이름을 변경할 수 있는데 name 속성을 사용하지 않으면 클래스 이름을 그대로 사용

Spring JPA

❖ Entity Class 와 JpaRepository

✓ Entity 관련 annotation

- @Table: Entity 와 매핑할 테이블을 지정 – 생략하면 클래스 이름과 동일한 테이블 매핑
 - ◆ @Entity annotation과 같이 사용할 수 있는 annotation 으로 데이터베이스상에서 Entity Class를 어떠한 Table로 생성할 것인지에 대한 정보를 담기 위한 annotation
 - ◆ 속성
 - name 속성을 이용해서 매핑할 테이블을 설정하는 것이 가능
 - catalog: 데이터베이스 카탈로그를 지정 – MySQL
 - schema: 데이터베이스 스키마를 지정 - Oracle
 - uniqueConstraints 속성에 여러 개의 컬럼이 결합되어 유일성을 보장해야 하는 경우 사용 - uniqueConstraints =
`{@UniqueConstraint(columnNames={"NAME", "MAILID"})}`
 - ◆ @Table(name="t_memo")와 같이 지정하는 경우에는 생성되는 Table의 이름이 t_memo Table로 생성됨
 - ◆ @Table annotation이 없는 경우에는 Entity 이름으로 테이블을 생성하는데 MySQL(Maria DB)에서는 중간에 대문자가 있으면 앞에 _를 붙이고 소문자로 변경함

Spring JPA

❖ Entity Class 와 JpaRepository

✓ Entity 관련 annotation

- @Table: Entity 와 매핑할 테이블을 지정 – 생략하면 클래스 이름과 동일한 테이블 매핑

◆ 인덱스 설정

- 인덱스 1개

```
@Table(name = "MEMBER", indexes = @Index(name = "idx_email_birthday", columnList = "email, birthday"))
```

- 인덱스 여러 개

```
@Table(name = "MEMBER", indexes = {  
    @Index(name = "idx_email_birthday", columnList = "email, birthday"),  
    @Index(name = "idx_name_age", columnList = "name, age")  
})
```

Spring JPA

❖ Entity Class 와 JpaRepository

✓ Entity 관련 annotation

- @Table: Entity 와 매핑할 테이블을 지정 – 생략하면 클래스 이름과 동일한 테이블 매핑

◆ 인덱스 설정

- 유일 인덱스

```
@Table(name = "MEMBER", indexes = {  
    @Index(name = "idx_email_birthday", columnList = "email,  
    birthday"), // index (unique = false (default))  
    @Index(name = "idx_name_age", columnList = "name, age"), //  
    index (unique = false (default))  
    @Index(name = "idx_unique_nickname", columnList = "nickname",  
    unique = true) // unique index  
})
```

Spring JPA

❖ Entity Class 와 JpaRepository

✓ Entity 관련 annotation

● @Column: 필드와 컬럼 매핑

- ◆ name: 컬럼 이름 설정으로 생각하면 변수 이름 과 동일한 컬럼 이름 사용
- ◆ unique: 기본값은 false
- ◆ nullable: 기본값은 false
- ◆ insertable, updatable: 삽입 과 수정 시 컬럼 포함 여부로 기본값은 true
- ◆ length: 문자열 타입의 길이
- ◆ precision: 숫자 타입의 전체 자릿수 지정
- ◆ scale: 소수점 자릿수 지정
- ◆ columnDefinition – 자료형 과 제약 조건을 직접 기재,
@Column(columnDefinition = "varchar(255) default 'Yes'")

Spring JPA

❖ Entity Class 와 JpaRepository

✓ Entity 관련 annotation

- @Lob: BLOB, CLOB 타입 매핑
- @CreationTimestamp: insert 시 시간 자동 저장
- @UpdateTimestamp: update 시 시간 자동 저장
- @Enumerated: enum 타입 매핑
- @Transient: 해당 필드는 테이블 매핑을 하지 않음
- @Temporal: java.util.Date 타입을 매핑할 때 사용하는 것으로 TemporalType.TIME 이나 TemporalType.DATE 이나 TemporalType.TIMESTAMP 설정 가능
- @CreateDate: Entity가 생성되어 저장될 때 시간 자동 저장
- @LastModifiedDate: Entity의 값을 변경할 때 시간 자동 저장
- @Access: SQL에 사용자가 입력한 값을 바인딩 할 때 접근하는 방식
 - ◆ @Access(AccessType.FIELD): 속성에 직접 접근
 - ◆ @Access(AccessType.PROPERTY): getter 와 setter로 접근

Spring JPA

❖ Entity Class 와 JpaRepository

✓ Entity 관련 annotation

● @Id 와 @GeneratedValue

- ◆ @Entity가 붙은 Class는 Primary Key(이하 PK)에 해당하는 특정 필드를 @Id로 지정해야만 하는데 @Id가 사용자가 입력하는 값을 사용하는 경우가 아니면 자동으로 생성되는 번호를 사용하기 위해서 @GeneratedValue라는 annotation을 활용
- ◆ @GeneratedValue(strategy = GenerationType.IDENTITY) 부분은 PK를 자동으로 생성하고자 할 때 사용하는데 연결되는 데이터베이스가 오라클이면 번호를 위한 별도의 Table을 생성하고 MySQL이나 MariaDB면 auto increment를 사용해서 새로운 레코드가 기록될 때 마다 다른 번호를 가질 수 있도록 처리

Spring JPA

❖ Entity Class 와 JpaRepository

✓ Entity 관련 annotation

● @Id 와 @GeneratedValue

◆ 키 생성 전략

- AUTO(default) – JPA 구현체(Spring Boot에서는Hibernate)가 생성 방식을 결정
- IDENTITY - 사용하는 데이터베이스가 키 생성을 결정하는데 MySQL이나 MariaDB의 경우 auto increment 방식을 이용하는데 오라클에 적용하면 테이블이 자동으로 생성되지 않음

Spring JPA

❖ Entity Class 와 JpaRepository

✓ Entity 관련 annotation

● @Id 와 @GeneratedValue

◆ 키 생성 전략

- SEQUENCE - 데이터베이스의 sequence를 이용해서 키를 생성하는데 @SequenceGenerator와 같이 사용

```
@Entity
@SequenceGenerator(
    name = "MEMBER_SEQ_GENERATOR",
    sequenceName = "MEMBER_SEQ", // 데이터베이스 시퀀스 이름
    initialValue = 1,
    allocationSize = 1)
public class Member {
    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE,
                    generator = "MEMBER_SEQ_GENERATOR")
    private Long id;
}
```

Spring JPA

❖ Entity Class 와 JpaRepository

✓ Entity 관련 annotation

● @Id 와 @GeneratedValue

◆ 키 생성 전략

- TABLE - 키 생성 전용 Table을 생성해서 키를 생성하는데 모든 DBMS에서 동일하게 동작하기를 원하는 경우 사용하는 값으로 @TableGenerator 어노테이션으로 테이블 정보를 설정

@Entity

@SequenceGenerator(

name = "MEMBER_SEQ_GENERATOR",

table = "MY_SEQUENCES", // 데이터베이스 이름

pkColumnName = "MEMBER_SEQ",

allocationSize = 1)

public class Member {

@Id

@GeneratedValue(strategy = GenerationType.TABLE,
generator = "MEMBER_SEQ_GENERATOR")

private Long id;

}

<https://gmlwjd9405.github.io/2019/08/12/primary-key-mapping.html>

Spring JPA

❖ Entity Class 와 JpaRepository

✓ Entity 관련 annotation

● @Id 와 @GeneratedValue

◆ 랜덤한 키(UUID) 사용

▪ 바이너리 코드

@Id

@GeneratedValue(strategy = GenerationType.UUID)

private UUID id;

