**자바 웹 개발 워크북(5~)**

**스프링 부트**

엔터프라이즈급 애플리케이션 개발을 위해 스프링 프레임워크를 지원하기 위한 개발 도구를 함께 제공하는 프로젝트이다. 서브 프로젝트에서 시작해 메인 프로젝트로 거듭난 개발 도구이다. 엄밀히 말하면 다르지만 스프링과 스프링 부트는 거의 동의어로 처리되기도 한다. 스프링 부트의 중요한 특징은 Auto Configuration이다. 관련 모듈만 추가하면 자동으로 관련 설정을 찾아 실행하는 간편화 기능이 다양하게 제공된다. 부수적이고 잡다한 모듈 설정이 필요하지 않고 설정 자체도 비교적 단순하다. 또한 내장 톰캣과 단독 실행이 가능한 도구이다. 별도의 서버 설정 없이도 개발/실행이 가능하며 이를 이용해 스프링 부트 프로젝트를 실행 가능한 jar파일로 만들고 다른 운영체제에서 실행하는 등의 작업이 가능하다.

**기존 개발에서 달라지는 점들**

톰캣이 내장된 채로 프로젝트가 생성되기 때문에 WAS의 추가 설정이 필요하지 않다. 또 빈 설정을 XML 대신 자바로 설정해 더 간단해진다. 스프링 MVC에서처럼 JSP도 사용 가능하지만 일반적으로는 템플린 엔진인 Thymeleaf를 사용한다. 화면을 구성하지 않고 데이터만을 제공하는 API 서버라는 형태를 사용하기도 한다. 스프링 웹 MVC의 기능과 설정을 사용할 수 있지만 간편하게 만들었다고 생각하면 된다.

**스프링 부트 프로젝트 생성 방식**

이니셜라이저를 사용해 자동 생성 하거나 Gradle같은 빌드 툴을 이용해 직접 생성할 수 있다. 보편적으로는 1번이 선호된다. 이니셜라이저가 프로젝트의 기본 템플릿 구조를 만들어주고 호환성 면에서 유리하기 때문이다.

스프링 부트는 main()을 실행해 프로젝트를 실행한다. 메인 에플리케이션을 실행하면 내장된 톰캣을 통해 단독으로 실행이 가능하다. 라이브러리만으로 설정을 인식하는 특성을 자동 설정(Auto Configuration)이라고 하는데 스프링 부트는 자동 설정이 적용된 프레임워크이다. 추가적인 설정을 위해서는 프로젝트 생성시 만들어진 application.properties 파일을 이용하거나 application.yml(YAML)을 이용하면 된다. 파일 설정을 피하고 싶으면 @Configuration이 있는 클래스 파일을 만들어 필요한 설정을 추가할 수도 있다.

초기에 application.properties에서 DB 드라이버 설정을 잡고 main()을 통해 실행하면 8080포트로 톰캣이 실행된다. 스프링 부트는 별도로 HikariCP 라이브러리를 가져오거나 HikariConfig 객체를 구성하지 않아도 기본적으로 이를 이용한다. 8080 포트가 점유 중이라면 application.properties에서 server.port를 지정해 변경할 수 있다.

롬복 라이브러리는 초기 설정에 추가하면 자동으로 추가되지만 테스트 환경에서는 빠져있다. Build.gradle에서 test관련 의존성을 추가해주면 된다.

Log4j2도 기본적으로 추가되어 있다. 프로퍼티에서 간단하게 로그 설정만 해두면 된다.

**스프링 부트에서 웹 개발**

Web.xml이나 servlet-context.xml과 같은 웹 설정 파일이 없기 때문에 이를 대신하는 클래스를 작성해야 한다는 점이 전과 다르다.

**JSON 데이터 만들기**

API서버는 JSP나 Thymeleaf처럼 화면과 관련된 내용을 만들어내지 않고 순수한 데이터만 전송하는 방식의 서버를 말한다. 대표적으로 모바일에서는 화면을 웹이 아닌 앱이 구성하기에 API 서버를 많이 사용한다. 과거에는 데이터 전송에 XML을 많이 사용했지만 최근에는 JSON을 사용하는 방식이 일반적이다.

