KISIA S-개발자 4기

개발자들의 약속: API 설계

API 중심 설계와 오픈소스 활용 — 1일차: 설계편

Day 1 of 5

Day 1 Day 2 Day 3 Day 4 Day 5

1일차 교육 일정

09:00-09:20	아이스브레이킹 & 오리엔테이션
09:20-10:00	API 개념과 API-First Design
10:00-10:10	🧉 휴식
10:10-11:00	JSON / YAML / HTTP 기초
11:00-11:10	🧉 휴식
11:10-12:00	OpenAPI Specification & Swagger Editor
12:00-13:00	🍽️ 점심 시간
13:00-15:00	[실습 1] 스타벅스 메뉴 조회 API 설계
15:00-15:10	🧉 휴식
15:10-17:30	[실습 2] 회원가입 & 주문 API 설계
17:30-18:00	1일차 요약 & Q&A

1일차 학습 목표

API-First Design의 개념 이해

코딩 전 설계도부터 확정하는 문화를 배웁니다

JSON / YAML / HTTP 기초 체득

API의 언어인 데이터 포맷과 통신 규약을 익힙니다

OpenAPI Specification 이해

전 세계 공통 API 설계 표준을 학습합니다

Swagger Editor로 API 설계도 완성

실제 도구를 사용하여 API 명세서를 작성합니다

※ OpenAPI(표준 규격)를 작성하기 위한 도구가 Swagger입니다

아이스브레이킹

여러분이 오늘 아침에 쓴 앱 중 API를 안 쓴 앱이 있을까요?

배달의민족 → 메뉴 API, 주문 API, 결제 API

카카오톡 → 메시지 API, 친구목록 API, 프로필 API

유튜브 → 영상 API, 검색 API, 추천 API

→ 우리가 쓰는 모든 앱은 API로 움직인다!

Section 1

API란 무엇인가?

자판기 비유로 쉽게 이해하기

API의 정의: 자판기 비유

- **API = Application Programming Interface**
  → 프로그램과 프로그램 사이의 '대화 규칙'
- **자판기 비유:**
  → 동전을 넣고(요청) → 버튼을 누르면(엔드포인트) → 음료수가 나온다(응답)
  → 자판기 내부 구조는 몰라도 됨 = 추상화
- **API도 마찬가지:**
  → 요청(Request)을 보내면 → 서버가 처리해서 → 응답(Response)을 보내줌
  → 서버 내부 로직은 몰라도 됨, 약속된 형식만 지키면 OK

왜 분리해서 개발하나?

- **프론트엔드(화면) + 백엔드(데이터) = 분업 구조**
  - → 프론트엔드: 사용자가 보는 화면, 버튼, UI
  - → 백엔드: 데이터 저장, 처리, 비즈니스 로직
- **왜 분리?**
  - → 1. 전문성: 프론트/백 개발자가 각자 잘하는 걸 한다
  - → 2. 독립성: 앱 화면만 바꿔도 서버는 그대로
  - → 3. 확장성: 웹 + 모바일 + IoT가 같은 백엔드 사용
- **API = 프론트와 백이 대화하는 유일한 통로**

실무 고충: 약속 없는 개발의 최후

"어제는 데이터가 왔는데 오늘은 왜 안 와요?"

프론트엔드 개발자:

"API 응답 형식이 바뀌어서 화면이 다 깨졌어요..."

백엔드 개발자:

"아 그거 어제 필드명 바꿨는데 안 알려드렸나..."

→ 해결책: 코딩 전에 API 설계도(명세서)를 먼저 합의!

API-First Design이란?

- **코딩 전 설계도부터 확정하는 개발 문화**
  - → Code First: 코드 먼저 → 문서 나중에 (전통적)
  - → API First: 설계도 먼저 → 코드 나중에 (현대적)
- **API-First의 장점:**
  - → 1. 프론트/백이 설계도를 보고 동시에 개발 시작 (병렬 개발)
  - → 2. 소통 비용 감소 — 설계도가 유일한 진실의 원천(SSOT)
  - → 3. 변경 시 설계도만 수정 → 자동으로 문서/코드 반영
- **실무에서 API-First를 채택하는 기업이 급증하는 추세**

설계도의 효과: 병렬 개발

설계도 없이 (순차 개발)

- 백엔드 먼저 완성 → 프론트 시작
- 응답 형식 변경 시 양쪽 수정
- 소통 비용 높음 (회의, 메신저)
- 총 소요: 8주
- 버그 발견 시 전면 수정

