스프링부트 바이브코딩 실습

1. 개발 환경 준비

1.1 기본 도구 설치

- JDK 25(LTS) 이상 설치
- Gradle (9.3.0) 이상 설치

1.1 Google Antigravity 설치

1. 다운로드: Google Antigravity 공식 사이트 에서 운영체제에 맞는 설치 파일을 내려 받아 설치한다.
2. 로그인: 구글 계정으로 로그인 한다. (프리뷰 기간 동안 대부분의 기능 무료 체험)

1.2 에이전트 브라우저 설정

1. 에이전트 매니저 실행: 상단 메뉴에서 'Open Agent Manager'를 클릭한다.
2. 브라우저 확장 프로그램 설치: 에이전트가 브라우저를 제어할 수 있도록 안내에 따라 Chrome 전용 확장 프로그램을 설치한다.
3. 권한 승인: 에이전트가 브라우저에서 버튼을 클릭하거나 폼을 입력할 수 있도록 'Setup' 버튼을 눌러 권한을 활성화한다.

1.3 워크스페이스 및 모드 설정

1. 샌드박스 폴더 생성 및 워크스페이스에 연결
    - 빈 폴더를 하나 만든다. 예) `$USER_HOME/git/exam01`
    - 방법1) `Open Agent Manager` 선택 > `Open Workspace` 선택 > `Open New Workspace` 선택 > `exam01` 폴더 선택
    - 방법2) `Open Folder` 선택 > `exam01` 폴더 선택
2. 작업 모드 선택
    - Planning Mode: 복잡한 기능을 구현할 때 추천. AI가 먼저 계획서를 쓰고 승인을 요청한다.
    - Fast Mode: 단순한 디자인 수정이나 빠른 프로토타이핑에 적합
3. 모델 선택
    - Gemini 3 Pro (High):
        - 추론 깊이: 최상 (Deep Thinking)
        - 응답 속도: 느림
        - 추천 용도: 아키텍처 설계, 복잡한 로직 구현
    - Gemini 3 Pro (Low):
        - 추론 깊이: 상 (Balanced)
        - 응답 속도: 보통
        - 추천 용도: 일반적인 코딩 및 기능 추가
    - Gemini 3 Pro (Flash):
        - 추론 깊이: 중상 (Fast Agent)
        - 응답 속도: 매우 빠름
        - 추천 용도: 이브 코딩, UI/UX 수정, 빠른 반복 테스트
4. 로컬 실행 환경 준비
    - 언어별 도구 설치: 프로젝트 성격에 따라 Node.js(20 버전 이상 추천)나 Python, JDK 등을 미리 설치해 둔다.

실습

실습 1 - 스프링부트 프로젝트 만들기

프로젝트 명세서 생성하기

```
다음은 스프링부트 기반 웹 애플리케이션 프로젝트를 생성에 대한 내용이다.
이를 바탕으로 프로젝트 명세서를 작성해줘.

- 프로젝트 설정
    - JDK: JDK 25 이상
    - Language: Java
    - Spring Boot: 4.0.1 이상
    - Build Tool: Gradle 9.3.0 이상, Groovy DSL 사용
    - Dependencies: 없음 (최소 기능 프로젝트)
- 플러그인
    - 'io.spring.dependency-management': Spring Boot 버전에 맞춰서 플러그인 추가
- Project Metadata
    - Group: com.example
    - Artifact: vibeapp
    - Main Class Name: VibeApp
    - Description: 최소 기능 스프링부트 애플리케이션을 생성하는 프로젝트다.
    - Configuration: YAML 파일 사용
```

프로젝트 생성하기

```
프로젝트 명세서(PROJECT_SPEC.md)의 내용대로 현재 작업 폴더를 스프링부트 프로젝트 폴더로 구성해줘.
```

"Hello, Vibe!" 출력하는 REST API 만들기

```
VibeApp 클래스에 "Hello, Vibe!" 문자열을 반환하는 REST API를 추가해줘.
- 엔드포인트: /api/hello
- HTTP 메서드: GET
- Method name: hello

REST API를 추가하는데 필요한 Starter나 라이브러리가 있다면 빌드 스크립트 파일에 추가해줘.
```

빌드 스크립트 파일 검토하기

```
생성된 빌드 스크립트 파일(build.gradle 또는 build.gradle.kts)을 검토해줘.
```

빌드 및 실행

- 터미널에서 다음 명령어를 실행하여 프로젝트를 빌드하고 실행한다.

```
# 빌드하기
gradle build

# 실행하기
gradle bootRun
```

- Agent에서 프로젝트를 빌드하고 실행한다.

```
프로젝트를 빌드하고 실행해줘.
```

현재 프로젝트 상태를 명세서에 저장하기

```
현재 프로젝트 상태를 프로젝트 명세서 파일(PROJECT_SPEC.md)에 저장해줘.
```

프로젝트 명세서 파일로 새 프로젝트 만들기

```
프로젝트 명세서(PROJECT_SPEC.md)의 내용대로 현재 작업 폴더를 스프링부트 프로젝트 폴더로 구성해줘.
```

실습 2 - 뷰 템플릿 도입하기

Thymeleaf 뷰 템플릿 엔진 추가하기

```
프로젝트에 Thymeleaf 뷰 템플릿 엔진을 추가해줘.
```

간단한 웹 페이지 만들기

```
"/" 경로로 접속하면 "Hello, Vibe!" 메시지를 보여주는 간단한 웹 페이지를 만들어줘.
- 컨트롤러 클래스: HomeController
- 뷰 템플릿 파일: home.html
```