▪ 문자열

@Id

@GeneratedValue(strategy = GenerationType.UUID)

private String id;

Spring JPA

❖ Entity Class 와 JpaRepository

✓ Entity 관련 annotation

● 복합키를 사용하고자 하는 경우

◆ 별도의 테이블을 생성해서 사용

@Embeddable

@NoArgsConstructor

@AllArgsConstructor

@Getter

@EqualsAndHashCode

@ToString

public class EmployeeId implements Serializable

{

private static final long serialVersionUID = 1L;

private Long id;

private String mailId;

}

Spring JPA

❖ Entity Class 와 JpaRepository

- ✓ Table과 매핑할 Class 생성 – 기본패키지.entity.Memo

@Entity

@Table(name= "tbl_memo")

@ToString

@Getter

@Builder

@AllArgsConstructor

@NoArgsConstructor

Spring JPA

❖ Entity Class 와 JpaRepository

- ✓ Table과 매핑할 Class 생성 – 기본패키지.entity.Memo

```
public class Memo {  
    @Id  
    //auto_increament 사용  
    //@GeneratedValue(strategy = GenerationType.IDENTITY)  
  
    //Sequence 생성  
    //@GeneratedValue(strategy = GenerationType.SEQUENCE, generator =  
        "SequenceGeneratorName")  
    //@SequenceGenerator(sequenceName = "SequenceName", name =  
        "SequenceGeneratorName", allocationSize = 1)  
  
    //생성 방법을 Hibernate 가 결정  
    @GeneratedValue(strategy = GenerationType.AUTO)  
    private Long mno;  
  
    @Column(length = 200, nullable = false)  
    private String memoText;  
}
```

Spring JPA

❖ Entity Class 와 JpaRepository

- ✓ Table과 매핑할 Class 생성 – 기본패키지.entity.EmployeeId

@Embeddable

@NoArgsConstructor

@AllArgsConstructor

@Getter

@EqualsAndHashCode

@ToString

public class EmployeeId implements Serializable

{

private static final long serialVersionUID = 1L;

private Long id;

private String mailId;

}

Spring JPA

❖ Entity Class 와 JpaRepository

- ✓ Table과 매핑할 Class 생성 – 기본패키지.entity.Employee

```
@Entity
@ToString
@Getter
@Builder
@AllArgsConstructor
@NoArgsConstructor
public class Employee {
    @Id
    private EmployeeId employeeId;

    @Column(length = 200, nullable = false)
    private String name;

    @Column
    private Long salary;
}
```

Spring JPA

❖ Entity Class 와 JpaRepository

- ✓ Table과 매핑할 Class 생성 – 기본패키지.entity.UUID1

```
@Entity
@ToString
@Getter
@Builder
@AllArgsConstructor
@NoArgsConstructor
public class UUID1 {
    @Id
    @GeneratedValue(strategy = GenerationType.UUID)
    private UUID id;
}
```

Spring JPA

❖ Entity Class 와 JpaRepository

- ✓ Table과 매핑할 Class 생성 – 기본패키지.entity.UUID1

```
@Entity
@ToString
@Getter
@Builder
@AllArgsConstructor
@NoArgsConstructor
public class UUID2 {
    @Id
    @GeneratedValue(strategy = GenerationType.UUID)
    private String id;
}
```

Spring JPA

❖ Spring Data JPA를 위한 Spring Boot 설정

✓ application.yml 파일에 설정 추가

#웹 서버 실행 포트 설정

server:

port: 80

#데이터베이스 접속 정보

spring:

datasource:

url: jdbc:mariadb://localhost:3306/adam

driver-class-name: org.mariadb.jdbc.Driver

username: root

password: wnddkd

Spring JPA

❖ Spring Data JPA를 위한 Spring Boot 설정

- ✓ application.yml 파일에 설정 추가

```
jpa:  
  hibernate:  
    ddl-auto: update  
  properties:  
    hibernate:  
      format_sql: true  
      show_sql: true
```

#쿼리에 물음표로 출력되는 바인드 파라미터 출력

```
logging:  
  level:  
    org.hibernate.type.descriptor.sql: trace
```

Spring JPA

❖ Spring Data JPA를 위한 Spring Boot 설정

✓ application.yml 파일 설정 내용

- spring.jpa.hibernate.ddl-auto: 프로젝트 실행 시에 자동으로 DDL(create, alter, drop 등)을 생성 할 것인지를 결정하는 설정 – 운영 환경에서는 none 이나 validate 사용
 - ◆ none: 아무것도 실행하지 않음(대부분의 DB에서 기본값)
 - ◆ create: SessionFactory가 시작될 때 기존 테이블을 drop하고 create
 - ◆ create-drop: SessionFactory가 시작될 때 create를 실행하고 SessionFactory가 종료될 때 drop을 실행(in-memory DB의 경우 기본값)
 - ◆ update: SessionFactory가 시작될 때 변경된 스키마를 적용
 - ◆ validate: Entity 와 Table 정상 매핑 확인(스테이징 및 운영 환경에서 주로 이용)

Spring JPA

❖ Spring Data JPA를 위한 Spring Boot 설정

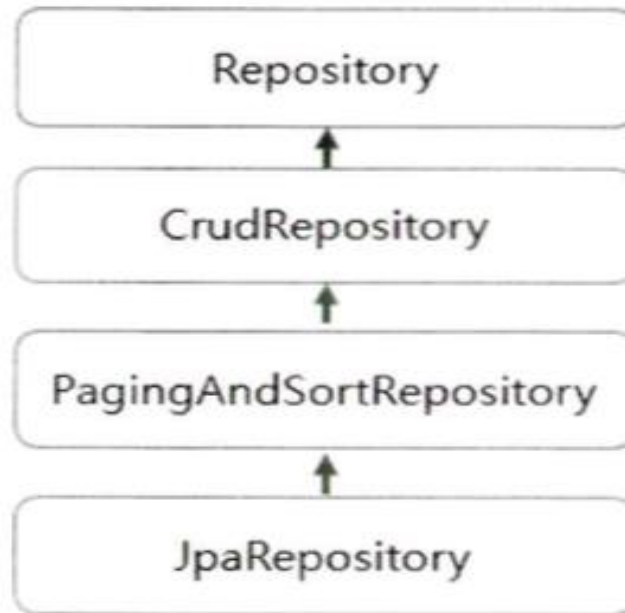
✓ application.yml 파일 설정 내용

- `spring.jpa.properties.hibernate.show_sql`: JPA 처리 시에 발생하는 SQL을 보여줄 것인지를 결정
- `spring.jpa.properties.hibernate.format_sql`: 실제 JPA의 구현체인 Hibernate가 동작하면서 발생하는 SQL을 포맷팅해서 출력하는 것으로
`spring.jpa.properties.hibernate.use_sql_comments`,
`spring.jpa.properties.hibernate.type.descriptor.sql` 등의 옵션도 있음
- `spring.jpa.hibernate.id.new_generator_mappings`: JPA 스펙에 맞는 새로운 키 생성 전략을 사용하는데 기존 버전과 호환성을 고려한다면 `false`

Spring JPA

❖ JpaRepository 인터페이스

- ✓ Spring Data JPA는 JPA의 구현체인 Hibernate를 이용하기 위한 여러 API를 제공
- ✓ Spring Data JPA에는 여러 종류의 인터페이스의 기능을 통해서 JPA 관련 작업을 별도의 코드 없이 처리할 수 있게 지원
- ✓ JpaRepository를 이용하는 경우가 많음