설계도 있으면 (병렬 개발)

- 설계도 합의 → 동시 개발 시작
- 설계도가 계약서 역할
- Mock 서버로 프론트 독립 개발
- 총 소요: 4주 (50% 단축!)
- 설계 단계에서 문제 미리 발견

Section 2

JSON, YAML, HTTP 기초

API의 언어를 배우자

JSON 문법 기초

- **JSON = JavaScript Object Notation**

  → API에서 데이터를 주고받는 표준 포맷

- **핵심 구조: Key-Value(키-값) 쌍**

  → { "이름": "홍길동", "나이": 25, "직업": "개발자" }

- **데이터 타입:**

  → 문자열: "hello", 숫자: 42, 불리언: true/false

  → 배열: [1, 2, 3], 객체: { "key": "value" }

- **규칙: 키는 반드시 큰따옴표, 마지막 항목에 콤마 금지**

JSON 실습: 내 인적사항을 JSON으로 작성해보기

JSON

```
{
  "이름": "홍길동",
  "나이": 28,
  "이메일": "hong@example.com",
  "직업": "백엔드 개발자",
  "기술스택": ["JavaScript", "Python", "Node.js"],
  "회사": {
    "이름": "S-개발자4기",
    "부서": "개발팀",
    "입사년도": 2023
  },
  "취미": ["코딩", "독서", "게임"],
  "결혼여부": false
}

// 여러분도 자신의 정보를 JSON으로 작성해보세요!
// jsonlint.com에서 검증할 수 있습니다
```

YAML 문법 기초

- **YAML = YAML Ain't Markup Language**

  → JSON보다 읽기 편한 구조, 설정 파일에 많이 사용

- **핵심: 들여쓰기(스페이스 2칸)로 구조 표현**
- **JSON과 비교:**

  → JSON: { "name": "홍길동", "age": 25 }

  → YAML: name: 홍길동 / age: 25

- **배열은 하이픈(-)으로 표현**

  → skills:
  - JavaScript
  - Python

- **주의: 탭(Tab) 사용 금지! 반드시 스페이스만 사용**

YAML vs JSON 비교

YAML / JSON

```yaml
# YAML 형식 (OpenAPI에서 사용)
name: 홍길동
age: 28
email: hong@example.com
skills:
  - JavaScript
  - Python
  - Node.js
company:
  name: S-개발자4기
  department: 개발팀

---
// 동일한 내용의 JSON 형식
{
  "name": "홍길동",
  "age": 28,
  "email": "hong@example.com",
  "skills": ["JavaScript", "Python", "Node.js"],
  "company": {
    "name": "S-개발자4기",
    "department": "개발팀"
  }
}
```

HTTP 메서드 (1) GET — 데이터 읽기

- **GET = 서버에서 데이터를 가져오는 요청**
  → 비유: 레스토랑에서 메뉴판 보기
- **예시: GET /api/menus**
  → 서버가 전체 메뉴 목록을 JSON으로 응답
- **특징:**
  → Body(본문)가 없음 — URL에 모든 정보 포함
  → 여러 번 요청해도 결과 동일 (멱등성)
  → 캐싱 가능 — 같은 요청은 저장된 결과 재사용
- **가장 많이 사용되는 HTTP 메서드 (전체의 약 60%)**

HTTP 메서드 (2) POST — 데이터 생성

- **POST = 서버에 새 데이터를 보내는 요청**
  → 비유: 레스토랑에서 주문하기
- **예시: POST /api/orders**
  → Body: { "menu": "아메리카노", "size": "tall" }
  → 서버가 주문을 생성하고 주문번호를 응답
- **특징:**
  → Body에 데이터를 담아서 전송
  → 매번 호출하면 새 데이터가 생성됨 (비멱등)
- **회원가입, 주문, 글 작성 등에 사용**

HTTP 메서드 (3) PUT/DELETE — 수정/삭제

- **PUT = 기존 데이터를 수정하는 요청**
  - → 비유: 주문 변경 — '아이스로 바꿔주세요'
  - → PUT /api/orders/42 Body: { size: 'grande' }
- **DELETE = 데이터를 삭제하는 요청**
  - → 비유: 주문 취소
  - → DELETE /api/orders/42 → 주문 삭제됨
- **PATCH = 데이터 일부만 수정**
  - → PUT은 전체 교체, PATCH는 일부만 변경
- **CRUD 매핑: Create=POST, Read=GET, Update=PUT, Delete=DELETE**

