**개인프로젝트(12/3~)**

목표 – 12월 내로 간단한 RESTful API 서버 만들기(필수 기능 – 회원 가입, 회원 관리, 상품 진열, 장바구니, 구매, 고객 게시판), 스프링 부트 익숙해지기, iOS 기초 학습, 웹 프론트 기본 구현

**12/3/2023**

커피 구독 서비스 레퍼런스 찾기

https://cafebox.kr/

https://ymcoffeeproject.com/

<https://www.koke.kr/>

<https://hikoco.co.kr/>

https://office.beanbrothers.co.kr/

https://coffeecg.com/index.html

멤버 엔티티 구상

* 필수이용약관, 선택약관을 두 개의 맵으로 선언해 가지고 있는다.(작성이 간단하지만 약관 변경에 대처 힘듦)
* 약관과 동의, 고객의 정보를 다른 엔티티로 분리해 관리한다.
* 약관과 동의가 간소한 경우 이를 하나의 엔티티로 묶어 고객의 동의 내역과 연계해 관리한다.

**내일 할 일**

프로젝트용 깃허브 리포지토리 만들기(v), 작업 진전도에 따라 나눠서 커밋하기(v)

보편적인 ERD 참조해 ERD 작성하기(v)

**12/4/2023**

**ERD 작성**

DB 설계를 해본 경험이 없기 때문에 혼자 고민하는 것보다 사람들이 일반적으로 많이 사용하는 방식을 차용하고 디테일한 부분에 개인적인 생각을 붙이는 방식으로 작성했다.

**회원**

회원 정보의 핵심이 되는 회원 엔티티(PK: mno)를 두고 회원 개인 정보, 회원 동의 내역, 필수 이용 약관 엔티티, 선택 약관 엔티티를 구분했다. 처음에는 맵을 통해 이용 약관을 회원 엔티티 안에 두는 방식을 생각했지만 그렇게 하면 전체 이용 약관 텍스트와 별개의, 임의로 정한 기호로 회원의 동의를 저장하는 것 같아 법적인 문제가 생길 것 같다는 생각이 들었다. 일단은 간단한 DB 설계가 최우선이기 때문에 필수 약관과 세부 약관을 각각의 엔티티로 두어 모든 약관 텍스트를 반영하고, 추후 변경 발생 시 이를 유연하게 수정할 수 있도록 회원 엔티티의 PK -> 동의 내역 -> 구체 약관으로 조인하는 구조를 생각했다.

**게시글과 댓글**

고객 문의 게시판으로 만들 게시글과 댓글 게시판은 이전에 공부했던 “자바 웹 개발 워크북”의 내용을 참조한다.

**장바구니와 상품**

장바구니는 “자바 웹을 다루는 기술”의 영상에 있는 일반적인 방법을 따르되 필요한 부분이 생기면 디테일한 조정을 거치려고 한다. 특히 상품 정보 엔티티는 우선 하나의 엔티티로 만들지만 상품 목록이 많아지거나 비즈니스 적인 요구사항이 발생할 시 세분화가 가능하도록 확장과 분화를 고려해서 만들 생각이다.

**내일 할 일**

도메인과 엔티티, DTO 패키지 어노테이션부터 관계 유형까지 제대로 공부하기, ERD로 작성한 개념 모델 도메인 엔티티와로 구현, 관련 엔티티 DTO 클래스, 리포지토리 작성하기, 스프링 시큐리티 적용해 구현 시작하기

**12/5/2023**

**도메인 관련 어노테이션**

**@Entity**

접근 제어자가 public이나 protected인 기본 생성자가 필수이다. final, enum, interface, inner 클래스인 경우 사용할 수 없다. 엔티티로 등록된 클래스는 JPA가 관리한다. JPA는 엔티티로 등록된 클래스의 이름을 DB의 테이블 명으로 설정해 클래스 속성과 DB 속성을 매핑한다. @Entity(name = “newName”)을 통해 이름을 지정할 수 있지만 일반적으로는 name 속성을 사용하지 않고 기본 값으로 사용한다. JPA는 내부적으로 JPQL이라는 객체지향 쿼리 언어(SQL추상화)를 사용한다.

**@Table(name = “newName”)**

위의 엔티티 네임 속성처럼 테이블에 특정한 이름을 부여할 때 사용한다.

**@Id**

테이블에 기본키로 지정할 속성에 사용한다. @GeneratedValue(strategy = GenerationType.Identity)와 같이 사용하면 점증하는 값을 자동으로 설정할 수 있다.

**@GeneratedValue(strategy = )**

기본키 생성 전략과 관련된 어노테이션이다. 세 가지 전략이 있다.

1. Identity – identity 작업에서 유일하게 persist와 동시에 insert 쿼리를 실행한다. 따라서 persist 직후에 바로 엔티티의 pk값을 알 수 없다. pk 생성이 데이터베이스에 위임되어 데이터베이스에서 생성되기 때문이다.
2. Sequence – allocation 사이즈를 정하고 할당량이 다 차면 시퀀스 번호를 DB에서 한 번에 조달한다.
3. Table – 데이터베이스에 키생성만 하는 테이블을 하나 만들어 키를 생성한다. 성능상 문제 때문에 잘 사용하지 않는다.

**@Column**

@Column(name = "email", nullable = false, length = 200)

private String email;

와 같이 사용한다. 테이블 컬럼의 특별한 이름이나 설정 등을 더할 수 있다. 컬럼 어노테이션을 사용하지 않으면 변수명이 컬럼명으로 지정된다.