Spring JPA

❖ JpaRepository 인터페이스

- ✓ JpaRepository를 상속받은 MemoRepository 인터페이스 생성 – JpaRepository<Entity Class, Entity 에서 Id 의 자료형> 을 상속받아서 생성

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

```
public interface MemoRepository extends JpaRepository<Memo, Long> {
```

```
}
```

Spring JPA

❖ JpaRepository 인터페이스

✓ CRUD 작업

- insert, update 작업: save(Entity 객체)
- select 작업: findById(키 타입), findAll()
- 데이터 개수 확인: count()
- delete 작업: deleteById(키 타입), delete(Entity 객체)
- save() 의 경우JPA의 구현체가 메모리(Entity Manager라는 존재가 Entity들을 관리하는 방식)에서 객체를 비교하고 없다면 insert 존재한다면 update 동작을 수행

Spring JPA

❖ JpaRepository 인터페이스

✓ CRUD 작업

- src/test 디렉토리에 자바 Class를 만들고 CRUD 테스트 - RepositoryTest

```
@SpringBootTest
public class RepositoryTest {
    @Autowired
    MemoRepository memoRepository;

    @Autowired
    UUID1Repository uuid1Repository;

    @Autowired
    UUID2Repository uuid2Repository;

    //주입 확인
    @Test
    public void testDependency(){
        System.out.println("주입 여부:" + memoRepository.getClass().getName());
    }
}
```

Spring JPA

❖ JpaRepository 인터페이스

✓ CRUD 작업

● 데이터 삽입 테스트

//삽입 확인

@Test

public void testInsert(){

 IntStream.rangeClosed(1,100).forEach(i -> {

 Memo memo = Memo.builder().memoText("Sample..." + i).build();

 memoRepository. save(memo);

 });

 UUID1 uuid1 = new UUID1();

 uuid1Repository.save(uuid1);

 UUID2 uuid2 = new UUID2();

 uuid2Repository.save(uuid2);

}

Spring JPA

❖ JpaRepository 인터페이스

✓ CRUD 작업

● 데이터 삽입 테스트 - 테이블 확인

◆ `SELECT * FROM tbl_memo;`

cord	테스트	그리드		123 mno	ABC memo_text
			1	1	Sample...1
			2	2	Sample...2
			3	3	Sample...3
			4	4	Sample...4
			5	5	Sample...5
			6	6	Sample...6
			7	7	Sample...7
			8	8	Sample...8

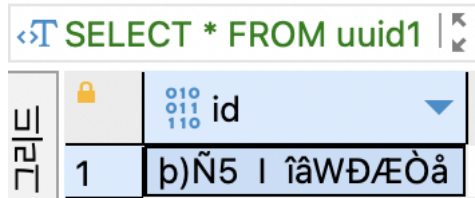
Spring JPA

❖ JpaRepository 인터페이스

✓ CRUD 작업

● 데이터 삽입 테스트 - 테이블 확인

◆ SELECT * FROM uuid1;



The screenshot shows a database query result for the query "SELECT * FROM uuid1". The result is displayed in a table with two columns: "id" and a column containing a UUID value. The "id" column has a value of 1, and the UUID column has a value that appears to be "p)Ñ5 I îâWĐÆÒå".

	id
1	p)Ñ5 I îâWĐÆÒå

Spring JPA

❖ JpaRepository 인터페이스

✓ CRUD 작업

● 데이터 삽입 테스트 - 테이블 확인

◆ SELECT * FROM uuid2;

<code><>T select * from uuid2</code> <i>Enter a SQL expression to f</i>		
그리드		ABC id ▼
		1 8bf6bfba-68b3-4757-9e56-a3ab819a2c62

Spring JPA

❖ JpaRepository 인터페이스

✓ CRUD 작업

● 데이터 조회 테스트

```
@Test
public void testSelect(){
    //데이터베이스에 존재하는 mno
    Long mno = 100L;
    Optional<Memo> result = memoRepository.findById(mno);
    System.out.println("=====");
    if(result.isPresent()){
        Memo memo = result.get();
        System.out.println(memo);
    }
}
```

Spring JPA

❖ JpaRepository 인터페이스

✓ CRUD 작업

● 데이터 수정 테스트

//데이터 수정

@Test

```
public void testUpdate() {
```

```
    Memo memo = Memo.builder().mno(100L).memoText("Update  
Text").build();
```

```
    System.out.println(memoRepository.save(memo));
```

```
}
```

Spring JPA

❖ JpaRepository 인터페이스

✓ CRUD 작업

● 데이터 삭제 테스트

```
//데이터 삭제
@Test
public void testDelete() {
    Long mno = 100L;
    memoRepository.deleteById(mno);
}
```

Spring JPA

❖ JpaRepository 인터페이스

✓ 페이징/정렬

- JPA를 사용하지 않는 경우 페이지 처리는 데이터베이스의 종류에 따라서 사용되는 기법이 달라서 별도의 학습이 필요
- 오라클은 인라인 뷰(inline view) 나 FETCH & OFFSET(12C 이후 사용 가능)를 사용하고 MySQL은 limit를 사용
- JPA는 내부적으로 이런 처리를 Dialect 라는 존재를 이용해서 처리하는데 JDBC 정보가 MySQL의 경우에는 자동으로 MySQL를 위한 Dialect가 설정되고 명령을 수행할 때 그 데이터베이스에 맞는 형태의 SQL로 변화해서 처리함
- JPA가 이처럼 실제 데이터베이스에서 사용하는 SQL의 처리를 자동으로 하기 때문에 개발자들은 SQL이 아닌 API의 객체와 메서드를 사용하는 형태로 페이징 처리를 할 수 있음
- Spring Data JPA에서 페이징 처리와 정렬은 findAll 이라는 메서드를 사용
- findAll 메서드는 JpaRepository 인터페이스의 상위인 PagingAndSortRepository의 메서드로 파라미터로 전달되는 Pageable이라는 타입의 객체에 의해서 실행되는 쿼리를 결정하고 리턴은 Page<T> 타입

Spring JPA

❖ JpaRepository 인터페이스

✓ 페이징/정렬

● Pageable 인터페이스

- ◆ Pageable 인터페이스는 페이지 처리에 필요한 정보를 전달하는 용도의 타입으로 인터페이스이기 때문에 실제 객체를 생성할 때는 구현체인 `org.springframework.data.domain.PageRequest` 라는 Class를 사용
- ◆ `PageRequest` Class의 생성자는 `protected`로 선언되어 `new`를 이용할 수 없으며 객체를 생성하기 위해서는 `static` 메서드인 `of()`를 이용해서 처리
- ◆ `PageRequest` 생성자를 보면 `page`, `size`, `Sort`라는 정보를 이용해서 객체를 생성
- ◆ `of` 메서드
 - `of(int page, int size)`: 0부터 시작하는 페이지 번호와 개수(size)를 이용해서 생성하는데 정렬이 지정되지 않음
 - `of(int page, int size, Sort.Direction direction, String ... props)`: 0부터 시작하는 페이지 번호와 개수, 정렬의 방향과 정렬 기준 필드들을 이용해서 생성
 - `of(int page, int size, Sort sort)`: 페이지 번호와 개수, 정렬 관련 정보를 이용해서 생성

Spring JPA

❖ JpaRepository 인터페이스

✓ 페이징/정렬

● Pageable 인터페이스

◆ 페이징 테스트

//페이징

@Test

```
public void testPageDefault() {
```

```
    //1페이지 10개
```

```
    Pageable pageable = PageRequest.of(0,10);
```

```
    Page<Memo> result = memoRepository.findAll(pageable);
```

```
    System.out.println(result);
```

