

## UMC\_Mission\_Week9

### API 만드는 순서

1. API 시그니처를 만든다!
  2. API 시그니처를 바탕으로 swagger에 명세를 해준다
  3. 데이터베이스와 연결하는 부분을 만든다
  4. 비즈니스 로직을 만든다
  5. 컨트롤러를 완성한다
  6. validation 처리를 한다
- API 시그니처 만들기
    1. 응답과 요청 DTO만들기
    2. 컨트롤러에서 어떤 형태로 리턴하는지, 어떤 파라미터가 필요한지, URI는 무엇인지 HTTP Method는 무엇인지만 정해준다.
    3. 컨버터 정의만 한다

큰 단위의 DTO를 하나의 클래스로 두고 하위 자잘한 DTO들은 static class로 둡니다.

API명세서는 빠르게 만들어 프론트엔드 개발자들이 확인할 수 있게 해야한다

즉 json으로 나가는 데이터들의 형태를 미리 알려줌으로 프론트와 백 동시에 작업을 진행할 수 있게된다.

Controller, Converter 정의만 해둬 return null;을 설정함으로 이름만 보이게 해준다

- 새로운 어노테이션 설명
  - @Operation은 이 API에 대한 설명을 넣게 되며 summary, description으로 설명을 적습니다.
  - @ApiResponses로 이 API의 응답을 담게 되며 내부적으로 @ApiResponse로 각각의 응답들을
  - @Parameters 는 프론트엔드에서 넘겨줘야 할 정보를 담는다  
에러 상황에 대해서만 content = 를 통해 형태를 알려줬고 (에러는 코드, 메시지만 중요하지 result는 필요 없어서!)  
성공에 대해서는 content를 지정하지 않았습니다  
Spring Data JPA에서 제공하는 Paging 관련 추상화는 여러분들이 찾아 볼 것을 추천
- @Builder
- @Getter

```

@NoArgsConstructor
@AllArgsConstructor
public static class ReviewPreViewListDTO {
    List<ReviewPreViewDTO> reviewList;
    Integer listSize;
    Integer totalPage;
    Long totalElements;
    Boolean isFirst;
    Boolean isLast;
}
@Builder
@Getter
@NoArgsConstructor
@AllArgsConstructor
public static class ReviewPreViewDTO {
    String ownerNickname;
    int score;
    String description;
    String title;
    LocalDate createdAt;
}
}

```

- ReviewPreViewDTO 해당 클래스는 리뷰를 간단하게 보여주는 클래스로 List에 담겨서 ReviewPreViewListDTO를 통해서 보여진다 페이지가 구현되어있다.
- ReviewPreViewListDTO는 여기에 한 가게의 리뷰들을 모아 넣은 리스트로 이 리스트의 상태를 보여주는 클래스이다. 페이지를 알 수 있다

오토와이어링 실패는 오류가 발생한 본 클래스의 어노테이션을 확인해야한다 빈으로 등록이 안되어 있을 가능성이 높다.

private final MemberQueryService memberQueryService; 와 같은 인터페이스를 다른 클래스에서 생성자를 만들어서 사용할때 해당 구현체 즉 여기서 StoreQueryServiceImpl 클래스에서 @Service ,@RequiredArgsConstructor ,@Transactional(readOnly = true) 어노테이션을 선언함으로 bean으로 등록해야지 다른 클래스에서 생성자를 사용할 수 있다.

**페이지 번호 -1 처리를 어디서 해야 할지는 책임 분리(Separation of Concerns) 관점에서 결정해야 합니다**  
 어노테이션으로 처리해야함

## 질문

프론트엔드 개발자를 위한 스웨거 명세서 만드는데 구현체를 어떻게 만들어야하는지  
쿼리스트링으로 조회하는 법 => RequestParam 어노테이션 부분이 쿼리스트링이고 이 쿼리 스트링은 url상에서 나타난다

- Argument Resolver-공부하기  
리퀘스트바디 어노테이션을 사용해서 리퀘스트요청을  
바인딩= 묶는다 => 값을 한번에 가져온다 이런느낌  
객체가 파라미터로 존재할때 원하는 값을 가져오는 것을 바인딩이라고 한다.

