

8주차 미션기록

미션 기록

미션 추가하기

```
{
  "missionName": "이식당에서 식사하기",
  "missionDetail": "15,000원 이상 결제하기",
  "reward": 500,
  "storeId": 1
}
```

Execute

Responses

Curl

```
curl -X 'POST' \
  'http://localhost:8080/mission/' \
  -H 'accept: */*' \
  -H 'Content-Type: application/json' \
  -d '{
    "missionName": "이식당에서 식사하기",
    "missionDetail": "15,000원 이상 결제하기",
    "reward": 500,
    "storeId": 1
  }'
```

Request URL

http://localhost:8080/mission/

Server response

Code

Details

200

Response body

```
{
  "isSuccess": true,
  "code": "COMMON200",
  "message": "성공입니다.",
  "result": {
    "missionId": 1,
    "createdAt": "2025-05-22T03:09:37.4709179"
  }
}
```

리뷰 추가하기

```
{
  "title": "jmt",
  "score": 5,
  "description": "음 맛있다~",
  "reply": "개췌",
  "storeId": 1,
  "missionId": 1
}
```

Execute

Responses

Curl

```
curl -X 'POST' \
  'http://localhost:8080/reviews/' \
  -H 'accept: */*' \
  -H 'Content-Type: application/json' \
  -d '{
    "title": "jmt",
    "score": 5,
    "description": "음 맛있다~",
    "reply": "개췌",
    "storeId": 1,
    "missionId": 1
  },'
```

Request URL

`http://localhost:8080/reviews/`

Server response

Code	Details
------	---------

200	Response body
-----	---------------

```
{
  "isSuccess": true,
  "code": "COMMON200",
  "message": "성공입니다.",
  "result": {
    "reviewId": 2,
    "createdAt": "2025-05-22T03:13:15.5825041"
  }
}
```

기존에 있는 가게 미션에 상태 추가하는 API

Request body required

```
{
  "statuses": 1,
  "missionId": 1
}
```

Execute

Responses

Curl

```
curl -X 'POST' \
  'http://localhost:8080/usermissionpointcounter/' \
  -H 'accept: */*' \
  -H 'Content-Type: application/json' \
  -d '{
    "statuses": 1,
    "missionId": 1
  }'
```

Request URL

```
http://localhost:8080/usermissionpointcounter/
```

Server response

Code	Details
200	<div>Response body</div> <pre>{ "isSuccess": true, "code": "COMMON200", "message": "성공입니다.", "result": { "phoneNumberMissionid": 1, "createdAt": "2025-05-22T17:54:34.6170211" } }</pre>

에러상태표 새로 만들어야함 preferenceshandler가 여러군데 설정됨 이거 바꿔야한다

어노테이션을 사용한 예외처리

```
curl -X 'POST' \
  'http://localhost:8080/members/' \
  -H 'accept: */*' \
  -H 'Content-Type: application/json' \
  -d '{
    "name": "",
    "nickname": null,
    "email": null,
    "gender": null,
    "preferCategory": [],
    "birth": "",
    "address": "",
    "phoneNumber": 0
  }
.'
```

Request URL

```
http://localhost:8080/members/
```

Server response

Code	Details
------	---------

400	Error: response status is 400
-----	-------------------------------

Undocumented

Response body

```
{
  "isSuccess": false,
  "code": "COMMON400",
  "message": "잘못된 요청입니다.",
  "result": {
    "email": "날이어서는 안됩니다",
    "address": "크기가 2에서 20 사이여야 합니다",
    "birth": "날이어서는 안됩니다",
    "nickname": "날이어서는 안됩니다",
    "name": "공백일 수 없습니다"
  }
}
```

조건4 어노테이션을 활용하여 이미 해당 미션이 이미 진행중인 미션인지 검증하는 어노테이션

Curl

```
curl -X 'POST' \
  'http://localhost:8080/usermissionpointcounter/' \
  -H 'accept: */*' \
  -H 'Content-Type: application/json' \
  -d '{
    "statuses": 1,
    "missionId": 1
  }
.'
```

Request URL

http://localhost:8080/usermissionpointcounter/

Server response

Code

Details

400

Undocumented

Error: response status is 400

Response body

```
{
  "isSuccess": false,
  "code": "COMMON400",
  "message": "잘못된 요청입니다.",
  "result": {
    "statuses": "ALREADY_CHALLENGED"
  }
}
```

