SpringBoot covid-19 Project

Handler Method: Spring Web 에서 사용자의 요청(request)을 받아 응답(response)을 리턴 하는 메소드

1. 매핑 정보
2. 요청
3. 응답

@RequestMapping:

* Name: 뷰 템플릿에서 식별할 때 쓰는 이름
* Value, path: URI
* Method: HTTP method (ex: GET, POST…)
* Params: 파라미터 검사
* Headers: 헤더 검사
* Consumes: 헤더의 Content=Type 검사
* Produces: 헤더의 Accept 검사

@RequestMapping shortcuts:

* @GetMapping
* @PostMapping
* @PutMapping
* @DeleteMapping
* @PatchMapping

Handler Method가 받을 수 있는 요청들: 메소드 파라미터로 적어 넣을 수 있는 타입들

* ServletRequest, ServeltResponse, HttpSession
* WebRequest, NativeWebRequest
* @RequestParam, @PathVariable
* @RequestBody, HttpEntity<B>
* @ModelAttribute, @SessionAttribute, Model, ModelMap
* @RequestHeader, @CookieValue
* Principal, Locale, TimeZone, InputStream, OutputStream, Reader, Writer…

Handler Method가 내보낼 수 있는 응답들: 메소드가 리턴할 수 있는 타입들

* ModelAndView
* String, View
* @ModelAttribute, Map, Model
* @ResponseBody
* HttpEntity<B>, ResponseEntity<B>
* HttpHeaders
* Void 등등

스프링 부트 기본 에러 응답의 응용

BasicErrorController: 스프링 부트의 기본 응답이 마음에 든다면

* BasicErrorController를 상속 받아서
* 특정 메소드만 오버라이드 하거나
* 특정 핸들러 메소드를 추가하는 식으로 응용
* BasicErrorController의 핸들러 메소드
* BasicErrorController.errorHtml() 🡺 뷰 응답
* BasicErrorController.error() 🡺 json body 응답

커스텀 에러 페이지: 기본

간단히 static html 이나 template 파일을 추가해서 커스텀 페이지를 등록하는 법

* 단일 기본 페이지
* /resources/static/error.html
* /resources/public/error.html
* /resources/template/error.[템플릿확장자]
* http status 별 기본 페이지
* /resources/[static|public|template]/error/{http status 번호}.[html|템플릿 확장자]
* /resources/[static|public|template]/error/4xx.[html|템플릿확장자]
* /resources/[static|public|template]/error/5xx.[html|템플릿확장자]

@ExceptionHandler

비즈니스 로직이 던진 예외에 반응하는 핸들러 메소드

* 위치: 특정 컨트롤러 클래스 내부 or @ControllerAdvice 컴포넌트 내부
* 특정 예외에 반응
* 예외 처리 범위
* 컨트롤러 안에 작성했을 경우: 해당 컨트롤러만
* @ControllerAdvice에 작성했을 경우: 프로젝트 전체

@ControllerAdvice: @ExceptionHandler를 모아서 글로벌하게 적용할 때 스는 annotation

* 종류:
* @ControllerAdvice
* @RestControllerAdvice = @ControllerAdvice + @ResponseBody
* 속성:
* Value == basePackages
* basePackages: 적용 범위를 문자열을 이용해 특정 패키지로 지정
* basePackageClasses: 적용 범위르 대표 클래스 한 개를 이용해 특정 패키지로 지정
* basePackages를 type-safe하게 사용하기 위해 제공하는 옵션
* assignableTypes: 적용 범위를 특정 클래스에 할당할 수 있는 컨트롤러로 지정
* anootations: 적용 범위를 특정 annotation을 사용한 컨트롤러로 지정

ResponseEntityExceptionHandler: Spring MVC에서 내부적으로 발생하는 예외들을 처리하는 클래스

* API 예외 처리를 담당하는 @ControllerAdvice 클래스에서 상속받아 사용
* 커스터마이징을 원하는 특정 메소드를 오버라이드

@ControllerAdvice 강의 요약:

* 보편적인 에러 페이지를 등록하는 법
* 공통 에러 페이지를 에러 컨트롤러 인터페이스를 구현해서 만드는 법
* View와 json body의 출력을 따로 떨어뜨리는 방법
* 세부적으로 만들기 위해서 exception handler를 사용하는 방법
* 클래스 레벨에서 구현
* 전역에 적용하기 위해서 @ControllerAdvice를 사용한 방법