JSON은 JavaScript Object Notation의 약자이다. 구조를 가진 데이터(객체)를 자바스크립트의 객체 표기법 기법으로 표현한 순수 문자열이다. 문자열이기 때문에 프로그래밍 언어에 독립적이라는 장점이 있다. 스프링에서 Jackson-databind라는 라이브러리를 통해 개발할 수 있지만 스프링 부트는 web을 추가할 때 자동으로 포함되어 별도 설정이 필요없다.

스프링 부트는 웹 개발에 있어 JSON과 Thymeleaf 형태를 많이 이용한다. 따라서 두 경우 모두 다 잘 익혀놔야 한다.

**Thymeleaf**

JSP와 비슷하게 서버에서 동작하지만 ‘템플릿 엔진’이기 때문에 정해진 템플릿에 맞춰서 데이터를 보여주는 방식이다. JSP는 그와 달리 직접 데이터를 생성한다. Thymeleaf는 HTML에 자신의 템플릿을 적용해 끼워맞춘다.

**타임리프의 특별한 기능들**

타임리프는 JSTL과 비슷한 것 같지만 인라인 처리와 레이아웃에 강점을 보인다. 특히 상황에 따라 같은 데이터를 다르게 출력해주는 인라인 기능은 자바스크립트를 사용할 때 편리하다.

레이아웃은 별도의 라이브러리가 필요해 의존성 설정을 추가해야 한다. 레이아웃을 활용하면 레이아웃을 만들어 놓고 필요할 때 특정한 부분만 추가해 활용할 수 있다.

**Spring Date JPA**

ORM은 Object Relational Mapping의 약어로 DB를 객체지향으로 구성한 시스템에 매핑하는 기법을 의미한다. JPA(Java Persistence API)는 ORM 기법을 도입해 적은 양의 객체지향 코드로 많은 일을 할 수 있는 라이브러리이다.

JPA의 핵심 아이디어는 프로그램 코드 따로 SQL 따로 생각하지 않고 객체지향 개념을 데이터(엔티티 객체)에 적용해 이를 바로 DB와 연동해 관리할 수 있게끔 하는 것이다. 이때 엔티티 객체는 기본키를 가지는 자바 객체가 된다. 엔티티 객체는 고유 식별을 위해 @Id를 이용해 객체를 구분하고 관리하게 된다. Spring Data JPA는 엔티티 객체를 이용해 JPA를 이용하는 편리한 방법들을 제공하는 스프링 관련 라이브러리이다. 자동으로 객체를 생성하고 이를 통해 예외 처리 또한 자동으로 처리하는데 이를 위해 제공하는 인터페이스가 JpaRepository이다.

**JpaRepository**

마이바티스를 쓸 때 매퍼 인터페이스만을 선언하고 매핑xml을 사용하던 것처럼 JpaRepository 인터페이스를 상속하는 인터페이스를 선언하는 것만으로 CRUD와 페이징 처리를 모두 완료할 수 있다. save(), findById()

일반적으로 엔티티 객체는 최소한의 변경이나 변경이 없는 불변 객체로 설계하는 게 좋다. 강제적인 건 아니다.

**Pageable 타입**

페이징 기능을 쓰고 싶으면 페이징 로직을 구현하지 않아도 Pageable 타입을 파라미터로 설정해 쓸 수 있다. Pageable은 pageRequest.of() 메서드를 지원하는데 이 메서드를 이용해 페이지 설정이 가능하다.

**쿼리메소드와 @Query**

SQL에서 사용하는 키워드와 칼럼을 결합해 메소드 이름을 구성하면 그 자체로 JPA가 사용하는 쿼리가 된다. 메소드 이름은 일반적으로 findBy… or get…으로 시작하고 칼럼명과 키워드를 결합해 사용한다. 하지만 요구사항과 조건이 많아질수록 메소드의 길이가 너무 길어져서 가독성을 저해시킬 수 있다. 단순한 쿼리를 작성할 때는 유용하지만 실제 개발에서는 많이 사용되지 않는다.