Response headers

조건2 어노테이션을 활용하여 리뷰 작성하는 가게가 존재하는지 검증하는 어노테이션

Curl

```
curl -X 'POST' \
  'http://localhost:8080/reviews/' \
  -H 'accept: */*' \
  -H 'Content-Type: application/json' \
  -d '{
    "title": "맛있는 카페 후기",
    "score": 5,
    "description": "커피가 정말 맛있었어요!",
    "reply": "감사합니다!",
    "storeId": 123456,
    "missionId": 1
  }
'
```

Request URL

```
http://localhost:8080/reviews/
```

Server response

Code	Details
400 <i>Undocumented</i>	Error: response status is 400
	Response body <pre>{ "isSuccess": false, "code": "COMMON400", "message": "잘못된 요청입니다.", "result": { "storeId": "STORE_NOT_FOUND" } }</pre>

미션을 진행하면서 발생한 오류들을 해결하는 과정과 부족한 부분들 공부한 내용

- domain 패키지 : DB에 entity에 해당하는 클래스
 - JPA에서 사용하기 위한 엔티티 클래스를 저장하기 위한 패키지
- controller 패키지 : http 요청에 대한 응답을 줌 post던 get이던
 - http 요청이 오면, 그에 대한 응답을 주는 클래스의 모임
 - 응답을 주기위한 과정들을 service에서 처리하도록 한다
- service 패키지 : 비즈니스 로직 구형
 - 비즈니스 로직이 필요한 클래스들의 모임 -> 가장 복잡한 코드가 들어간다
 - controller에서 service의 메소드를 호출하게 된다
 - service는 repository의 메소드를 호출하게 됩니다
 - repository -> service -> controller 순서로 진행된다

- repository 패키지
 - database와 통신을 하는 계층으로 Spring Data Jpa를 이용해서 만든 repository를 이용할 것입니다.
- dto 패키지
 - 클라이언트가 body에 담아서 보내는 데이터를 받기 위한 클래스
 - Database에서 받아온 데이터를 클라이언트에게 보여주기 위한 클래스
 - DB에서 받아온 데이터(엔티티)를 그대로 응답으로 주게되면 문제가 생기게 된다.
 - 요구사항의 변경이 생겨 DB의 설계가 바뀌게 되어 티의 변화가 생겼을 시, 엔티티를 그대로 응답으로 줄 경우 DB의 변경사항이 프론트엔드 개발자에게까지 영향을 주게된다.
 - dto를 통해 응답 데이터를 결정하게 되면, DB의 변경이 생길 경우 dto만 변경하면 되기에 더 좋은 설계가 된다.
- converter 패키지
 - converter는 데이터 형식 간의 변환을 수행하는 역할이다
 - entity to dto를 해야한다
 - repository에서 받아온 엔티티를 dto로 바꾸는 과정을 converter에서 하게된다 추가로 entity의 생성 역시 converter에서 수행하기도 한다
 - converter에서 entity의 생성을 하지 않고 service에서 하도록 하는 경우도 있습니다.
 - converter에서 엔티티의 생성을 하게 되면, service는 순수하게 비즈니스 로직에만 집중할 수 있어 단일 책임 원칙 측면에서 더 좋다
 - converter에서 entity를 생성하도록 하자
- API
 - Controller, DTO 클래스는 API 계층에 해당된다
- 영속화
 - Review 객체를 JpaRepository.save(review) 같은 방식으로 영속화해야 합니다.
 - `reviewId` 는 **자동으로 DB가 생성**합니다.
 - 개발자가 ID를 지정할 필요 없이 JPA가 **DB에 insert를 요청**하면서 자동으로 생성됩니다.
 - 컨버터 클래스에서 구현시 id는 지정하지 않았기에 **reviewId는 null**입니다. -> 메모리 상에서만 존재한다
 - reviewRepository.save(review); 를통해서 db에
 - 왜 `save()` 가 반드시 필요할까?
 - Review 객체를 만들고 `reviewId` 를 수동으로 넣지 않았다면 -> **JPA는 DB에 저장하면서 ID를 받아오는 과정**을 통해서만 ID를 알 수 있습니다.
 - 이걸 하지 않으면 `reviewId` 는 null인 상태로 남고, -> `ReviewConverter.toJoinResultDTO(review)` 등에서 NPE가 날 수 있어요.
- 스트림 함수
 - `Collection` 인터페이스(List, Set 등)의 기본 메서드
 - 컬렉션 데이터를 스트림 형태로 변환해서 처리할 수 있게 해줍니다

