14. REST API 설계 및 문서화

RESTful API 설계 원칙

🧩 1. REST란?

> **REpresentational State Transfer**의 약자
> 자원(Resource)을 URI로 표현하고, 상태(행위)는 HTTP 메서드로 전달하는
> **웹 기반의 아키텍처 스타일**

📌 핵심 철학:

- URL은 **자원(Resource)**을 표현하고
- HTTP Method는 **행위(Action)**를 표현함

📐 2. RESTful URI 설계 원칙

원칙	설명
자원 중심	동사는 쓰지 않고, 명사로 표현
계층 구조	하위 자원을 포함할 땐 중첩 표현
복수형 명사 사용	일관성 있게 설계 (`users`, `orders`, `products`)
쿼리는 필터링/검색에만 사용	검색 조건, 정렬, 페이지 등은 query parameter로 처리

✅ 예시

기능	URI
사용자 전체 조회	`GET /users`
사용자 단건 조회	`GET /users/{id}`
사용자 생성	`POST /users`
사용자 수정	`PUT /users/{id}`
사용자 삭제	`DELETE /users/{id}`
조건 검색	`GET /users?age=20&gender=male`
하위 자원 조회	`GET /users/{id}/orders`

📦 3. HTTP 메서드 설계 원칙

Method	용도	특징
`GET`	조회	안전하고 멱등
`POST`	생성	리소스를 생성, 상태 변경
`PUT`	전체 수정	멱등, 리소스 없으면 생성 가능
`PATCH`	부분 수정	일부 필드만 수정
`DELETE`	삭제	멱등, 없는 리소스를 삭제해도 OK

> 📌 "멱등(idempotent)"이란: 같은 요청을 여러 번 보내도 결과가 같다는 뜻

🎯 4. HTTP 응답 상태 코드 설계

상태 코드	의미	사용 예
`200 OK`	정상 처리	일반 조회, 수정
`201 Created`	생성 성공	`POST /users` 후
`204 No Content`	성공했지만 반환할 데이터 없음	삭제 후
`400 Bad Request`	잘못된 요청	필수 필드 누락 등
`401 Unauthorized`	인증 필요	토큰 없음
`403 Forbidden`	권한 부족	접근 불가 리소스
`404 Not Found`	자원 없음	없는 ID 조회
`409 Conflict`	중복 충돌	이미 존재하는 데이터
`500 Internal Server Error`	서버 오류	예외 미처리 시

🧪 5. 응답 구조 설계 (표준화 추천)

```
{
  "success": true,
  "data": {
    "id": 1,
    "name": "Jeongseok"
  },
  "error": null
}
```

실무에서는 다음과 같은 표준 구조가 많이 쓰임:

```json
{
  "code": "USER_NOT_FOUND",
  "message": "사용자를 찾을 수 없습니다.",
  "status": 404
}
```

🧠 6. 계층적 자원 설계

✅ 중첩된 자원

자원	URI
특정 사용자 주문 목록	`GET /users/{userId}/orders`
특정 사용자 특정 주문	`GET /users/{userId}/orders/{orderId}`

> 필요 시 `orders/{id}` 단독 경로도 병행 설계 가능

🔐 7. 보안 & 버전 관리

✅ 버전 관리

방식	예시
URI 방식	`/api/v1/users` ✅ 실무에서 가장 흔함
헤더 방식	`Accept: application/vnd.company.v1+json`
파라미터 방식	`GET /users?version=1` (권장 안 함)

✅ 보안

항목	설명
인증	JWT, OAuth2
HTTPS 사용	모든 API는 HTTPS 필수
CORS 설정	프론트엔드 연동 시 `@CrossOrigin` 또는 글로벌 설정
입력 검증	`@Valid`, `BindingResult`, 예외 처리 (`@ControllerAdvice`) 필수

🧩 8. 기타 실무 설계 포인트

항목	설계 기준
페이징	`GET /users?page=1&size=10`
정렬	`sort=createdDate,desc`
필터링	`GET /products?category=shoes&brand=nike`
상태 변경 API	`PATCH /orders/{id}/cancel`, 또는 `/orders/{id}` + `"status": "CANCELLED"`
파일 업로드	`POST /files` (Content-Type: multipart/form-data)
응답 캐싱	ETag, Cache-Control 헤더 사용

✅ 마무리 요약표

항목	원칙
URI	자원 중심, 복수형 명사 사용
HTTP 메서드	행위 표현, 의미에 맞게 사용 (`GET`, `POST`, `PUT`, `DELETE`)
상태 코드	적절한 응답 코드로 클라이언트에게 피드백
응답 구조	통일된 JSON 구조 사용
버전 관리	URI 방식 추천 (`/api/v1`)
보안	인증 필수, HTTPS 기본, 검증 처리 명확히
REST 철학	상태 없는, 클라이언트-서버 분리 구조 유지

상태 코드 정의

🧩 1. 상태 코드란?