패스베리어블은 값을 하나만 가져오기에 단일조회만 사용하고  
쿼리스트링은 여러개의 데이터 조회 가능 => 리퀘스트파라미터 이 단일조회가 아니라 여러개 조회를 할때 사용하는 것

레포지토리에서 `Page<UserMissionPointcounter> findAllByMemberAndStatus(Member member, int status, Pageable pageable);`  
이렇게 만든다고 하면 해당 메소드의 파라미터를 JPA가 자동으로 파악해서 파라미터 이름이랑 같은 SQL 쿼리를 자동으로 실행해준다

**Spring Data JPA에서 "메서드 이름"에 포함된 필드명이 엔티티 클래스의 필드명과 같아야 한다.**  
즉 도메인 클래스내 필드명이랑 같아야한다

status

JPA는 다음을 기준으로 자동 매핑합니다:

기준	설명
<code>@Column(name = "status_code")</code>	명시된 경우 이 DB 컬럼명 사용
필드명이 <code>status</code>	명시 없으면 <code>status</code> 라는 컬럼으로 인식 (자동 네이밍 룰 적용)

Repository 메서드의 `findByXxxAndYyy` 에서 `Xxx` , `Yyy` 는 엔티티 클래스의 필드명과 일치해야 함

DB 컬럼명과 일치할 필요는 없음

DB 컬럼명이 다르면 `@Column(name = "db_column_name")` 으로 매핑하면 됨

현재 mission post가 작동이 안됨 마찬가지로 ump도 작동이 하지 않음

지금 id값이 null이라고 에러가 발생하는 이유는 어노테이션 alreadychallenging이 지금처럼 단일 필드( Integer )를 검증하는 대신, **전체 DTO(JoinDTO)** 를 대상으로 검증하도록 설계를 바꾸는 게 가장 안정적이고 안전합니다.

해결 방안 1: Validator 를 Object 단위로 검증

@Target({ ElementType.METHOD, ElementType.FIELD, ElementType.PARAMETER })  
이 설정은 어노테이션을 필드, 메서드, 파라미터에만 붙일 수 있게 제한합니다. 하지만:

현재 @AlreadyChallenging 어노테이션은 아래처럼 클래스 레벨에 붙어야 합니다:

```
@AlreadyChallenging // 👉 클래스 위에 붙임
public static class JoinDTO {
    ...
}
```

클래스에 붙을 수 있도록 설정

@Target({ ElementType.TYPE })

이나

@Target({ ElementType.TYPE, ElementType.METHOD, ElementType.FIELD, ElementType.PARAMETER }) 형식으로 해야한다

## 🔧 해결 방법: 리포지토리 메서드 이름 수정

`member.id` 가 아닌 `member.memberId` 를 참조하도록 메서드 이름을 고쳐야 합니다.

### ✅ 변경 전 (오류 발생)

java

📄 복사

✎ 편집

```
boolean existsByMemberIdAndMissionIdAndStatus(Long memberId, Long missionId, Statuses status);
```

### ✅ 변경 후 (정상 작동)

java

📄 복사

✎ 편집

```
boolean existsByMember_MemberIdAndMission_MissionIdAndStatus(Long memberId, Long missionId, Statuses status);
```

Spring Data JPA는 `member.memberId`, `mission.missionId`, `status` 를 자동 해석합니다.

## 📌 추가 팁

- `memberId`, `missionId` 와 같은 필드명을 엔티티에서 기본 키로 지정할 경우, 리포지토리에서 참조할 때는 `member_MemberId` 와 같이 **객체명 + 필드명** 형태로 써야 합니다.
- 만약 `memberId` → `id`, `missionId` → `id` 로 필드명을 통일했다면 `member.id` 처럼 간단하게 쓸 수 있습니다.

### ✅ 최종 수정 예시 (Repository)

java

📄 복사

✎ 편집

```
public interface UmpRepository extends JpaRepository<UserMissionPointcounter, Long>, UmpRepository {  
    boolean existsByMember_MemberIdAndMission_MissionIdAndStatus(Long memberId, Long missionId, Statuses status);  
}
```

이렇게 수정하면 더 이상 `"No property 'id' found for type 'Member'"` 오류가 발생하지 않으며, JPA가 경로를 제대로 추적할 수 있습니다.

필요하면 추가적으로 `Mission` 이나 `Statuses` 도 확인해드릴게요.

📄 🌟 🔗 ↻ ⌵

- `memberId`, `missionId` 와 같은 필드명을 엔티티에서 기본 키로 지정할 경우, 리포지토리에서 참조할 때는 `member_MemberId` 와 같이 **객체명 + 필드명** 형태로 써야 합니다.
- 만약 `memberId` → `id`, `missionId` → `id` 로 필드명을 통일했다면 `member.id` 처럼 간단하게 쓸 수 있습니다.
- 커스텀 어노테이션을 만드는 작업