HTTP 상태 코드 — 서버의 대답

- **200 OK — 요청 성공! 정상 응답**

  → 비유: '주문하신 음료 나왔습니다~'

- **201 Created — 새 데이터 생성 성공**

  → 비유: '새 주문이 접수되었습니다'

- **400 Bad Request — 잘못된 요청**

  → 비유: '메뉴에 없는 음료를 주문하셨어요'

- **404 Not Found — 요청한 데이터 없음**

  → 비유: '42번 주문은 존재하지 않습니다'

- **500 Internal Server Error — 서버 에러**

  → 비유: '주방에 불이 났습니다...'

Section 3

OpenAPI & Swagger Editor

전 세계 공통 API 설계 표준

OpenAPI Specification (OAS)

- **OpenAPI = 전 세계 공통 API 설계 표준 (규격)**

  → OpenAPI Specification(OAS)은 API를 어떻게 기술할지 정의한 **국제 표준 규격**입니다
  → **Swagger**는 이 OpenAPI 규격을 작성하고 시각화하는 **도구 모음**입니다 (Swagger Editor, Swagger UI, Codegen)
  ★ 핵심: OpenAPI를 만들기 위해 Swagger 도구를 활용한다!

- **역할: API의 설계도를 YAML/JSON으로 작성**

  → 어떤 URL이 있고, 어떤 데이터를 보내고, 어떤 응답이 오는지

- **누가 읽나?**

  → 개발자: API 구조 파악 및 구현

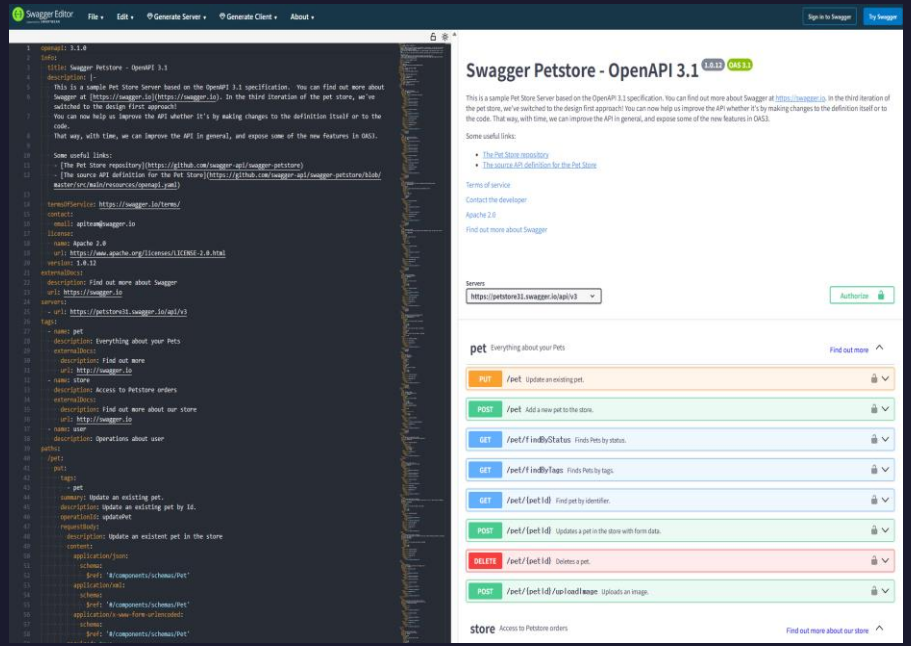
  → 프론트엔드: Mock 서버로 선행 개발

  → QA: 테스트 케이스 자동 생성

  → 도구: Swagger UI가 자동으로 인터랙티브 문서 생성

Swagger Editor 소개

- **Swagger Editor = 웹에서 바로 API를 설계하는 도구**
  - → https://editor.swagger.io
- **좌측: YAML 코드 작성 영역**
- **우측: 실시간 미리보기 (인터랙티브 문서)**
- **기능**
  - → 실시간 문법 검증 — 오류 즉시 표시
  - → Try it out — 설계한 API를 바로 테스트
  - → 코드 생성 — 서버/클라이언트 코드 자동 생성
- **설치 없이 브라우저에서 바로 사용 가능!**