- 사용이유 : 기존의 반복문(for/foreach)을 사용하지 않고 **함수형 스타일**로 데이터를 처리할 수 있기 때문입니다.

```
java

List<String> list = Arrays.asList("apple", "banana", "cherry");

// 기존 방식
for (String s : list) {
    if (s.startsWith("b")) {
        System.out.println(s.toUpperCase());
    }
}

// Stream 방식
list.stream()
    .filter(s -> s.startsWith("b"))
    .map(String::toUpperCase)
    .forEach(System.out::println);
```

- 스트림은 데이터를 저장하지 않고, 파이프라인처럼 처리(파이프라인 : 데이터나 작업 흐름이 여러 단계로 나뉘어 순차적으로 처리되는 구조)
- 연산은 **중간 연산**(예: `filter`, `map`)과 **최종 연산**(예: `forEach`, `collect`)으로 구분된다
- 스트림은 한 번만 사용할 수 있습니다 (재사용 불가).

```
java

List<Integer> nums = Arrays.asList(1, 2, 3, 4, 5);



int sum = nums.stream()
    .filter(n -> n % 2 == 1) // 홀수 필터링
    .mapToInt(n -> n * 2)    // 2배로 만들기
    .sum();                  // 합계 구하기

System.out.println(sum); // 결과: 18 (1*2 + 3*2 + 5*2)
```

- `filter` : 말그대로 인자들을 선택(필터, 걸러냄)

- map : 걸러진 인자들을 가지고 로직 구현
- 결과
 - collect(Collectors.toList()) 는 Java Stream API에서 스트림(Stream)을 리스트(List)로 변환하는 역할을 합니다.
 - collect(...)
 - => - Stream의 최종 연산 중 하나로, 스트림 요소들을 원하는 자료구조로 수집(collect) 합니다.
 - Collectors.toList()
 - 요소들을 List 형태로 수집하는 수집기(Collector) 를 리턴합니다.
 - forEach(...)
 - 자바의 Stream 또는 List 에 존재하는 최종 연산
- Controller클래스에 사용된 어노테이션의 설명
 - @PostMapping("/")
 - 반환값을 JSON 형태로 응답 (HTML이 아님)
 - 주로 REST API 만들 때 사용
 - @Controller + @ResponseBody 조합
 - @RequiredArgsConstructor
 - final 이 붙은 필드(memberCommandService)를 자동으로 생성자 주입
 - 의존성 주입을 간편하게 만들어 줌
 - @RequestMapping("/members")
 - 이 컨트롤러의 모든 경로는 /members 로 시작을 의미한다
- remove(this) 나 add(this) 는 리스트나 컬렉션에서 사용하는 메서드

1. StoreInfo 입장에서 리뷰는 여러 개니까:

 설명
 

```

java
@OneToMany(mappedBy = "storeInfo")
private List<Review> reviewList;
  
```

이때 **getter**는 아래처럼 만들어야죠:

```

java
public List<Review> getReviewList() {
    return reviewList;
}
  
```

즉, storeInfo.getReviewList() 로 여러 개의 리뷰를 다룹니다.

•

2. 반면, Review 는 하나의 StoreInfo 만 가지니까:

java

복사

설명 편집

```
@ManyToOne
@JoinColumn(name = "store_info_id")
private StoreInfo storeInfo;
```

이건 `getStoreInfo()` 가 맞아요. 왜냐면 1:1 연결이기 때문이에요.