쿼리 메소드와 유사한 @Query를 사용하면 JPQL을 이용할 수 있기 때문이다. JPQL은 @Query의 어노테이션의 value로 작성하는 문자열을 말한다. 여기서 JPQL은 SQL과 유사한 JPA의 쿼리 언어이다. SQL은 객체지향에 최적화된 언어가 아니므로 이를 추상화 해 JPA에 맞는 객체지향 쿼리로 만든 게 JPQL이다. JPQL은 테이블 대신 엔티티 타입을 사용하고 칼럼 대신 엔티티 속성을 이용한다. SQL과 상당히 유사하지만 **조인**, **원하는 속성만 추출**해 **Object[] 혹은 DTO로 처리**하는 기능, **nativeQuery 속성값을 true로 지정**해 **특정 DB에서 동작하는 SQL을 사용하는 기능**들을 **수행할 수** **없다.**

**Querydsl을 이용한 동적 쿼리 처리**

JPQL은 어노테이션에 붙박이로 작성해서 정적이다. 검색 기능이 필요할 경우 다양한 경우의 수를 모두 미리 JPQL로 작성해둬야 이에 대비할 수 있다. 이 문제를 해결하기 위해 사용되는 방식이 Querydsl이다.

Querydsl은 JPA의 구현체인 Hibernate 프레임워크가 사용하는 HQL(Hibernate Query Language)을 동적으로 생성할 수 있는 프레임워크이다. 자바 코드를 사용해 작성하기 때문에 타입 안정성을 유지하면서 쿼리를 작성할 수 있다. Querydsl을 이용하기 위해 Q도메인이라는 존재가 필요한데 Q도메인은 Querydsl의 설정을 통해 기존의 엔티티 클래스를 Querydsl에서 사용하기 위해 별도의 코드로 생성하는 클래스이다.

Querydsl 설정을 마치면 Q도메인이 소스 파일로 생성된다. 구현을 원한다면 Search인터페이스와 Search 인터페이스와 QuerydslRepositorySupport를 상속하는 Impl클래스를 만들어 구현하면 된다. super(instance.class) 명령을 통해 QuerydslRepositrySupport의 생성자 매개변수로 Querydsl을 이용할 엔티티 클래스를 넣어주면 된다. 그리고 인터페이스의 메소드를 구현해준다.

Querydsl은 ‘타입’기반 ‘코드’이용 ‘쿼리’생성을 목표로 한다. 이때 코드를 만드는 클래스가 Q도메인 클래스이다.

페이징을 원하면 Pageable을 매개변수로 받아 구현한 뒤 Page<T>타입을 반환하게 하면 되는데 Querydsl에서는 이를 직접 처리해야 하는 불편함이 있다. Spring Data JPA가 이를 위해 지원하는 PageImpl 클래스를 이용하면 간편하다. PageImpl은 3개의 파라미터로 Page<T>를 생성할 수 있다.

* List<T>: 실제 목록 데이터가 들어감
* Pageable: 페이지 관련 정보를 가진 객체
* long: 전체 개수

Querydsl은 JDBC나 마이바티스에 비해 객체지향 코드로 쿼리를 간편하게 작성할 수 있고 동적으로 활용할 수 있다는 강점을 가진다.

**JPARepository**

JPA에서 리포지토리는 JDBC에서 DAO와 같은 역할을 한다. 앞서 보았지만 단순히 상속만 해도 JPA의 기본적인 CRUD를 가져다 쓸 수 있다. 여기에 추가적인 커스텀 쿼리를 만들고 싶다면 쿼리를 표현할 인터페이스와 Impl 클래스를 만들어 JPARepository에 더해 인터페이스를 상속시키면 된다.

JDBC를 이용할 때 서비스 클래스에서 로직을 구성하고 해당 로직에서는 DAO를 이용해 DB에 접근했던 것처럼 Spring Data JPA에서는 Mapper와 Repositoy를 통해 DB에 접근할 수 있다. DB에 직접 접근해 가져와야 하므로 JDBC를 사용할 때처럼 DTO를 VO인 Entity로 매핑해주는 과정이 필요하다.

**@Transactional**

트랜잭셔널 어노테이션을 사용하면 스프링이 해당 객체를 감싸는 별도의 클래스를 생성한다. 여러 번의 데이터 연결이 있을 수 있으므로 트랜잭션 처리는 기본으로 적용해두는 것이 좋다.