Spring JPA

❖ JpaRepository 인터페이스

✓ 페이징/정렬

● Pageable 인터페이스

◆ 페이징 테스트

```
        System.out.println ("-----  
----- ");  
        System.out.println("Total Pages: "+result.getTotalPages()); //전체  
페이지 개수  
        System.out.println("Total Count: "+result.getTotalElements());  
//전체 데이터 개수  
        System.out.println("Page Number: "+result. getNumber ()); //현재  
페이지 번호 0부터 시작  
        System.out.println("Page Size: "+result .getSize()); //페이지당  
데이터 개수  
        System.out.println("Has next page?:"+ result.hasNext()); //다음  
페이지존재 여부  
        System.out.println("First page?: "+result.isFirst()); //시작 페이지 (0)  
여부  
        System.out.println("-----  
-----");
```

Spring JPA

❖ JpaRepository 인터페이스

✓ 페이징/정렬

● Pageable 인터페이스

◆ 페이징 테스트

```
        //데이터 순회
        for (Memo memo : result.getContent()) {
            System.out.println(memo);
        }
    }
```

Spring JPA

❖ JpaRepository 인터페이스

✓ 페이징/정렬

● Pageable 인터페이스

◆ 정렬 테스트 – mno의 내림차순 정렬

```
@Test
public void testSort() {
    Sort sort1 = Sort.by("mno").descending();
    Pageable pageable = PageRequest.of(0, 10, sort1);
    Page<Memo> result = memoRepository.findAll(pageable);
    result.get().forEach(memo -> {
        System.out.println(memo);
    });
}
```

Spring JPA

❖ JpaRepository 인터페이스

✓ 페이징/정렬

● Pageable 인터페이스

◆ 정렬 테스트 - 결합된 조건

@Test

```
public void testSortConcate() {  
    Sort sort1 = Sort.by("mno").descending();  
    Sort sort2 = Sort.by("memoText").ascending();  
    Sort sortAll = sort1.and(sort2); //and를 이용한 연결  
    Pageable pageable = PageRequest.of(0, 10, sortAll); //결합된 정렬 조건  
    사용  
  
    Page<Memo> result = memoRepository.findAll(pageable);  
    result.get().forEach(memo -> {  
        System.out.println(memo);  
    });  
}
```

Spring JPA

❖ JpaRepository 인터페이스

✓ SQL 실행

- JpaRepository 가 제공하는 findById 나 findAll 메서드를 이용한 조회
- Query Method: 메서드의 이름 자체가 쿼리의 구문으로 처리되는 기능을 이용해서 메서드를 생성해서 조회
- @Query: SQL과 유사하게 Entity Class의 정보를 이용해서 쿼리를 작성하는 기능으로 Native SQL 사용 가능
- Querydsl: 동적 쿼리(상황에 따라 조건이 변경되는 쿼리) 처리

Spring JPA

❖ JpaRepository 인터페이스

✓ SQL 실행

● Query Methods

- ◆ <https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#jpa.query-methods>
- ◆ 쿼리 메서드는 메서드의 이름 자체가 질의(query)문이 되는 기능
- ◆ 쿼리 메서드에서 조회하는 메서드의 이름 규칙
 - find + (Entity 이름) + By + 변수이름
 - Entity 이름은 생략 가능
 - By 이후에 필요한 필드 조건이나 And, Or와 같은 키워드를 이용해서 메서드의 이름을 질의 조건을 이용해서 생성
 - Item Entity에서 num을 가지고 조회하는 경우 - findByNum
- ◆ 쿼리 메서드의 관련 키워드는 SQL에서 사용되는 키워드와 동일하게 작성
- ◆ 쿼리 메서드는 사용하는 키워드에 따라서 파라미터의 개수를 결정
- ◆ 리턴 타입
 - select의 결과는 List 타입이나 배열
 - select 에서 파라미터에 Pageable 타입을 넣는 경우에는 Page<E>
 - select 이외의 작업은 void

Spring JPA

❖ JpaRepository 인터페이스

✓ SQL 실행

● Query Methods

◆ 쿼리 메서드 Sample 및 JPQL snippet

Keyword	Sample	JPQL snippet
Distinct	<code>findDistinctByLastnameAndFirstname</code>	<code>select distinct ... where x.lastname = ?1 and x.firstname = ?2</code>
And	<code>findByLastnameAndFirstname</code>	<code>... where x.lastname = ?1 and x.firstname = ?2</code>
Or	<code>findByLastnameOrFirstname</code>	<code>... where x.lastname = ?1 or x.firstname = ?2</code>
Is , Equals	<code>findByFirstname</code> , <code>findByFirstnameIs</code> , <code>findByFirstnameEquals</code>	<code>... where x.firstname = ?1</code>
Between	<code>findByStartDateBetween</code>	<code>... where x.startDate between ?1 and ?2</code>
LessThan	<code>findByAgeLessThan</code>	<code>... where x.age < ?1</code>
LessThanEqual	<code>findByAgeLessThanEqual</code>	<code>... where x.age <= ?1</code>
GreaterThan	<code>findByAgeGreaterThan</code>	<code>... where x.age > ?1</code>
GreaterThanEqual	<code>findByAgeGreaterThanEqual</code>	<code>... where x.age >= ?1</code>

Spring JPA

❖ JpaRepository 인터페이스

✓ SQL 실행

● Query Methods

◆ 쿼리 메서드 Sample 및 JPQL snippet

After	findByStartDateAfter	... where x.startDate > ?1
Before	findByStartDateBefore	... where x.startDate < ?1
IsNull, Null	findByAge(Is)Null	... where x.age is null
IsNotNull, NotNull	findByAge(Is)NotNull	... where x.age not null
Like	findByFirstnameLike	... where x.firstname like ?1
NotLike	findByFirstnameNotLike	... where x.firstname not like ?1
StartingWith	findByFirstnameStartingWith	... where x.firstname like ?1 (parameter bound with appended %)
EndingWith	findByFirstnameEndingWith	... where x.firstname like ?1 (parameter bound with prepended %)
Containing	findByFirstnameContaining	... where x.firstname like ?1 (parameter bound wrapped in %)
OrderBy	findByAgeOrderByLastnameDesc	... where x.age = ?1 order by x.lastname desc

Spring JPA

❖ JpaRepository 인터페이스

✓ SQL 실행

● Query Methods

◆ 쿼리 메서드 Sample 및 JPQL snippet

Not	<code>findByLastnameNot</code>	<code>... where x.lastname <> ?1</code>
In	<code>findByAgeIn(Collection<Age> ages)</code>	<code>... where x.age in ?1</code>
NotIn	<code>findByAgeNotIn(Collection<Age> ages)</code>	<code>... where x.age not in ?1</code>
True	<code>findByActiveTrue()</code>	<code>... where x.active = true</code>
False	<code>findByActiveFalse()</code>	<code>... where x.active = false</code>
IgnoreCase	<code>findByFirstnameIgnoreCase</code>	<code>... where UPPER(x.firstname) = UPPER(?1)</code>

Spring JPA

❖ JpaRepository 인터페이스

✓ SQL 실행

● Query Methods

◆ Memo 객체의 mno 값이 70 부터 80 사이의 객체를 검색하고 mno의 역순으로 정렬

- JpaRepository 인터페이스에 메서드 추가

```
List<Memo> findByMnoBetweenOrderByMnoDesc(Long from, Long to);
```