> HTTP 응답의 **상태를 숫자 코드로 나타내는 표준 규격**
> 클라이언트는 이 숫자를 통해 **요청이 성공했는지, 실패했는지, 왜 실패했는지**를 이해함

📦 2. 상태 코드 5대 분류

범위	분류명	의미
1xx	**Informational**	처리 중 (거의 안 씀)
2xx	**Success**	요청 성공
3xx	**Redirection**	요청 리다이렉션 필요

범위	분류명	의미
4xx	**Client Error**	클라이언트 잘못
5xx	**Server Error**	서버 오류 (개발자 책임)

✅ 3. 실무에서 자주 사용하는 상태 코드 정리

✔ 2xx: 성공

코드	의미	설명
200 OK	일반 성공	데이터 조회, 수정 등
201 Created	리소스 생성 성공	`POST /users` 이후
202 Accepted	요청 수락 (비동기 처리)	즉시 처리 X
204 No Content	성공하지만 응답 본문 없음	`DELETE`, `PUT` 완료 후 자원 반환 불필요

✔ 4xx: 클라이언트 잘못

코드	의미	설명
400 Bad Request	잘못된 요청	파라미터 오류, JSON 구조 오류 등
401 Unauthorized	인증 필요	로그인 필요, 토큰 누락
403 Forbidden	접근 금지	인증은 됐지만 권한 없음
404 Not Found	자원 없음	존재하지 않는 ID 등
405 Method Not Allowed	허용되지 않은 HTTP 메서드	`POST`만 가능한데 `PUT` 요청
409 Conflict	리소스 충돌	이미 존재하는 값, 중복 요청 등
422 Unprocessable Entity	유효성 검사 실패	JSON 형식은 맞지만 내용이 유효하지 않음 (입력 검증 실패)

✔ 5xx: 서버 오류

코드	의미	설명
500 Internal Server Error	일반 서버 에러	NPE, DB 오류 등
502 Bad Gateway	게이트웨이 오류	프록시나 API Gateway 연동 실패
503 Service Unavailable	서비스 일시 중지	유지보수, 과부하 등

코드	의미	설명
504 Gateway Timeout	응답 지연	타임아웃 발생 (DB, 외부 API 응답 없음 등)

🧠 4. 실무 설계 기준

상황	권장 상태 코드
조회 성공	`200 OK`
생성 성공	`201 Created` + `Location` 헤더 (생성된 URI)
수정 후	`200 OK` 또는 `204 No Content`
삭제 후	`204 No Content`
요청 파라미터 오류	`400 Bad Request`
인증 안됨	`401 Unauthorized` (토큰 없음)
권한 없음	`403 Forbidden` (토큰은 유효하나 Role 미달)
리소스 없음	`404 Not Found`
중복 등록 시	`409 Conflict`
입력 유효성 실패	`422 Unprocessable Entity`
서버 예외	`500 Internal Server Error`

🔐 5. 사용자 정의 상태 코드? → ❌

> ⚠️ 612, 777, 999 같은 **비표준 코드 사용은 절대 지양**
> → HTTP는 **표준 프로토콜**이며, **모든 클라이언트/프록시/도구가 3자리 숫자**만 인식

✅ 대신 **응답 바디에 비즈니스 에러 코드를 정의**하자:

```
{
  "code": "USER_ALREADY_EXISTS",
  "message": "이미 가입된 사용자입니다.",
  "status": 409
}
```

🔁 6. 응답 예시

✅ 201 Created + Location

```
POST /users
→

HTTP/1.1 201 Created
Location: /users/1004
```

✅ 204 No Content (삭제 후)

```
DELETE /users/1004
→

HTTP/1.1 204 No Content
```

✅ 400 Bad Request (입력 오류)

```
HTTP/1.1 400 Bad Request

{
  "code": "INVALID_INPUT",
  "message": "이메일 형식이 유효하지 않습니다."
}
```

✅ 마무리 요약표

범위	의미	대표 코드	사용 용도
2xx	성공	`200`, `201`, `204`	CRUD 성공
4xx	클라이언트 잘못	`400`, `401`, `403`, `404`, `409`	입력/인증/권한 오류
5xx	서버 잘못	`500`, `503`, `504`	코드 예외, 타 시스템 오류
❌ 사용자 정의	❌	`999`, `777` 등	❌ 절대 지양 – 바디로 비즈니스 코드 전달