Spring Boot Test

@SpringBootTest 기본 구성: 일단 스프링 컨테이너와 스프링 부트 환경을 포함하는 테스트는 이것으로 다 작성 가능

* 통합 테스트를 할 때 적절한 선택
* Application context를 로드하는데 시간이 걸리므로 테스트가 다소 무거운편
* @SpringBootTest
* 포함한 annotation: @BooststrapWith + @ExtendWith
* 흥한 실수: Junit 5 쓸 때 @ExtendWith(SpringExtension.calss) 쓰지 않기
* @Test
* Junit 5 annotation
* 단위 테스트 메소드를 지정

@SpringBootTest

* Value, properties: 프로퍼티 설정
* Args: application 실행 시 커맨드 라인으로 입력하는 인자(옵션) 설정
* Classes: ApplicationContext 로딩을 위한 설정 클래스를 직접 지정
* webEnvironment: ApplicationContext의 웹 환경 설정
* WebEnvironment.Mock: mock servlet, embedded server 동작 X
* @AutoConfigureMockMvc, @AutoConfigureWebTestClient 와 함께 써서 mock test 가능
* WebEnvironment.RANDOM\_PORT: 랜덤 포트, Embedded server 동작
* WebEnvironment.DEFINED\_PORT: 포트 지정(server.port), embedded server 동작
* WebEnvironment.NONE: 웹 환경 구성 안함, embedded server 동작 X

@WebMvcTest: Spring MVC 컨트롤러 레이어를 슬라이스 테스트할 때 사용

* MockMvc 빈을 자동 설정하고 테스트에 사용
* 로드 할 컨트롤러 클래스를 지정 가능 (기본 동작: 전체 컨트롤러 로드)

Test Driven Development (TDD)

테스트 주도 개발 기법

* 프로그램의 설계와 구현, 사고의 흐름을 테스트 중심으로 생각하는 개발 방법
* 개발 순서의 변화
* As-is: 구현한다 🡺 테스트한다
* To-be: 테스트를 만든다 🡺 구현한다.
* 주요 키워드: 익스트림 프로그래밍 (XP), 애자일, 폭포수 모델, Test-First Programming

TDD 개발 사이클

1. RED: (실패하는) 테스트를 짠다. (요구사항의 명세)
2. Green: 테스트를 성공시킨다. (구현)
3. REFACTOR: 구현 코드를 고도화(리팩토링)한다.

Given – When – Then: 테스트의 구조를 표현하는 방법 (a.k.a 3A, Arrange – Act – Assert)

* Given (Arrange): 상태(state)의 정의 – 테스트를 수행할 때 전제 조건
* When (Act): 동작 – 테스트 실행
* Then (Assert): 검증 – 동작의 결과 (actual) vs. 예상값(expected)

Why TDD? Why Test?

* 개발자가 지금 뭘 하려는지 명확히 안다는 사실을, 스스로 지속적으로 확인한다.
* 개발이 지연되는 이유 중 하나는, 막막해서 멍때리기 때문
* 지금 뭘 하려는지 명확히 안다는 사실을, 팀원과 소스 코드로 공유하고 소통(코드 리뷰)한다.
* As-is 1: 개발 계획을 별도의 문서로 공유함
* As-is 2: 개발 계획을 구현 코드로 공유함
* To-be: 개발 계획을 테스트 코드로 공유함

최고의 효율로 테스트 개발을 하려면: TDD가 기존 개발 방식보다 효율적이기 위해서는

* 테스트 설계 흐름에 익숙해야 한다.
* 사람의 요구사항을 프로그램이 할 수 있는 기능으로 변환하기
* 기능을 단위 기능으로 세분화하기
* 기능의 관계와 상호작용을 설계하기
* 테스트 작성 기술에 익숙해야 한다.

테스트 내용의 발전 과정(가이드)

1. 메인 요구사항의 기본 목표 위주로만 우선 테스트를 작성

* 날짜를 yyyymmdd 포맷으로 입력하면, 정산일 (D+3)을 계산해준다.

1. 메인 요구사항 기본 + 세부 목표를 테스트로 작성

* 날짜를 yyyymmdd 포맷으로 입력하면, 정산일 (D+3)을 계산해준다.
* 입력값이 없으면, 사용자에게 적절한 경고를 표시
* 2000.01.01 이전 날짜 입력은, 최초 정산일(2000.01.01)을 출력
* 매월 말일은, 빠른 정산일(D+2)을 계산

1. 메인 요구사항 기본 + 세부 + 더욱 구체적인 기능적 고려 요소

* 날짜를 yyyymmdd 포맷으로 입력하면, 정산일 (D+3)을 계산해준다.
* yyyy-mm-dd는? Yymmdd는? 20210000은? 20210231은?
* 입력값이 없으면, ~~사용자에게 적절한 경고를 표시해준다~~. 🡺 에러로 응답한다.
* Null 은? “”는? “ “는? “ “는?
* 2000.01.01 이전 날짜 입력은, 최초 정산일(2000.01.01)을 출력
* 19991230? 19991231? 20000101?
* 매월 말일은 빠른 정산일 (D+2)을 계산
* 20210101? 20210102? 20210130?