- Test Class에 테스트 메서드 추가

```
@Test
```

```
public void testQueryMethods(){
```

```
    List<Memo> list =
```

```
    memoRepository.findByMnoBetweenOrderByMnoDesc(70L,80L);
```

```
    for (Memo memo : list) {
```

```
        System.out.println(memo);
```

```
    }
```

```
}
```

Spring JPA

❖ JpaRepository 인터페이스

✓ SQL 실행

● Query Methods

- ◆ Memo 객체의 mno 값이 10 부터 50 사이의 객체를 내림차순 정렬해서 검색하고 페이지징

- JpaRepository 인터페이스에 메서드 추가

```
Page<Memo> findByMnoBetween(Long from, Long to, Pageable  
pageable);
```

- Test Class에 테스트 메서드 추가

```
@Test
```

```
public void testQueryMethodsPaging(){  
    Pageable pageable = PageRequest.of(0, 10,  
    Sort.by("mno").descending());
```

```
    Page<Memo> result =  
    memoRepository.findByMnoBetween(10L,50L, pageable);  
  
    result.get().forEach(memo -> System.out.println(memo));  
}
```

Spring JPA

❖ JpaRepository 인터페이스

✓ SQL 실행

● Query Methods

◆ deleteBy 로 시작하는 메서드를 이용한 mno 가 10보다 작은 데이터 삭제

- JpaRepository 인터페이스에 메서드 추가

```
void deleteMemoByMnoLessThan(Long num);
```

- Test Class에 테스트 메서드 추가

```
//작업을 완료하기 위해서 설정
```

```
@Commit
```

```
//설정하지 않으면 에러
```

```
@Transactional
```

```
@Test
```

```
public void testDeleteQueryMethods() {  
    memoRepository.deleteMemoByMnoLessThan(10L);  
}
```

Spring JPA

❖ JpaRepository 인터페이스

✓ SQL 실행

● @Query

- ◆ Spring Data JPA가 제공하는 쿼리 메서드는 검색과 같은 기능을 작성할 때 편리함을 제공하기는 하지만 조인이나 복잡한 조건을 처리해야 하는 경우에는 And, Or 등이 사용되면서 불편할 때가 많음
- ◆ @Query의 경우는 메서드의 이름과 상관없이 메서드에 추가한 annotation을 통해서 원하는 처리가 가능
- ◆ @Query의 value는 JPQL(Java Persistence Query Language)로 작성하는데 객체 지향 쿼리 라고 함
- ◆ @Query를 이용해서는 가능한 작업
 - 필요한 데이터만 선별적으로 추출
 - 데이터베이스에 맞는 순수한 SQL(Native SQL)
 - insert, update, delete 와 같은 select가 아닌 DML 처리 - @Modifying 과 같이 사용

Spring JPA

❖ JpaRepository 인터페이스

✓ SQL 실행

● @Query

◆ 작성

- 객체 지향 쿼리는 Table 대신에 Entity Class를 이용하고 Table의 칼럼 대신에 Class에 선언된 필드를 이용해서 작성
- JPQL은 SQL과 상당히 유사하기 때문에 간단한 기능을 제작하는 경우에는 추가적인 학습 없이도 적용 가능
- mno의 역순으로 정렬하라는 기능을 @Query를 이용해서 제작하면 다음과 같은 형태가 됨

```
@Query("select m from Memo m order by m.mno desc")
```

```
List<Memo> getListDesc();
```

- JPQL이 SQL과 유사하듯이 실제 SQL에서 사용되는 함수들도 JPQL에서 동일하게 사용하는데 avg(), count(), group by, order by 등 SQL에서 사용한 익숙한 구문들을 사용할 수 있음

Spring JPA

❖ JpaRepository 인터페이스

✓ SQL 실행

● @Query

◆ 파라미터 바인딩

- ?1, ?2 와 같이 인덱스를 이용해서 파라미터의 순서를 설정하는 방식 – 인덱스는 1부터 시작
- @Param을 이용해서 이름을 설정하고 :xxx 와 같이 이름을 활용하는 방식
- #{ } 과 같이 자바 빈 스타일을 이용하는 방식

Spring JPA

❖ JpaRepository 인터페이스

✓ SQL 실행

● @Query

◆ 파라미터 바인딩 – 수정하는 메서드를 MemoRepository 인터페이스에 작성

```
@Transactional
```

```
@Modifying
```

```
@Query("update Memo m set m.memoText = :memoText where m.mno  
= :mno ")
```

```
int updateMemoText(@Param("mno") Long mno, @Param("memoText")  
String memoText );
```

```
@Transactional
```

```
@Modifying
```

```
@Query("update Memo m set m.memoText = :#{#param.memoText} where  
m.mno = :#{#param.mno} ")
```

```
int updateMemoText(@Param("param") Memo memo );
```

Spring JPA

❖ JpaRepository 인터페이스

✓ SQL 실행

● @Query

◆ 파라미터 바인딩 – 수정하는 메서드를 테스트 클래스에서 테스트

@Test

```
public void testUpdateQuery(){  
    System.out.println(memoRepository.updateMemoText(11L, "@Query를  
    이용한 수정");  
    System.out.println(memoRepository.updateMemoText(Memo.builder().mn  
    o(12L).memoText("@Query를 이용한 수정").build()));  
}
```

Spring JPA

❖ JpaRepository 인터페이스

✓ SQL 실행

● @Query

◆ 페이징 처리

- @Query를 이용할 때는 별도의 countQuery라는 속성에 데이터 개수를 조회하는 쿼리를 작성하고 Pageable 타입의 파라미터를 전달하면 됨

```
@Query(value = "select m from Memo m where m.mno > :mno",  
      countQuery = "select count(m) from Memo m where  
      m.mno > :mno" )
```

```
Page<Memo> getListWithQuery(@Param("mno") Long mno, Pageable  
    pageable);
```

Spring JPA

❖ JpaRepository 인터페이스

✓ SQL 실행

● @Query

◆ 페이징 처리

```
@Test
public void testSelectQuery(){
    Pageable pageable = PageRequest.of(0, 10,
    Sort.by("mno").descending());
    Page<Memo> page = memoRepository.getListWithQuery(50L,
    pageable );
    for(Memo memo : page){
        System.out.println(memo);
    }
}
```

Spring JPA

❖ JpaRepository 인터페이스

✓ SQL 실행

● @Query

◆ Object [] 리턴

- 쿼리 메서드의 경우에는 Entity 타입의 데이터만을 추출하지만 @Query를 이용하는 경우에는 현재 필요한 데이터만을 Object [] 의 형태로 선별적으로 추출 가능
- JPQL을 이용해서 JOIN이나 GROUP BY 등을 이용하는 경우가 있는데 그럴 때는 적당한 Entity 타입이 존재하지 않는 경우가 많기 때문에 이런 상황에서 유용
- mno와 memoText 그리고 현재 시간을 같이 얻어오고 싶다면 Memo Entity Class에는 시간 관련된 부분의 선언이 없기 때문에 추가적인 구문이 필요
- JPQL에서는 CURRENT_DATE, CURRENT_TIME, CURRENT_TIMESTAMP와 같은 구문을 통해서 현재 데이터베이스의 시간을 구할 수 있는데 이를 적용하면 다음과 같은 형태로 가능

```
@Query(value = "select m.mno, m.memoText, CURRENT_DATE from  
Memo m where m.mno > :mno", countQuery = "select count(m)  
from Memo m where m.mno > :mno" )
```

```
Page<Object[]> getListWithQueryObject(@Param("mno") Long mno,  
Pageable pageable);
```