**REST 방식의 서비스**

모바일에서 앱으로 데이터를 받기 시작하면서 HTML을 구성하던 기존의 서버 사이드 프로그래밍은 순수한 데이터만을 제공하고 나머지는 클라이언트 쪽에서 처리하는 방식으로 변화하기 시작했다. REST 방식은 특정 URI와 HTTP 메소드를 결합해 특정한 자원에 특정한 작업을 지정하는 방식이다. REST 방식을 이해하고 Swagger UI를 이용해 테스트하고 문서화하는 방법을 학습하자.

**Ajax와 REST 방식의 이해**

Ajax(Asynchronous JavaScript And XML) 방식은 브라우저에서 서버를 호출하지만 모든 작업이 브라우저 내부에서 이루어진다. 이와 같은 방식의 장점은 현재 브라우저의 화면 변화가 없어도 서버와 통신할 수 있다는 점이다. 예를 들어 자동 완성이나 지도 서비스 등이 이에 속한다. Ajax가 적용되면서 자바스크립트를 중심으로 데이터를 다루는 기술의 수요가 늘어났고 이는 프론트 엔드 중심의 개발 방식으로 이어졌다.

Ajax는 초기에 자바스크립트를 이용해 XML을 주고받는 방식을 이용했지만 최근에는JSON(JavaScript Object Notation)이 더 선호된다.

**+추가**

전체 서버 애플리케이션을 완전한 REST API로 작성하는 건 상당히 힘들다고 한다. 지켜야할 제약 사항이나 규약 등도 많아 첫 프로젝트부터 너무 무리하게 REST API로 작성하려 하는 건 욕심일 수 있다. 현업에서는 다양한 기술이 쓰이고 있으니 프로젝트 진행 시에는 하나만 고집하지 말고 JSP, Thymeleaf, RESTful API 등 다양한 방식을 적용하자. +POST, GET 외 메서드 보안 이슈

**클라이언트 중심의 개발**

모바일 환경이 등장하고 리액트, 뷰, 앵귤러와 같은 대규모 웹 애플리케이션 지원 라이브러리나 프레임워크가 발전하면서 클라이언트 사이드와 서버 사이드의 역할 분배가 진행됐다. 핵심 아이디어는 순수한 데이터만을 취급한다면 클라이언트의 환경에 상대적으로 덜 제약받으면서 서버 사이드의 작업을 해낼 수 있다는 것이다. 클라이언트 사이드의 구현은 전문화된 영역으로 넘긴다.

**JSON 문자열**

핵심은 문자열이다. 문자열은 특정 프로그래밍 언어나 기술에 종속되지 않는다. 다만 복잡한 데이터를 표현하기에 어려운데 이를 위해 고려되는 것이 XML이나 JSON 포맷이다.

JSON은 자바 스크립트에서 객체 리터럴을 표현할 때 쓰는 {키:밸류} 형식을 적용한 문자열이다. 원래 자바에서는 클래스와 인스턴스를 만들어 데이터를 표현해야 했는데(DTO, VO) JSON을 사용하면 자바의 언어적 특성에 종속되지 않는(어느 클라이언트에서나 인식 가능한) 문자열로 데이터를 처리할 수 있다. 스프링 부트에는 jackson-databind라는 JSON 라이브러리가 포함되어 있어 별도 설정 없이 바로 JSON 데이터를 만들 수 있다.

**REST 방식**

이전 웹 개발 방식에서 URL은 원하는 ‘행위나 작업’을 의미하고 GET/POST 등은 데이터를 전송하는 위치를 의미했다. 그러나 Ajax를 이용함에 따라 URL 이동 없이 서버와 데이터를 교환할 수 있게 되었으므로 URL은 ‘행위나 작업’에서 ‘원하는 대상’ 그 자체를 의미하게 됐고 해당 URL에서 사용되는 HTTP 메서드가 ‘행위나 작업’을 의미하게 됐다.

링크는 자원 자체가 되고 HTTP 메서드는 해당 자원을 통해 어떤 이룰 목적이 되었다. 그리고 이렇게 바뀐 패러다임의 방식을 REST(Representational State Transfer)방식이라고 한다.