REST 예외 처리 전략

🧩 1. 목표

- 클라이언트가 **예외 상황을 명확하게 이해할 수 있도록**
- HTTP 상태 코드 + 명확한 메시지 + 표준화된 응답 형식 제공

✅ 2. 기본 처리 흐름

```
예외 발생
  ↓
Spring Exception Resolver
  ↓
→ @ExceptionHandler (개별)
→ @ControllerAdvice (글로벌)
→ Default Exception 처리
```

🎯 3. 예외 처리 방식 3단계

방식	설명
✅ `@ExceptionHandler`	특정 컨트롤러에서 예외를 직접 처리
✅ `@ControllerAdvice`	애플리케이션 전체에 대한 글로벌 예외 처리
✅ `ResponseEntity<T>`	예외 발생 여부에 관계없이 명확한 응답 포맷 반환 가능

📘 4. ResponseEntity 기본 예제

```
@GetMapping("/users/{id}")
public ResponseEntity<UserDto> getUser(@PathVariable Long id) {
    User user = userRepository.findById(id)
        .orElseThrow(() -> new NotFoundException("사용자를 찾을 수 없습니다."));

    return ResponseEntity.ok(UserDto.of(user));
}
```

📦 5. 예외 클래스 정의 (Custom Exception)

```
public class NotFoundException extends RuntimeException {
    public NotFoundException(String message) {
        super(message);
    }
}
```

✅ 6. 글로벌 예외 처리 with `@ControllerAdvice`

```
@RestControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(NotFoundException.class)
    public ResponseEntity<ErrorResponse> handleNotFound(NotFoundException ex) {
```

```
        return ResponseEntity
            .status(HttpStatus.NOT_FOUND)
            .body(new ErrorResponse("NOT_FOUND", ex.getMessage()));
    }

    @ExceptionHandler(MethodArgumentNotValidException.class)
    public ResponseEntity<ErrorResponse>
handleValidation(MethodArgumentNotValidException ex) {
        String message = ex.getBindingResult().getFieldError().getDefaultMessage();
        return ResponseEntity
            .badRequest()
            .body(new ErrorResponse("INVALID_INPUT", message));
    }

    @ExceptionHandler(Exception.class)
    public ResponseEntity<ErrorResponse> handleAll(Exception ex) {
        return ResponseEntity
            .status(HttpStatus.INTERNAL_SERVER_ERROR)
            .body(new ErrorResponse("INTERNAL_ERROR", "예상치 못한 오류가 발생했습니다."));
    }
}
```

7. 표준 에러 응답 구조

```
{
  "code": "INVALID_INPUT",
  "message": "이메일 형식이 올바르지 않습니다.",
  "status": 400
}
```

ErrorResponse 클래스 예시

```
public class ErrorResponse {
    private final String code;
    private final String message;
    private final int status;

    public ErrorResponse(String code, String message) {
        this.code = code;
        this.message = message;
        this.status = resolveStatus(code);
    }

    // Optional: code → status 매핑 로직 추가
}
```

🔒 8. 실무 설계 기준

항목	설계 방식
예외는 반드시 코드/메시지/상태값 포함	ErrorResponse 클래스 고정화
컨트롤러는 비즈니스 로직만	예외 처리 로직은 전부 `@ControllerAdvice` 로 분리
클라이언트가 해석할 수 있는 구조	code 값은 문자열 (`USER_NOT_FOUND`, `TOKEN_EXPIRED`) 형태
상태 코드와 일치	404 예외 → 404 응답, 400 → 400
유효성 검증 예외 처리	`MethodArgumentNotValidException`, `BindException` 대응
Exception은 마지막 catch	로그 남기고 모호한 예외는 `500` 처리

🚫 9. 절대 하면 안 되는 것

잘못된 패턴	이유
예외를 무시하고 null 반환	디버깅, 로깅, 보안 위험
모든 예외에 200 OK 반환	클라이언트가 정상으로 인식
상태 코드 없이 "message"만 전달	기계가 해석 불가능함
사용자 정의 상태코드 (e.g. 999)	HTTP 표준 위반, 프록시·API Gateway에서 오류 발생

✅ 마무리 요약표

항목	설명
`@ControllerAdvice`	전역 예외 처리 담당
`@ExceptionHandler`	개별 예외 클래스 처리
`ResponseEntity`	상태 코드 + 응답 구조 지정 가능
Error 구조	code, message, status 포함
실무 기준	표준 응답 포맷, 공통 코드 관리, 명확한 상태 매핑
테스트 전략	예외 발생 케이스 단위 테스트 필수

표준 응답 구조 설계

🧩 1. 왜 필요한가?