현재 프로젝트 상태를 명세서에 저장하기

```
추가된 내용을 프로젝트 명세서 파일(PROJECT_SPEC.md)에 저장해줘.
```

실습 3 - CSS 프레임워크 도입하기

Bootstrap 5 CSS 프레임워크 추가하기

```
Thymeleaf 템플릿에 Bootstrap 5 CSS 프레임워크를 적용해줘.
- CDN 방식을 사용해줘.
```

현재 프로젝트 상태를 명세서에 저장하기

```
추가된 내용을 프로젝트 명세서 파일(PROJECT_SPEC.md)에 저장해줘.
```

실습 4 - 게시글 CRUD 구현하기 (without DBMS)

게시글 목록 조회 기능 구현하기

```
"게시글 목록 조회" 기능을 추가해줘.
- 게시글 속성:
    - 번호: no(Long)
    - 제목: title(String)
    - 내용: content(String)
    - 생성일: createdAt(LocalDateTime)
    - 수정일: updatedAt(LocalDateTime)
    - 조회수: views(Integer)
- 게시글 목록 조회
        - URL: /posts
        - 출력 항목: 번호, 제목, 생성일, 조회수
- 데이터 저장:
    - Java Collection API ArrayList를 사용하여 메모리 기반 저장
    - 예제 데이터 10개 미리 추가
- 컨트롤러 클래스: PostController
- 서비스 클래스: PostService
- 엔티티 클래스: Post
- 리포지토리 클래스: PostRepository
- 뷰 템플릿 파일:
    - 게시물 목록 화면: posts.html
```

게시글 상세 조회 기능 구현하기

```
"게시글 상세 조회" 기능을 추가해줘.
- 게시글 목록 조회 페이지 변경
    - 각 게시글 제목을 클릭하면 상세 조회 페이지로 이동
- 게시글 상세 조회
    - URL: /posts/{no}
    - 출력 항목: 번호, 제목, 내용, 생성일, 수정일, 조회수
```

```
    - 목록 버튼: 클릭하면 게시글 목록으로 페이지 이동
- 컨트롤러 클래스: PostController
- 서비스 클래스: PostService
- 엔티티 클래스: Post
- 리포지토리 클래스: PostRepository
- 뷰 템플릿 파일:
    - 게시물 상세 화면: post_detail.html
```

새 게시글 작성폼 기능 구현하기

```
"새 게시글 작성폼" 페이지를 추가해줘.
- "새 게시글 작성폼"
    - URL: /posts/new
    - 입력 항목: 제목(필수 입력), 내용(필수 입력)
    - 등록 버튼: 클릭하면 "새 게시글 등록" alert 창만 출력
    - 취소 버튼: 클릭하면 목록 조회 화면으로 이동
    - 컨트롤러 클래스: PostController
    - 뷰 템플릿 파일:
        - 게시글 작성 화면: post_new_form.html
- 게시글 목록 페이지 변경
    - 게시글 목록 페이지에 "새 글" 버튼 추가
    - "새 글" 버튼: 클릭하면 "새 게시글 작성폼" 페이지로 이동
```

새 게시글 등록 기능 구현하기

```
"새 게시글 등록" 기능을 추가해줘.
- 새 게시글 등록
    - URL: /posts/add
    - 새 게시글 작성폼 페이지의 입력 값을 받아서 저장 처리
    - createdAt: 현재 시각으로 설정
    - updatedAt: null 로 설정
    - views: 0 으로 설정
    - 처리 후 목록 조회 화면으로 이동
- 새 게시글 작성폼 변경
    - 등록 버튼: 클릭하면 "새 게시글 등록"을 요청한다.
- 컨트롤러 클래스: PostController
- 서비스 클래스: PostService
- 엔티티 클래스: Post
- 리포지토리 클래스: PostRepository
- 뷰 템플릿 파일: 없음
```

게시글 수정폼 기능 구현하기

```
"게시글 수정폼" 페이지를 추가해줘.
- "게시글 수정폼"
    - URL: /posts/{no}/edit
```

```
    - 출력 항목:
        - 번호(read-only)
        - 제목
        - 내용
        - 생성일(read-only, YYYY-MM-DD)
        - 수정일(read-only, YYYY-MM-DD, null이면 빈칸으로 표시)
        - 조회수(read-only)
    - 저장 버튼: 클릭하면 alert 창만 출력
    - 취소 버튼: 클릭하면 상세 조회 화면으로 되돌아 간다.
- "게시글 상세 페이지" 변경
    - 수정 버튼 추가
    - 수정 버튼 클릭하면 수정폼으로 페이지 이동
- 컨트롤러 클래스: PostController
- 서비스 클래스: PostService
- 엔티티 클래스: Post
- 리포지토리 클래스: PostRepository
- 뷰 템플릿 파일:
    - 게시글 수정 화면: post_edit_form.html
```

게시글 수정 기능 구현하기

```
"게시글 수정" 기능을 추가해줘.
- 게시글 수정
    - URL: /posts/{no}/save
    - 게시글 수정폼 페이지에서 보낸 '제목', '내용'을 받아서 변경 처리
    - updatedAt: 현재 시각으로 설정
    - 처리 후 상세 조회 화면으로 이동
- 게시글 수정폼 변경
    - 저장 버튼: 클릭하면 "게시글 수정"을 요청한다.
- 컨트롤러 클래스: PostController
- 서비스 클래스: PostService
- 엔티티 클래스: Post
- 리포지토리 클래스: PostRepository
- 뷰 템플릿 파일: 없음
```

