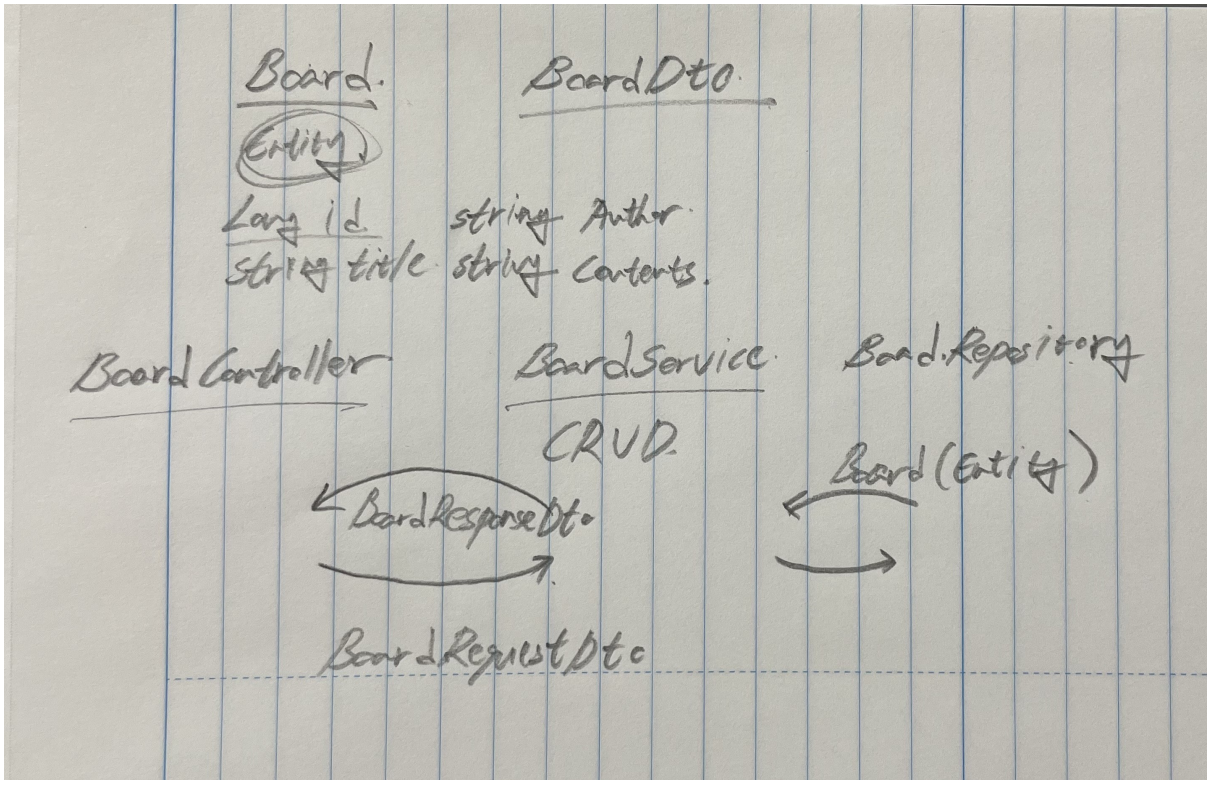
Spring 실습

게시글 작성 및 목록 조회

UseCase 작성



제작 순서

1. Board Entity 생성 : Board 테이블에 생성할 Column을 생성한다.

   → Id, Title, Author, Contents

```
@Getter
@NoArgsConstructor
@Entity
public class Board extends TimeStamped{
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(nullable = false)
    private String title;

    @Column(nullable = false)
    private String author;

    @Column(nullable = false)
    private String contents;

    // Board 클래스의 생성자
    // -> CourseRequestDto 객체를 인자로 받아 해당 객체의 필드 값을 가져와
    //     title, instructor, cost 필드에 할당
    public Board(BoardRequestDto boardRequestDto){
        //this.id = boardRequestDto.getId();
        this.title = boardRequestDto.getTitle();
        this.author = boardRequestDto.getAuthor();
        this.contents = boardRequestDto.getContents();
    }

    // update() 메서드는 이미 생성된 객체를 업데이트할 때 사용
    // -> CourseRequestDto 객체를 인자로 받아 해당 객체의 데이터로
    //    title, instructor, cost를 업데이트하는 로직이 들어감.
    public void update(BoardRequestDto boardRequestDto){
        //this.id = boardRequestDto.getId();
```

```
        this.title = boardRequestDto.getTitle();
        this.author = boardRequestDto.getAuthor();
        this.contents = boardRequestDto.getContents();
    }
}
```

→ 이처럼 생성자와 업데이트 메서드에서 사용되는 데이터가 다를 경우, 두 개의 DTO를 사용하여 각각의 메서드에서 필요한 데이터를 전달하는 것이 일관성 있는 객체 초기화와 업데이트를 할 수 있도록 도와줌.