- 다양한 클라이언트(iOS, Android, Web)가 API를 동일한 방식으로 파싱할 수 있게 하기 위함
- 상태 코드만으로는 부족한 **비즈니스 로직 오류 판단** 가능
- ✅ 유지보수, 디버깅, 문서화에 **일관성과 안정성** 제공

📦 2. 응답 분류: 3가지

유형	설명	상태 코드
✅ 성공 응답	비즈니스 로직 성공	200, 201, 204 등
✅ 실패 응답	요청은 성공했지만 비즈니스 실패	200 (실무에서 자주 사용) or 400+
✅ 예외 응답	시스템 오류, 유효성 오류 등	400, 404, 500 등

📐 3. 기본 응답 포맷 (공통 구조)

```
{
  "success": true,
  "code": "SUCCESS",
  "message": "요청이 성공적으로 처리되었습니다.",
  "data": {
    "id": 1,
    "name": "jeongseok"
  }
}
```

❌ 실패 또는 예외일 때

```
{
  "success": false,
  "code": "USER_NOT_FOUND",
  "message": "해당 사용자를 찾을 수 없습니다.",
  "data": null
}
```

> ☝️ `HTTP status`는 404, 400, 500 등 실제 상태 코드로 설정하고
> 본문은 이처럼 **표준화된 메시지 구조**로 응답

✅ 4. 응답 클래스 설계 (Generic 기반)

```
public class ApiResponse<T> {

    private final boolean success;
    private final String code;
    private final String message;
    private final T data;

    private ApiResponse(boolean success, String code, String message, T data) {
        this.success = success;
        this.code = code;
        this.message = message;
        this.data = data;
    }

    public static <T> ApiResponse<T> success(T data) {
        return new ApiResponse<>(true, "SUCCESS", "요청이 성공적으로 처리되었습니다.", data);
    }

    public static <T> ApiResponse<T> error(String code, String message) {
        return new ApiResponse<>(false, code, message, null);
    }

    // Getter 생략
}
```

🎯 5. 응답 코드 관리 방식

```
public enum ErrorCode {
    USER_NOT_FOUND("USER_NOT_FOUND", HttpStatus.NOT_FOUND, "사용자를 찾을 수 없습니다."),
    INVALID_INPUT("INVALID_INPUT", HttpStatus.BAD_REQUEST, "입력값이 올바르지 않습니다."),
    INTERNAL_ERROR("INTERNAL_ERROR", HttpStatus.INTERNAL_SERVER_ERROR, "서버 오류가 발생했습니다.");

    private final String code;
    private final HttpStatus status;
    private final String message;
}
```

→ 예외 발생 시 `ErrorCode`를 기반으로 `ApiResponse.error()` 생성

💻 6. 컨트롤러에서 사용 예시

```
@GetMapping("/users/{id}")
public ResponseEntity<ApiResponse<UserDto>> getUser(@PathVariable Long id) {
    User user = userRepository.findById(id)
        .orElseThrow(() -> new CustomException(ErrorCode.USER_NOT_FOUND));

    return ResponseEntity.ok(ApiResponse.success(UserDto.of(user)));
}
```

📘 7. 글로벌 예외 처리 적용

```
@ExceptionHandler(CustomException.class)
public ResponseEntity<ApiResponse<Void>> handleCustom(CustomException ex) {
    ErrorCode code = ex.getErrorCode();
    return ResponseEntity
        .status(code.getStatus())
        .body(ApiResponse.error(code.getCode(), code.getMessage()));
}
```

🧠 8. 실무 설계 기준

항목	기준
응답 형식	항상 같은 구조 (`success`, `code`, `message`, `data`)
성공 응답	200~204 + `success: true`, `code: SUCCESS`
실패 응답	400, 404, 409 등 + `success: false`, 명확한 code
에러 코드 관리	enum 기반 관리 (중앙 집중화)
예외 메시지	사용자 친화적 메시지 제공 (보안 정보 숨기기)
Swagger 연동	응답 구조 명세화 (`@ApiResponse`, `@Schema`) 가능

✅ 마무리 요약표

항목	설명
응답 구조	`success`, `code`, `message`, `data` 4가지 필드 고정
성공	`success: true` + 2xx 상태 코드
실패	`success: false` + 4xx/5xx + 명확한 code/message
제네릭 응답 객체	`ApiResponse<T>` 활용

항목	설명
코드 관리	`ErrorCode enum`으로 중앙 집중

API 버전 관리 전략

🧩 1. 왜 API 버전 관리가 필요한가?