- isValid

- @Constraint : java에서 제공하는 사용자가 validation을 커스텀 어노테이션을 통해 할 수 있도록 제공하는 어노테이션
- @Constraint의 파라미터로 validatedBy가 있고 CategoriesExistValidator.class를 지정
- CategoriesExistValidator라는 클래스를 통해 @ExistCategories가 붙은 대상을 검증
- 클래스 CategoriesExistValidator 내부
 - 인터페이스 ConstraintValidator 인터페이스에 대한 구체화 클래스로서 만들어야 합니다
 - ExistCategories 어노테이션에 대한 로직을 담을 것이며 검증 대상이 List<Long>임을 명시
 - public class CategoriesExistValidator implements ConstraintValidator<ExistCategories, List<Long>>
 - isValid 메소드만 우리가 원하는 형태로 바꾸면 된다
 - ConstraintValidator : isValid 메소드의 반환 값을 확인하여 검증이 되었는지 실패했는 지를 알려준다

```
@Override
public void initialize(ExistCategories constraintAnnotation) {
    ConstraintValidator.super.initialize(constraintAnnotation);
}
```

- DTO상

- @ExistCategories 어노테이션이 Request Body로 받아 올 Dto의 필드에 붙어 있다

```
@PostMapping("/{id}")
public ApiResponse<MemberResponseDTO.JoinResultDTO> join(@RequestBody @Valid MemberRequestDTO.JoinDTO request){
    Member member = memberCommandService.joinMember(request);
    return ApiResponse.onSuccess(MemberConverter.toJoinResultDTO(member));
}
```

- 컨트롤러에서 RequestBody를 받아오는 과정에서 @ExistCategories 가 붙어 있는 DTO로 인해

```

@Override
public boolean isValid(List<Long> values, ConstraintValidatorContext context) {
    boolean isValid = values.stream()
        .allMatch( Long value -> preferencesRepository.existsById(value));

    if (!isValid) {
        context.disableDefaultConstraintViolation();
        context.buildConstraintViolationWithTemplate(
            HttpStatus.FOOD_CATEGORY_NOT_FOUND.toString()).addConstraintViolation();
    }
}

```

- 위의 코드가 실행
- 이제 isValid 메소드에서 false를 리턴하면 ConstraintViolationException 를 발생

```

@Override @가장의 사용위치
public ResponseEntity<Object> handleMethodArgumentNotValid(MethodArgumentNotValidException e
    ,HttpHeaders headers, HttpStatusCode status, WebRequest request) {

    Map<String, String> errors = new LinkedHashMap<>();

    e.getBindingResult().getFieldErrors().stream()
        .forEach( FieldError fieldError -> {
            String fieldName = fieldError.getField();
            String errorMessage = Optional.ofNullable(fieldError.getDefaultMessage()).orElse("");
            errors.merge(fieldName, errorMessage, (String existingErrorMessage, String newErrorMessage) -> existingErrorMessage + ", " + newErrorMessage);
        });

    return handleExceptionInternalArgs(e,HttpHeaders.EMPTY,HttpStatus.valueOf( name: "_BAD_REQUEST"),request,errors);
}

```

- 따라서 최종적으로 advice에서 MethodArgumentNotValidException를 감지

• 미션 의문점

🔍 왜 "nickname", "email", "gender", "preferCategory", "phoneNumber" 는 오류 안 나올까?

필드명	어노테이션	전달된 값	설명
nickname	@NotNull	"" (빈 문자열)	빈 문자열은 null 아님 → @NotNull 통과
email	@NotNull	"" (빈 문자열)	동일하게 통과됨
gender	@NotNull	0	int 기본값 0 → null 아님 → 통과
preferCategory	@ExistCategories (커스텀)	[] (빈 배열)	null 아님 → 통과 커스텀 validator에서 빈 배열 검사 안 하면 예외 없음
phoneNumber	@NotNull	0	int 기본값 0 → 통과

✅ 반면 오류가 나온 항목

필드명	어노테이션	전달된 값	오류 이유
name	@NotBlank	""	빈 문자열은 @NotBlank 위반
birth	@NotNull	"" → 파싱 실패	문자열을 LocalDate 로 파싱 실패 = 실제로는 바인딩 실패로 null 로 간주됨
address	@Size(min=2, max=20)	""	길이 0 → 실패

- @NotNull 대신 @NotBlank 또는 @NotEmpty 를 써야 문자열의 빈 값도 막을 수 있어요.