REST는 **효율적, 안정적**이며 **확장가능**한 **분산 시스템**을 구현할 수 있는 소프트웨어 아키텍처 디자인 모음을 이야기 한다. 이 디자인 제약을 준수했을 때 해당 시스템은 RESTful하다고 일컬어진다.

REST 방식은 하나의 자원을 하나의 주소로 표현이 가능하고 해당 자원이 유일무이해야 한다. 한URL이 한 자원을 식별할 수 있게 하는 고유값이고 GET/POST 등은 이에 대한 ‘작업’을 수행할 수 있어야 한다.

**REST 방식의 URL 설계**

표준이나 원칙이 존재하지는 않지만 관례적인 방식은 있다.

**GET -> /boards/123**

**POST -> /boards/** -> 신규 게시물 데이터

**PUT -> /boards/123** -> 수정이 필요한 데이터

**DELETE -> /boards/123**

**Swagger UI**

REST는 화면을 구성하지 않기에 POST 기능을 처리하는 API를 테스트하기 힘들다. 따라서 화면 역할을 대신해주는 Postman이나 Swagger UI 등을 사용해야 한다. Swagger UI는 어노테이션 설정으로 API 문서와 테스트 할 수 있는 화면을 생성할 수 있다.

**REST 방식의 댓글 처리**

REST 방식의 댓글 처리는 다음과 같은 단계로 진행한다.

1. URL의 설계와 데이터 포맷 결정
2. 컨트롤러의 JSON/XML 처리
3. 동작 확인
4. 자바스크립트를 통한 화면 처리

REST 방식은 XML이나 JSON 형태의 문자열을 전송하고 이를 컨트롤러에서 처리하는 방식을 많이 사용한다. JSON을 이용해 DTO에 맞는 데이터를 전송하고 스프링을 통해 이를 DTO로 처리하도록 구성하는 것이다.

**다대일 연관관계 실습**

댓글과 게시물은 전형적인 다대일 관계에 있다. DB상에서 PK/FK로 처리되는 관계를 JPA로 어떻게 처리하는지 학습하고 이를 실습하는 것이 이번 장의 핵심이다.

**연결관계 결정하기**

일반적으로 DB 설계 시에는 PK를 가진 테이블을 먼저 작성하고 이후 해당 테이블의 PK를 FK로 사용하는 테이블을 작성한다. 이런 방식의 데이터베이스 설계 방식에는 우선 순위가 있고 처리 방식에도 일정한 규칙이 존재한다.

하지만 객체지향을 활용하는 JPA는 조금 다르다. 우선 방향성을 결정하는 것이 어렵다. 회원 객체가 아이템을 가지고 있을 때, 회원이 아이템을 참조할 것인지, 아이템이 회원을 참조할 것인지 결정해야 한다. JPA의 연관 관계 판단 기준은 다음과 같은 기준을 적용하는 것이 좋다.

* 연관 관계의 기준은 항상 변화가 많은 쪽을 기준으로 결정
* ERD의 FK를 기준으로 결정

**변화가 많은 쪽을 기준**

변화가 많은 쪽이 연관관계의 핵심이 된다. 변화가 많은 쪽에 많은 테이블이 조인되는 걸 생각하면 자연스럽다.

**그럼에도 애매할 때**

그럴 때는 ERD(엔티티 관계 다이어그램)을 그려 확인하는 것이 확실하다.

**단방향과 양방향**

관계형 데이터베이스와 JPA의 큰 차이점 중 하나는 객체가 다른 객체를 참조하는 방식에 있다. 관계형 DB에서는 조인을 통해 다른 테이블을 참조하지만 객체지향에서는 (접근 제한자가 허용한다면) 어느 객체에서나 다른 객체의 속성을 참조할 수 있다. 이렇게 양방향으로 객체 참조를 구현하면 객체가 서로 참조를 유지하기 때문에 변경사항이 생길 경우 모든 관리를 양쪽에 적용해야 한다는 불편함이 생긴다. 하지만 JPA에서 필요한 데이터를 탐색하는 작업은 편리해진다.

반면에 단방향으로 객체 참조를 구현하면 에러를 많이 줄일 수 있다는 장점이 있다. 하지만 관계형 데이터베이스의 조인 처리와 같이 다른 엔티티 객체의 내용을 사용하기 어려워진다.

