개념

@SpringBootApplication

스프링 부트의 자동 설정, 스프링 Bean 읽기와 생성을 모두 자동으로 설정

이 어노테이션이 있는 위치부터 설정을 읽어가기 때문에 이 클래스는 항상 프로젝트의 최상단에 위치하여야 한다

main 메서드에서 실행하는 SpringBootApplication.run으로 인해 내장 WAS(Web application server, 웹 애플리케이션 서버)를 실행한다.

내장 WAS란 별도로 외부에 WAS를 두지 않고 애플리케이션을 실행할 때 내부에서 WAS를 실행하는 것이다.

@Autowired

스프링이 관리하는 빈(Bean)을 주입 받는다.

@RequestParam

외부에서 API로 넘긴 파라미터를 가져오는 어노테이션이다.

여기선 외부에서 name(@RequestParam("name"))이란 이름으로 넘긴 파라미터를 메소드 파라미터 name(String name)에 저장하게 된다.

```
@GetMapping("/hello/dto")
  public HelloResponseDto helloDto(@RequestParam("name") String name,
                                   @RequestParam("amount") int amount){
        return new HelloResponseDto(name, amount);
 }
```

@Entity

테이블과 링크될 클래스임을 나타낸다.

기본값으로 클래스의 카멜케이스 이름을 언더스코어 네이밍(_)으로 테이블 이름을 매칭한다.

@id

해당 테이블의 PK필드를 나타낸다.

@GeneratedValue

PK의 생성 규칙을 나타낸다.

스프링 부트 2.0버전에서는 GenerationType.IDENTITY 옵션을 추가해야만 auto_increment가 된다.

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

@Column

테이블의 칼럼을 나타내며, 굳이 선언하지 않아도 해당 클래스의 필드는 모두 칼럼이 된다.

사용하는 이유는 기본값 외에 추가로 변경이 필요한 옵션이 있으면 사용한다.

문자열의 경우 VARCHAR(255)가 기본값인데, 사이즈를 500으로 늘리고 싶거나, 타입을 TEXT로 변경하고 싶거나 등의 경우에 사용된다.

```
@Column(length = 500, nullable = false)
  private String title;
@Column(columnDefinition = "TEXT", nullable = false)
  private String content;
```

DB Layer 접근자

JPA에서는 Repository라고 부르며 인터페이스로 생성한다.

단순히 인터페이스를 생성한 후, JpaRepository<Entity 클래스, PK 타입>를 상속하면 기본적인 CRUD 메서드가 자동으로 생성된다.

Entity클래스와 기본 Entity Repository는 함께 위치해야 한다.

Entity클래스는 기본 Repository없이는 제대로 역할 수행이 불가능하다.

save

테이블에 insert/update 쿼리를 실행한다.

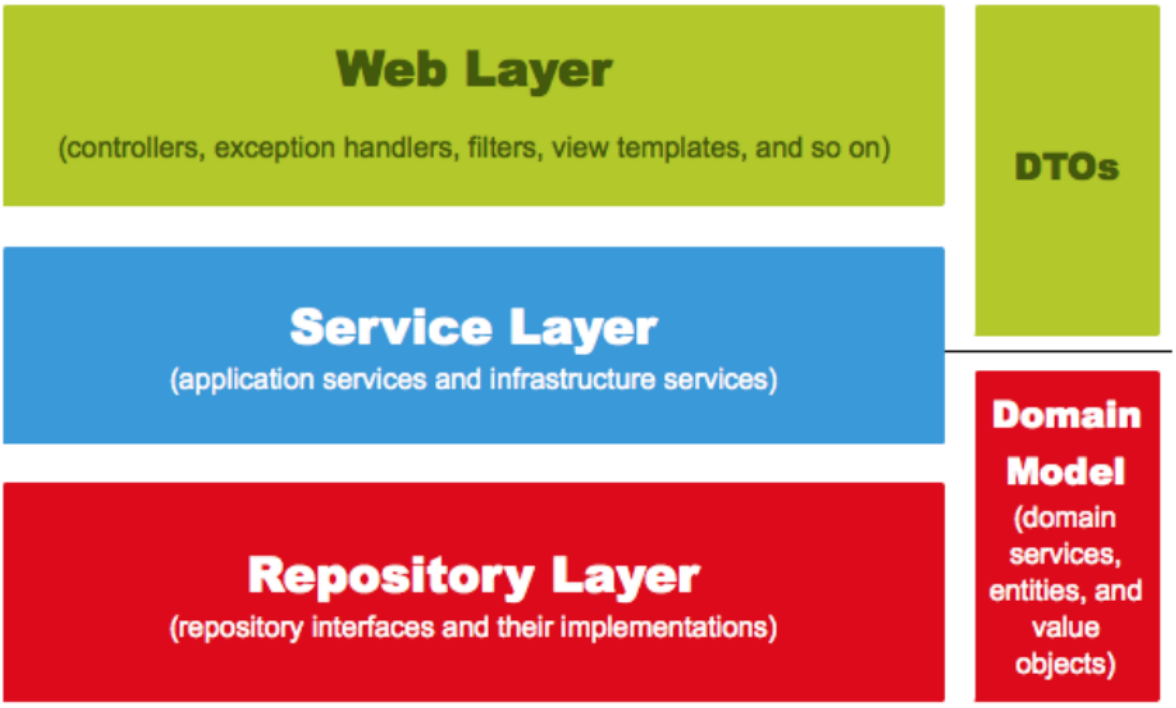
id 값이 있다면 update가, 없다면 insert가 수행이 된다.

```
postsRepository.save(Posts.builder()
                        .title(title)
                        .content(content)
                        .author("sjk6437@gmail.com")
                        .build());
```