- @NotNull은 어떤 상황에 작동하는가 :

✓ @NotNull 이 작동하는 JSON 상황

JSON 필드 상태	Java에서 받은 값	@NotNull 반응	설명
"field": null	null	✗ 유효성 검사 실패	JSON에서 명시적으로 null 전달한 경우
필드 아예 없음	null	✗ 유효성 검사 실패	JSON 요청에 필드 자체가 없을 경우
"field": ""	""	✓ 유효성 검사 통과	빈 문자열이지만 null은 아님
"field": 0	0	✓ 유효성 검사 통과	0은 int 타입의 유효한 값

- 검사 실패시 작동함

```
curl -X 'POST' \
  'http://localhost:8080/members/' \
  -H 'accept: */*' \
  -H 'Content-Type: application/json' \
  -d '{
    "name": "",
    "nickname": null,
    "email": null,
    "gender": null,
    "preferCategory": [],
    "birth": "",
    "address": "",
    "phoneNumber": 0
  }
'
```

Request URL

http://localhost:8080/members/

Server response

Code	Details
400 <i>Undocumented</i>	Error: response status is 400

Response body

```
{
  "isSuccess": false,
  "code": "COMMON400",
  "message": "잘못된 요청입니다.",
  "result": {
    "email": "날이어서는 안됩니다",
    "address": "크기가 2에서 20 사이여야 합니다",
    "birth": "날이어서는 안됩니다",
    "nickname": "날이어서는 안됩니다",
    "name": "공백일 수 없습니다"
  }
}
```

int와 integer의 차이

Integer (참조 타입)

- 자바의 래퍼 클래스(Wrapper Class)로, `int` 를 객체처럼 다룰 수 있도록 해줌.
- `java.lang.Integer` 클래스에 정의되어 있음.
- 메모리에 값이 아니라 객체의 참조(주소)가 저장됨.
- `null` 저장 가능 (ex. DB 처리 시 유용).
- 다양한 유틸리티 메서드 제공 (`Integer.parseInt()` , `toString()` , `compareTo()` 등).

```
int a = 5;
Integer b = a; // 오토팩싱 (int → Integer)
int c = b; // 언박싱 (Integer → int)
```

LocalDate를 사용해서 날짜 받기

```
import java.time.LocalDate;

public class Main {
    public static void main(String[] args) {
        LocalDate today = LocalDate.now(); // 현재 날짜
        LocalDate birthday = LocalDate.of(1995, 5, 15); // 특정 날짜 생성
        System.out.println("오늘 날짜: " + today);
        System.out.println("생일: " + birthday);
    }
}
```

@DateTimeFormat : **Spring Framework**에서 날짜와 시간 값을 문자열로 변환하거나 문자열을 날짜로 변환할 때 사용하는 애노테이션

@OneToMany 와 @ManyToOne 은 JPA에서 두 엔티티 간의 방향성과 관계를 설정할 때 사용하는 애노테이션

|항목| @OneToMany | @ManyToOne |

의미	한 개가 여러 개를 가짐	여러 개가 하나를 가짐
----	---------------	--------------

위치	1 쪽에서 선언	N 쪽에서 선언
----	----------	----------

외래키	보통 반대쪽(N)에 존재	해당 엔티티 테이블에 외래키 생성
-----	---------------	--------------------

연관관계 주인 여부	❌ 기본적으로 주인이 아님	✅ 연관관계의 주인 (외래키 관리)
------------	----------------	---------------------

사용하는 경우	부모 → 자식 목록 보기	자식 → 부모 참조할 때
---------	---------------	---------------

매핑 테이블을 만든다고 할때 연관관계 어노테이션

매핑 테이블(조인 테이블)을 명시적으로 만들 때는 JPA에서 일반적으로 중간 엔티티(매핑 테이블)를 따로 만들고, @ManyToOne 을 두 번 사용해서 관계를 설정

Member (1) ← MemberPreferdFood → (1) PreferdFood 인 상황일때

즉, MemberPreferdFood 테이블이

@ManyToOne Member, @ManyToOne PreferdFood 를 가짐

서로 참조(CRUD 포함)를 위해서는 양쪽 설정이 필요합니다.

- **단방향 연관관계:** 한쪽만 객체 참조가 가능
- **양방향 연관관계:** 양쪽에서 모두 객체 참조 가능 → 양방향 탐색, CRUD 편리
mappedBy 는 **양방향 연관관계에서 연관관계의 주인이 아닌 쪽**에서 사용하는 속성입니다.
즉, **외래키를 관리하지 않는 쪽**이 mappedBy 를 써서 **"나는 연관관계의 주인이 아니야"** 라고 명시
mappedBy = ~
~에 들어갈 부분은 상대 클래스에 있는 필드명

@JoinColumn(name = "...") 안의 name 은 **DB의 컬럼명**을 의미

findById의미 PA Repository에서 특정 ID를 가진 엔티티를 **데이터베이스에서 조회하는 메서드**입니다.