- 클라이언트가 사용하는 **기존 API를 깨지 않고**,
- 새로운 기능이나 변경된 규칙을 **점진적으로 적용**할 수 있게 하기 위해

예:

> `GET /users/1` 응답 구조가 바뀌면 구 클라이언트는 파싱에 실패함 → **호환성 깨짐**

🎯 2. API 버전 관리 방식 3가지

방식	예시	특징
✅ URI 버전 방식	`/api/v1/users`	명확, 직관적, 브라우저/문서화 쉬움
헤더 기반 방식	`Accept: application/vnd.company.v1+json`	REST 철학 지향, 구현 복잡
파라미터 기반 방식	`/users?version=1`	쉬움, RESTful하지 않음 (❌ 권장 안 함)

✅ 3. URI 버전 방식 (권장)

```
GET /api/v1/users
GET /api/v2/users
```

- 각 버전마다 컨트롤러 클래스를 **별도로 분리**
- Swagger 문서도 버전별 생성 가능
- 클라이언트가 명확히 버전 호출 가능

```
@RestController
@RequestMapping("/api/v1/users")
public class UserControllerV1 { ... }

@RestController
@RequestMapping("/api/v2/users")
public class UserControllerV2 { ... }
```

📦 4. 헤더 기반 방식 (Header Versioning)

```
GET /users
Accept: application/vnd.myapp.v1+json
```

- 컨트롤러에서 헤더 조건으로 분기

```
@GetMapping(value = "/users", headers = "X-API-VERSION=1")
public UserResponseV1 getV1() { ... }

@GetMapping(value = "/users", headers = "X-API-VERSION=2")
public UserResponseV2 getV2() { ... }
```

> 📌 Spring에서는 `@RequestMapping`의 `headers` 속성으로 조건 제어 가능

🧪 5. Media Type 방식 (Content Negotiation)

```
Accept: application/vnd.company.user.v2+json
```

- 복잡하지만 엄밀한 REST 스타일
- `produces` 속성으로 컨트롤러 분기 가능

```
@GetMapping(value = "/users", produces = "application/vnd.company.v2+json")
```

> 하지만 브라우저에서 호출 테스트 불편, 도입 어려움

⚠️ 6. 방식 비교 요약

항목	URI 방식	헤더 방식	Media Type 방식
직관성	✅ 매우 높음	❌ 낮음	❌ 낮음
브라우저 친화성	✅ 좋음	❌ 헤더 추가 필요	❌
Swagger 문서화	✅ 쉬움	⚠️ 설정 필요	❌ 어려움
구현 난이도	쉬움	중간	복잡
REST 철학	약간 위반	✅ 적합	✅ 적합
실무 적합성	✅ 최고	⚠️ 대규모 기업용	❌ 거의 안 씀

> 실무에서는 **URI 방식** + Swagger 문서화가 가장 많이 쓰임

🧠 7. 실무 운영 전략

전략	설명
버전은 최대 2개 이하 유지	v1 + v2까지만 유지하고, 이후엔 하나 교체
v1을 Deprecated 표시 후 v2 공지	문서화 도구(Swagger)에 “deprecated” 표시
응답 포맷 일관 유지	v1과 v2는 구조만 바뀌지, 형태는 유사하게 유지
버전마다 DTO 분리	`UserResponseV1`, `UserResponseV2` 등으로 설계 분리
테스트 자동화	버전별 통합 테스트 구성 필수

📘 8. Swagger 문서 분리 예시 (Springdoc 기준)

```yaml
springdoc:
  group-configs:
    - group: v1
      paths-to-match: /api/v1/**
    - group: v2
      paths-to-match: /api/v2/**
```

```java
@RestController
@RequestMapping("/api/v2/users")
@Tag(name = "User API v2", description = "신규 사용자 API")
public class UserControllerV2 { ... }
```

🔁 9. DTO 버전 분리 패턴

```java
public class UserResponseV1 {
    private String name;
    private String email;
}

public class UserResponseV2 {
    private String name;
    private String email;
    private LocalDateTime registeredAt;
}
```

→ 버전에 따라 DTO, 컨트롤러, 서비스 계층까지 **독립 유지**
→ 클라이언트 오류 방지

✅ 마무리 요약

항목	설명
기본 방식	`URI 방식 (/api/v1)`이 실무에서 가장 안정적
버전 전략	DTO/Controller 완전 분리, 테스트도 버전별로
문서화	Swagger에서 group 별 분리 추천
운영 정책	v1은 Deprecated 표시 후 일정 기간 유지
권장 조합	`URI + Swagger 문서 + 버전별 DTO + 자동 테스트`

Swagger UI 설정

🧩 1. Swagger란?

