CH3 JPA in SpringBoot

정승혜

1. 소개

왜 JPA를 사용할까?

- iBatis(MyBatis)
  - SQL Mapper(쿼리 매핑)을 통해서 작성하던 DB query

⇒테이블 모델링에만 집중하고, 객체를 테이블에 맞추게 됨
⇒ 관계형 DB를 이용하는데, 객체지향 프로그래밍이 어려운 문제 발생

1. 단순 반복 문제 – CRUD SQL의 반복
2. 패러다임 불일치

- RDB : 데이터를 어떻게 저장할지
- 객체지향언어: 메시지를 기반으로 기능과 속성을 한 곳에서 관리

=> 상속, 1:N 등 객체 모델링을 완전히 DB로 구현할 수 없음

⇒ JPA의 등장
  자바표준 ORM(Object Relational Mapping) - 객체 매핑

=> 객체 프로그래밍을 하면 JPA가 이를 RDB에 맞게 SQL을 대신 생성하여 실행

왜 JPA를 사용할까?

- Spring Data JPA

- JPA는 인터페이스로서 자바 표준명세서
- 인터페이스인 JPA를 사용하기 위해서는 구현체가 필요
- 쉽게 사용하기 위해서 추상화된 Spring Data JPA라는 모듈을 이용해서 JPA를 다룸

JPA<- Hibernate <- Spring Data JPA

Hibernate와 Spring Data JPA의 기능은 큰차이가 없으나,
Spring Data JPA를 권장하는 이유는

1. 구현체의 용이성 : 구현체 매핑을 지원하기 때문에 Hibernate외에 다른 구현체로 쉽게 교체가능
2. 저장소 교체의 용이성 : RDB 외에 다른 저장소로 쉽게 교체가능. 제공하는 CRUD 인터페이스가 같기때문에 의존성 교체만으로 해결

2. 구현

JPA를 사용해보자

- build.gradle 의존성 추가

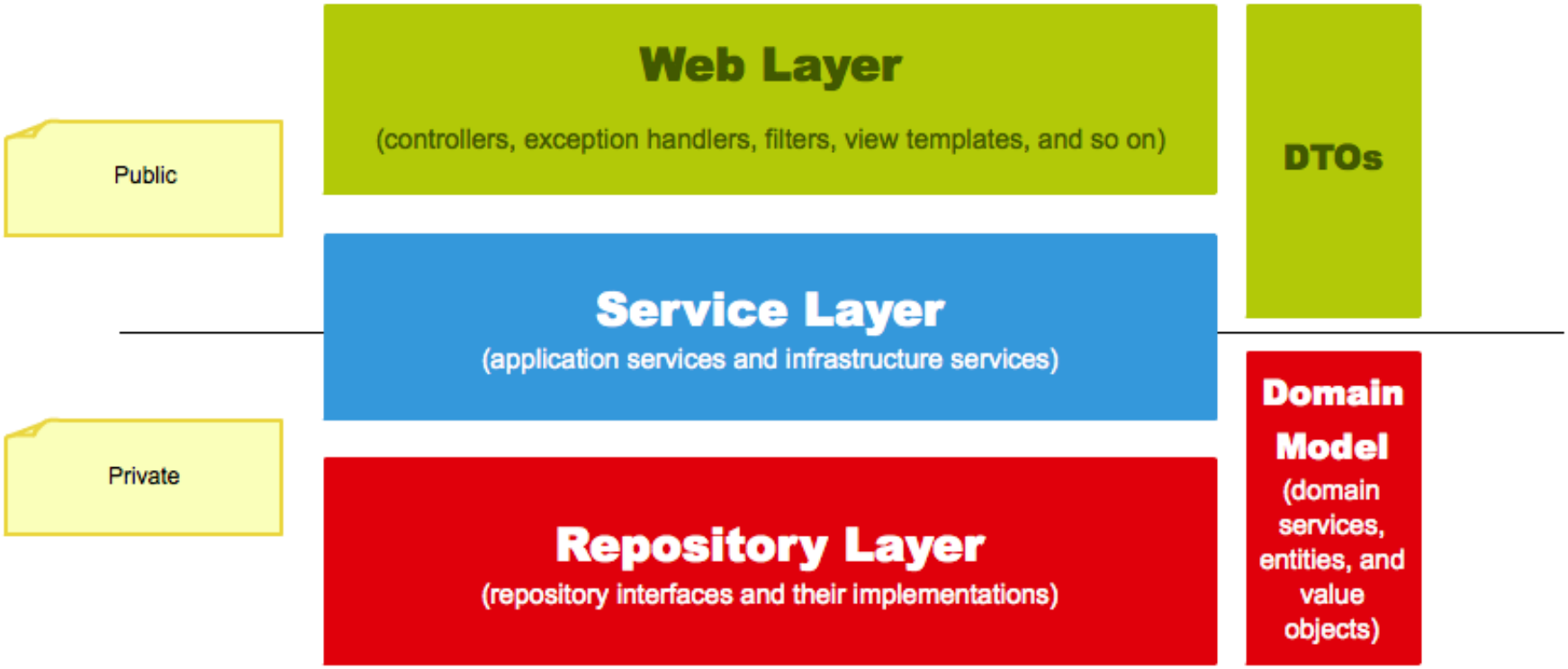
- org.springframework.book:spring-boot-starter-data-jpa
- com.h2database:h2 (인메모리 RDB, 테스트용으로 많이 사용)