Spring JPA

❖ JpaRepository 인터페이스

✓ SQL 실행

● @Query

◆ Object [] 리턴

▪ 메서드 테스트

@Test

```
public void testSelectQueryObjectReturn(){
    Pageable pageable = PageRequest.of(0, 10,
    Sort.by("mno").descending());
    Page<Object []> page =
    memoRepository.getListWithQueryObject(50L, pageable );
    for(Object [] ar : page){
        System.out.println(Arrays.toString(ar));
    }
}
```

Spring JPA

❖ JpaRepository 인터페이스

✓ SQL 실행

● @Query

◆ Native SQL

- JPA 자체가 데이터베이스에 독립적으로 구현이 가능하다는 장점을 잃어버리기는 하지만 경우에 따라서는 복잡한 JOIN 구문 등을 처리하기 위해서 어쩔 수 없는 선택을 하는 경우에 사용

```
@Query(value = "select * from tbl_memo where mno > 0",  
      nativeQuery = true)
```

```
List<Object[]> getNativeResult();
```

- @Query의 nativeQuery 속성 값을 true로 지정하고 일반 SQL을 그대로 사용할 수 있음

```
@Test
```

```
public void testSelectNativeQuery(){
```

```
    List<Object []> list = memoRepository.getNativeResult();
```

```
    for(Object [] ar : list){
```

```
        System.out.println(Arrays.toString(ar));
```

```
    }
```

```
}
```

Spring JPA

- ❖ Oracle 로 변경
 - ✓ 오라클의 의존성 확인
 - build.gradle 파일의 dependencies에 추가
runtimeOnly 'com.oracle.database.jdbc:ojdbc8'



Spring JPA

❖ Oracle 로 변경

- ✓ application.yml 파일의 데이터베이스 접속 위치 변경

spring:

datasource:

#url: jdbc:mariadb://localhost:3306/adam

#driver-class-name: org.mariadb.jdbc.Driver

#username: root

#password: wnddkd

url: jdbc:oracle:thin:@localhost:1521:xe

driver-class-name: oracle.jdbc.driver.OracleDriver

username: system

password: oracle

MyBatis

❖ MyBatis 사용

✓ 의존성 확인

- build.gradle 의 dependencies 에 추가

implementation 'org.mybatis.spring.boot:mybatis-spring-boot-starter:3.0.1'



MyBatis

❖ MyBatis 사용

- ✓ dto 패키지에 MemoDTO 클래스 생성

@ToString

@Getter

@Builder

@AllArgsConstructor

@NoArgsConstructor

```
public class MemoDTO {  
    private Long mno;  
    private String memoText;  
}
```

MyBatis

❖ MyBatis 사용

- ✓ mybatis.mapper 패키지에 MemoMapper 인터페이스 생성

```
@Repository
public interface MemoMapper {
    @Select("select * from tbl_memo")
    public List<MemoDTO> listMemo();
}
```

MyBatis

❖ MyBatis 사용

- ✓ config 패키지에 MyBatisConfig 클래스 생성

```
@Configuration
@EnableTransactionManagement
@MapperScan(basePackages= {"com.kakao.jpa.mybatis.mapper"})
public class MyBatisConfig {
    @Bean
    public DataSource batisDataSource() {
        HikariConfig hikariConfig = new HikariConfig();
        hikariConfig.setDriverClassName("org.mariadb.jdbc.Driver");
        hikariConfig.setJdbcUrl("jdbc:mariadb://localhost:3306/adam");
        hikariConfig.setUsername("root");
        hikariConfig.setPassword("wnddkd");
        hikariConfig.setMinimumIdle(5);
        hikariConfig.setPoolName("springHikariCP");
        hikariConfig.addDataSourceProperty("dataSource.cachePrepStmts", "true");
        hikariConfig.addDataSourceProperty("dataSource.prepStmtCacheSize", "200");
        hikariConfig.addDataSourceProperty("dataSource.prepStmtCacheSqlLimit", "2048");
        hikariConfig.addDataSourceProperty("dataSource.useServerPrepStmts", "true");
        HikariDataSource dataSource = new HikariDataSource(hikariConfig);
        return dataSource;
    }
}
```

MyBatis

❖ MyBatis 사용

- ✓ config 패키지에 MyBatisConfig 클래스 생성

@Bean

```
public SqlSessionFactory batisSqlSessionFactory(@Qualifier("batisDataSource")
DataSource batisDataSource, ApplicationContext applicationContext) throws
Exception {
    SqlSessionFactoryBean sqlSession = new SqlSessionFactoryBean();
    sqlSession.setDataSource(batisDataSource);
    return sqlSession.getObject();
}
```

}

@Bean

```
public SqlSessionTemplate batisSqlSessionTemplate(SqlSessionFactory
batisSqlSessionFactory) throws Exception {
    return new SqlSessionTemplate(batisSqlSessionFactory);
}
```

}

MyBatis

❖ MyBatis 사용

- ✓ MyBatisTest 클래스에 테스트 메서드를 만들어서 확인

```
@SpringBootTest
public class MyBatisTest {
    @Autowired
    MemoMapper memoMapper;

    @Test
    public void testMyBatis(){
        System.out.println(memoMapper);
        System.out.println(memoMapper.listMemo());
    }
}
```

Test

❖ Service 계층 테스트

- ✓ service 패키지에 Service 인터페이스를 생성하고 메서드 선언

```
public interface MemoService {  
    public MemoDTO saveMemo(MemoDTO memoDTO);  
}
```

Test

❖ Service 계층 테스트

- ✓ service 패키지에 ServiceImpl 클래스를 생성하고 메서드 구현

```
@Service
@RequiredArgsConstructor
public class MemoServiceImpl implements MemoService {
    private final MemoRepository memoRepository;

    @Override
    public MemoDTO saveMemo(MemoDTO memoDTO) {
        Memo memo =
Memo.builder().memoText(memoDTO.getMemoText()).build();
        Memo saveMemo = memoRepository.save(memo);

        MemoDTO result =
MemoDTO.builder().mno(saveMemo.getMno()).memoText(saveMemo.getMemoText()).b
uild();
        return result;
    }
}
```

Test

❖ Service 계층 테스트

- ✓ src/test/java 디렉토리에 Service 테스트를 위한 클래스를 생성하고 메서드 구현 후 테스트 수행

```
@SpringBootTest
public class ServiceTest {
    @Autowired
    MemoService memoService;

    @Test
    public void testSave(){
        MemoDTO memoDTO = MemoDTO.builder().memoText("서비스
테스트").build();
        MemoDTO result = memoService.saveMemo(memoDTO);
        System.out.println(result);
    }
}
```

Test

❖ Controller 계층 테스트

- ✓ build.gradle 파일의 implementation 영역에 gson 라이브러리의 의존성 설정
implementation 'com.google.code.gson:gson:2.9.0'



Test

❖ Controller 계층 테스트

- ✓ controller 패키지에 MemoController 클래스를 생성하고 작성

```
@RestController
@RequiredArgsConstructor
public class MemoController {
    private final MemoService memoService;

    @PostMapping("/save")
    public ResponseEntity<MemoDTO> saveMemo(@RequestBody MemoDTO
memoDTO){
        MemoDTO result = memoService.saveMemo(memoDTO);
        return ResponseEntity.status(HttpStatus.OK).body(result);
    }
}
```

Test

❖ Controller 계층 테스트

- ✓ src/test/java 디렉토리에 Controller 테스트를 위한 클래스를 생성하고 메서드 구현 후 테스트 수행

```
@WebMvcTest(MemoController.class)
```

```
public class ControllerTest {
```

```
    @Autowired
```

```
    private MockMvc mockMvc;
```

```
//Controller에서 잡고 있는 Bean 객체에 대해 Mock 형태의 객체를 생성해줌
```

```
@MockBean
```

```
MemoService memoService;
```