> REST API 명세를 **자동 생성하고 UI로 문서화**해주는 도구.
> Spring Boot에서는 Springfox 대신 **Springdoc OpenAPI 3** 사용을 권장 (Spring Boot 3 호환성 ↑)

📦 2. 의존성 설정

✅ Maven

```
<dependency>
  <groupId>org.springdoc</groupId>
  <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
  <version>2.2.0</version>
</dependency>
```

✅ Gradle

```
implementation("org.springdoc:springdoc-openapi-starter-webmvc-ui:2.2.0")
```

🔗 3. Swagger UI 접속 경로

Spring Boot 실행 후:

```
http://localhost:8080/swagger-ui.html
```

or (자동 리다이렉트됨)

```
http://localhost:8080/swagger-ui/index.html
```

📘 4. 기본 설정 예시 (`application.yml`)

```yaml
springdoc:
  api-docs:
    path: /v3/api-docs
  swagger-ui:
    path: /swagger-ui.html
    tags-sorter: alpha
    operations-sorter: method
    display-request-duration: true
```

✅ 5. API 문서 커스터마이징 어노테이션

어노테이션	설명
`@Tag(name = "User", description = "사용자 관련 API")`	API 그룹 분류
`@Operation(summary = "회원 조회", description = "회원 ID로 조회합니다.")`	메서드 설명
`@Parameter(name = "id", description = "회원 ID", required = true)`	파라미터 설명
`@Schema(description = "사용자 응답 DTO")`	DTO 필드 설명
`@ApiResponse(responseCode = "200", description = "성공")`	상태 코드 설명

📂 6. 예시 Controller

```java
@RestController
@RequestMapping("/api/v1/users")
@Tag(name = "User", description = "사용자 API")
public class UserController {

    @Operation(summary = "회원 조회", description = "회원 ID로 조회")
    @ApiResponse(responseCode = "200", description = "조회 성공")
    @GetMapping("/{id}")
    public ResponseEntity<UserDto> getUser(
        @Parameter(description = "회원 ID") @PathVariable Long id) {
        return ResponseEntity.ok(...);
    }
}
```

📊 7. 그룹 / 버전 분리 (멀티 API)

```yaml
springdoc:
  group-configs:
    - group: v1
      paths-to-match: /api/v1/**
    - group: v2
      paths-to-match: /api/v2/**
```

➡️ UI에서 `v1`, `v2` 그룹을 선택할 수 있게 됨

🔐 8. JWT 보안 헤더 추가 (Authorization 설정)

```yaml
springdoc:
  swagger-ui:
    persist-authorization: true
components:
  securitySchemes:
    bearer-key:
      type: http
      scheme: bearer
      bearerFormat: JWT
security:
  - bearer-key: []
```

```java
@SecurityRequirement(name = "bearer-key")
@Operation(summary = "보호된 API", security = { @SecurityRequirement(name = "bearer-key")
})
```

→ Swagger UI에 **Authorize** 버튼이 생기고, JWT 토큰을 입력하면 `Authorization: Bearer xxx` 헤더가 자동 추가됨

🧠 9. 실무 팁

팁	설명
DTO마다 `@Schema(description = "...")` 붙이면 자동 문서화됨	
테스트용 Token 입력은 `Authorize` 버튼으로 자동 처리 가능	
응답 예시는 `@ExampleObject` 또는 `@ApiResponse`로 명시 가능	
문서 숨기기: `@Hidden` 어노테이션 사용	
서버 URL 설정: `@OpenAPIDefinition(servers = ...)` 사용	

✅ 마무리 요약

항목	내용
UI 접속 경로	`/swagger-ui.html` or `/swagger-ui/index.html`
의존성	`springdoc-openapi-starter-webmvc-ui`
기본 설정	`application.yml`에서 경로 및 정렬 설정 가능
커스터마이징	`@Tag`, `@Operation`, `@Schema` 등으로 문서화
그룹화	`group-configs`로 v1/v2 등 문서 분리
보안 헤더	JWT/OAuth2 인증 헤더 추가 가능 (`@SecurityRequirement`)

SpringDoc OpenAPI 설정

🧩 1. SpringDoc OpenAPI란?