게시글 삭제 기능 구현하기

```
"게시글 삭제" 기능을 추가해줘.
- 게시글 삭제
    - URL: /posts/{no}/delete
    - 삭제 처리 후 목록으로 페이지로 이동
- 게시글 상세 조회 페이지 변경
    - 삭제 버튼 추가
    - 삭제 버튼 클릭하면 "게시글 삭제"를 요청한다.
- 컨트롤러 클래스: PostController
- 서비스 클래스: PostService
- 엔티티 클래스: Post
```

```
- 리포지토리 클래스: PostRepository
- 뷰 템플릿 파일: 없음
```

게시글 목록 페이징 처리 기능 구현하기

```
"게시글 목록 페이징 처리" 기능을 추가해줘.
- 게시글 목록 조회 페이지 변경
    - 한 페이지에 5개 게시글씩 출력
    - 페이지 네비게이션 추가
- 컨트롤러 클래스: PostController
- 서비스 클래스: PostService
- 리포지토리 클래스: PostRepository
- 엔티티 클래스: Post
- 뷰 템플릿 파일:
    - 게시물 목록 화면: posts.html
```

자바 패키지 구조를 "기능형 구조"로 변경하기

```
현재 자바 패키지 구조를 다음과 같이 "기능형 구조"로 변경해줘.
com.example.vibeapp
├── VibeApp.java
├── home
│   └── HomeController.java
└── post
    ├── Post.java
    ├── PostController.java
    ├── PostRepository.java
    └── PostService.java
```

뷰 템플릿 파일 위치 변경하기

```
뷰 템플릿 파일들을 기능별로 다음과 같이 위치를 변경해줘.
templates/home/
    └── home.html
templates/post/
    ├── post_detail.html
    ├── post_edit_form.html
    ├── post_new_form.html
    └── posts.html
```

리팩토링 하기

```
```

```
자바 클래스 파일을 검토해줘. 메서드 이름이 실무의 관례를 따르는지 확인하고, 사용하지 않는 메서드는 제거해줘.
```

현재 프로젝트 상태를 명세서에 저장하기

```
변경 사항이나 프로젝트 상태를 프로젝트 명세서(PROJECT_SPEC.md)에 적용해줘.
```

실습 5 - DTO 패턴 적용하기

```
현재 프로젝트에 DTO 패턴을 적용해줘.
현재 상태:
- PostController가 Entity를 직접 받고 반환함
- 입력 검증 없음
- 메모리 기반 PostRepository 사용

요구사항:
- PostCreateDto, PostUpdateDto, PostResponseDTO, PostListDto 생성
- Bean Validation 적용 (제목 필수, 최대 100자)
- Service는 DTO만 입출력
- Entity → DTO 변환은 정적 팩토리 메서드 from() 사용
- DTO → Entity 변환은 DTO 내부의 toEntity() 메서드 사용
```

실습 6 - DTO를 record 문법으로 리팩토링하기

```
DTO 클래스를 Java의 record 문법으로 리팩토링해줘.
```

실습 7 - DBMS + Mybatis SQL Mapper 도입하기

프로젝트에 H2 데이터베이스와 MyBatis 설정하기

- 프롬프트 작성 방법을 묻는다.

  ```
  프로젝트에 H2 Database와 MyBatis SQL Mapper 설정을 추가하고 싶다.
  예시 프롬프트를 작성해줘.
  ```

- AI가 알려준 예시 프롬프트를 적절하게 수정한다.

  ```
  Spring Boot 프로젝트에 H2 데이터베이스와 MyBatis를 도입해줘.

  요구사항:
  1. build.gradle에 다음 의존성 추가:
  ```

```
    - H2 Database (runtime)
    - MyBatis Spring Boot Starter

2. application.yml 설정:
    - H2 파일 모드로 설정 (jdbc:h2:file:./data/testdb)
    - H2 Console 활성화 (/h2-console)
        - JakartaWebServlet 등록 포함
    - MyBatis 설정:
        * Mapper XML 위치: classpath:mapper/**/*.xml
        * Type aliases 패키지: com.example.vibeapp.post
        * Camel case 자동 변환 활성화
    - 디버깅 모드 활성화
    - 로깅
        - 스프링부트 autoconfigure 레벨: debug
        - 로그 파일명: ./logs/app.log

3. 프로젝트 루트에 .gitignore 업데이트:
- /data/ 디렉토리 제외
- *.mv.db, *.trace.db 제외

현재 Spring Boot 버전은 4.0.1이고, Java 25를 사용 중이야.
설정 후 정상 동작 여부를 확인할 수 있는 방법도 알려줘.
```

H2 데이터베이스가 SpringBoot 4.0.1 에서 자동 설정 안되는 문제 해결하기(필요 시)

```
Spring Boot 4.0.1에서 H2 데이터베이스가 자동 설정되지 않는 문제를 해결해줘.

요구사항:
1. 신규 설정 추가: com.example.vibeapp.config 패키지에 H2ConsoleConfig 클래스를 생성해줘.
  - H2 콘솔을 /h2-console 경로에 직접 매핑하는 설정을 포함해야 해.

현재 Spring Boot 4.0.1 버전은 Jakarta EE 네임스페이스를 사용한다. H2ConsoleConfig 클래스를 만들 때 이 점을 유의해줘.
```

H2 데이터베이스에 웹 콘솔로 접속하기

- 웹 브라우저에서 `http://localhost:8080/h2-console` 에 접속하여 H2 데이터베이스 웹 콘솔 페이지를 띄운다.
- 로그인 정보:
    - JDBC URL: `jdbc:h2:file:./data/testdb`
    - 사용자 이름: `sa`
    - 비밀번호: (빈칸)
    - Connect 버튼 클릭
    - 로그인이 안 된다면?
        - 에이전트를 통해 해결!