요구사항 분석

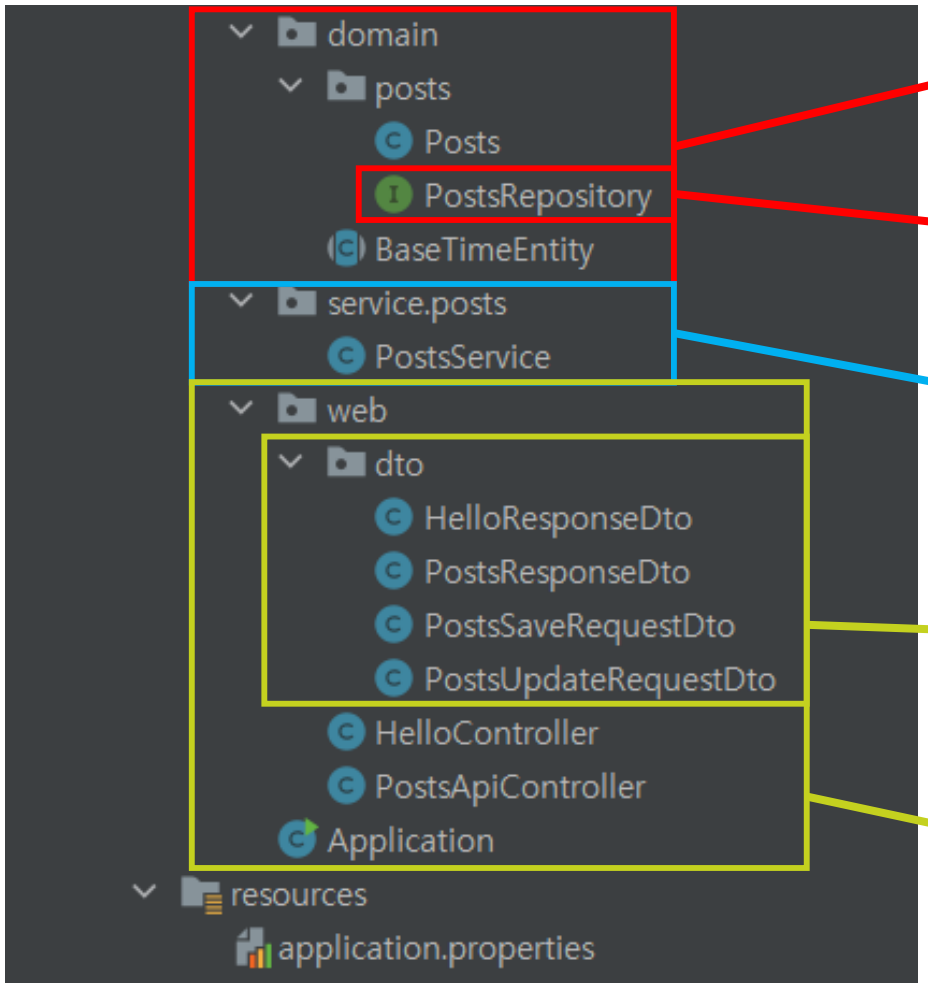
- 게시판 기능
- 게시글 조회
- 게시글 등록
- 게시글 수정
- 게시글 삭제