회원가입 응답에서는 일반적으로 다음만 포함합니다:

- 식별자 (memberId)
- 닉네임 또는 이름 (name, nickname)
- 이메일 (email)
- 생성 시간 (createdAt) ← BaseEntity 에 있을 것으로 추정
- (선택) 주소, 생년월일, 성별 등

절대 포함하지 말아야 할 것들:

- 비밀번호 (없긴 하지만 혹시라도)
- 민감 정보 (전화번호 등)
- 내부용 설정 필드 (isActive, deletedAt 등)

요청용 DTO — 클라이언트가 서버에 보내는 데이터를 담는 클래스

`getId()` 함수는 일반적으로 **객체의 고유 식별자(ID)**를 반환하는 메서드입니다. Java에서는 엔티티(Entity)나 도메인 객체에서 자주 사용되며, Lombok 의 `@Getter` 어노테이션이나 IDE에서 자동 생성되는 경우가 많습니다

1. DTO에서 PreferCategory 를 그대로 사용하는 방법

이름 클래스 DTO에서 사용방법

```
1. public class JoinDto {
    private List preferCategories;
}
```

위처럼 리스트로 처리하면 된다 해당 이넘클래스를 타입으로 하는

Java의 `stream()` 은 컬렉션(List, Set 등)의 데이터를 **선언적이고 간결하게** 처리할 수 있게 해주는 기능입니다. **Java 8 이상에서 사용** 가능합니다.

`stream()` 은 컬렉션의 요소들을 하나씩 꺼내서, **가공하거나 필터링하거나 매핑하는 처리 흐름(파이프라인)**을 만드는 데 사용됩니다.

Stream() 동작

```
List names = Arrays.asList("Alice", "Bob", "Charlie");
```

```
names.stream()
    .filter(name -> name.startsWith("A"))
    .map(String::toUpperCase)
    .forEach(System.out::println);
```

▶ 동작 순서:

`.stream()` – 리스트에서 스트림 생성
`.filter(...)` – "A"로 시작하는 이름만 남김
`.map(...)` – 대문자로 바꿈
`.forEach(...)` – 출력

메서드	설명
<code>filter()</code>	조건에 맞는 요소만 통과시킴
<code>map()</code>	요소를 변형 (예: 문자열 → 길이, <code>String</code> → <code>enum</code> 등)
<code>collect()</code>	결과를 리스트 등으로 다시 수집
<code>forEach()</code>	각 요소에 대해 반복 실행
<code>sorted()</code>	정렬
<code>distinct()</code>	중복 제거
<code>count()</code>	요소 개수 세기
<code>anyMatch()</code>	조건 만족하는 요소가 하나라도 있는지
<code>allMatch()</code>	모든 요소가 조건을 만족하는지

해당 레포지토리의 역할

```
package umc.study.repository.PreferencesRepository; import
org.springframework.data.jpa.repository.JpaRepository; import
umc.study.domain.Preferences; public interface PreferencesRepository extends
JpaRepository<Preferences, Long> { }
```

Spring Data JPA를 사용해서 데이터베이스와 연동하는 **레포지토리 인터페이스**를 정의한 것입니다. 하나씩 풀어볼게요.

- **PreferencesRepository** :
DB에서 `Preferences` 엔티티를 조회, 저장, 삭제 등 작업을 담당하는 객체를 정의한 인터페이스입니다.
- **extends JpaRepository<Preferences, Long>** :
`JpaRepository` 는 Spring Data JPA에서 제공하는 인터페이스로,
기본적인 CRUD(생성, 조회, 수정, 삭제) 메서드를 자동으로 만들어 줍니다.
여기서 제네릭 타입 파라미터는:
 - `Preferences` : 이 레포지토리가 관리할 **엔티티 클래스 타입**
 - `Long` : `Preferences` 엔티티의 **주 키(Primary Key)**의 타입

`ong` 과 `Long` 은 비슷해 보여도 Java에서 중요한 차이가 있어요.