게시글 테이블 생성하기

```
자바 Post 클래스의 필드 값을 저장할 테이블 생성 SQL문을 작성해줘. 컬럼의 타입은 Post 클래스의 필드 타입에 맞춰줘.
- 테이블명: POSTS
- 컬럼:
    - NO: Pimary key, 자동 증가
    - TITLE: 최대 200자, 필수 입력
    - CONTENT: 10MB까지 입력 가능, 필수 입력
    - CREATED_AT: 기본값은 현재 시각
    - UPDATED_AT: 기본 값은 NULL
    - VIEWS: 기본 값은 0
```

- H2 데이터베이스 웹 콘솔에서 SQL문을 실행하여 POSTS 테이블을 생성한다.
- 스프링부트를 재시작할 때 POSTS 테이블을 자동 생성되게 만들고 싶다면?

  ```
  스프링부트를 재시작할 때 POSTS 테이블이 없으면 자동으로 생성되게 설정해줘.
  ```

  - schema.sql 파일에 POSTS 테이블 생성 SQL문이 추가된 것을 확인하라.
  - application.yml 파일에서 변경된 설정을 확인하라.

PostRepository를 MyBatis 기반으로 변경하기

```
현재 메모리 기반으로 구현된 PostRepository를 MyBatis SQL Mapper 기반으로 변경해줘.
- Mapper XML 파일 생성
    - 위치: src/main/resources/mapper/post/PostMapper.xml
    - 네임스페이스: com.example.vibeapp.post.PostRepository
    - SQL 매핑: CRUD 및 페이징 처리에 필요한 SQL문 작성
- PostRepository 인터페이스 수정
    - @Mapper 어노테이션 추가
    - 메서드 시그니처는 기존과 동일하게 유지
```

게시글 조회할 때 조회수 증가 기능 구현하기

```
게시글 조회할 때 조회수가 1 증가하는 기능을 추가해줘.
```

실습 8 - 트랜잭션 적용하기

게시글 태그 정보를 저장할 테이블 생성

```
게시글의 태그 정보를 저장할 테이블 생성 SQL문을 작성해줘.
- 테이블명: POST_TAGS
- 컬럼:
    - ID: Primary key, 자동 증가
```

```
    - POST_NO: 게시글 번호, POSTS 테이블의 NO 컬럼을 참조하는 외래 키
    - TAG_NAME: 태그 이름, 최대 50자, 필수 입력
```

게시글 태그 엔티티 클래스 만들기

```
게시글 태그 정보를 다루는 PostTag 엔티티 클래스를 만들어줘.
- 패키지: com.example.vibeapp.post
- 필드:
    - id(Long): 태그 고유 번호
    - postNo(Long): 게시글 번호
    - tagName(String): 태그 이름
```

게시글 태그 Repository 만들기

```
PostTag 엔티티를 다루는 PostTagRepository 인터페이스를 만들어줘.
- 패키지: com.example.vibeapp.post
- 메서드: 태그 추가, 태그 삭제 기능만 구현.

PostTagRepository의 MyBatis SQL Mapper 파일을 만들어줘.
- Mapper XML 파일:
    - 위치: src/main/resources/mapper/post/PostTagMapper.xml
    - 네임스페이스: com.example.vibeapp.post.PostTagRepository
    - DB 테이블: POST_TAGS
    - SQL 매핑: 태그 추가, 태그 삭제에 필요한 SQL문 작성

PostTagRepository 인터페이스와 Mybatis Mapper를 연동해줘.
```

게시글 등록폼과 수정폼에 태그 입력 기능 추가하기

```
게시글 등록폼과 수정폼에 태그를 입력할 수 있는 기능을 추가해줘.
- 입력 방식: 쉼표(,)로 구분된 태그 문자열 입력
- 등록폼:
    - URL: /posts/new
    - 입력 항목에 태그 입력 필드 추가
- 수정폼:
    - URL: /posts/{no}/edit
    - 입력 항목에 태그 입력 필드 추가
- 컨트롤러 클래스: PostController
- 서비스 클래스: PostService
- 리포지토리 클래스: PostRepository, PostTagRepository
- 뷰 템플릿 파일:
    - 게시글 작성 화면: post_new_form.html
    - 게시글 수정 화면: post_edit_form.html
```

게시글 상세 조회 페이지에 태그 출력 기능 추가하기

```
게시글 상세 조회 페이지에 해당 게시글의 태그를 출력하는 기능을 추가해줘.
- 출력 위치: 게시글 제목 아래
- 컨트롤러 클래스: PostController
- 서비스 클래스: PostService
- 리포지토리 클래스: PostRepository, PostTagRepository
- 뷰 템플릿 파일:
    - 게시물 상세 화면: post_detail.html
```

트랜잭션 미적용 시 문제점 확인하기

- 게시글을 등록할 때 태그 추가에 실패하는 상황을 시뮬레이션한다.
    - 게시글을 입력 시 3개의 태그를 입력한다.
    - 두 번째 태그는 DB 컬럼의 길이를 초과한 값을 입력한다.
    - 게시글 등록할 때 발생하는 예외를 확인한다.
    - 게시글 목록에서 방금 등록한 게시글이 있는지 확인한다.
    - 게시글 상세 조회에서 태그가 몇 개 저장되었는지 확인한다.