> Spring Boot 애플리케이션의 REST API를 분석해
> **자동으로 OpenAPI(Swagger) 문서와 UI를 생성**해주는 도구

✅ `Springfox`의 후속 라이브러리로 Spring Boot 3.x 호환성과 유지보수성이 뛰어남

📦 2. 의존성 추가

✅ Gradle (Kotlin DSL 기준)

```
implementation("org.springdoc:springdoc-openapi-starter-webmvc-ui:2.2.0")
```

> ✅ Spring Boot 3 이상은 `springdoc-openapi-starter-*` 버전 사용

🔗 3. Swagger UI 접속 경로

서버 실행 후:

```
http://localhost:8080/swagger-ui.html
or
http://localhost:8080/swagger-ui/index.html
```

OpenAPI 스펙(JSON):

```
http://localhost:8080/v3/api-docs
```

⚙️ 4. application.yml 설정

```yaml
springdoc:
  swagger-ui:
    path: /swagger-ui.html
    display-request-duration: true
    operations-sorter: method
    tags-sorter: alpha
    try-it-out-enabled: true
  api-docs:
    enabled: true
    path: /v3/api-docs
```

📂 5. API 문서 그룹화 (버전별, 도메인별)

```yaml
springdoc:
  group-configs:
    - group: v1
      display-name: Version 1
      paths-to-match: /api/v1/**
    - group: v2
      display-name: Version 2
      paths-to-match: /api/v2/**
```

➡️ Swagger UI에서 `v1`, `v2` 문서를 선택 가능

🛡️ 6. 보안 설정 (JWT Bearer)

✅ application.yml

```yaml
springdoc:
  swagger-ui:
    persist-authorization: true
components:
  security-schemes:
    bearer-key:
      type: http
      scheme: bearer
      bearer-format: JWT
security:
  - bearer-key: []
```

✅ 컨트롤러에 적용

```
@Operation(summary = "사용자 조회", security = { @SecurityRequirement(name = "bearer-key")
})
@GetMapping("/users")
public UserResponse getUser() { ... }
```

➡️ Swagger UI 상단에 **Authorize 버튼** 생성됨

📘 7. DTO & Controller 주석 어노테이션

대상	어노테이션	예시
API 그룹	`@Tag(name, description)`	컨트롤러 단에 적용
메서드	`@Operation(summary, description)`	API 설명
파라미터	`@Parameter(description)`	`@PathVariable`, `@RequestParam`
DTO 필드	`@Schema(description, example, required)`	요청/응답 DTO에 적용
응답 코드	`@ApiResponse(responseCode, description)`	성공/실패 명세 추가

✅ 예시

```
@Tag(name = "User", description = "사용자 관련 API")
@RestController
@RequestMapping("/api/v1/users")
public class UserController {

    @Operation(summary = "사용자 단건 조회", description = "ID로 사용자 정보 조회")
    @ApiResponse(responseCode = "200", description = "조회 성공")
    @GetMapping("/{id}")
    public UserDto getUser(@Parameter(description = "사용자 ID") @PathVariable Long id)
{
        ...
    }
}
```

📑 8. 예외 응답 문서화

SpringDoc은 기본적으로 `@ExceptionHandler`까지 문서화하지 않지만,
`@ApiResponse`, `@ApiResponses`로 명시 가능:

```
@ApiResponses({
    @ApiResponse(responseCode = "404", description = "사용자 없음"),
    @ApiResponse(responseCode = "400", description = "요청 오류")
})
```

또는 전체 공통 예외는 `@ControllerAdvice` + Swagger 주석 조합 필요

🧠 9. 실무 팁

팁	설명
DTO에 `@Schema(description, example)` 추가 시 자동 문서화	
`@Hidden` 어노테이션으로 Swagger에서 제외 가능	
Swagger UI에서 Try it out → 테스트 가능	
문서 비공개 시에는 `springdoc.api-docs.enabled=false` 설정 가능	
클라이언트 SDK 생성 가능 (OpenAPI Generator 사용)	

✅ 마무리 요약

항목	설명
기본 URL	`/swagger-ui.html`, `/v3/api-docs`
그룹화	`group-configs`로 버전별/도메인별 문서 분리
보안 연동	JWT 설정 시 `@SecurityRequirement(name = "bearer-key")`
문서 커스터마이징	`@Tag`, `@Operation`, `@Schema`, `@Parameter`, `@ApiResponse` 등
실무 활용도	UI 기반 테스트 + 문서화 + API 버전 관리 가능

REST Docs

🧩 1. Spring REST Docs란?

> **테스트 코드 기반으로 실제 API 요청/응답을 문서화**하는 Spring 공식 문서화 도구.
> Swagger처럼 실행 중인 서버에서 동작하는 UI가 아니라,
> ✅ **정적 문서(HTML, PDF 등)로 배포 가능**하며,
> ✅ **문서 = 실제 테스트 코드 결과**라는 점이 가장 강력한 장점.