@Service의 이해

비즈니스 로직을 담당하는 컴포넌트

* 도메인 모델(데이터)과 컨트롤러 사이에 위치
* 클래스 네이밍을 반드시 `-Service`로 할 필요 없음
* 하지만 관례적으로 실무에서 즐겨 사용되는 네이밍
* 구현 방식

1. 인터페이스 + 클래스 : 정석

* 인터페이스를 요구사항에 따른 기능 문서 (ex:Javadoc)로 작성
* 같은 기능을 하는 다양한 구현체를 작성할 니즈가 예상될 때 적합

1. 클래스

* 실무에서는 즐겨 사용되기도 하는 방식
* 인터페이스를 추가로 작성하는 수고를 덜게 됨
* 아무래도 컨트롤러 레이어와 결합도는 좀 더 증가됨

Validation

사용자 입력값에 있을지 모르는 오류를 처리하려면

* 모든 입력단 앞에 방어 코드가 추가됨
* 방어 코드는 복잡하고 반복적임 – 대표적인 boilerplate code

Annotation 기반으로 데이터 검증을 돕는 JSR-303, JSR-380 표준이 도입

* 검증 구현과 비즈니스 로직을 분리하고, 비즈니스 로직에 더 집중 가능
* 간결한 코드 표현, 더 나은 가독성

Validation in Boot: 사용 패턴

1. @Validated \_ 메소드 파라미터 검증

* 메소드 파라미터에 validation annotation 을 직접 사용해서 검증하는 방법
* 클래스에 @Validated 필요
* 발생 예외: ConstraintViolationException, 직접 처리해 줘야 하는 예외
* ConfigurationProperties 클래스에도 적용 가능

1. @Valid + Data Object

* 검증 하려는 데이터 오브젝트에만 검증 로직을 적용
* @Validated 필요하지 않음
* 발생 예외: MethodArgumentNotValidException
* ResponseEntityExceptionHandler 지원을 받을 수 있음

**Spring Boot Properties**

자주 등장하거나 유용한 프로퍼티들

1. Logging

* Debug
* Trace
* Logging.level.원하는.패키지.이름: logging.level.com.kim.getinline = debug

1. Config

* Spring.config.activate.on-profile: spring.profiles가 deprecated 되었으니 주의
* Spring.config.import
* Spring.config.use-legacy-processing

1. Main

* Spring.main.lazy-initialization: 모든 빈 등록을 spring 컨테이너가 처음에 뜰 때, 빈 로딩을 할 때 인스턴스 하는 걸 최대한 미룬다

**Thymeleaf**

Thymeleaf만의 특징, 강점: HTML5 웹 표준을 준수하는 템플릿

* 전체적인 문법이 html5 마크업 표준을 최대한 해치지 않게끔 설계
* Decoupled logic: 템플릿 문법을 아예 템플릿에서 분리 가능
* 순수한 마크업만 남음 -> 템플릿 엔진이 작동하지 않아도 렌더링되는 정적 목업 페이지
* 디자이너가 이해하기 쉬운 코드

템플릿 문법 적용 방법:

1. “th:” tag
2. “data-th-“ attribute
3. Decoupled logic

**Expressions: 변수에 대한 참조 등**

Variable Expressions: ${…} (변수)

Selection Variable Expressions: \*{…}

Message Expressions: #{…} (외부 프로퍼티에서 메시지 읽는법)

Link URL Expressions: @{…}

Fragment Expressions: ~{…}

**Literals: 인라인으로 코드 바로 표현하는 법**

Text literals: ‘one text’, ‘Another one!’

Number literals: 0, 34, 3.0, 12.3 ….

Boolean literals: true, false

Null literal: null

Literal tokens: one, sometext, main …

**Text operations**

String concatenation: +

Literal substitutions: |The name is ${name}| (치환)

**Arithmetic operations**

Binary operators: +,-,\*,/,%

Minus sign \*unary operator): -

**Conditional operators**

If-then: (if) ? (then)

If-then-else: (if) ? (then) : (else)

Default: (value) ?: (defaultvalue)