게시글 등록과 태그 등록을 하나의 트랜잭션으로 묶기

```
- 프로젝트의 트랜잭션 기능을 활성화해줘.
- 게시글 등록과 태그 등록을 하나의 트랜잭션으로 묶어줘.
- 게시글 수정과 태그 수정을 하나의 트랜잭션으로 묶어줘.
```

트랜잭션 적용 후 문제점 해결 여부 확인하기

- 게시글 등록할 때 태그 추가에 실패하는 상황을 다시 시뮬레이션한다.
    - 게시글을 입력 시 3개의 태그를 입력한다.
    - 두 번째 태그는 DB 컬럼의 길이를 초과한 값을 입력한다.
    - 게시글 등록할 때 발생하는 예외를 확인한다.
    - 게시글 목록에서 방금 등록한 게시글이 있는지 확인한다.

실습 9 - SQL Mapper를 ORM으로 바꾸기

MyBatis SQL Mapper에서 JPA ORM으로 전환하는 프롬프트 생생하기

```
현재 프로젝트를 MyBatis(SQL Mapper) 기반 구현에서 "순수 JPA ORM(EntityManager)사용"
방식으로 전환하고 싶다. 코딩 에이전트에게 요청할 프롬프트 예시를 작성해줘.
```

코드 생성하기

```

```

현재 프로젝트를 MyBatis(SQL Mapper) 기반 구현에서 "순수 JPA ORM(EntityManager)사용" 방식으로 전환해줘.
현재 프로젝트 상태는 프로젝트 명세서(PROJECT_SPEC.md) 파일에 저장되어 있어.

작업 목표:
    - MyBatis를 제거하고 순수 JPA(EntityManager)로 전환한다.
    - 주의: Spring Data JPA Repository는 사용하지 말라. EntityManager를 직접 사용해야 한다.

작업 요청사항:
1. 의존성 변경해줘. (build.gradle)
    - mybatis는 제거해줘.
2. JPA 설정 (application.yml)
    - 다음 설정을 YAML 형식에 맞춰서 변환하여 추가해줘:
    ```properties
    # JPA/Hibernate 설정
    spring.jpa.hibernate.ddl-auto=validate
    spring.jpa.show-sql=true
    spring.jpa.properties.hibernate.format_sql=true
    ```
3. Entity 클래스 변경
    - DB 테이블: POSTS, POST_TAGS
    - DB 테이블에 맞춰 기존 Post 클래스를 JPA Entity로 변경해줘.
4. Repository 계층 재구현
    - 중요: EntityManager를 직접 사용하여 PostRepository 클래스를 리팩토링해줘.
5. Mybatis 관련 코드 제거
    - Mpper 인터페이스 삭제
    - SQL Mapper XML 삭제
6 src/main/resources/schema.sql은 그대로 둔다.

제약사항 및 주의사항
반드시 지켜야 할 사항:
1. Spring Data JPA의 JpaRepository 인터페이스는 절대 사용하지 말라.
2. JPQL 사용
   - findAll() 같은 목록 조회는 JPQL로 작성
   - 예: `em.createQuery("SELECT p FROM Post p", Post.class).getResultList()`

3. 트랜잭션 유지
   - @Transactional은 Service 계층에만 유지
   - Respository 계층에는 선언하지 말 것.
   - JPA는 트랜잭션 내에서 동작해야 함

4. 영속성 컨텍스트 활용
   - update() 메서드는 변경 감지(Dirty Checking) 또는 merge() 사용
   - 주석으로 동작 원리 간단히 설명

코드 품질 요구사항
- 초급 학습자를 위한 명확하고 읽기 쉬운 코드
- 각 메서드에 JPA 동작 원리 주석 추가 (한 줄 정도)
- 예외 처리는 간단하게 (Optional, orElseThrow 활용)

실습 10 - Spring Data JPA로 리팩토링하기

```
# 프로젝트 전환 요청: 순수 JPA → Spring Data JPA

## 프로젝트 현재 상태
- EntityManager를 직접 사용하는 PostRepository 클래스 구현 완료
- JPQL로 쿼리 작성
- persist(), find(), createQuery() 등 직접 호출
- 페이징: setFirstResult(), setMaxResults() 사용
- 모든 CRUD 메서드를 수동으로 구현

## 목표
순수 JPA Repository를 Spring Data JPA로 리팩토링한다.
코드 감소와 생산성 향상을 체감하는 것이 핵심이다.

## 작업 요청사항

### 1. 의존성 확인 (build.gradle)
현재 `spring-boot-starter-data-jpa`를 이미 포함하고 있다.
추가할 의존성이 있는지 확인하여 추가해줘.

### 2. 기존 Repository 백업
리팩토링 전 기존 코드를 비교할 수 있도록:
- 현재 Repository 클래스들의 이름 뒤에 접미사 "Old"를 붙여서 별도 파일로 백업
- Before/After 비교를 위해 보존 필수

### 3. 기존 Repository를 Spring Data JPA Repository 로 변경

## 제약사항 및 주의사항

### 반드시 지켜야 할 사항

1. 인터페이스로 작성
   - 기존 Repository 클래스는 interface여야 함
   - @Repository 어노테이션 불필요 (JpaRepository 상속 시 자동)
   - 구현 클래스를 만들지 말라. (Spring Data JPA가 자동 생성)
2. 메서드 이름 규칙 준수
    - findBy + 필드명 + 조건
    - 예: findByTitle, findByTitleContaining, findByTitleAndContent

### 코드 품질 요구사항

- 각 Query Method에 주석으로 동작 설명 추가
- Before/After 비교 가능하도록 기존 코드 보존
```