- 회원기능
- 구글/네이버 로그인
- 로그인 사용자 글 작성 권한
- 본인 작성 글에 대한 권한 관리

DDD (Domain-Driven Design)



디렉토리 구조



Domain layer – 비즈니스 로직
(domain services, entities, Value objects)

Repository layer
(repository interface, DB에 접근하는, dao 영역)

Service layer
(@Transactional, 컨트롤러와 dao의 중간 단계)

DTO layer
(계층간 데이터 교환 객체 다룸, Web에서 사용될 혹은 repository에서 결과로 넘겨진 객체)

Web layer
(Controller, JSP 등 뷰템플릿, 필터, 인터셉터, 어드바이스)

2-1. Domain & Repository

Posts.java – domain

```java
@Getter
@NoArgsConstructor // 기본 생성자 자동추가
@Entity // 주요 어노테이션을 클래스에 가깝게 , 테이블과 매핑되는 클래스, JPA가 제공
public class Posts extends BaseTimeEntity {
```

- @Entity : 테이블과 링크될 클래스임을 명시
- 스키마 정의, 생성을 담당함

@Id	해당 테이블의 PK 필드 @Id
@GeneratedValue	PK의 생성규칙. 스프링 부트 2.0에서 GeneratedType.IDENTITY 옵션을 추가해야만 auto_increment가 됨. Entity의 PK는 웬만하면 auto_increment 권장 @GeneratedValue(strategy = GenerationType.*IDENTITY*)
@Column	테이블의 컬럼을 나타냄. 기본값 외에 추가로 변경이 필요한 옵션이 있는 경우 사용 (사이즈 확장 or 타입변경) @Column(length = 500, nullable = false)

클래스의 Setter 메소드에 대해

```
@Getter
@NoArgsConstructor // 기본 생성자 자동추가
@Entity // 주요 어노테이션을 클래스에 가깝게 , 테이블과 매핑되는 클래스, JPA가 제공
public class Posts extends BaseTimeEntity {
```

⇒@Getter만 두고 기본 생성자 자동추가 어노테이션을 사용한 모습

왜?

- 자바빈 규약을 따르려고 getter/setter를 무작정 생성하면 해당 클래스의 인스턴스가 언제 어디서 변해야하는지 코드상으로 명확하게 구분이 어려움
- Entity 클래스에는 절대 Setter 메소드를 만들지 않음
- 대신 해당 필드의 값 변경이 필요하면 명확히 그 목적과 의도를 나타낼 수 있는 메소드를 추가해야함

생성자로 값을 채워 DB에 삽입하기

```java
@Builder // 해당 클래스의 빌더 패턴 클래스 생성, 생성자 상단에 선언시 생성자에 포함된 필드만 빌더에 포함
public Posts(String title, String content, String author){
    this.title = title;
    this.content = content;
    this.author = author;
}

public void update(String title, String content){
    this.title = title;
    this.content = content;
}
```

- 생성자를 통해 최종값을 채운 후 DB에 삽입
- Posts에는 생성자 대신에 @Builder을 통해 제공되는 빌더 클래스를 사용함

=> Builder와 생성자 모두 생성지점에 값을 채워줌. 그러나 생성자는 각각에 채워야 할 필드가 무엇인지 명확히 지정불가

PostsRepository.java – repository

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

public interface PostsRepository extends JpaRepository<Posts,Long> {
}
```

- JPA interface 클래스
- JpaRepository를 상속받음

2-1. Test

PostsRepositoryTest.java

- Entity 클래스 테스트에서는 @WebMvcTest 대신에 @SpringBootTest 사용
- @SpringBootTest는 h2 데이터베이스를 자동으로 실행해줌

cleanup() 메소드