OpenAPI 기본 구조

YAML

```yaml
openapi: 3.0.3            # OpenAPI 버전
info:                     # API 정보
  title: 스타벅스 API
  description: 스타벅스 메뉴 조회 및 주문 API
  version: 1.0.0

servers:                  # 서버 주소
  - url: http://localhost:3000
    description: 개발 서버

paths:                    # API 엔드포인트 목록
  /api/menus:             # 경로
    get:                  # HTTP 메서드
      summary: 메뉴 목록 조회
      description: 전체 메뉴를 조회합니다
      responses:
        '200':
          description: 성공
          content:
            application/json:
              schema:
                type: array
                items:
                  $ref: '#/components/schemas/Menu'
```

응답 스키마(Schema) 정의

YAML

```yaml
components:
  schemas:
    Menu:                        # 메뉴 데이터 구조
      type: object
      properties:
        id:
          type: integer
          example: 1
        name:
          type: string
          example: "아메리카노"
        category:
          type: string
          example: "에스프레소"
          enum: [에스프레소, 프라푸치노, 티, 주스]
        price:
          type: integer
          example: 4500
        sizes:
          type: array
          items:
            type: string
          example: ["tall", "grande", "venti"]
        available:
          type: boolean
          example: true
```

Section 4

[실습 1] 스타벅스 메뉴 조회 API 설계

Swagger Editor로 직접 만들어보기

HANDS-ON

실습 1 준비: Swagger Editor 접속

1. 브라우저에서 https://editor.swagger.io 접속
2. 기존 내용 모두 삭제 (File → Clear Editor)
3. 새 API 명세 시작 준비
4. 좌측에 YAML 코드 작성 → 우측에 미리보기 확인
5. 오류가 있으면 빨간색으로 표시됨

실습 1-1: 기본 정보 작성

YAML

```yaml
# Step 1: 기본 정보 작성
openapi: 3.0.3
info:
  title: 스타벅스 메뉴 API
  description: |
    스타벅스 매장의 메뉴를 조회하는 API입니다.
    음료, 푸드, 상품 카테고리를 지원합니다.
  version: 1.0.0
  contact:
    name: 교육생 이름
    email: student@example.com

servers:
  - url: http://localhost:3000
    description: 로컬 개발 서버
  - url: https://api.starbucks-demo.com
    description: 테스트 서버

# ✅ 우측 미리보기에서 API 제목이 보이면 성공!
```

실습 1-2: 전체 메뉴 조회 (GET /menus)

YAML

```yaml
paths:
  /api/menus:
    get:
      tags:
        - 메뉴
      summary: 전체 메뉴 목록 조회
      description: 스타벅스의 전체 메뉴를 조회합니다
      parameters:
        - name: category
          in: query
          description: 카테고리 필터
          schema:
            type: string
            enum: [에스프레소, 프라푸치노, 티, 푸드]
      responses:
        '200':
          description: 메뉴 목록 반환 성공
          content:
            application/json:
              schema:
                type: array
                items:
                  type: object
                  properties:
                    id: { type: integer, example: 1 }
                    name: { type: string, example: "아메리카노" }
                    price: { type: integer, example: 4500 }
                    category: { type: string, example: "에스프레소" }
```

실습 1-3: 특정 메뉴 조회 (GET /menus/{id})

YAML

```yaml
/api/menus/{id}:
  get:
    tags:
      - 메뉴
    summary: 특정 메뉴 상세 조회
    description: 메뉴 ID로 상세 정보를 조회합니다
    parameters:
      - name: id
        in: path
        required: true
        description: 메뉴 고유 ID
        schema:
          type: integer
          example: 1
    responses:
      '200':
        description: 메뉴 상세 정보
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/MenuDetail'
      '404':
        description: 메뉴를 찾을 수 없음
        content:
          application/json:
            schema:
              type: object
              properties:
                error: { type: string, example: "Menu not found" }
```

실습 1-4: 메뉴 상세 스키마 정의

YAML

```yaml
components:
  schemas:
    MenuDetail:
      type: object
      required: [id, name, price, category]
      properties:
        id:
          type: integer
          example: 1
        name:
          type: string
          example: "카페 아메리카노"
        description:
          type: string
          example: "진한 에스프레소에 뜨거운 물을 더해 깔끔한 맛"
        category:
          type: string
          enum: [에스프레소, 프라푸치노, 티, 푸드]
        price:
          type: integer
          example: 4500
        sizes:
          type: array
          items:
            type: object
            properties:
              size: { type: string, example: "tall" }
              ml: { type: integer, example: 355 }
              price: { type: integer, example: 4500 }
        available:
          type: boolean
          example: true
```

HANDS-ON

실습 1 체크포인트

1. 우측 미리보기에서 'GET /api/menus' 확인
2. 우측 미리보기에서 'GET /api/menus/{id}' 확인
3. 'Try it out' 버튼이 보이는지 확인
4. Schema 섹션에서 MenuDetail이 보이는지 확인
5. 에러 표시(빨간색)가 없는지 확인
6. 문제가 있으면 손을 들어주세요!