실습 11 - 세션 기반 사용자 인증하기(without Spring Security)

사용자 테이블 생성하기

```
사용자 정보를 저장할 테이블 생성 SQL문을 작성해줘.
- 테이블명: USERS
- 컬럼:
    - NO: Pimary key, 자동 증가
    - NAME: 최대 50자, 필수 입력
    - EMAIL: 최대 50자, 필수 입력, 고유 값
    - PASSWORD: 최대 200자, 필수 입력
    - CREATED_AT: 기본값은 현재 시각
```

회원가입 구현하기

```
USERS 테이블에 맞춰서 회원가입을 처리하는 기능을 추가할 것이다. 적절한 프롬프트를 알려줘.
```

```
# 요청: 회원가입 기능 구현
## 1. 개요
최근 [schema.sql]에 추가된 `USERS` 테이블을 바탕으로 회원가입 기능을 구현해줘.
기존 프로젝트의 디자인 가이드라인(Glassmorphism, Bootstrap 5)을 엄격히 준수해야 해.

## 2. 작업 상세
### 백엔드 (Java)
1. Entity: `com.example.vibeapp.user.User` 엔티티 클래스 작성 (`@Entity`,
`@Table(name="USERS")` 적용)
2. Repository: `UserRepository` 인터페이스 작성 (`Spring Data JPA` 사용)
    - 이메일 중복 확인을 위한 `existsByEmail(String email)` 메서드 포함
3. DTO: 회원가입 요청을 처리할 `UserSignupDto` (Spring Validation 적용: 이름/이메일
필수, 이메일 형식 체크 등)
4. Service: `UserService` 작성
    - 이메일 중복 시 예외 발생 처리
    - 비밀번호 암호화 로직 (현재 단계에서는 단순화를 위해 평문 또는 간단한 해싱 처리 로직만 위치
확보)
5. Controller: `UserController` 작성
    - `GET /signup`: 회원가입 폼 화면 반환
    - `POST /signup`: 회원가입 처리 후 메인 화면(`/`)으로 리다이렉트 (성공 메시지 포함)
### 프론트엔드 (Thymeleaf/CSS)
1. View: `src/main/resources/templates/user/signup.html` 생성
    - 기존 `home.html`의 'Glassmorphism 스타일'과 '애니메이션 효과'를 그대로 적용
    - 이름, 이메일, 비밀번호, 비밀번호 확인 필드 포함
    - Bootstrap 5의 검증 스타일(is-invalid)을 사용해 에러 메시지 표시
## 3. 제약 및 요구사항
- [PROJECT_SPEC.md]의 파일 구조 및 프로젝트 상태 섹션을 업데이트할 것.
- 모든 날짜 처리는 기존 규칙대로 `yyyy-MM-dd` 형식을 고려할 것.
- 패키지 구조는 `com.example.vibeapp.user`를 사용하여 기능별로 분리할 것.
```

로그인 및 로그아웃 기능 구현하기

로그인 및 로그아웃 기능을 추가할 것이다. 적절한 프롬프트를 알려줘.

```
# 요청: 로그인 기능 구현 (HttpSession 기반)

## 1. 개요
이미 구현된 `USERS` 테이블과 회원가입 기능을 바탕으로 로그인 기능을 구현해줘.
Spring Security 없이 `HttpSession`을 직접 사용하는 방식으로 진행하며, 기존 디자인 가이드라인(Glassmorphism)을 준수해야 해.

## 2. 작업 상세

### 백엔드 (Java)
1. DTO: `UserLoginDto` (이메일, 비밀번호 필수 검증 포함)
2. Service: [UserService]에 `login` 메서드 추가
    - 이메일로 사용자 조회 후 비밀번호 일치 여부 확인
    - 인증 실패 시 적절한 예외 발생
3. Controller: [UserController]에 로그인 처리 추가
    - `GET /login`: 로그인 폼 화면 반환
    - `POST /login`: 로그인 처리 후 성공 시 `HttpSession`에 사용자 정보 저장 및 `/`으로 리다이렉트
    - `GET /logout`: 세션 만료 처리 및 홈 화면으로 리다이렉트

### 프론트엔드 (Thymeleaf/CSS)
1. View: `src/main/resources/templates/user/login.html` 생성
    - `signup.html`과 동일한 **Glassmorphism 디자인** 적용
    - 이메일, 비밀번호 입력 필드 및 로그인 버튼 포함
2. Navigation/Home: `home.html` 및 공통 레이아웃 반영
    - 세션에 로그인 정보가 있는 경우: "로그아웃", "XX님 환영합니다" 표시
    - 세션에 정보가 없는 경우: "로그인", "회원가입" 링크 표시

## 3. 제약 및 요구사항
- [PROJECT_SPEC.md]의 파일 구조 및 프로젝트 상태 섹션을 업데이트할 것.
- 로그인 성공/실패 시 `RedirectAttributes`를 사용하여 사용자에게 알림 메시지를 전달할 것.
- `src/main/java/com/example/vibeapp/user` 패키지 내에 관련 클래스들을 위치시킬 것.
```

인가 기능 구현하기

```
'인가(authorization)'' 기능을 추가할 것이다.
- 로그인한 사용자만 게시글 작성할 수 있다.
- 게시글 수정과 삭제는 그 게시글을 작성한 사용자만 할 수 있다.
- 이런 '인가' 기능은 중앙에서 관리한다.
이런 작업을 지시할 적절한 프롬프트를 알려줘.
```

```
현재 프로젝트에 Spring Security 없이 '세션 기반 인가(Authorization)' 기능을 중앙 집중식으로 구현해줘. 구체적인 요구사항은 다음과 같아:
```

1. 데이터 스키마 수정:
- POSTS 테이블에 작성자를 식별할 수 있는 user_no 컬럼을 추가하고 USERS 테이블에 대한 외래 키(FK)를 설정해줘.
- Post 엔티티와 관련 DTO(PostCreateDto 등)에도 작성자 정보를 반영해줘.