실습에서는 단방향 참조 관계를 유지하면서 다른 엔티티를 사용할 때 JPQL을 통해 조인 처리하는 방식으로 진행한다.

**연관관계 설정 시 주의할 점**

* @ToString을 사용할 때 참조하는 객체를 사용하지 않도록 exclude 속성을 지정한다.
* @ManyToOne과 같이 연관 관계를 나타낼 때 fetch 속성은 반드시 LAZY로 지정한다.

@ToString의 exclude 속성을 지정하지 않으면 참조하는 객체의 쿼리를 실행할 때 참조되는 객체에도 쿼리를 실행하려고 한다. Exclude 처리를 하지 않은 엔티티를 사용하는 메소드에 @Transactional이 붙어있지 않다면 no session 에러가 발생한다.

Fetch속성 LAZY는 지연 로딩을 의미한다. 지연 로딩은 자바 코드만을 실행해 객체의 값을 조정한 뒤 모든 작업을 완료했다고 파악하는 시점에 이를 일괄적으로 DB에 반영한다. 지연 로딩의 반대는 EAGER로 즉시 로딩이다. 즉시 로딩은 해당 엔티티를 로딩할 때 DB 커넥션을 동시에 로딩해 반영한다. 기본값을 LAZY로 사용하되 필요 시에만 EAGER를 고려하는 게 좋다.

**\*주의(VO와 Entity의 차이)**

여태 VO와 Entity를 혼용해서 생각했는데 VO는 식별자가 없고 Entity는 식별자가 존재한다. 따라서 엔티티는 식별자를 제외한 값이 바뀌어도 이전의 객체와 동일한 객체라고 할 수 있지만 VO는 처음부터 불변 객체로 생성되어 값이 변했으면 객체 자체가 다른 객체인 것이다.

구체적으로 둘을 구분해서 개발해야 하는 상황은 아직 없기에 VO는 일종의 기준이라고 생각하고 엔티티는 기준위에서 가변값을 허용하는 프로그래밍 단위라고 생각하자. 막 도로공사가 완공되어 10년 내에는 변할 일이 없는 도로 구간 위에서 교통 상황을 반영해 최적의 길 찾기를 해주는 서비스를 운영한다고 가정했을 때 트래픽을 고려하지 않은 도로 자체의 길이나 너비 등은 VO로 설정하고 도로 위 차량의 수나 기상상황 등을 반영한 객체는 엔티티로 설정하는 방식이라면 둘을 구분해서 사용할 것이다.

**단방향 구현의 장점**

단방향 구현을 할 때 참조 당하는 객체는 자신을 참조하는 객체에 반영되는 변경과 무관하다. 양방향일 경우 변경사항이 발생할 때마다 양방향 객체에 모두 쿼리를 날려야 하고 이를 트랜잭션 처리까지 해줘야 하기 때문에 번잡하다.

**단방향 구현의 단점**

이전에 단방향 구현은 하나의 객체로 양방향의 객체에 접근할 수 없으며 조인을 사용할 수 없다는 단점이 있다고 했다. 이 문제를 해결하기 위해 JPQL을 이용한 left outer join이나 inner join을 고려할 수 있다.

**\*DTO(레이어 사이를 오가는 객체), Entity(DB와 직접 매핑되는 객체)**

텍스트, 스크린샷, 폰트, 디자인이(가) 표시된 사진

자동 생성된 설명

**사용자 귀책인 에러를 코드 500에서 400으로 바꾸기**

사용자가 데이터베이스에 없는 자료를 조회했을 때 메시지가 서버에 전달된 뒤 서버가sqlException 에러를 내며 500 코드를 클라이언트에 전송할 수 있다. 하지만 본질적인 문제는 서버 쪽 로직에서 잘못된 sql을 작성해 발생한 게 아니라 사용자가 DB에 존재하지 않는 데이터를 조회한 것이다. 이를 바로 알리려면 advice 패키지 등에 구현한 CustomAdvice 클래스에서 @ExceptionHandler 어노테이션을 이용해 DataIntegrityViolationException 등이 발생했을 때 이를 핸들링하는 메소드를 만들어 400 메시지를 반환하게 해야 한다. 메소드 설정을 통해 ResponseEntity에 다른 메시지를 담아 반환하게 한다. RESTful API를 만들 때 이렇게 에러 메시지를 컨트롤하려면 에러 사항을 확인하고 이를 advice 클래스에서 핸들링해줘야 한다.