2. BoardRepository 생성 : 데이터베이스와의 상호작용을 위해 Board repository 생성

   → JpaRepository 인터페이스 사용

```
public interface BoardRepository extends JpaRepository<Board, Long> {
    Optional<Board> findById(Long id);
}
```

→ 근데, findById 옵션은 이미 구성되어 있지 않나?

3. BoardRequestDto 생성 : Controller와 Service 간의 데이터 흐름을 위해 생성

   → DTO(Data Transfer Object)란, 계층간 데이터 교환을 위해 사용하는 객체

   → DTO는 각 비즈니스 로직에 맞춘 필드들만 생성함으로써 DTO를 보면 어떤 값들이 매핑되는지 쉽게 파악할 수 있고, 만약 API 설계 상황에서 필드에 다른 이름을 부여하거나 하는 상황에서도 유연하게 대처할 수 있다.

   → **Controller와 Service 사이에서 강한 의존을 방지하기 위해**서 DTO를 사용!!

```
@Getter
//일반적으로 DTO 클래스는 단순히 데이터를 저장하기 위한 용도로 사용되기 때문에, 별도의 생성자가 필요하지 않음
public class BoardRequestDto {
    private Long id;
    private String title;
    private String author;
    private String contents;
}
```

4. BoardResponseDto 생성 : 응답 Dto 클래스로 버에서 처리된 강좌 정보를 클라이언트에게 반환하기 위해 사용

```
@Getter
@NoArgsConstructor
public class BoardResponseDto {
    private Long id;
    private String title;
    private String author;
    private String contents;

    public BoardResponseDto(Board board){
        this.id = board.getId();
        this.title = board.getTitle();
        this.author = board.getAuthor();
        this.contents = board.getContents();
    }
}
```

→ `BoardResponseDto`는 외부로부터 `Board`엔티티의 데이터를 전달하기 위한 목적으로 사용

→ DTO 클래스에서 생성자를 만들어 사용하는 것은 데이터 전달, 불변성 유지, 재사용성 향상, 클래스 초기화 등의 목적을 달성하기 위해

- RequestDto와 ResposeDto로 나누어서 구성한 이유
  - **요청(Request)과 응답(Response)의 데이터 필요성 차이**
    - 클라이언트로부터의 요청은 보통 클라이언트가 제공하는 입력 데이터를 담고 있어야 하고, 서버로부터의 응답은 클라이언트에게 반환되는 결과 데이터를 담고 있어야 한다.

      → 이렇게 요청과 응답의 데이터 필요성이 다르기 때문에, 요청(Request)과 응답(Response)용으로 별도의 DTO 클래스를 사용하여 데이터를 전달하면 목적에 맞게 더 명확하고 유연한 코드를 작성할 수 있다.
  - 이 뿐만 아니라, 데이터의 가공과 보안 & API 버전 관리를 위해 나누어서 구성

---

5. BoardService 구성 : 비지니스 로직 수행과 데이터의 처리와 관리를 위해 작성

- **비즈니스 로직 처리 → 로직 : 데이터의 가공, 검증, 계산, 처리, 외부 시스템과의 연동 등등**
- **트랜잭션 관리 → 트랜잭션 : 데이터베이스의 상태를 일관성 있게 유지하기 위한 기술 Service는 트랜잭션의 시작, 종료, 롤백 등을 처리하여 데이터의 일관성과 무결성을 보장**
- **데이터의 처리와 관리 → CRUD 수행**

```
@Service
public class BoardService {
    private final BoardRepository boardRepository;

    @Autowired
    public BoardService(BoardRepository boardRepository){
        this.boardRepository = boardRepository;
    }

    public String createBoard(BoardRequestDto boardRequestDto){
        Board board = new Board(boardRequestDto);

        boardRepository.save(board);

        return "게시글 등록을 완료하였습니다.";
    }

    public List<BoardResponseDto> getBoardList() {
        List<BoardResponseDto> responseDtoList = new ArrayList<>();
        //작성 날짜 기준 내림차순 정렬
//        List<Board> boards = boardRepository.findAll(Sort.by(Sort.Direction.DESC, "createdAt"));
//
//        for(Board board : boards){
//            BoardResponseDto boardResponseDto = new BoardResponseDto(board);
//            responseDtoList.add(boardResponseDto);
//        }
//
//        return responseDtoList;
        return boardRepository.findAll(Sort.by(Sort.Direction.DESC, "createdAt")).stream().map(BoardResponseDto::new).collect(Collectors.toList());
    }

    public BoardResponseDto getBoard(Long id){
        Board board = boardRepository.findById(id).orElseThrow(
                () -> new NullPointerException("선택한 게시글이 없습니다.")
        );

        return new BoardResponseDto(board);
    }

    @Transactional
    public BoardResponseDto updateBoard(Long id, BoardRequestDto boardRequestDto){
        Board board = boardRepository.findById(id).orElseThrow(
                () -> new NullPointerException("선택한 게시글이 없습니다.")
        );

        board.update(boardRequestDto);

        return new BoardResponseDto(board);
    }

    public String deleteBoard(Long id){
        Board board = boardRepository.findById(id).orElseThrow(
                () -> new NullPointerException("선택한 게시글이 없습니다.")
        );

        boardRepository.delete(board);

        return "선택한 게시글을 삭제하였습니다.";
    }
}
```