🔄 2. Swagger vs REST Docs 비교

항목	Swagger (SpringDoc)	Spring REST Docs
작동 방식	코드/어노테이션 기반 자동 생성	테스트 결과 기반 문서화
UI	✅ Swagger UI	❌ 없음 (HTML/PDF로 생성)
신뢰도	어노테이션 → 실제와 다를 수 있음	✅ 테스트 결과 = 문서 내용
문서 형식	JSON(OpenAPI) → Swagger UI	AsciiDoc → HTML/PDF
실무 적합성	초기 개발, 인터랙티브	✅ 대기업, 금융권, 릴리즈 문서 등

> ✅ REST Docs는 "진짜 문서 = 실제 테스트 성공 응답"을 보장함

📦 3. 의존성 설정 (Gradle 기준)

```
dependencies {
    implementation("org.springframework.boot:spring-boot-starter-web")
    testImplementation("org.springframework.restdocs:spring-restdocs-mockmvc")
    testImplementation("org.springframework.boot:spring-boot-starter-test")
}

plugins {
    id("org.asciidoctor.jvm.convert") version "3.3.2"
}
```

🛠️ 4. 설정 추가 (`build.gradle.kts`)

```
tasks.test {
    outputs.dir(snippetsDir)
    useJUnitPlatform()
}

val snippetsDir by extra { file("build/generated-snippets") }

tasks.named<org.asciidoctor.gradle.jvm.AsciidoctorTask>("asciidoctor") {
    inputs.dir(snippetsDir)
    dependsOn(tasks.test)
    sourceDir.set(file("src/docs/asciidoc"))
    outputDir.set(file("build/docs/asciidoc"))
    attributes(
        mapOf("snippets" to snippetsDir)
    )
}
```

📄 5. 테스트 기반 문서 생성 예시

✅ 컨트롤러

```
@RestController
@RequestMapping("/api/users")
public class UserController {
    @GetMapping("/{id}")
    public ResponseEntity<UserResponse> getUser(@PathVariable Long id) {
        return ResponseEntity.ok(new UserResponse(id, "jeongseok"));
    }
}
```

✅ 테스트 코드 (MockMvc + REST Docs)

```
@WebMvcTest
@AutoConfigureRestDocs
class UserControllerDocsTest {

    @Autowired
    private MockMvc mockMvc;

    @Test
    void 사용자_조회_문서화() throws Exception {
        mockMvc.perform(get("/api/users/{id}", 1L)
                .accept(MediaType.APPLICATION_JSON))
            .andExpect(status().isOk())
            .andDo(document("get-user",
                pathParameters(
                    parameterWithName("id").description("사용자 ID")
                ),
                responseFields(
                    fieldWithPath("id").description("사용자 ID"),
                    fieldWithPath("name").description("사용자 이름")
                )
            ));
    }
}
```

➡️ 실행 시: `build/generated-snippets/get-user/` 경로에 `.adoc` 스니펫 생성

📘 6. AsciiDoc 문서 작성 (`src/docs/asciidoc/index.adoc`)

```
= REST API 문서
:toc: left
:toclevels: 2
:sectlinks:

== 사용자 API

=== 사용자 단건 조회

include::{snippets}/get-user/path-parameters.adoc[]
include::{snippets}/get-user/response-fields.adoc[]
```

📊 7. 문서 빌드 명령

```
./gradlew clean build asciidoctor
```

➡️ `build/docs/asciidoc/index.html` 생성됨

🔐 8. 보안 테스트 포함 시

- 인증 헤더 추가:

```
.header("Authorization", "Bearer sample.token")
```

- `@AutoConfigureRestDocs(uriScheme = "https", uriHost = "api.example.com", uriPort = 443)` 로 실제 배포 환경 경로 설정 가능

🧠 9. 실무 설계 팁

팁	설명
REST Docs는 실제 API 응답 기반이라 문서 신뢰도 최고	
DTO가 바뀌면 테스트 실패 → 문서 자동 변경됨	
문서 자동화 + CI 통합 가능 (`./gradlew asciidoctor` 포함)	
Spring REST Docs + Swagger UI 병행도 가능 (초기 + 정식 버전 문서 분리)	
`spring-restdocs-webtestclient` 도 WebClient 기반 문서화 지원함	

✅ 마무리 요약

항목	내용
문서화 방식	테스트 코드 기반 (`MockMvc` or `WebTestClient`)
문서 형식	`.adoc` → HTML, PDF 등
신뢰도	✅ 테스트 결과 = 문서 내용
빌드	`./gradlew asciidoctor`
출력 위치	`build/docs/asciidoc/index.html`
Swagger와 비교	UI는 없음, 정적 문서화 + 배포에 적합