**댓글의 자바스크립트 처리**

**비동기 처리와 Axios**

동기와 비동기의 차이를 명확히 알고 가자. 동기는 연쇄적인 로직의 결과를 말한다. 로직2을 수행하기 위해 로직1이 끝나야 하고 로직3을 수행하기 위해서는 로직 2가 끝나야 한다. 이에 반해 비동기는 작업을 분산해 동시에 진행할 수 있다. 비동기는 작업이 완료된 후 이를 대기자에게 통보하는 방식을 사용하는데 이를 전문 용어로 “콜백”이라고 한다.

**Promise**

대부분의 개발자들은 절차적인 동기식 프로그래밍에 익숙하다. 비동기 방식에서 함수를 일급 객체로 사용해 콜백을 이용하는 것은 조금만 단계가 많아져도 상당히 복잡해진다.

자바스크립트는 Promise라는 개념을 도입해 비동기 호출을 동기화된 방식으로 작성할 수 있게 해준다. Axios는 이를 활용하는 라이브러리이다. Axios를 사용하면 Ajax를 호출하는 코드를 작성할 때 동기화된 방식처럼 작성할 수 있어 자바스크립트 기반의 프레임워크나 라이브러리에서 많이 사용된다.

async 는 해당 함수가 비동기 함수라는 것을 나타대고 await는 async 함수 내에서 비동기 호출 부분에 사용한다.

해당 함수를 실행해 로그를 찍어보면 반환값을 반환하기 전에 Promise라는 객체를 먼저 반환한다. 이는 비동기 함수가 비동기이므로 호출한 시점에 확정적으로 예정된 반환값을 돌려주는 것이 아니기 때문에 미리 무엇인가를 반환하기로 한 ‘약속’을 반환하는 것이다.

then을 사용하면 프로미스가 끝난 뒤 원하던 값을 반환할 때 반환값을 받아볼 수 있다. catch를 사용하면 reject 발생 시 에러를 캐치해 특정 로직을 수행한다.

**비동기 처리 방식 결정**

비동기 함수는 일반 함수와 동작 방식이 다르기 때문에 데이터 처리 방식을 지정할 수 있다.

1. 첫번째 방법은 비동기 함수에서 순수하게 비동기 통신을 처리하게 하고 호출한 쪽에서 then(), catch()를 이용해 처리하는 방식
2. 두번째 방법은 비동기 함수를 호출할 때 나중에 처리할 내용들을 같이 별도의 함수로 구성해 파라미터로 전달하는 방식이다.

**파일 업로드 기본 설정**

서블릿 3부터 서블릿 API 자체에 파일 업로드를 처리할 수 있는 API가 제공되어 추가적인 라이브러리가 필요하지 않다.

**업로드 설정시 유의할 점**

이미지 파일이 아닌 경우 섬네일 생성을 하면 안 된다. 이미지 파일은 섬네일을 생성하듯이 파일 유형마다 별도로 처리해줘야 할 작업이 있다면 처리하고 삭제시에도 이를 반영해 추가 생성된 파일들을 같이 삭제해줘야 한다.

**@OneToMany**

게시물과 댓글의 관계와 게시물과 첨부파일의 관계는 테이블 구조상 같지만 JPA에서는 무엇을 중심으로 해석하는지에 따라 다른 결과가 나올 수 있다.

앞에서 게시물과 댓글은 JPA에서 가장 많이 사용하는 @ManyToOne을 따랐다. 이번에는 게시물의 관점에서 첨부 파일을 바라보는 @OneToMany 방식을 사용할 것이다.

@OneToMany는 하나의 상위 엔티티와 여러 개의 하위 엔티티의 구조로 이루어진다. @ManyToOne과의 차이점은 매니투원은 다른 엔티티 객체의 참조로 FK를 가지는 쪽에서 사용하는 방식이고 원투매니는 PK를 가진 쪽에서 사용하는 방식이라는 점이다.