2. 중앙 집중식 인증 체크 (Authentication Interceptor):
- HandlerInterceptor를 구현하여 로그인 여부를 체크하는 기능을 중앙에서 관리해줘.
- 게시글 작성(/posts/new, /posts/add), 수정, 삭제와 같이 로그인이 필요한 경로(PathPatterns)에 대해 인터셉터를 적용하고, 비로그인 사용자는 로그인 페이지로 리다이렉트해줘.

3. 게시글 소유권 확인 로직 (Ownership Check):
- 게시글 수정 및 삭제 시, **'현재 로그인한 유저'**와 **'게시글 작성자'**가 일치하는지 확인하는 로직을 추가해줘.
- 이 기능은 PostService에서 처리하거나 별도의 권한 관리 컴포넌트를 통해 중앙에서 수행될 수 있도록 해줘. 권한이 없는 경우 적절한 예외를 던지고 유저에게 안내 메시지를 보여줘야 해.

4. UI 동적 렌더링:
게시글 상세 페이지(post_detail.html)에서 현재 로그인한 사용자가 게시글의 작성자일 경우에만 '수정' 및 '삭제' 버튼이 보이도록 Thymeleaf 코드를 수정해줘.

5. 문서 업데이트:
변경된 데이터 구조와 인가 정책을
PROJECT_SPEC.md
에 반영해줘.

사용자 비밀번호 암호화하기

사용자 비밀번호가 DB에 평문으로 저장되어 있다. 이것을 암호화하여 저장하고 싶다. 로그인할 때도 암호화된 비밀번호와 비교하도록 처리하고 싶다. 암호화는 Spring Security의 BCryptPasswordEncoder를 사용하여 처리한다. 다만, Spring Security를 적용하지는 않는다. 이런 요구사항을 프롬프트로 작성해줘.

[목표] 현재 프로젝트의 사용자 비밀번호를 평문 저장 방식에서 BCrypt 암호화 저장 방식으로 리팩토링한다.

[상세 요구사항]

1. 의존성 추가: build.gradle에 BCryptPasswordEncoder를 사용하기 위한 의존성을 추가한다. Spring Security 전체를 추가하지 않고, 암호화 기능만 사용할 수 있도록 한다.
3. Bean 등록:
WebConfig.java
 또는 별도의 설정 클래스에서 BCryptPasswordEncoder를 @Bean으로 등록한다.
4. 회원가입 로직 수정: UserService.signup()에서 사용자가 입력한 평문 비밀번호를 passwordEncoder.encode()를 사용하여 암호화한 뒤 DB에 저장한다.
5. 로그인 로직 수정: UserService.login()에서 String.equals() 대신 passwordEncoder.matches(rawPassword, encodedPassword)를 사용하여 암호화된 비밀번호와 입력값을 검증한다.
6. 데이터베이스 확인:
schema.sql

```
의 users 테이블 password 컬럼 길이가 암호화된 해시값(약 60자 이상)을 충분히 담을 수 있는지
확인한다. (이미 VARCHAR(200)인 경우 유지)
```

실습 12 - Spring Security 도입하기

```
현재 프로젝트에 Spring Security를 도입하여 인증 및 인가 기능을 구현하고 싶다.
다음 요구사항을 반영한 프롬프트를 작성해줘.
- 사용자 인증: Spring Security의 인증 매커니즘을 사용하여 로그인 및 로그아웃 기능 구현
- 비밀번호 암호화: BCryptPasswordEncoder를 사용하여 비밀번호 암호화
- 인가(Authorization): 특정 URL 경로에 대한 접근 제어 설정
- 세션 관리: 동시 세션 제어 및 세션 고정 보호 설정
- CSRF 보호: CSRF 공격 방지를 위한 설정 활성화
- 커스텀 로그인 페이지: 기존 로그인 페이지를 Spring Security와 통합
```

```
# VibeApp Spring Security 통합 구현 요청

## 프로젝트 환경:
PROJECT_SPEC.md 파일에 명시된 현재 프로젝트 상태를 기준으로 작업을 진행해줘.

## 요구사항 상세:

### Spring Security 의존성 추가 및 기본 설정:

- build.gradle에 spring-boot-starter-security 의존성을 추가해줘.
- SecurityConfig 설정을 통해 Security Filter Chain을 구성해줘.

### 사용자 인증 (Authentication):

- Spring Security의 표준 인증 메커니즘을 사용해줘.
- 기존 User 엔티티와 UserRepository를 활용하여 UserDetailsService 및 UserDetails를
구현해줘.
- 사용자 식별값은 email을 사용해줘.
- 로그아웃 기능을 구현하고, 로그아웃 시 세션 무효화 및 홈 화면(/) 리다이렉트를 처리해줘.

### 비밀번호 암호화:

- BCryptPasswordEncoder를 PasswordEncoder 빈으로 등록하여 사용해줘.
- 회원가입 시 비밀번호를 암호화하여 저장하고, 로그인 시 암호화된 비밀번호와 비교하도록 설정해줘.

### 인가 (Authorization):

- 특정 URL 경로에 대한 접근 제어를 설정해줘.
    - 전체 허용: /, /signup, /login, /css/**, /js/**, /images/**, /h2-
console/**
    - 인증 필요: /posts/** (게시글 작성, 수정, 삭제, 상세 조회 등 모든 게시글 관련 활동)

### 세션 관리 (Session Management):

- 동시 세션 제어: 사용자당 최대 세션 수를 1개로 제한하고, 기존 세션을 만료시키는 방식을 적용해
```

```
줘.
- 세션 고정 보호: 로그인 시 세션 ID를 새로 생성하는 migrateSession 설정을 적용해줘.

### CSRF 보호:

- CSRF 공격 방지를 위한 설정을 활성화해줘.
- 개발 편의를 위해 /h2-console/** 경로는 CSRF 검사에서 제외해줘.
- Thymeleaf의 th:action을 사용하여 폼(Form)에서 CSRF 토큰이 자동으로 포함되도록 확인해줘.

### 커스텀 로그인 페이지 통합:

- 기존에 작성된 user/login.html 페이지를 Spring Security의 로그인 페이지로 지정
(loginPage("/login"))해줘.
- 로그인 성공 시 /로 리다이렉트되도록 설정해줘.

### 기존 코드 정리 (Legacy Cleanup):

- UserController에서 HttpSession을 직접 관리하던 로그인/로그아웃 로직을 제거해줘.
- 기존의 LoginInterceptor 및 WebConfig 내 인터셉터 등록 코드를 Spring Security 설정
으로 대체하며 삭제해줘.
- WebConfig에 선언된 PasswordEncoder 빈을 SecurityConfig로 이동해줘.
- 위 요구사항을 바탕으로 build.gradle, SecurityConfig, UserDetailsService 구현체,
그리고 수정이 필요한 Controller 및 View 파일들을 순서대로 작성해줘.
```