**Special tokens:**

No-Operation: \_

**Rest repository**

Repository 로부터 REST API를 자동으로 만들어주는 기술

* Spring Data Rest를 사용
* 페이징, 정렬, 리미트 사용 가능
* QueryDSL Extension을 함께 사용하면 큰 노력 없이 컬럼별 검색 기능까지 사용 가능
* 실무에서는 많이 사용하지 않는다.
* 도메인을 직접 api에 노출하는 것을 꺼림
* 복잡한 요구사항에 맞춰 api 설게를 하고 싶은 경우에는 부적합
* 설정을 잘 하지 않으면 불필요한 api가 노출될 수도 있음
* 실무에서 API는 필요에 따라 직접 만드는 편
* 사용할 경우
* 내부에서 사용하는 애플리케이션을 만들 때
* 도메인이 복잡하지 않을 때
* 요구사항이 복잡하지 않고 심플한 CRUD로 구성되어 있을 때
* 빠르게 만들어야 할 때

**Spring Boot Actuator**

스프링 부트의 다양한 설정과 지표를 모니터링할 수 있는 도구

* 만든 빈이 잘 등록되었는지 확인
* 만든 빈이 다른 빈들과 어떤 연관 관계를 맺고 있는지 확인
* 설정한 환경변수, 프로퍼티가 잘 등록되었는지 확인
* 의도한 엔드포인트가 잘 노출되고 있는지 확인
* 의도한 로깅이 잘 일어나고 있는지 확인
* 헬스 체크: 시스템이 정상 동작하고 있는지 확인
* 캐시가 잘 동작하고 있는지 확인

기본 엔드포인트:

* Auditevents, beans, caches, conditions, configprops, env,…

웹애플리케이션에서 추가로 활성화되는 엔드포인트

* Heapdump, jolokia, logfile, Prometheus

**JPA**

ORM (object Relational Mapping):

객체 지향 언어를 이용하여, 서로 호환되지 않는 타입 간의 데이터를 변환하는 기술

* 좁은 의미: DB(RDBMS) 테이블 데이터를 (자바) 객체와 매핑하는 기술
* 효과: RDBMS를 객체 지향 유로 가상화하는 것
* ORM으로 얻고자 하는 것
* DB의 추상화: 특정 DB에 종속된 표현(ex:SQL)이나 구현이 사라지고, DB 변경에 좀 더 유연해짐
* 객체의 이점을 활용: 객체간 참조, type-safety
* 관심사 분리: DB 동작에 관한 코드 작성의 반복을 최소화하고 비즈니스 로직에 집중

JPA(Jakarta(java) Presistence API): 자바에서 ORM 기술을 사용해 RDBMS를 다루기 위한 인터페이스 표준 명세

* API + JPQL + metadata ( + Criteria API)
* 기본적으로 관계형 데이터베이스의 영속성(persistence)만을 규정
* JPA 구현체 중에 다른 유형의 데이터베이스 모델을 지원하는 경우가 있지만, 원래 JPA 스펙과는 무관
* 이름의 변화
* Java Persistence API -> Jakarta Persistence API
* Spring Boot: 2.2부터 Jakarta EE로 의존성이 변경됨
* 현재: JPA 2.2.3 (패키지명은 아직 javax.persistence.\*)
* 미래: JPA 3.0이 도입되면 패키지명이 완전히 JAKARTA.PERSISTENCE.\*로 변경될 전망

Persistence (영속성):

프로세스가 만든 시스템의 상태가 종료된 후에도 사라지지 않는 특성

* 구현 방법: 시스템의 상태를 데이터 저장소에 데이터로 저장한다.
* 사라지는 데이터 – 주기억장치(휘발성 스토리지)에 저장된 데이터
* 프로세스 메모리 안의 데이터 (변수, 상수, 객체, 함수 등)
* 사라지지 않는 데이터 – 보조기억장치(비휘발성 스토리지)에 저장된 데이터
* 하드디스크, SSD에 기록된 데이터 (파일, 데이터베이스 등)
* 영속성 프레임워크: 영속성을 관리하는 부분을 persistence layer로 추상화하고, 이를 전담하는 프레임워크에게 관리를 위임
* JPA에서 persistence 란: 프로세스가 DB로부터 읽거나 DB에 저장한 정보의 특성

Jakarta(Java) Persistence Query Language:

플랫폼으로부터 독립적인 객체 지향 쿼리 언어

* JPA 표준의 일부로 정의됨
* RDBMS의 엔티티(Entity)를 다루는 쿼리를 만드는데 사용
* SQL의 영향을 받아서 형식이 매우 유사

JPQL Example:

SELECT DISTINCT a

FROM Author a

INNER JOIN a.books b

WHERE b.publisher.name = ‘XYZ Press’

AND a.lastName IS NULL

;

SQL(MySQL) Example:

SELECT DISTINCT a.\*

FROM author a

INNER JOIN book b on b.id = a.book\_id

INNER JOIN publisher p on p.id = b.publisher\_id

WHERE p.name = ‘XYZ Press’

AND a.last\_name IS NULL

SQL 과 JPQL은 다른 언어이다.