```java
@Autowired
PostsRepository postsRepository;

@After // Junit 단위 테스트가 끝날 때마다 수행되는 메소드를 지정, 배포전 전체 테스트시 테스트간 데이터 침범을 막음
public void cleanup() { postsRepository.deleteAll(); }
```

테스트 메소드에 @Test 어노테이션
assertThat 검증

```java
// 테이블 posts에 insert or update 쿼리 실행
postsRepository.save(Posts.builder()
        .title(title)
        .content(content)
        .author("jojoldu@gmail.com")
        .build());

//when - 테이블의 모든 요소 조회
List<Posts> postsList = postsRepository.findAll();

//then
Posts posts = postsList.get(0);
assertThat(posts.getTitle()).isEqualTo(title);
assertThat(posts.getContent()).isEqualTo(content);
```

2-2. Dtos & Service

등록/수정/조회 API 만들기

1. Request 데이터를 받을 DTO – PostSaveRequestDto.java
2. API 요청을 받을 Controller – PostsApiController.java
3. 트랜잭션, 도메인 기능 간의 순서를 보장하는 – PostsService.java

스프링에서 Bean을 주입받는 방법

1. @Autowired
2. Setter
3. 생성자 – 가장 권장하는 방식 => @RequiredArgsConstructor : final이 선언된
   => @NoArgsConstructor : 모든 필드 값

=> final이 선언된 모든 필드를 인자값으로 하는 생성자를 롬복이 자동 생성

PostsService.java

```java
@RequiredArgsConstructor
@Service
public class PostsService {
    private final PostsRepository postsRepository;

    @Transactional
    public long save(PostsSaveRequestDto requestDto) { return postsRepository.save(requestDto.toEntity()).getId(); }

    @Transactional
    public Long update(Long id, PostsUpdateRequestDto requestDto){ // JPA의 영속성 컨텍스트, 엔티티 영구 저장
        Posts posts = postsRepository.findById(id).orElseThrow(()-> new IllegalArgumentException("해당 게시글이 없습니다." +
                " id="+id));
        posts.update(requestDto.getTitle(), requestDto.getContent());
        return id;
    }