- initialize(ExistCategories constraintAnnotation)
  - `ConstraintValidator<A, T>` 인터페이스에서 제공되는 초기화 메서드입니다. 여기서 `A` 는 어노테이션 타입 ( 해당 커스텀 어노테이션의 인터페이스입니다 )입니다.  
→ 제약 조건 어노테이션에서 전달된 값을 바탕으로 초기화 로직을 수행할 때 사용됩니다.
  - `ConstraintValidator.super.initialize(constraintAnnotation)`  
→ Java 8부터는 인터페이스에서도 default 메서드가 가능해졌기 때문에, `super` 를 통해 인터페이스의 기본 구현을 호출할 수 있습니다.  
→ 이 코드는 부모 인터페이스의 기본 `initialize()` 구현을 호출하는 것입니다.  
`ConstraintValidator` 인터페이스는 `initialize()` 의 **default 구현이 없습니다**. 즉, 이 호출은 의미가 없거나 컴파일 오류가 날 수 있습니다 (사용 중인 Validator 구현체에 따라 다름).
- 커스텀 어노테이션 사용법
  - DTO나 Controller에서 사용한다
    - DTO

## DTO 예시

```
java

public class PageRequestDto {

    @NonZeroPage
    private Integer page;

    // 생성자, getter/setter 등
}
```

- - `@NonZeroPage` 같은 커스텀 유효성 검사 어노테이션은 일반적으로 DTO에서 사용합니다.
  - 클라이언트에서 들어오는 데이터를 **검증하는 핵심 위치**
  - `@Valid` 또는 `@Validated` 와 함께 사용하면 자동 검증됨
  - → 검증 실패 시 예외( `MethodArgumentNotValidException` ) 발생
- Controller

## Controller 예시

java

복사

설명

편집

```
@PostMapping("/items")
public ResponseEntity<?> getItems(@Valid @RequestBody PageRequestDto dto) {
    // dto.getPage()는 1 이상이어야 유효함
    ...
}
```

- 컨트롤러에서는 보통 DTO 전체를 검증 대상으로 받고,
  - 검증 결과는 Spring이 자동 처리함
  - 컨트롤러의 파라미터 자체에 `@NonZeroPage` 를 붙이는 건 비효율적이고 잘 안 씀
- DTO나 controller 둘중 하나만 선택해서 사용하면된다.
- `@RestControllerAdvice` 는 **Spring Boot**에서 예외 처리, 전역 설정 등을 담당하는 **전역 컨트롤러 어드바이스(조언자)**입니다. `@ControllerAdvice` 에 `@ResponseBody` 가 결합된 형태로, **REST API에서 주로 사용됩니다.**
- 어노테이션으로 파라미터 값을 받아서 변환해주는 로직
  - UMC워크북 9주차 페이지징관련
  - `umc_study -> validator` 패키지 -`ZeroBasedPageResolver`클래스 확인하기
    - `supportsParameter`
      - 스프링이 컨트롤러 메서드 파라미터마다 이 리졸버가 처리할 수 있는지 물어봅니다.
      - 여기서는 파라미터에 `@ZeroBasedPage` 어노테이션이 붙어있고 타입이 `Integer` 일 때만 처리합니다.
    - `resolveArgument`
      - 실제 HTTP 요청에서 파라미터 값을 읽어서, 원하는 형태로 변환한 후 반환합니다.
      - 여기서는 `page` 파라미터를 읽고, 1 이상인지 체크하며, 내부 로직에 맞게 `page - 1` 값을 반환합니다.
  - 예외 처리
    - `page` 가 1 미만이면 `IllegalArgumentException` 을 던집니다.
    - 나중에 `@RestControllerAdvice` 에서 이 예외를 잡아 적절히 클라이언트에 에러 메시지를 응답해 줄 수 있습니다.
  - `WebConfig`에 리졸버 등록
    - -스프링 MVC 설정에 내가 만든 `ZeroBasedPageResolver` 를 등록합니다.
    - 이 등록이 없으면 스프링이 리졸버를 전혀 알지 못해, 작동하지 않습니다.
- 전체 동작 흐름
  - 클라이언트가 `/example?page=3` 요청 보냄

- 스프링이 컨트롤러 메서드의 `page` 파라미터에 `@ZeroBasedPage` 어노테이션이 있는지 검사
- `ZeroBasedPageResolver.supportsParameter()` 가 `true` 반환
- `resolveArgument()` 가 호출되어 `page=3` 문자열을 읽고, `3 - 1 = 2` 로 변환 후 반환
- 컨트롤러 메서드가 `page = 2` 를 받음
- 만약 클라이언트가 `page=0` 이나 `-1` 같은 값 보내면, `IllegalArgumentException` 이 발생
  - 전역 예외 처리기가 예외를 잡아 적절한 에러 응답을 만들어 클라이언트에 반환
- 에러 발생 시 반드시 `RestControllerAdvice`와 연계를 해야 함 => 커스텀 예외처리를 통해 예외를 처리해야한다
- `@Validated` // 반드시 필요! (RequestParam 유효성 검사를 작동시키는 핵심)
- `@Valid` 는 `@RequestBody` 용, `@Validated` 는 `@RequestParam`, `@PathVariable` 등 메서드 파라미터에 대해 작동
 