실습 1-5: 응답 예시(Example) 추가

YAML

```yaml
# responses의 content 하위에 example 추가

responses:
  '200':
    description: 메뉴 목록 반환 성공
    content:
      application/json:
        schema:
          type: array
          items:
            $ref: '#/components/schemas/MenuDetail'
        example:
          - id: 1
            name: "카페 아메리카노"
            category: "에스프레소"
            price: 4500
            available: true
          - id: 2
            name: "카페 라떼"
            category: "에스프레소"
            price: 5000
            available: true
          - id: 3
            name: "자바 칩 프라푸치노"
            category: "프라푸치노"
            price: 6300
            available: false

# ✅ 미리보기에서 Example Value가 보이면 성공!
```

Section 5

[실습 2] 회원가입 & 주문 API 설계

수강생 개별 실습

실습 2-1: 회원가입 API (POST /users)

YAML

```yaml
/api/users:
  post:
    tags:
      - 회원
    summary: 회원가입
    description: 새 사용자를 등록합니다
    requestBody:
      required: true
      content:
        application/json:
          schema:
            type: object
            required: [name, email, password]
            properties:
              name:
                type: string
                minLength: 2
                maxLength: 20
                example: "홍길동"
              email:
                type: string
                format: email
                example: "hong@example.com"
              password:
                type: string
                minLength: 8
                example: "MyPass123!"
              phone:
                type: string
                example: "010-1234-5678"
    responses:
      '201':
        description: 회원가입 성공
      '400':
        description: 입력값 오류
```

실습 2-2: 로그인 API (POST /auth/login)

YAML

```yaml
/api/auth/login:
  post:
    tags:
      - 인증
    summary: 로그인
    description: 이메일과 비밀번호로 로그인합니다
    requestBody:
      required: true
      content:
        application/json:
          schema:
            type: object
            required: [email, password]
            properties:
              email:
                type: string
                format: email
                example: "hong@example.com"
              password:
                type: string
                example: "MyPass123!"
```

```yaml
/api/auth/login:
    responses:
      '200':
        description: 로그인 성공
        content:
          application/json:
            schema:
              type: object
              properties:
                token:
                  type: string
                  example: "eyJhbGciOiJIUzI1NiIs..."
                user:
                  type: object
                  properties:
                    id: { type: integer }
                    name: { type: string }
      '401':
        description: 인증 실패
```

실습 2-3: 주문 생성 API (POST /orders)

YAML

```yaml
/api/orders:
  post:
    tags:
      - 주문
    summary: 새 주문 생성
    security:
      - BearerAuth: []      # 로그인 필요!
    requestBody:
      required: true
      content:
        application/json:
          schema:
            type: object
            required: [items]
            properties:
              items:
                type: array
                items:
                  type: object
                  properties:
                    menuId: { type: integer, example: 1 }
                    size: { type: string, example: "grande" }
                    quantity: { type: integer, example: 2 }
                    options:
                      type: array
                      items:
                        type: string
                      example: ["얼음 적게", "시럽 추가"]
```

```yaml
/api/orders:
    responses:
      '201':
        description: 주문 성공
      '401':
        description: 로그인 필요
```

실습 2-4: 주문 조회 & 취소 API

YAML

```yaml
/api/orders/{id}:
  get:
    tags:
      - 주문
    summary: 주문 상세 조회
    security:
      - BearerAuth: []
    parameters:
      - name: id
        in: path
        required: true
        schema:
          type: integer
    responses:
      '200':
        description: 주문 상세 정보
      '404':
        description: 주문을 찾을 수 없음

  delete:
    tags:
      - 주문
    summary: 주문 취소
    security:
      - BearerAuth: []
    parameters:
      - name: id
        in: path
        required: true
        schema:
          type: integer
```

```yaml
/api/orders/{id}:
    responses:
      '204':
        description: 주문 취소 성공
      '404':
        description: 주문을 찾을 수 없음
```