Test

❖ Controller 계층 테스트

- ✓ src/test/java 디렉토리에 Controller 테스트를 위한 클래스를 생성하고 메서드 구현 후 테스트 수행

```
@Test
void createMemoTest() throws Exception {
    MemoDTO memoDTO = MemoDTO.builder()
        .memoText("Controller 테스트")
        .build();

    Gson gson = new Gson();
    String content = gson.toJson(memoDTO);

    mockMvc.perform(
        post("/save")
            .content(content)
            .contentType(MediaType.APPLICATION_JSON))
        .andExpect(status().isOk())
        .andDo(print());
}
```

Mongo DB

- ❖ build.gradle 파일에 Mongo DB 의존성 추가
implementation 'org.springframework.boot:spring-boot-starter-data-mongodb'



Mongo DB

- ❖ application.yml 파일을 생성하고 데이터베이스 접속 정보 작성

```
spring:  
  data:  
    mongodb:  
      host: localhost  
      port: 27017  
      database: adam
```



Mongo DB

❖ Entity 클래스 생성 – domain.Board

```
import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;

import lombok.Data;

@Data
@Document(collection = "board")
public class Board {

    @Id
    private String _id;
    private String title;
    private String content;
}
```

Mongo DB

- ❖ Repository 인터페이스 생성 – persistence.BoardRepository

```
import org.springframework.data.mongodb.repository.MongoRepository;  
  
public interface BoardRepository extends MongoRepository<Board, String>{  
  
}
```

Mongo DB

❖ DTO 클래스 생성 – dto.BoardDTO

```
import lombok.Data;
@Data
public class BoardDTO {
    private String title;
    private String content;

    public Board toEntity() {
        Board board = new Board();
        board.setTitle(title);
        board.setContent(content);
        return board;
    }
}
```

Mongo DB

❖ Controller 클래스 생성 – controller.BoardController

```
import java.util.List;
```

```
import org.springframework.web.bind.annotation.DeleteMapping;  
import org.springframework.web.bind.annotation.GetMapping;  
import org.springframework.web.bind.annotation.PathVariable;  
import org.springframework.web.bind.annotation.PostMapping;  
import org.springframework.web.bind.annotation.PutMapping;  
import org.springframework.web.bind.annotation.RequestBody;  
import org.springframework.web.bind.annotation.RestController;
```

```
import lombok.RequiredArgsConstructor;
```

```
@RequiredArgsConstructor // DI  
@RestController // 데이터 리턴 서버  
public class BoardController {
```

```
    // DI  
    private final BoardRepository boardRepository;
```

Mongo DB

❖ Controller 클래스 생성 – controller.BoardController

```
@PostMapping("board/{id}")
public void update(@RequestBody BoardDTO dto, @PathVariable String id) {

    Board board = dto.toEntity();
    board.set_id(id); // save함수는 같은 아이디면 수정한다.

    boardRepository.save(board);
}

@DeleteMapping("board/{id}")
public int deleteById(@PathVariable String id) {

    boardRepository.deleteById(id); // 내부적으로 실행되다가 오류 Exception 발동

    return 1; // 1 : 성공, -1 : 실패
}

@GetMapping("/board/{id}")
public Board findById(@PathVariable String id) {
    return boardRepository.findById(id).get();
}
```

Mongo DB

❖ Controller 클래스 생성 – controller.BoardController

```
@GetMapping("/board")
public List<Board> findAll() { // 리턴을 JavaObject로 하면 스프링 내부적으로 JSON으로
    자동 변환 해준다.
    return boardRepository.findAll();
}

@PostMapping("/board")
public Board save(@RequestBody BoardDTO dto) { // {"title":"제목3","content":"내용3"}
    Board boardEntity = boardRepository.save(dto.toEntity());

    return boardEntity; // MessageConverter 발동 -> 자바오브제그를 Json 변환해서 응답함.
}
}
```

Mongo DB

❖ POSTMAN 등을 이용해서 테스트



Redis

❖ Redis 사용

- ✓ build.gradle 파일에 의존성 설정

implementation 'org.springframework.boot:spring-boot-starter-data-redis'

- ✓ application.yml 파일에 redis 접속 서버 설정

spring:

redis:

lettuce:

pool:

max-active: 5 # pool에 할당될 수 있는 커넥션 최대수

max-idle: 5 # pool의 'idle' 커넥션 최대수

min-idle: 2

host: localhost

port: 6379

Redis

❖ Redis 사용

✓ redis 접속 옵션

● 변수	기본값	설명
● spring.redis.database 인덱스	0	커넥션 팩토리에 사용되는 데이터베이스
● spring.redis.host	localhost	레디스 서버 호스트
● spring.redis.password		레디스 서버 로그인 비밀번호
● spring.redis.pool.max-active (음수로 하면 무제한)	8	pool에 할당될 수 있는 커넥션 최대수
● spring.redis.pool.max-idle 무제한)	8	pool의 "idle" 커넥션 최대수 (음수로 하면 무제한)
● spring.redis.pool.max-wait 할당 차단 최대 시간 (단위 밀리세컨드, 음수는 무제한 차단)	-1	pool이 바닥났을 때 예외 발생 전, 커넥션 할당 차단 최대 시간 (단위 밀리세컨드, 음수는 무제한 차단)
● spring.redis.pool.min-idle (양수일 때만 유효)	0	풀에서 관리하는 idle 커넥션의 최소수 대상 (양수일 때만 유효)
● spring.redis.port	6379	레디스 서버 포트
● spring.redis.sentinel.master		레디스 서버 이름
● spring.redis.sentinel.nodes		호스트: 포트 쌍 목록 (콤마로 구분)
● spring.redis.timeout	0	커넥션 타임아웃 (단위 밀리세컨드)

Redis

❖ Redis 사용

✓ Redis 설정 클래스 생성 및 작성

```
@Configuration
@EnableRedisRepositories
@RequiredArgsConstructor
public class RedisConfig {

    private final RedisProperties redisProperties;

    @Bean
    public RedisConnectionFactory redisConnectionFactory() {
        return new LettuceConnectionFactory(redisProperties.getHost(),
            redisProperties.getPort());
    }

    @Bean
    public RedisTemplate<String, Object> redisTemplate() {
        RedisTemplate<String, Object> redisTemplate = new RedisTemplate<>();
        redisTemplate.setConnectionFactory(redisConnectionFactory());
        redisTemplate.setKeySerializer(new StringRedisSerializer());
        redisTemplate.setValueSerializer(new StringRedisSerializer());
        return redisTemplate;
    }
}
```

Redis

❖ Redis 사용

✓ Redis 설정 클래스 생성 및 작성

- RedisConnectionFactory 인터페이스를 통해 LettuceConnectionFactory를 생성하여 반환
- setKeySerializer, setValueSerializer 설정해주는 이유는 RedisTemplate를 사용할 때 Spring - Redis 간 데이터 직렬화, 역직렬화 시 사용하는 방식이 JDK 직렬화 방식이기 때문인데 동작에는 문제가 없지만 redis-cli을 통해 직접 데이터를 보려고 할 때 알아볼 수 없는 형태로 출력되기 때문에 적용한 설정
- RedisProperties를 통해서 properties에 저장한 host, port를 가지고 와서 연결