Long (대문자 L)

- 참조 타입(reference type), 즉 클래스(wrapper class)입니다.
- `java.lang.Long` 클래스이고, 내부에 `long` 값을 감싸고 있어요.
- 객체로서 존재하며, `null` 값을 가질 수 있어요.
- 컬렉션(List, Map 등)에 저장하려면 참조 타입이어야 함.
- 메서드도 포함하고 있음 (`Long.parseLong()` , `toString()` 등).

|특징|long (primitive)|Long (wrapper)|

|---|---|---|

|값 저장 방식|실제 숫자를 직접 저장|숫자를 감싼 객체 저장|

|기본값|0|null 가능|

|null 허용 여부|불가|가능|

|사용 위치|산술 연산, 변수 선언|제네릭 타입, 컬렉션 등|

|성능|빠름|느림 (객체 생성 비용)|

양방향 연관 관계 설정은 converter 보다는 service에서 하는 것이 좋습니다!

단방향은 컨버터에서 설정을 해도 괜찮습니다.

따라서 MemberPrefer 생성 시 member를 넣지 않았습니다.

이제 서비스에서 뭘 해야 할까요?

양방향 연관관계 설정

연관 관계를 설정하고, 양방향 매핑이 된 경우 연관 관계 편의 메서드를 이용해 양방향 매핑을 해줍니다.

참고로, 제가 사용하는 연관 관계 편의 메서드는 아래와 같은 모양을 가집니다.

@Transactional을 통해 트랜잭션을 설정해줍니다.

이 어노테이션이 무엇인지, 어떤 역할을 하는지 궁금하다면 구글링을 해서 찾아보세요!

바꿔야 할 것: `MemberCommandService` 인터페이스에 메서드 정의 추가

그러면 `MemberCommandService` 인터페이스에는 반드시 다음 메서드가 정의돼 있어야 합니다:

java

복사편집

```
import umc.study.dto.member.MemberRequestDTO; import umc.study.domain.Member;
public interface MemberCommandService { Member
joinMember(MemberRequestDTO.JoinDto request); }
```

- 현재 컨트롤러는 인터페이스(`MemberCommandService`)를 참조하고 있으므로,
- 구현 클래스(`MemberCommandServiceImpl`)에서 이 메서드를 구현하려면,

- 인터페이스에 메서드 시그니처가 존재해야 합니다.

[클라이언트 요청]

↓

MemberRestController.join() ← 컨트롤러에서 서비스 호출

↓

MemberCommandService.joinMember() ← 인터페이스 메서드 선언 필요

↓

MemberCommandServiceImpl.joinMember() ← 실제 비즈니스 로직 수행

- JoinDto 는 단순한 구조인데,
- 실제 보내신 JSON은 member → preferCategory → memberPrefer → member → review → storeInfo ... 무한 구조.

→ Jackson이 순환 참조 또는 너무 복잡한 구조를 만나서 Failed to read request (400) 발생.

실전 팁: DTO는 절대로 Entity 전체 넣으면 안 됨

JoinDto 는 사용자로부터 받는 최소 정보만 포함하고 있어야 하고,

Member, Review, StoreInfo, Mission 등 다른 엔티티 객체는 절대로 포함시키면 안 됩니다.

그 구조는 Entity 간의 관계를 위한 것이지, 클라이언트와 통신용(JSON 직렬화/역직렬화)에 적합하지 않습니다.

여기서 Member → MemberPrefer → Preferences → Member 식으로 순환 참조가 발생할 수 있습니다.

그리고 newMember 객체는 memberPreferList 와 연결되어 있지 않기 때문에, JPA에서 연관관계가 완전히 설정되지 않은 상태로 저장하게 되어 문제가 발생할 수 있습니다.

DTO에서는 Preferences 나 MemberPrefer 전체 구조를 절대 담지 마세요. 필요한 ID만 받고 Entity 내에서 조회해서 사용합니다.

이 두 개는 insert와 update 시 null 인 경우는 그냥 쿼리를 보내지 않도록 해줍니다.

☀️ ****@Documented**** - ****이 어노테이션은 사용자 정의 어노테이션을 만들 때 붙입니다.

****@Target**** - ****이 어노테이션은 어노테이션의 적용 범위를 지정합니다. 각 파라미터의 자세한

역할은 구글링을 해서 찾아보세용. ****@Retention**** - ********이 어노테이션은 어노테이션의 생명 주기를 지정합니다. 위의 코드는 RUNTIME이기에 실행 하는 동안에만 유효하게 됩니다.