**@ManyToOne, @OneToMany, @OneToOne, @ManyToMany**

엔티티 관계 매핑 어노테이션

**@JoinColumn(name = ”department\_id”, nullable = false)**

엔티티 관계를 조인하기 위한 칼럼을 명시한다.

**@Enumerated**

enum타입과 데이터베이스 컬럼을 매핑한다. ORDINAL로 매핑하면 이넘의 순서를 매핑하고 STRING으로 매핑하면 이넘의 값을 매핑한다.

**@Transient**

엔티티에 등록한 필드를 DB에 반영하지 않으려면 이 어노테이션을 사용한다. 지금은 활용할 일이 없으니 필요시에 자세하게 공부하자.

**@NoArgsConstructor**

**@AllArgsConstructor**

**@RequiredArgsConstructor**

주석의 이름과 기능이 같다.

그에 더해 @NoArgsConstructor은 JPA와 Hibernate에서 lazy loading을 위한 프록시 객체 형성을 위해 선언해야 한다. 일반적으로 No와 All을 같이 선언한다. 이렇게 하면 원하는 상황에 따라 객체의 초기화를 선택할 수 있다.

**\*추가사항 (직렬화와 반직렬화)**

직렬화는 객체 데이터를 직렬화하여 전송 가능한 스트림 형태로 만드는 것이다. 반직렬화는 반대로 스트림 형태의 객체 데이터를 DB에 저장 가능한 형태로 만듦을 뜻한다.

@NoArgsConstructor는 위와 같은 반직렬화의 경우에 객체의 데이터를 담을 빈 생성자를 생성해주기 때문에 엔티티에 선언해주는 것이 좋다. @NoArgsConstructor의 설정은 많은 프레임워크나 라이브러리에서 권장하는 사항이기도 하다. @AllArgsConsturtor는 빌더 패턴을 만들 때 사용한다. @RequiredArgsConstructor은 엔티티 내에 반드시 초기화되는 속성이 있을 경우 @NoArgsConstructor 대신 쓰일 수 있다.

**@Builder**

엔티티 클래스에 많이 쓰는 롬복 어노테이션이다. 빌더 패턴을 통해 생성자를 대신하는 데는 크게 몇 가지 이유가 있다.

1. **가독성**

* 입력 칸에 어떤 필드가 대응되는지 한 눈에 알 수 있다. 일부 IDE는 생성자 파라미터에도 이 기능을 제공하지만 빌더 패턴을 이를 보다 뚜렷하게 보여준다.

1. **불변성**

* 빌더 객체는 한 번 설정하면 값을 바꿀 수 없는 불변객체를 생성한다.

1. **옵셔널 파라미터**

* 생성자에서 옵셔널을 허용하는 여러 개의 파라미터를 만드는 대신 빌더 클래스가 제공하는 옵셔널 메소드를 통해 옵셔널 필드를 설정할 수 있다.

**@ToString**

Object 클래스에 있는 toString을 오버라이딩 해서 객체의 정보를 문자열에 담아 반환하는 메서드를 생성한다.

**@ElementCollection**

해당 컬렉션이 엔티티의 컬렉션이 아닌 밸류 타입의 컬렉션임을 알리는 어노테이션. 컬렉션이 다루는 값이 그 자체로 엔티티가 아닌 원시값이거나 임베더블 타입이라는 것을 알린다.

@ElementCollection

@CollectionTable(

name = “role”

joinColumns = @JoinColumn(name = “member\_id”)

)

private Set<RoleName> roleSet = new HashSet<>();

와 같이 선언해 사용한다. 데이터 베이스에서 값으로 컬렉션을 저장할 수 없기 때문에 컬렉션은 별도의 테이블을 만들어 조인 후 사용한다. 위 코드에서는 member\_id를 외래키로 하는 role 테이블이 엔티티에서 별도로 만들어지는데 role 테이블은. 멤버 테이블과 일대다 관계로 묶이게 된다. ElementCollection은 엔티티의 생명 주기와 일치해야 하기 때문에 엔티티 테이블과 cascade = ALL, orphanRemoval = true가 설정된 것과 같다. 엔티티 테이블을 삭제하게 되면 elementCollection도 같이 삭제가 된다.

값 타입 컬렉션 테이블은 데이터를 수정하는 경우 데이터를 아예 삭제했다가 수정된 데이터를 삽입한다. 만약 하나의 엔티티와 여러 개의 값 타입 컬렉션이 연결된 경우, 하나의 엔티티에 연결된 어떤 element collection에 변경이 일어나는지 추적이 불가능하기 때문이다. 따라서 값 타입 컬렉션 테이블은 정말 값 타입인 경우에만 쓰고 컬렉션이 단순하지 않은 경우, 예를 들면 식별자가 필요하고 지속해서 값을 추적, 변경해야 하는 경우라면 새 엔티티를 만들어 조인해서 사용해야 한다.

**@Embedable, @Embedded**

위에서 @ElementCollection은 말 그대로 컬렉션 값 타입을 저장하는 테이블이었다. @Embedable 어노테이션은 어노테이션이 붙은 필드 객체를 값 타입의 테이블로 만든다. 이 테이블은 @Embedded가 붙은 속성에 매핑되어 값 타입으로 활용된다. 위와 같이 별도의 생명 주기를 가지지 않고 엔티티에 종속되어 있다.

**내일 할 일**

ERD로 작성한 개념 모델 도메인 엔티티와로 구현, 관련 엔티티 DTO 클래스, 리포지토리 작성하기, 스프링 시큐리티 적용해 구현 시작하기