Redis

❖ Redis 사용

- ✓ Controller 클래스를 만들고 테스트

```
@RestController
@RequiredArgsConstructor
public class RedisController {
    private final RedisTemplate<String, String> redisTemplate;

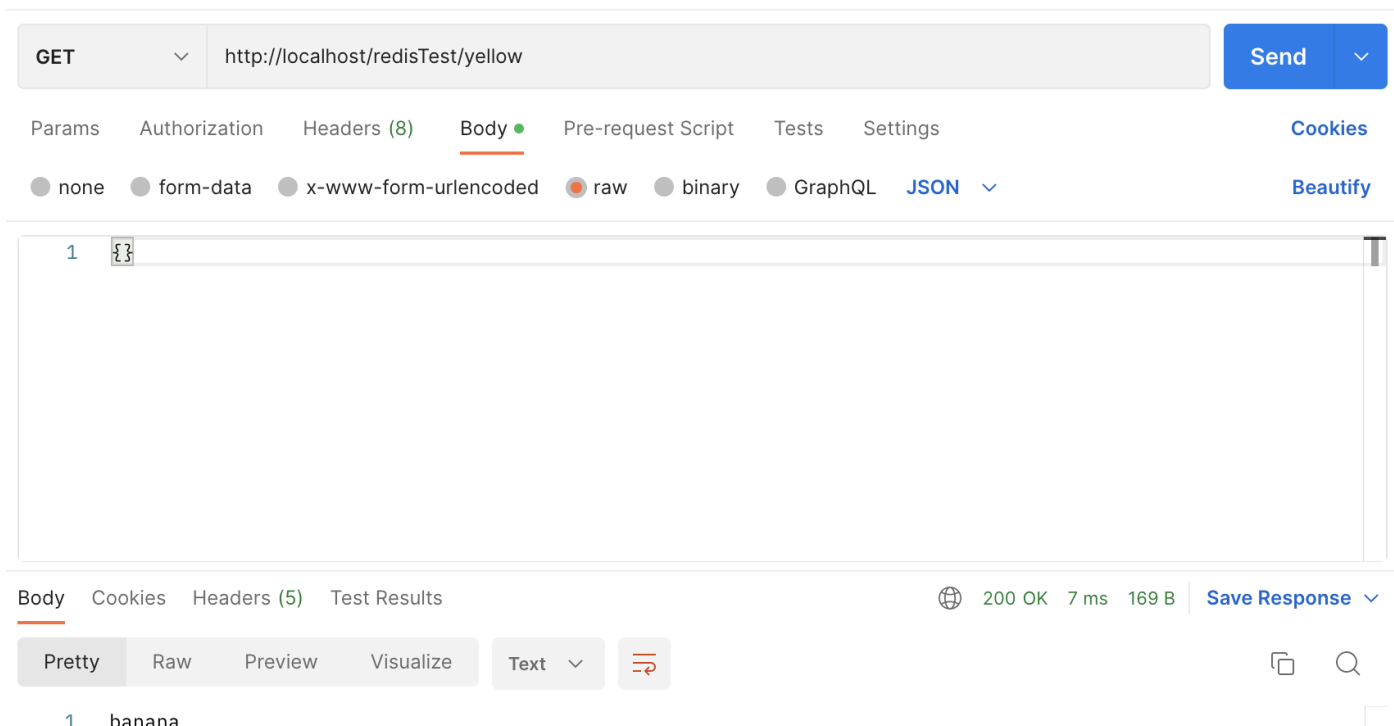
    @PostMapping("/redisTest")
    public ResponseEntity<?> addRedisKey() {
        System.out.println(redisTemplate);
        ValueOperations<String, String> vop = redisTemplate.opsForValue();
        vop.set("yellow", "banana");
        vop.set("red", "apple");
        vop.set("green", "watermelon");
        return new ResponseEntity<>(HttpStatus.CREATED);
    }

    @GetMapping("/redisTest/{key}")
    public ResponseEntity<?> getRedisKey(@PathVariable String key) {
        ValueOperations<String, String> vop = redisTemplate.opsForValue();
        String value = vop.get(key);
        return new ResponseEntity<>(value, HttpStatus.OK);
    }
}
```

Redis

❖ Redis 사용

- ✓ Controller 클래스를 만들고 테스트



```
redis:6379> get "yellow"
"banana"
```

Redis

❖ Redis 사용

✓ 토큰 저장

```
String refreshToken = jwtTokenProvider.createRefreshToken();
```

```
ValueOperations<String, String> valueOperations = redisTemplate.opsForValue();
```

```
valueOperations.set(rRTKey + user.getIdx(), refreshToken);
```

```
log.info("redis RT : {}", valueOperations.get(rRTKey + user.getIdx()));
```

- Service 계층에서 사용하였고 RedisTemplate을 생성자 주입 방식으로 추가하여 사용
- 프로젝트에서 로그인 시 RefreshToken을 cache 서버인 Redis에 저장할 용도로 사용하였고, 'rRTKey'라는 값은 따로 정의하여 가져온 String 값

Redis

❖ Redis 사용

✓ 자료 구조에 따른 메서드

- opsForValue Strings를 쉽게 Serialize / Deserialize
- opsForList List를 쉽게 Serialize / Deserialize
- opsForSet Set을 쉽게 Serialize / Deserialize
- opsForZSet ZSet을 쉽게 Serialize / Deserialize
- opsForHash Hash를 쉽게 Serialize / Deserialize

Redis

❖ 객체 저장

✓ DTO 클래스 생성 – dto.Member

```
@Getter
@Setter
@RedisHash("member")
public class Member {

    @Id
    private String id;
    private String name;
    private int age;

    public Member(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```

- Redis를 통해 저장할 Member Class
- @RedisHash 어노테이션을 통해 설정한 값을 Redis의 key 값 prefix로 사용
- @Id 어노테이션은 JPA와 동일한 역할을 수행하는데 member:{id} 의 위치에 자동으로 generate 값이 대입됨

Redis

❖ 객체 저장

- ✓ Repository 인터페이스 생성 – persistence.MemberRedisRepository

```
public interface MemberRedisRepository extends CrudRepository<Member, String> {  
}
```

Redis

❖ 객체 저장

- ✓ Service 클래스 생성 – service.MemberService

@RequiredArgsConstructor

@Service

public class MemberService {

private final MemberRedisRepository memberRedisRepository;

public void addMember() {

Member member = new Member("jan", 99);

memberRedisRepository.save(member);

}

}

Redis

❖ 객체 저장

- ✓ Controller 클래스에 처리 코드 추가

```
private final MemberService memberService;

@GetMapping("/addmember")
public ResponseEntity<?> addMember() {
    memberService.addMember();
    return new ResponseEntity<>("success", HttpStatus.OK);
}
```

Redis

- ❖ 객체 저장
- ✓ 확인

The screenshot shows a REST client interface with the URL `http://localhost/addmember`. The request method is `GET`. The `Body` tab is selected, showing an empty JSON object `{}`. The response status is `200 OK` with a response time of `298 ms` and a size of `170 B`. The response body is `success`.

URL: `http://localhost/addmember`

Method: `GET`

Body: `{}`

Response: `200 OK` (298 ms, 170 B)

Response Body: `success`

Redis

- ❖ 객체 저장
 - ✓ 확인

```
[redis:6379> keys *
1) "name"
2) "key5"
3) "member"
4) "green"
5) "key1"
6) "member:0c0f8007-00f4-4a28-8a0e-02c06615e376"
7) "key"
8) "red"
9) "a"
10) "yellow"
11) "key3"
12) "key2"
13) "b"
[redis:6379> hgetall member:0c0f8007-00f4-4a28-8a0e-02c06615e376
1) "_class"
2) "com.adamsoft.jpa.dto.Member"
3) "age"
4) "99"
5) "id"
6) "0c0f8007-00f4-4a28-8a0e-02c06615e376"
7) "name"
8) "jan"
redis:6379> █
```