* SQL: 표준 ANSI SQL을 기준으로 만든, 특정 DB에 종속적인 언어
* JPQL: 특정 DB에 종속적인 언어가 아님
* JPA 프레임워크를 사용한다면
* 특별한 요구사항이 있지 않는 한, JPQL을 몰라도 된다

SQL Mapper: RDBMS 쿼리문의 실행 결과를 자바 코드에 매핑하는 프레임워크

* JDBC API를 사용
* Persistence framework
* 프로그램 코드와 sql을 분리

Apache iBATIS: SQL 데이터베이스와 객체 간 매핑을 지원해주는 persistence framework

* 지원 언어: Java, .NET, Ruby
* SQL 문을 별도의 XML 문서로 작성하여 프로그램 코드와 분리한 형식
* DAO 패턴이 발전하던 시기
* Data Access Object 패턴: 애플리케이션 비즈니스 레이어와 영속성 레이어를 추상화된 API를 이용하여 분리
* DB 접근 구현 클래스를 ~~~Dao라고 네이밍 하는 관례가 많았던 시기

MyBatis

* iBATIS 3.0에서 출발한 persistence framework
* 스프링, 스프링 부트와 연동을 지원
* 스프링: org.mybatis:mybatis-spring
* 스프링 부트: org.mybatis.spring.boot:mybatis-spring-boot-sterter
* 다양한 프레임워크와 연동을 지원
* Freemarker, Velocity, Hazelcast, Memcached, Redis, Ignite, Guice
* ORM vs MyBatis
* ORM: 자바 객체를 DB 테이블과 매핑
* MyBatis: 자바 메소드를 SQL 실행 결과와 매핑

JDBC API (Spring JDBC) : 스프링에서 제공하는 JDBC 기반 persistence framework

* Spring-boot-starter-jdbc (spring-boot-starter-data-jdbc랑 다름)
* JdbcTemplate: Spring JDBC에서 제공하는 템플릿 클래스. 쿼리 실행과 결과 전달 기능을 제공

SQL Mapper: 요약

* 프로그램 코드에서 아직 SQL을 완전히 분리하지 못함
* 개발자가 여전히 SQL을 알아야 한다
* 프로그램이 SQL을 알아야 한다 -> 전체 코드가 특정 DB 기술과 결합을 가짐
* XML 관리: SQL 을 분리하는 목적으로 만들었지만 XML을 알아야 한다.
* Type-safety를 온전히 활용하지 못한다: 쿼리 실행 결과는 대체로 Map. ResultSet 구조로 넘어옴
* 결국 매핑은 구현해 줘야 함
* Map 구조가 데이터 클래스와 비교해서 갖는 단점

1. 어떤 “필드”(맵에서는 key)가 있음을 보장하지 않는다
2. 각 데이터의 타입을 보장하지 않는다

* 결론: 객체 지향적이지 않음

Hibernate vs Spring data JPA

Hibernate:

* 자바 생태계를 대표하는 ORM framework
* 스프링 부트에서 채택한 메인 ORM framework
* JPA 표준 스팩을 구현한 JPA Provider
* 고성능, 확장성, 안정성을 표방
* 다양한 하위 제품들로 나뉨
* Hibernate ORM
* Hibernate Validator: 검증에 필요한 도구들을 지원
* Hibernate Reactive

Hibernate Query Language: hibernate가 사용하는 sql 스타일 비표준 쿼리 언어

* 객체 모델에 초점을 맞춰 설계됨
* JPQL의 바탕이 됨
* JPQL은 완벽한 QHL 문장이지만, 반대로는 성립하지 않음

EX): Query query = entityManager.createQuery(

“select p from Person p where p.name like: name”

);

TypedQuery<Person> typedQuery = entityManager.createQuery(

“select p from Person p where p.name like:name”, Person.class

);

Spring Data JPA: 스프링에서 제공하는 JPA 추상화 모듈

* JPA 구현체의 사용을 한 번 더, Repository라는 개념으로 추상화
* JPA 구현체의 사용을 감추고, 다양한 지원과 설정 방법을 제공
* Jpa 기본 구현체로 Hibernate 사용
* Querydsl 지원

Spring Data JPA 를 사용한다면: JPA, hibernate를 몰라도 된어야 한다.