    public PostsResponseDto findById (Long id){
        Posts entity = postsRepository.findById(id)
                .orElseThrow(()->new IllegalArgumentException("해당 게시글이 없습니다. id="+id));
        return new PostsResponseDto(entity);
    }
}
```

requestDto에서 넘어온 게시글을 Repository가 저장(DB)

requestDto가 요청한 게시글이 repository를 통해 존재를 확인하면 entity 클래스로 게시글의 대한 정보를 업데이트(DB)

requestDto가 요청한 게시글을 Repository가 찾아서 반환

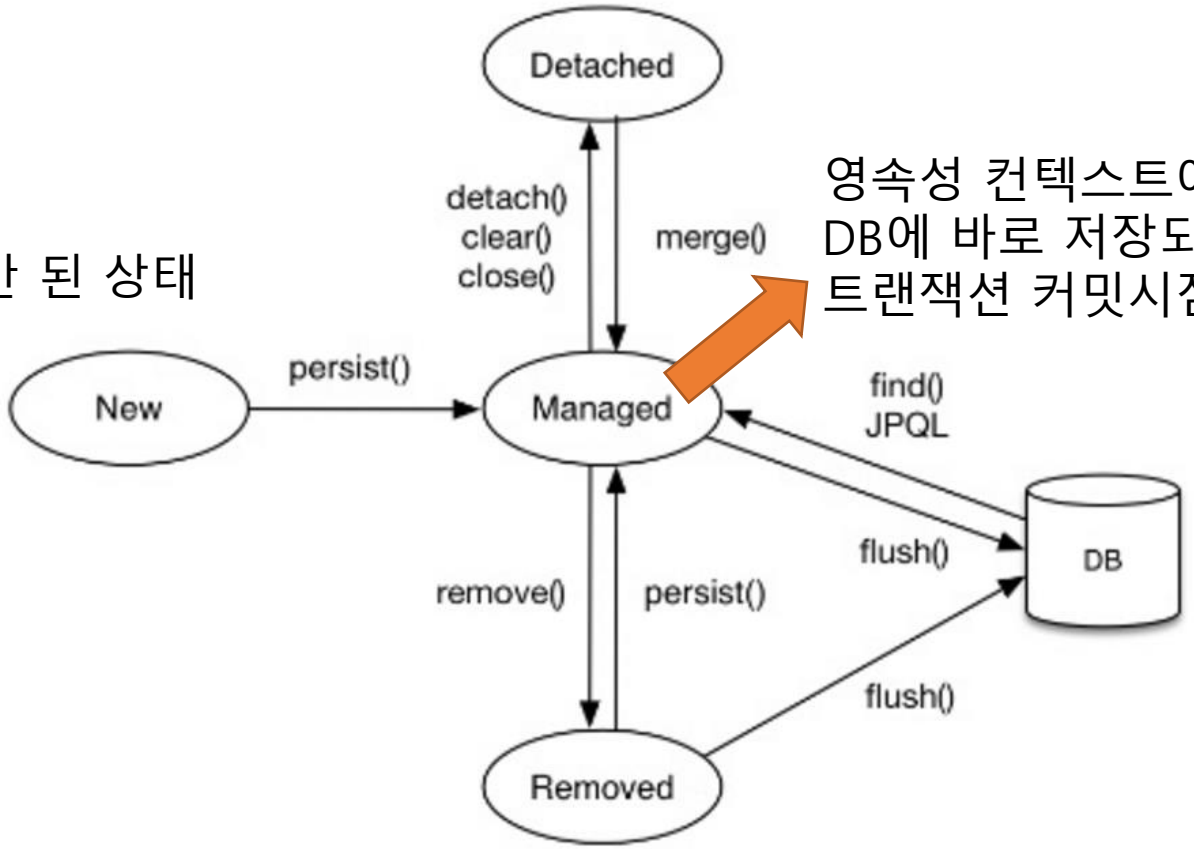
영속성 컨텍스트

- Service의 update와 같은 메서드에 쿼리를 날리는 부분이 없음

- 영속성 컨텍스트
- 논리적인 개념
- 엔티티를 영구 저장하는 환경, 영속성 컨텍스트는 JPA의 엔티티 매니저가 활성화된 상태로 JPA를 사용하면 트랜잭션안에서 DB의 데이터를 가져오고, 이때 데이터는 영속성 컨텍스트가 유지된 상태라함
- 엔티티가 영속성 컨텍스트안에 저장되어있으면, 데이터 값에 변경이 일어나면 트랜잭션이 끝나는 시점에 해당 테이블에 변경분을 반영함
- Entity 객체의 값만 변경하면 별도로 update 쿼리를 날릴 필요가 없음

Entity의 생명주기

영속성 컨텍스트에서 분리된 상태



삭제된 상태. DB에서 날림

PostsSaveRequestDto.java

```java
@Getter
@NoArgsConstructor
public class PostsSaveRequestDto {
    private String title;
    private String content;
    private String author;

    @Builder
    public PostsSaveRequestDto(String title,String content,String author){
        this.title = title;
        this.content = content;
        this.author = author;
    }

    public Posts toEntity(){
        return Posts.builder()
                .title(title)
                .content(content)
                .author(author)
                .build();
    }
}
```

DTO 클래스의 특징

- Entity 클래스(Posts)와 유사한 형태이지만, Entity 클래스를 절대 Request/Response 클래스로 사용해서는 안됨
- Entity 클래스는 DB와 맞닿은 핵심 클래스이기 때문에 테이블 생성/스키마 변경이 일어남
- View Layer와 DB Layer의 역할 분리를 철저하게 할 것
- Entity는 변경이 잦지 않지만 Dto 클래스는 View와도 맞닿아 있어서 변경이 잦음
- Controller에서 결괏값으로 여러 테이블을 조인해서 줘야 하는 경우가 빈번하므로, Entity와 Controller에서 쓸 Dto는 분리해서 사용해야함

2-3. Controller

PostsApiController.java

