DTO와 Record 그리고 Validation

DTO(Data Transfer Object)가 뭘까?

이름 그대로 데이터를 전달하는 객체!

Entity로 그대로 전달하는게 코드짜기도 편하고 좋지않나? 라는 의문을 가질 수 있음!

우리가 짠 게시글 CRUD로 예를 들어보자. 다음과 같이 Entity를 구성했다고 치자.

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

@Column(nullable = false, columnDefinition = "TEXT")
private String content;

@Column(nullable = false, length = 100)
private String authorName;

@Column(nullable = false)
private int likes = 0;

@CreatedDate
@Column(updatable = false)
private LocalDateTime createdAt;

@LastModifiedDate
private LocalDateTime updatedAt;
```

게시글을 생성할 때, 우리가 사용자에게 받아내는 것은 제목, 내용, 작성자명 정도면 충분하다.

하지만, Entity를 그대로 사용한다면 어떻게 될까? 저기서 id, createdAt, updatedAt 등 생성 시에는 필요가 없는 변수까지도 공간을 내어 들고오게된다. 즉, 필요없는 변수까지 생성한다고 비용을 소모하게 된다.

또한, Entity 사용을 위해 Setter를 사용한다던가 하면 앞에서 보았듯 일관성에 문제가 되는 코드가 되어버린다.

그러므로, 우리는 DTO라는 것으로 필요한 상자만을 만들고, 필요한 것만을 담아 통신을 하는 것이 굉장히 중요하다.

```
@Getter
@Setter
public class PostRequestDto{
    private String content;
    private String authorName;
}
```

Record란 무엇일까?

불변이 뭐지?

> 한 번 값이 설정되면 값이 변하지 않는 것! == final

왜 불변이여야 하지?

- 예측가능한 동작이 수행된다.
  - 객체에서 값이 중간에 변하여 예상치 못한 결과가 나오는 경우를 배제할 수 있음.
- 여러 Thread에서 동시에 접근해도 같은 결과를 보여줌.
  - 여러 Thread에서 접근하더라도, 이미 값이 설정되어있으면 수정도 불가하기에 모두 같은 결과를 내보냄. - race condition을 사전에 방지
- 객체를 어디서든 공유할 수 있다.
  - 수정될 가능성이 없기 때문에 어디서든 그냥 전달만 하면됨.

그래서 Record가 뭔데?

불변(Immutable) 데이터를 표현하기 위한 객체를 간결하게 정의할 때 주로 사용

자동으로 **생성자, getter, toString(), equals(), hashCode()** 등을 제공

```
public record PostRequestDto( //위의 DTO class를 이렇게 변환가능!
    String content;
    String authorName;
){}

---- 사용할 때 ----

@Service
public class PostService{

    public Long createPost(PostRequestDto postRequestDto) {
        Post post = new Post();
          post.createPost(
            postRequestDto.content(),
            postRequestDto.autherName()
        );
          // 아예 Dto를 파라미터로 받아서
          // post.createPost(postRequestDto); 라고하면 더 깔끔함!

        postRepository.save(post);
        return post.getId();
    }
}
```

그러면 DTO class는 안쓰나요?

NO. 특정 상황에서는 사용합니다.

- 커스터마이징한 메서드가 필요한 경우
    - 데이터를 전달할 때, 빈 값(null)이면 이를 “” 로 채워두고 싶어!

→ 위 과정을 Class는 내부에서 제약없이 구성가능하지만, Record는 불변 데이터 구조를 유지하기 위해서 다음과 같은 제한사항이 존재

- 필드는 암묵적으로 `private final`이며 추가적인 인스턴스 필드를 선언할 수 없음.
- 상속 불가 (다른 클래스를 상속할 수 없고, `abstract`나 `final`이 아님).
- 모든 필드는 생성자 파라미터로 받아야 함.
- mutable한 상태(필드 등)를 가질 수 없음.

아래 코드는 이해를 돕기 위한 예시이며, 다음과 같이 메서드를 추가할 때는 합당한 논리가 존재해야합니다!!