* EntityManager를 직접 사용하지 않는다
* JPQL을 직접 사용하지 않는다.
* Persist(), merge(), close(0를 직접 사용하지 않는다
* 트랜잭션을 getTransaction(), commit(), rollback()으로 관리하지 않는다.
* 코드가 hibernate를 직접 사용하고 있다면
* 꼭 필요한 코드인지, 아니면 Spring data JPA로 할 수 있는 일인지 확인
* 코드는 하이버네이트와 직접적인 연관 관게를 가지게 됨
* 추상화의 이점을 포기하게 되는 셈

**@Repository**

스프링 스테레오타입 어노테이션

Persistence layer를 구현하는 클래스에 사용

* @Component 와 마찬가지로 해당 클래스를 빈으로 등록
* DAO 패턴을 적용한 클래스에도 사용 가능
* Persistence layer 에서 발생하는 예외를 잡아서 DataAccessException으로 처리해줌
* persistenceExceptionTranslationPostProcessor
* Spring Data JPA 를 사용한다면, “직접 사용할 일은 없다”고 봐도 무방함

Spring Data JPA 인터페이스: 단계별로 필요한 기능까지만 사용 가능

* Repository: 기본 repository 인터페이스. 어떤 메소드도 제공하지 않음
* CrudRepository: Repository + CRUD 기능 제공
* PagingAndSortingRepository: CrudRepository + 페이징, 정렬 기능 제공
* JpaRepository: PagingAndSortingRepository + Spring Data JPA repository 전체 기능

인터페이스에 작성한 메소드 이름이 곧 쿼리 표현이 됨

Ex: List<Event> findByEventStatusAndEventNameOrCapacity(String eventStatus, String eventName, Integer capacity);

* 다이나믹 쿼리를 만들 수는 없음
* 사용 가능한 키워드:
* Distinct, and, or, is, not, between, lessThan, lessThanEqual, greaterThen, greaterThenEqual
* Null, isNotNull, like, startingWith, endingWith, containing, orderBy, in, true, false, ignoreCase
* Join 등 복잡한 표현은 불가
* Annotations:
* @Param – 쿼리 메소드 입력 파라미터에 사용하여 annotation 기반 파라미터 바인딩할 때 사용
* @Query – 직접 JPQL을 작성하고 싶을 때 사용
* NoRepositoryBean: 빈으로 등록하고 싶지 않은 인터페이스를 지정할 수 있음
* 특정 쿼리 메소드를 기본 메소드로 지정하는 방식으로 운영 가능
* 특정 메소드를 선택적으로 사용하거나 api에 노출하고자 할 때도 사용하는 테크닉

**@Entity**

Entity class annotation: 데이터베이스에 저장(persist)할 자바 객체를 정의

* 다양한 annotation을 이용해 보다 자세한 테이블 스키마 정보를 표현
* Annotation으로 표현한 스키마 정보와 실제 테이블 스키마가 완벽히 일치해야 할 필요는 없음
* 하나의 도메인(domain)으로 간주

@Entity 클래스 안에서 사용되는 주요 JPA annotation

* @Table, @Index, UniqueConstraint: 테이블 기본 정보와 인덱스, unique 키를 설정
* @Id, @GeneratedValue:primary key 설정
* @Column: 각 컬럼 설정
* 컬럼 설정할 때 쓰는데, column annotation은 생략이 가능.
* JPA의 엔티티 클래스 안에 있는 멤버 필드들은 기본적으로 엔티티에 등록될 때 해당 엔티티의 테이블 컬럼으로 인식 되기때문
* 세부적인 디테일을 넣을 때, 안에있는 attribute를 옵션을 넣을 때 (not null) 사용한다.
* @Enumerated: enum을 처리하는 방법을 설정
* @Transient: 특정 필드를 DB 영속 대상에서 제외
* @OneToOne, @OneToMany, @ManyToOne, @ManyToMany: 연관 관계 설정
* @MappedSuperClass: 상속을 이용한 공통 필드 정의
* @Embedded, @Embeddable: 클래스 멤버를 이용한 공통 필드 정의
* @DateTimeFormat: 스프링에서 제공하는 annotation, 날짜 입력의 포맷을 지정

@Entity: JPA 엔티티의 lifecycle event 를 활용한 auditing 테크닉

JPA 엔티티에 생성일시, 수정일시 같이 일정하게 작성하는 메타데이터를 처리 가능

* @PrePersist, @PostPersist, @PreRemove, @PostRemove, @PreUpdate, @PostUpdate, @PostLoad

Entity의 생성일시, 수정일시, 생성자, 수정자를 자동으로 관리해주는 annotation

* 설정
* @EnableJpaAuditing
* @EntityListeners(AuditingEntityListener.class)
* 활용
* CreatedBy
* CreatedDate
* LastModifiedBy
* LastModifiedDate