```java
@RequiredArgsConstructor
@RestController
public class PostsApiController {

    private final PostsService postsService;

    @PostMapping("/api/v1/posts")
    public Long save(@RequestBody PostsSaveRequestDto requestDto) { return postsService.save(requestDto); }

    @PutMapping("/api/v1/posts/{id}")
    public Long update(@PathVariable Long id, @RequestBody PostsUpdateRequestDto requestDto){
        return postsService.update(id, requestDto);
    }

    @GetMapping("/api/v1/posts/{id}")
    public PostsResponseDto findById (@PathVariable Long id) { return postsService.findById(id); }
}
```

2-3. Test

PostsApiControllerTest.java

```java
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
public class PostsApiControllerTest {

    @LocalServerPort
    private int port;

    @Autowired
    private TestRestTemplate restTemplate;

    @Autowired
    private PostsRepository postsRepository;

    @After
    public void tearDown() throws Exception {
        postsRepository.deleteAll();
    }
```

cleanup()과 동일

MockTest와의 차이는 실제 서블릿 컨테이너 실행 여부이며
RestTestTemplate은 컨테이너를 직접 실행함

PostsApiControllerTest.java

```java
@Test
public void Posts_등록된다() throws Exception{
    //given
    String title = "title";
    String content = "content";
    PostsSaveRequestDto requestDto = PostsSaveRequestDto.builder()
            .title(title)
            .content(content)
            .author("author")
            .build();

    String url = "http://localhost:"+port+"/api/v1/posts";

    //when
    ResponseEntity<Long> responseEntity = restTemplate.postForEntity(url, requestDto, Long.class);

    //then
    assertThat(responseEntity.getStatusCode()).isEqualTo(HttpStatus.OK);
    assertThat(responseEntity.getBody()).isGreaterThan(0L);

    List<Posts> all = postsRepository.findAll();
    assertThat(all.get(0).getTitle()).isEqualTo(title);
    assertThat(all.get(0).getContent()).isEqualTo(content);
}
```

2-4. JPA Auditing

JPA Auditing

- 보통 엔티티에는 데이터의 생성/수정 시간을 포함함
- 자동이 아닐경우 DB 삽입, 갱신 전 날짜 데이터를 등록/수정하는 코드가 여기저기 반복되고 지저분해짐

⇒JPA Auditing 사용

LocalDate

- java8부터 LocalDate, LocalDateTime 등장
- LocalDate, LocalDateTime이 제대로 DB에 매핑되지 않는 이슈도 Hibernate 5.2.10에서 해결됨

BaseTimeEntity

```java
@Getter
@MappedSuperclass // JPA entity 클래스들이 지금 클래스를 상속할 경우 필드들을 지금 클래스의 칼럼으로 인식하게함
@EntityListeners(AuditingEntityListener.class) // BaseTimeEntity 클래스에 Auditing 기능 포함
public abstract class BaseTimeEntity {

    @CreatedDate // 생성시간 자동저장
    private LocalDateTime createdDate;

    @LastModifiedDate // 값 변경시 자동저장
    private LocalDateTime modifiedDate;
}
```

```java
@Getter
@NoArgsConstructor // 기본 생성자 자동추가
@Entity // 주요 어노테이션을 클래스에 가깝게 , 테이블과 매핑되는 클래스, JPA가 제공
public class Posts extends BaseTimeEntity {
```

```java
@EnableJpaAuditing // JPA AUditing 활성화
@SpringBootApplication // 스프링 부트의 자동 설정, Bean 읽기와 생성이 모두 자동
public class Application {
    public static void main(String[] args){
        SpringApplication.run(Application.class, args); // 내장 WAS 사용, 톰캣 필요 X, jar파일로 실행
    }
}
```