→ 여기서 공통된 코드를 따로 빼서 작성 가능

```
    private Course checkCourse(Long id) {
        return courseRepository.findById(id).orElseThrow(
                () -> new NullPointerException("선택한 강의가 존재하지 않습니다.")
        );
    }
```

→ 앞서 설명한 요청Dto와 응답Dto를 위 코드에서 살펴보면 요청Dto는 매개체로 응답Dto는 리턴 값으로 사용!!

6. BoardController 구성 : 웹 애플리케이션에서 사용자의 요청을 처리하고, 응답을 반환하는 역할을 수행하기 위해 Controller 구성
    - 웹 애플리케이션의 프론트 엔드와 백 엔드 사이의 인터페이스 역할을 수행
    - RestController
        - RESTful 웹 서비스를 구현하기 위해 사용되는 컨트롤러로, JSON, XML 등의 형식으로 데이터를 반환

→ 간결하고 가독성이 좋은 데이터 표현, 다양한 플랫폼과 언어에서의 지원, 웹 브라우저와의 호환성, RESTful 웹 서비스의 표준 형식을 따르기 위해

- 주로 웹 API를 구현하는데 사용, 요청과 응답이 HTTP 프로토콜을 기반으로 이루어짐

```
@RestController
@RequestMapping("/board")
//Restful 웹 서비스 특징으로 사용 가능
public class BoardController {
    private final BoardService boardService;

    @Autowired
    public BoardController(BoardService boardService){
        this.boardService = boardService;
    }

    @PostMapping("/create")
    // 데이터를 붙이기 위해 PostMapping
    public String createBoard(@RequestBody BoardRequestDto boardRequestDto){
        return boardService.createBoard(boardRequestDto);
    }

    @GetMapping("/list")
    // 데이터를 얻기 위해 GetMapping
    public List<BoardResponseDto> getBoardList() {
        return boardService.getBoardList();
    }

    @GetMapping("/{id}")
    public BoardResponseDto getBoard(@PathVariable Long id){
        return boardService.getBoard(id);
    }

    @PutMapping("/update/{id}")
    // 데이터 수정을 위해 PutMapping
    public BoardResponseDto updateBoard(@PathVariable Long id, @RequestBody BoardRequestDto boardRequestDto){
        //@PathVariable Long id: id는 URL 경로(Path)에서 추출되는 경로 변수(Path Variable)로, Long 타입의 값을 받아옴
        //requestDto는 HTTP 요청의 본문(Request Body)에 담긴 데이터를 바인딩하여 객체로 받아옴
        return boardService.updateBoard(id, boardRequestDto);
    }

    @DeleteMapping("/delete/{id}")
    public String deleteBoard(@PathVariable Long id){
        return boardService.deleteBoard(id);
    }
}
```

→ @PostMapping("/create") 또는 @GetMapping("/update/{id}") 이런 Mapping은 `@RestController` 선언을 통해 사용할 수 있는 것이며,

만약, `@Controller`를 사용하면 일반적인 웹 애플리케이션에서 사용되는 기능들을 사용할 수 있지만, RESTful 웹 서비스의 특징인 자원 기반의 통신, HTTP 메서드를 통한 작업 수행 등을 명시적으로 구현하기 어려울 수 있음

→ Dto 사용으로 인해, Controller가 (비즈니스 로직이나 데이터 접근을 처리하는) Repository에 대한 의존성 주입은 필요하지 않다.

entity 추가!!

- 비밀번호 추가하기!!
  - BoardRequestDto에서 password 필드를 추가 후 Service 클래스 속 update 메서드에

  아래 코드를 추가한다. 이로써, 기존 데이터에 저장된 password와 업데이트 요청 password를 비교해서 같으면 업데이트 되고 다르면 예외처리를 하도록 설정하였다.

  ```
        if (!board.getPassword().equals(boardRequestDto.getPassword())) {
            throw new IllegalArgumentException("비밀번호가 일치하지 않습니다.");
        }
  ```

  이뿐만 아니라 삭제 메서드에서도 같은 방식으로 처리되도록 수정하였다.