@OneToMany구조 특징

* 상위 엔티티가 하위 엔티티를 관리한다.
* JPA Repository를 상위 엔티티 기준으로 생성한다. 하위 엔티티의 변경은 상위 엔티티에도 반영되어야 한다.
* 상위 엔티티가 변경되면 하위 엔티티에도 동일한 처리를 해야한다.
* 상위 엔티티 하나와 하위 엔티티 여러 개를 처리할 때 ‘N + 1’ 문제가 발생할 수 있어 주의해야 한다.

@OneToMany는 각 엔티티에 해당하는 테이블을 독립적으로 생성하고 중간에 매핑해주는 테이블을 생성한다. board와 board\_image가 있으면 중간에 board\_image\_set이라는 테이블이 @OneToMany를 처리하기 위해 생성된다.

**mappedBy를 이용한 구조 변경**

위의 set테이블을 매핑테이블이라고 한다. 테이블을 생성하지 않으려면 단방향으로 @OneToMany를 이용하거나 @JoinColumn을 이용하거나 mappedBy 속성을 이용할 수 있다. mappedBy의 경우 Board와 BoardImage가 서로 참조를 유지하는 양방향 참조 상황에서 사용하는데 mappedBy는 ‘어떤 엔티티의 속성으로 매핑되는지’를 의미한다.

mappedBy는 연관관계의 주인으로 해석되기도 한다.

**영속성의 전이(cascade)**

이전에 상위 엔티티와 하위 엔티티의 연관 관계를 상위 엔티티에서 관리하는 경우 상위 엔티티 객체 상태의 변경이 하위 엔티티에도 반영해야 한다고 했다. 이를 JPA에서는 영속성의 전이(cascade)라고 표현한다. 위에서 Board와 BoardImage처럼 양방향 연관관계가 설정되어 있는 경우 서로 영속성 전이가 필요하다. JPA에서는 이런 경우 cascade 속성을 부여해 이를 제어하도록 한다.

mappedBy를 통해 하위 엔티티를 관리하는 상위 엔티티는 자체적으로 하위 엔티티를 관리하는 메소드를 가진다. 이 메소드를 통해 상위 엔티티의 모든 변화가 하위 엔티티에도 영향을 미치도록 관리하는 것이다.

**Lazy 로딩과 @EntityGraph**

@OneToMany의 로딩은 기본적으로 지연 로딩이다. 게시물을 조회할 때 Board 객체와 BoardImage 객체를 생성해야 하므로 2번의 select가 필요하다.

**N+1 문제와 @BatchSize**

상위 엔티티에서 @OneToMany 연관관계를 유지하는 경우 트랜잭션 단위로 상, 하위 엔티티의 작업을 동시에 처리할 수 있다는 장점이 있지만 목록을 처리할 때 예상하지 못한 문제를 만들어 낼 수 있다.

N+1문제는 하나의 질의를 수행하기 위해 훨씬 더 많은 하위 질의를 수행하게 되는 문제를 말한다. @OneToMany에서 ManyToOne으로 엮인 경우 테이블 단위인 관계형 DB라면 left join을 수행한 후 한 번의 쿼리로 질의를 끝내겠지만 JPQL은 엔티티 단위이기 때문에 하나의 엔티티를 기준으로 질의를 날리게 된다. 이 문제를 해결하는 다양한 방법들이 있지만 여기서는 @BatchSize를 통해 해결하는 법을 소개한다.

**@BatchSize**

N+1로 실행되는 쿼리는 데이터 베이스에 너무 많이 접근해 효율이 떨어진다. @BatchSize는 이에 대한 가장 간단한 보완책이 될 수 있다. @BatchSize에는 size 속성을 지정해 N번에 해당하는 쿼리를 모아서 한 번에 실행할 수 있다. 여기서 size로 설정된 속성값은 해당 칼럼을 조회할 때 한 번의 in 조건으로 사용된다. 달리 말하면 여러 개의 프록시 객체를 조회할 때 where절이 같은 여러 개의 select쿼리를 하나로 묶어 in 쿼리로 만드는 것이다.

**컨트롤러와 화면 처리**

**서버에는 이미지 업로드(O), DB에는 업데이트(X)**