findAll

테이블에 있는 모든 데이터를 조회해오는 메서드이다.

```
List<Posts> postsList = postsRepository.findAll();
```



Web Layer

흔히 사용하는 컨트롤러(Controller)와 JSP/Freemarker 등의 뷰 템플릿 영역이다.

이외에도 필터(@Filter), 인터셉터, 컨트롤러 어드바이스(@ControllerAdvice) 등 외부 요청과 응답에 대한 전반적인 영역을 의미한다.

@Controller

프레젠테이션 레이어, 웹 어플리케이션에서 웹 요청과 응답을 처리하는 클래스에 사용한다.

Service Layer

@Service에 사용되는 서비스 영역이다.

일반적으로 Controller와 Dao의 중간 영역에서 사용된다.

@Transactional이 사용되어야하는 영역이다.

@Service

서비스 레이어, 비즈니스 로직을 가진 클래스에 사용된다.

Repository Layer

Database와 같이 데이터 저장소에 접근하는 영역이다.

Dao(Data Access Object) 영역이라고 생각하면 된다.

ORM (Mybatis, Hibernate)를 주로 사용하는 계층

DAO 인터페이스와 @Repository 어노테이션을 사용하여 작성된 DAO 구현 클래스가 이 계층에 속함

Dabase에 data를 CRUD(Create, Read, Update, Drop)하는 계층

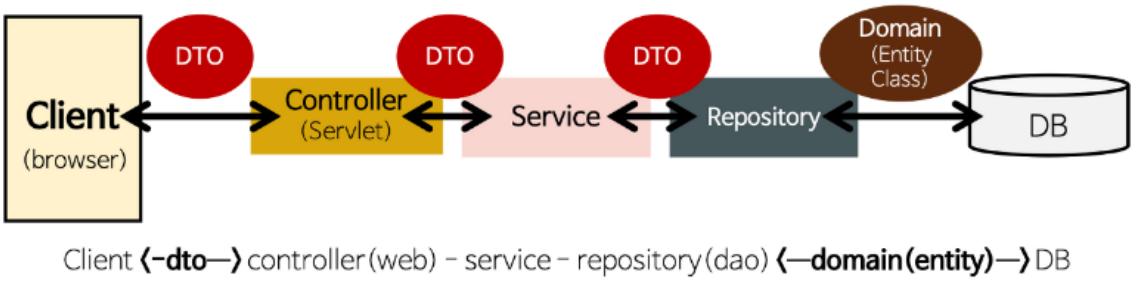
@Repository

퍼시스턴스 레이어, 영속성을 가지는 속성(파일, 데이터베이스)

Dto

Dto(Data Transfer Object)는 계층 간에 데이터 교환을 위한 객체를 의미하며, Dtos는 이들의 영역을 의미한다.

예를 들어 뷰 템플릿 엔진에서 사용될 객체나 Repository Layer에서 결과로 넘겨준 객체 등이 이들을 이야기 한다.



Domain Model

도메인이라 불리는 개발 대상은 모든 사람이 동일한 관점에서 이해할 수 있고 공유할 수 있도록 단순화 시킨 것을 도메인 모델이라 한다.

이를테면 택시 앱이라고 하면 배차 ,탑승, 요금 등이 모두 도메인이 될 수 있다.

@Entity가 사용된 영역 역시 도메인 모델이라고 이해하면 된다.

다만 무조건 데이터 베이스의 테이블과 관계가 있어야 하는 것은 아니다.

Domain 영역에서 비지니스 처리를 담당한다.

모든 로직은 서비스 클래스 내부에서 처리된다. 그러다 보니 서비스 계층이 무의미 하며, 객체란 단순히 데이터 덩어리 역할만 하게 된다 .

반면 도메인 모델에서 처리할 경우

각각 본인의 이벤트 처리를 하며, 서비스 메서드는 트랜잭션과 도메인 간의 순서만 보장을 해준다.

Domain 클래스와 Dto 클래스를 분리하는 이유

- View Layer와 DB Layer의 역할을 철저하게 분리하기 위해서
- 테이블과 매핑되는 Entity 클래스가 변경되면 여러 클래스에 영향을 끼치게 되지만 View와 통신하는 DTO 클래스는 자주 변경되므로 분리해야 한다.
- 즉 DTO는 Domain Model을 복사한 형태로, 다양한 Presentation Logic을 추가한 정도로 사용하며 Domain Model 객체는 Persistent만을 위해서 사용한다.