  - 수정 코드

  ```
      @Transactional
      public BoardResponseDto updateBoard(Long id, BoardRequestDto boardRequestDto, String password){


          Board board = boardRepository.findById(id).orElseThrow(
                  () -> new NullPointerException("선택한 게시글이 없습니다.")
          );

          if (!board.getPassword().equals(password)) {
              throw new IllegalArgumentException("비밀번호가 일치하지 않습니다.");
          }
  ```

```
        board.update(boardRequestDto);

        return new BoardResponseDto(board);
    }

    public String deleteBoard(Long id, String password){
        Board board = boardRepository.findById(id).orElseThrow(
                () -> new NullPointerException("선택한 게시글이 없습니다.")
        );

        if (!board.getPassword().equals(password)) {
            throw new IllegalArgumentException("비밀번호가 일치하지 않습니다.");
        }

        boardRepository.delete(board);

        return "선택한 게시글을 삭제하였습니다.";
    }
```

```
    @PutMapping("/update/{id}/{password}")
    public BoardResponseDto updateBoard(@PathVariable Long id, @RequestBody BoardRequestDto boardRequestDto, @PathVariable String password){
        //@PathVariable Long id: id는 URL 경로(Path)에서 추출되는 경로 변수(Path Variable)로, Long 타입의 값을 받아옴
        //requestDto는 HTTP 요청의 본문(Request Body)에 담긴 데이터를 바인딩하여 객체로 받아옴
        return boardService.updateBoard(id, boardRequestDto, password);
    }

    @DeleteMapping("/delete/{id}/{password}")
    public String deleteBoard(@PathVariable Long id, @PathVariable String password){
        return boardService.deleteBoard(id, password);
    }
```

- 여기서 더 나아가 API 명세서를 확인 중에 내가 구성했던 url에 password를 입력해서 password가 일치하는지 확인하는 것이 아니라, BoardPasswordDto를 생성하여 password 값만 가져오도록 하는 Dto클래스를 생성해서 비교 가능하도록 수정하였다.
- 수정 코드

```
//BoardPasswordDto 클래스
@Getter
public class BoardPasswordDto {
    private String password;
}

//Board entity 클래스
    public boolean check(BoardPasswordDto boardPasswordDto){
        return this.password.equals(boardPasswordDto.getPassword());
    }

// BoardController 클래스
    @PutMapping("/update/{id}")
    public BoardResponseDto updateBoard(@PathVariable Long id, @RequestBody BoardRequestDto boardRequestDto){
        //@PathVariable Long id: id는 URL 경로(Path)에서 추출되는 경로 변수(Path Variable)로, Long 타입의 값을 받아옴
        //requestDto는 HTTP 요청의 본문(Request Body)에 담긴 데이터를 바인딩하여 객체로 받아옴
        return boardService.updateBoard(id, boardRequestDto);
    }

    @DeleteMapping("/delete/{id}")
    public String deleteBoard(@PathVariable Long id, @RequestBody BoardPasswordDto boardPasswordDto){
        return boardService.deleteBoard(id, boardPasswordDto);
    }

//BoardService 클래스
    @Transactional
    public BoardResponseDto updateBoard(Long id, BoardRequestDto boardRequestDto){


        Board board = boardRepository.findById(id).orElseThrow(
                () -> new NullPointerException("선택한 게시글이 없습니다.")
        );

        if (!board.getPassword().equals(boardRequestDto.getPassword())) {
            throw new IllegalArgumentException("비밀번호가 일치하지 않습니다.");
        }

        board.update(boardRequestDto);

        return new BoardResponseDto(board);
    }

    public String deleteBoard(Long id, BoardPasswordDto boardPasswordDto){
        Board board = boardRepository.findById(id).orElseThrow(
                () -> new NullPointerException("선택한 게시글이 없습니다.")
        );

        if (!board.check(boardPasswordDto)) {
            throw new IllegalArgumentException("비밀번호가 일치하지 않습니다.");
        }

        boardRepository.delete(board);
```

```
        return "선택한 게시글을 삭제하였습니다.";
    }
```

→ update 메서드는 BoardRequestDto를 통해 얻은 데이터 정보 상에 password 정보가 들어있으므로 이를 통해 비교하도록 변경하였다.

오류 사항!!!

여기서 update 시 중요한 사항은 업데이트 시 Board entity 상에서 update 메서드가 호출되는데 이때 id도 가져오게 되면 변경되면 안되는 id가 변경되는 오류가 생긴다. 이로써 update 메서드에는 아래 코드를 생략해 줘야 한다.

```
this.id = boardRequestDto.getId();
```