Data Source: 물리적인 데이터소스(데이터베이스) 정보를 담는 인터페이스

* 하나의 물리 데이터베이스를 표현
* 다양한 구현체를 사용
* EmbeddedDatabaseBuilder: HSQL, Derby, H2 등 임베디드 DB 세팅할 때 사용
* DataSourceBuilder: JDBC DataSource 빌더
* DriverManagerDataSource: JDBC 드라이버로 세팅하는 DataSource
* SimpleDriverDataSource: DriverManagerDataSource 를 간편하게 만든 버전
* HikariDataSource: HikariCP를 connection pool로 사용하는 DataSource

TransactionManager: 스프링 트랜잭션 관리 기능을 담당하는 인터페이스

* 용도에 따라 다양한 인터페이스와 구현체들
* PlatformTransactionManager, ReactiveTransactionManager
* JpaTransactionManager: Spring Data JPA 일반적인 상황에 사용하는 구현체, 단일 EntityManagerFactory를 사용
* DataSourceTransactionManager: 단일 JDBC DataSource 를 사용하는 구현체
* HibernateTransactionManager: 하이버네이트 SessionFactory를 사용하는 구현체
* ChainedTransactionManager: 여러 개의 트랜잭션 매니저를 묶어서 사용하는 구현체 (@Deprecated : Boot 2.5 지워짐)

JPA DB 수동 설정: 자바 코드로 DataSource, TransactionManager를 수동 세팅해야 하는 경우가 있다.

* Configuration properties 로 커버되지 않는 세밀한 옵션을 줄 때
* 다중 DataSource
* 세팅해야 하는 요소
* DataSource
* EntityManagerFactory -> LocalContainerEntityManagerFactoryBean
* 추가적인 예외 처리 기능 때문에, 인터페이스 말고 구현체를 직접 빈으로 등록해야 한다
* PlatformTransactionManager
* 세팅 구성: DataSource (DB 설정) -> EntityManagerFactory (JPA Entity 관리) -> PlatformTransactionManager (트랜잭션 관리)

@Transactional: 스프링이 annotation 기반 transaction 관리 기능을 제공

* EntityManager 불러오고 -> 구역 지정하고 -> commit(), rollback() 직접 할 필요가 없다.
* 서비스 클래스, 메소드에 적용하는 것으로 간단히 트랜잭션 구역을 설정
* 동시에 설정하면 메소드가 우선순위
* JpaRepository 는 메소드 단위 @Transactional 이 이미 붙어있음
* 관련 annotation
* @Commit/ @Rollback
* [Javax.transaction.@Transactional](mailto:Javax.transaction.@Transactional): 스프링 패키지가 아님, 기대하는 기능을 주지 않으므로 주의

@Transactional 이 제공하는 다양한 옵션들

* transactionManager(value): 사용할 트랜잭션 매니저를 이름으로 특정
* label: 트랜잭션 구분 짓고 식별하는 레이블
* propagation: 트랜잭션이 중첩될 경우 동작(트랜잭션 효과의 전파) 규칙(default:REQUIRED)
* isolation: 트랜잭션 내부 데이터의 격리 레벨 (default: DEFAULT)
* timeout, timeoutString: 시간제한을 거는 것이 가능
* readOnly: “이 트랜잭션 안에서는 select만 일어난다”를 표현
* 강제성이 없음을 주의 – 힌트로 생각
* 이 옵션을 처리하지 않는 트랜잭션 매니저 구현체를 사용할 경우, 별도의 예외처리를 안함
* rollbackFor, rollbackForClassNmae
* noRollbackFor, noRollbackForClassNmae

@Transactional: Propagation: 중첩된 트랜잭션의 동작 규칙

* REQUIRED(default): 현재 있으면 보조, 없으면 새로 만들기
* SUPPORTS: 현재 있으면 보조, 없으면 트랜잭션 없이 실행
* MANDATORY: 있으면 보조, 없으면 예외 처리
* REQUIRES\_NEW: 트랜잭션 생성하고 실행, 현재 있던 것은 미룸
* NOT\_SUPPORTED: 트랜잭션 없이 실행, 현재 있던 것은 미룸
* NEVER: 트랜잭션 없이 실행, 현재 트랜잭션이 있었으면 예외처리
* NESTED: 현재 있으면 그안에서 중첩된 트랜잭션 형성

**Querydsl**

Querydsl: 자바 코드(entity) -> DB 쿼리 생성 도구

* HQL 생성 라이브러리
* Type-safety 가 부족한 HQL(JPQL)의 대안
* 읽기 어려운 Criteria API의 대안
* Querydsl 코드
* 보다 readable 한 쿼리 작성
* 편리한 join
* 스프링 pageable과 매끄러운 연동