실습 13 - SSR 방식에서 CSR 방식으로 전환하기

```
현재 프로젝트를 SSR(서버 사이드 렌더링) 방식에서 CSR(클라이언트 사이드 렌더링) 방식으로 전환하고
싶다. 다음 요구사항을 반영한 실행 계획을 작성해줘.
- Controller 클래스:
    - @Controller → @RestController 변경
    - ResponseEntity 사용하여 JSON 응답 반환
    - REST API 설계 원칙에 따라 엔드포인트 설계
    - REST API 엔드포인트 예시:
        - 게시글 목록 조회: GET /api/posts
        - 게시글 상세 조회: GET /api/posts/{no}
        - 새 게시글 등록: POST /api/posts
        - 게시글 수정: PATCH /api/posts/{no}
        - 게시글 삭제: DELETE /api/posts/{no}
- 예외 처리:
    - @RestControllerAdvice 사용하여 전역 예외 처리
    - JSON 형식으로 에러 응답 반환
- 프론트엔드:
    - 기존의 Thymeleaf 뷰 템플릿을 정적 리소스 파일(HTML, CSS, JS)로 변경
    - Fetch API를 사용하여 REST API와 통신
    - Thymeleaf 뷰 템플릿 제거
```

실습 14 - 토큰(JWT) 기반 인증 방식으로 전환하기

현재 프로젝트의 인증 방식을 세션 기반에서 토큰(JWT) 기반 인증 방식으로 변경하려 한다.
예시 프롬프트를 작성해줘.

요청: 현재 프로젝트의 세션 기반 인증을 JWT 기반 인증으로 전환해줘

1. 개요
- 현재 HttpSession을 사용하는 세션 기반 인증 방식을 JWT(JSON Web Token) 기반의 무상태(Stateless) 인증 방식으로 변경하고 싶어. 프론트엔드와 백엔드 모두에 필요한 작업을 수행해줘.

2. 백엔드 요구사항

- 의존성 추가: JWT 구현을 위해 jjwt 라이브러리(또는 최신 Java 25와 호환되는 JWT 라이브러리)를 build.gradle에 추가해줘.
- JWT 유틸리티 클래스 구현: 토큰 생성, 검증, 클레임 추출 기능을 가진 JwtTokenProvider 클래스를 만들어줘. (Secret Key는 환경 설정 파일에서 관리하도록 설정)
- 로그인 API 수정:
    - 로그인 성공 시 세션을 생성하는 대신 JWT 액세스 토큰을 생성해서 JSON 응답 바디로 반환해줘.
    - 응답 바디 예시: {"accessToken": "...", "tokenType": "Bearer", "userName": "..."}
- SecurityConfig 수정:
    - 세션 정책을 SessionCreationPolicy.STATELESS로 변경해줘.
    - 모든 요청에서 JWT를 검증할 수 있도록 JwtAuthenticationFilter를 커스텀 필터로 등록해줘.
    - 기존의 formLogin 핸들러 대신 JWT 기반의 커스텀 필터나 컨트롤러 로직을 사용하도록 변경해줘.

3. 프론트엔드 요구사항

- 로그인 처리 (login.html): 로그인 성공 시 서버에서 받은 accessToken을 localStorage에 저장하도록 수정해줘.
- 인증 요청 처리: 모든 fetch 요청 시 헤더에 Authorization: Bearer <token>을 포함하도록 공통 로직을 수정해줘.
- 상태 체크 및 로그아웃:
    - index.html에서 인증 상태를 체크할 때 localStorage의 토큰 유무를 확인하고, 로그아웃 시 토큰을 삭제하도록 처리해줘.

4. 기타
- 전환 과정에서 PROJECT_SPEC.md와 walkthrough.md를 최신 상태로 업데이트해줘.
- 기존의 세션 기반 인터셉터나 설정 중 더 이상 필요 없는 코드는 정리해줘.