```
@Getter
@Setter
public class PostRequestDto{
    private String content;
    private String authorName;

    public String fillNullWithEmptyString(String content){
        //null값이면 이를 ""로 채우는 로직 구성
    }
}

public record PostRequestDto(
    String content,
    String authorName
    ) {

      // 생성자로 값 설정해야함
      // 이렇게 만들면 값이 ""로 고정
    public PostRequestDto {
        if (content == null) {
            content = "";
        }
        if (authorName == null) {
            authorName = "";
        }
    }

    // 필요시 이런식으로 커스텀 메서드 추가 가능
    public boolean isEmpty() {
        return content.isEmpty() && authorName.isEmpty();
    }
}

----------------------------

@Service
public class PostService{

    public Long createPost(PostRequestDto postRequestDto) {
```

```
        Post post = new Post();
        post.createPost(
            postRequestDto.fillNullWithEmptyString(postRequestDto.content),
            postRequestDto.autherName
        );
        postRepository.save(post);
        return post.getId();
    }
}
```

DTO Validation

validation?

특정 변수가 비어있다면? null값이라면? 문제가 될 것이다.

이를 사전에 막기 위해 DTO내에서 유효성(validation)을 검사할 수 있다.

어떻게?

이렇게.

```
@NoArgsConstructor(access = AccessLevel.PROTECTED)
@Getter
public class SignupDto {
    @NotBlank
    private String email;
    @NotBlank
    private String password;
    @NotBlank
    private String nickname;

    public static SignupDto newInstance(
            String email,
            String password,
            String nickname
    ) {
        SignupDto signupDto = new SignupDto();
        signupDto.email = email;
        signupDto.password = password;
        signupDto.nickname = nickname;
        return signupDto;
    }
}

//문자열의 경우 @NotBlank를 문자열이 Null이거나, 비어있거나, ' '인지 확인합니다.
//객체의 값 검증인 경우에는 @NotNull
//Collection의 경우에는 @NotEmpty를 사용해서 값이 존재하지 않는지를 검증할 수 있습니다.
```

코드 출처: 참고문헌 3

이를 Controller에 쓰겠다고 미리 언질을 주어야 하기때문에 @Valid 라는 것을 붙입니다

```
@RestController
@RequiredArgsConstructor
@RequestMapping("/api")
public class MemberController {
    private final MemberService memberService;
    private final JwtCommunicationServlet jwtCommunicationServlet;
    private final JwtProvider jwtProvider;

    @PostMapping("/auth/signup")
    public ResponseEntity<SignupDto> signup(@RequestBody @Valid final SignupDto
request) {
        SignupDto response = memberService.createMember(request);

        return ResponseEntity.status(HttpStatus.CREATED).body(response);
    }

    ...
}
```

코드 출처: 참고문헌 3

이걸로 어디까지 막아야할까?

이제 좀 어려운 내용...

위에서 DTO로 유효성 검사를 하는데, 이 범위에 관한 문제를 다루는 것

예를 들어보자.

1. 나는 해당 값이 Null값이 아니였으면 좋겠어!
2. 나는 제목에 욕설이 포함되지 않았으면 좋겠어!
3. 나는 해당 리스트가 비어있는 리스트가 아니였으면 좋겠어!

- ?

  1,3번은 DTO에서, 2번은 Service단에서 처리하는게 일반적!

왜? 나는 DTO에서 다 하고싶은데? 라고 할 수 있음.

→ 이러면 SOLID원칙 중 S에 해당하는 단일 책임 원칙(SRP, Single Responsibility Principle) 이라는 것을 위반함.

그게 뭐임? - 하나의 객체는 하나의 책임만을 가져야 한다는 것.

즉, DTO는 데이터 전달만 잘하면 됐지, 필터링까지 해줄 의무는 없다는 것!

억지로 추가해서 짜게되면 나중에 코드 수정 시에 지옥을 보게 될 것... ~~직접 해보시는 것을 추천드립니다^^~~

참고 문헌

https://yozm.wishket.com/magazine/detail/2814/

https://00shin.tistory.com/200

https://velog.io/@imkkuk/Spring-Boot-DTO-Validation