JPQL 작성 라이브러리

* Spring Data JPA 와 조합하여 보다 복잡한 쿼리를 type-safe 하게 작성 가능
* 커스텀 key join
* 자유로운 query projection
* Spring Data JPA Repository interface와 매끄럽게 연동
* Spring Data에서 다양한 서포트 지원
* QuerydslRepositorySupport: EntityManager를 노출하지않고, Querydsl 필요 기능 직접 지원
* QuerydslPredicateExecutor: Predicate 을 이용한 dynamic select, Spring Data REST 지원
* QuerydslBinderCustomizer: 파라미터 바인딩의 세부 기능 조절 지원

Fetch: 애플리케이션이 DB로부터 데이터를 가져오는 것

DB와 통신하여 데이터를 읽는 것에는 큰 비용이 소모되기 때문에, 효율성 있게 가져오는 전략이 필요

* Eager: 프로그램 코드가 쿼리를 날리는 시점에 데이터를 즉시 가져오기
* Ex: select a.id from A a inner join B b on a.b\_id = b.id (b를 보지 않았지만 일단 다 가져옴)
* Lazy: 가져오려는 데이터를 애플리케이션에서 실제로 접근할 때 가져오기
* Ex: select a.id from A; (select b from B b where b.id = ?)
* Lazy: 전략은 기본적으로
* ORM 의 특징이자 기능적 장점
* 더 빠르고 경제적인 쿼리
* 잘못 사용하면 데이터 접근 에러

Fetch 기본 전략 (default setting): 각 JPA 연관관계 annotation은 기본 fetch 전략을 가지고 있다. 기본 세팅의 핵심은 “어느 쪽이 효율적인가”

* @OneToOne: FetchType.EAGER
* @ManyToOne: FetchType.EAGER
* @OneToMany: FetchType.LAZY
* @ManyToMany: FetchType.LAZY

Fetch 전략의 설정 (실전): 효율성 – 데이터가 어느 쪽으로 더 자주 사용될 것 같은가 예측

* Default 내버려두기: 필요한 시점에 최선의 방식으로 데이터를 가져옴
* LAZY 사용: 연관 관계가 있는 인티티에서 자식 엔티티만 가져오는 시나리오일 때
* 프로그래머가 로직 흐름에서 join을 의식하고 있지 않음
* Lazy 세팅이 후속 쿼리 발생 방지를 보장하지는 않음
* Ex: 불러들인 자식 엔티티가 서비스 레이어 어딘가에서 결국 부모 엔티티 필드를 건드렸을 경우
* EAGER 사용: 연관 관게가 있는 엔티티에서 무조건 다 가져오는 시나리오일 때
* 프로그래머가 join 을 사용해야 하는 상황임을 인지하고 있음
* EAGER 세팅이 join 동작을 보장하지는 않음
* Ex: Spring Data JPA 쿼리 메소드 findAll()
* JPQL 을 직접 작성해서 JOIN을 영속성 컨텐스트에 알려줘야 함

N + 1 query problem: 한 번 쿼리를 날렸을 뿐인데, 1 + N개의 쿼리가 더 생겼다.

N + 1 query problem 해결: 3가지 방법

* 똑똑한 lazy
* 비즈니스 로직을 면밀히 분석하여, 불필요한 연관 관계 테이블 정보를 불러오는 부분을 제거
* 가장 효율적인 방법
* Eager fetch + join jpql
* Join 쿼리를 직접 작성하는 방법은 다양(@Query, querydsl, …)
* 후속 쿼리를 in으로 묶어주기: N + 1 -> 1 + 1로 i/o 줄일 수 있음
* 하이버네이트 프로퍼티: default\_batch\_fetch\_size
* 스프링 부트에서 쓰는 법: spring.jpa.properties.hibernate.default\_batch\_fetch\_size
* 100 ~ 1000 사이를 추천
* 모든 쿼리에 적용되고, 복잡한 도메인에서 join 쿼리를 구성하는 것이 골치 아플 때 효율적

**순환 참조 문제**

StackOverFlowError:

각 모듈이 서로를 의존하고 있는 상태, 스프링을 사용하다보면, JPA뿐만 아니라 다양한 위치에서 간혹 경험하게 됨

* 스프링 컨포넌트끼리 참조하는 경우
* JPA 에서 가장 흔하게 발생하는 순환 참조: toString() (Lombok)

해결 방법:

* 한 쪽의 참조를 제거하는 것으로 간단히 해결
* Lombok @ToString: 한 쪽 사용을 제거

Get-in-line project: Spring security todo list

* 디펜던시 추가
* 기본 보안 설정: WebSecurityConfigurerAdapter 구현
* 패스워드 암호화: PasswordEncoder 빈 등록
* AdminService를 AuthenticationManagerBuilder 인증 설정에 등록
* Admin 데이터에 패스워드 정보 변경 (우선은 noop으로)
* @WebMvcTest 설정이 시큐리티 설정에 영향 받지 않게 하기