스프링에서 Bean을 주입받는 방식

@Autowired

setter

생성자

이중 권장하는 방식은 생성자로 주입하는 방식이다.

생성자로 Bean 객체를 받도록 하면 @Autowired와 동일한 효과를 볼 수 있다.

Entity 클래스는 데이터베이스와 맞닿은 핵심 클래스이다.

Entity클래스를 기준으로 테이블이 생성되고, 스키마가 변경된다.

Requset와 Response 용 Dto는 View를 위한 클래스라 자주 변경이 필요하다.

영속성 컨텍스트

엔티티를 영구 저장하는 환경

JPA에 존재한다.

JPA의 핵심 내용은 엔티티가 영속성 컨텍스트에 포함되어 있냐 아니냐로 갈린다.

JPA의 엔티티 매니저가 활성화된 상태로 트랜잭션 안에서 데이터베이스에서 데이터를 가져오면 이 데이터는 영속성 컨텍스트가 유지된 상태이다.

이 상태에서 해당 데이터의 값을 변경하면 트랜잭션이 끝나는 시점에 해당 테이블에 변경분을 반영한다.ㅁ

즉 Entity 객체의 값만 변경하면 별도로 Update쿼리를 날릴 필요가 없다. 이 개념을 더티 체킹이라고 한다.

@MappedSuperclass

JPA Entity 클래스들이 이 클래스를 상속할 경우 필드들도 칼럼으로 인식하게 해준다.

@EntityListeners(AuditingEntityListener.class)

이 클래스에 Auditing 기능을 포함시킨다.

롬복

@Getter

롬복 어노테이션, 선언된 모든 필드의 get 메서드를 생성

@Setter

롬복 어노테이션, 선언된 모든 필드의 set 메서드를 생성

@Data

롬복어노테이션, 선언된 모든 필드의 set, get 메서드를 생성

@RequiredArgsConstructor

롬복 어노테이션, 선언된 모든 final 필드가 포함된 생성자를 생성해준다.

final이 없는 필드는 생성자에 포함되지 않는다.

@Builder

해당 클래스의 빌더 패턴 클래스를 생성

생성자 상단에 선언시 생성자에 포함된 필드만 빌더에 포함한다.

```
public Posts(String title, String content, String author){
        this.title = title;
        this.content = content;
        this.author = author;
    }
```

Test

@WebMvcTest

여러 스트링 테스트 어노테이션 중, Web(Spring MVC)에 집중할 수 있는 어노테이션이다.

선언할 경우 @Controller, @ControllerAdvice 등을 사용할 수 있다.

단, @Service, @Component, @ControllerAdvice등을 사용할 수 없다.

@RunWith(SpringRunner.class)

테스트를 진행할 때 JUnit에 내장된 실행자 외에 다른 실행자를 실행시킨다.

여기서는 SpringRunner라는 스프링 실행자를 사용한다.

즉, 스트링 부트 테스트와 JUnit 사이에 연결자 역할을 한다.

assertThat

assertj라는 테스트 검증 라이브러리의 검증 메소드이다.

검증하고 싶은 대상을 메서드 인자로 받는다.

메서드 체이닝이 지원

isEqualTo

assertj의 동등 비교 메서드이다.

assertThat에 있는 값과 isEqualTo의 값을 비교해서 같을 때만 성공이다.

```
@Test
    public void 롬복_기능_테스트(){
        //given
        String name = "test";
        int amount = 1000;

        //when
        HelloResponseDto dto = new HelloResponseDto(name, amount);

        //then
        assertThat(dto.getName()).isEqualTo(name);
        assertThat(dto.getAmount()).isEqualTo(amount);


   }
```

param

API테스트할 떄 사용될 요청 파라미터를 설정한다.

단, 값은 String만 허용된다.

숫자/날짜 등의 데이터도 등록할 때는 문자열로 변경해야만 가능하다.

jsonPath

JSON 응답값을 필드별로 검증할 수 있는 메서드이다.

$를 기준으로 필드명을 명시한다.

여기서는 name과 amount를 검증하니, $.name, $.amount로 검증한다.

```
@Test
    public void helloDto가_리턴된다() throws Exception{
        String name = "hello";
        int amount = 1000;

        mvc.perform(get("/hello/dto").param("name",name).param("amount", String.valueOf(amount)))
                .andExpect(status().isOk())
                .andExpect(jsonPath("$.name", is(name)))
                .andExpect(jsonPath("$.amount", is(amount)));
    }
```

@After

Junit에서 단위 테스트가 끝날 때마다 수행되는 메서드를 지정한다.

보통은 배포 전 전체 테스트를 수행할 때 테스트간 데이터 침범을 막기 위해 사용한다. 여러 테스트가 동시에 수행되면 테스트용 데이터베이스인 H2에 데이터가 그대로 남아있어 다음 테스트 실행시 테스트가 실패할 수 있다.

```
@After
     public void cleanup(){
        postsRepository.deleteAll();
    }
```

@SpringBootTest

별 다른 설정없이 실행하면 H2 데이터베이스가 실행된다.