*`!@RequestParam(name = "page")` / `@ZeroBasedPage` 앞에 `@RequestPream` 어노테이션 때문에 뒤에 있는 커스텀 어노테이션이 동작을 하지 않았음*
- 커스텀 어노테이션을 활용하여 쿼리스트링을 바인딩함 커스텀 어노테이션 로직으로 해당 파라미터를 반환 또는 검증 순서
  1. Client Request 요청
  2. Dispatcher Servlet에서 해당 요청 처리
  3. Client Request에 대한 Handler Mapping
    - 3.1 RequestMapping에 대한 매칭 (RequestMappingHandlerAdapter가 수행)
    - 3.2 Interceptor 처리
    - 3.3 Argument Resolver 처리 <-- Argument Resolver 실행 지점
    - 3.4 Message Converter 처리
  4. Controller Method invoke
    - 정리하자면 특정 Request가 Handler로 Mapping되는 과정에서 invoke 되기 전, Interceptor > Resolver > MessageConverter 순으로 처리된 후, Controller의 Method가 invoke 된다.
    - 다음으로 `WebMvcConfigurer`를 구현한 `WebConfig` 클래스에서 위와 같이 우리가 만든 `LoginUserDetailsResolver` 를 Argument Resolver로 등록한다.

## • 커스텀 어노테이션 기반 파라미터 처리 로직 순서

### 1. 요청 → DispatcherServlet 진입

- 클라이언트로부터 HTTP 요청이 들어오면 `DispatcherServlet` 이 먼저 요청을 가로칩니다.



## 2. `HandlerMapping` 이 적절한 Controller 메서드 매핑

예를 들어, URL `/members?page=2` 이면:

java

복사편집

```
@GetMapping("/members") public ResponseEntity<?> getList(@ZeroBasedPage Integer page) { ... }
```

이 메서드가 선택됩니다.

---

## 3. 컨트롤러 메서드의 각 파라미터에 대해 `ArgumentResolver` 순회

Spring은 등록된 모든 `HandlerMethodArgumentResolver` 를 순회하며 `supportsParameter()` 를 호출합니다.

▶ 순회 중 조건에 맞는 리졸버 발견 시:

- `supportsParameter()` 가 `true` 를 반환하면,
- `resolveArgument()` 가 호출되어 해당 파라미터를 가공/변환해서 컨트롤러 메서드에 주입합니다.

즉:

java

복사편집

```
@Override public boolean supportsParameter(MethodParameter parameter) { return parameter.hasParameterAnnotation(ZeroBasedPage.class) && Integer.class.equals(parameter.getParameterType()); }
```

이게 `true` 면 `resolveArgument()` 실행.

---

#### 4. `resolveArgument()` 실행: 실제 가공 처리

java

복사편집

```
@Override public Object resolveArgument(...) { // 1. 요청에서 query param 추출
String value = webRequest.getParameter("page"); // 2. 검증 if (value == null
|| Integer.parseInt(value) < 1) { throw new GeneralException(...); } // 3. 값
가공 후 반환 return Integer.parseInt(value) - 1; }
```

이 반환값이 `@ZeroBasedPage Integer page` 로 들어갑니다.


---

#### 5. 컨트롤러 메서드 실행

- 이제 `page` 파라미터는 변환된 값( 0-based )으로 주입되어 컨트롤러 비즈니스 로직이 실행됩니다.
- 

#### 6. 응답 반환 → `DispatcherServlet` 이 HTTP 응답으로 변환

---

 요약: 전체 흐름

text

복사편집

```
요청 → DispatcherServlet → HandlerMapping → 컨트롤러 메서드 매핑 →
HandlerMethodArgumentResolver 순회 → supportsParameter() → true이면 →
resolveArgument() 호출 → 커스텀 처리 → 컨트롤러 메서드 실행 → 응답 반환
```

---

⚠ 주의: `@RequestParam` 같이 쓰면 안 되는 이유

- `@RequestParam` 이 붙으면, Spring의 기본 리졸버 (`RequestParamMethodArgumentResolver`)가 먼저 작동하여 **커스텀 리졸버가 처리할 기회를 잃습니다.**

→ `@ZeroBasedPage` 단독 사용해야 함.

---