실습 2-5: 인증 스키마 추가

YAML

```yaml
# components에 보안 스키마 추가
components:
  securitySchemes:
    BearerAuth:
      type: http
      scheme: bearer
      bearerFormat: JWT
      description: "로그인 후 발급받은 JWT 토큰"

# ✅ 전체 API 설계 완성 체크:
# - GET  /api/menus          → 메뉴 목록 조회
# - GET  /api/menus/{id}     → 메뉴 상세 조회
# - POST /api/users          → 회원가입
# - POST /api/auth/login     → 로그인
# - POST /api/orders         → 주문 생성 (인증 필요)
# - GET  /api/orders/{id}    → 주문 조회 (인증 필요)
# - DELETE /api/orders/{id}  → 주문 취소 (인증 필요)

# 🎉 Swagger Editor에서 7개 엔드포인트가 보이면 완성!
```

HANDS-ON

실습 2 개별 도전 과제

1. 메뉴 검색 API 추가: GET /api/menus/search?q=아메리카노
2. 주문 내역 목록 API: GET /api/orders (내 주문 전체 조회)
3. 메뉴 즐겨찾기 API: POST /api/favorites
4. 리뷰 작성 API: POST /api/menus/{id}/reviews
5. 각자 1~2개 추가 엔드포인트 설계해보기 (15분)
6. 완성 후 옆 사람과 설계도 비교 & 피드백

HANDS-ON

실습 2 체크포인트 & 제출

1. Swagger Editor에서 모든 엔드포인트 에러 없는지 확인
2. 각 응답에 Example이 포함되어 있는지 확인
3. 인증이 필요한 API에 security 설정이 있는지 확인
4. Schema에 required 필드가 정의되어 있는지 확인
5. File → Save as YAML로 파일 저장
6. 저장한 파일을 강사에게 제출

Section 6

1일차 요약 & Q&A

1일차 핵심 요약

✓ API = 프로그램 간 약속된 대화 규칙 (자판기 비유)

✓ 프론트/백 분리 개발 → API가 소통의 다리

✓ API-First Design: 설계도 먼저 → 코딩은 나중에

✓ JSON: key-value 구조, API의 데이터 포맷

✓ YAML: 들여쓰기 기반, OpenAPI 명세에 사용

✓ HTTP 메서드: GET(읽기), POST(생성), PUT(수정), DELETE(삭제)

✓ 상태 코드: 200(성공), 404(없음), 500(서버에러)

✓ OpenAPI(표준) + Swagger Editor(도구)로 API 설계도 작성 — Swagger는 OpenAPI를 만들기 위한 도구!

"설계도가 없으면 코딩은 노동일 뿐이다."

오늘 배운 것: API의 언어(JSON/HTTP)와 설계 도구(Swagger)

내일 예고: 2일차 — 거인의 어깨 위에 올라타기 (오픈소스 활용)

Q & A

"설계도가 없으면 코딩은 노동일 뿐이다."

내일 예고: 2일차 - 오픈소스 활용 (Build)

S-개발자 4기 교육

API 실생활 사례 모음

- **카카오 로그인 API: 다른 앱에서 카카오 계정으로 로그인**

  → OAuth 2.0 기반 — 내일 배울 인증 방식

- **네이버 지도 API: 앱에 지도 기능 삽입**

  → 좌표 → 주소 변환, 경로 탐색 등

- **날씨 API (OpenWeatherMap): 도시명 → 현재 날씨**

  → 무료 API Key 발급 → 바로 사용 가능

- **결제 API (토스페이먼츠): 결제 처리 자동화**

  → 복잡한 결제 로직을 API 호출 한 번으로 처리

좋은 API 설계의 7가지 원칙

- **1. 직관적인 URL: 명사 사용, 복수형 권장**

  → ✓ /api/users/123 ✗ /api/getUser?id=123
- **2. HTTP 메서드로 행위 표현 (URL에 동사 금지)**
- **3. 일관된 응답 형식: { data, error, meta }**
- **4. 적절한 상태 코드 반환 (200, 201, 400, 404)**
- **5. 버전 관리: /api/v1/menus, /api/v2/menus**
- **6. 페이지네이션: ?page=1&limit=20**
- **7. 에러 메시지는 친절하게**

  → { "error": "email 형식이 올바르지 않습니다" }

REST API URL 네이밍 규칙

- **리소스는 복수형 명사 사용**

  → ✓ /api/users ✗ /api/user

- **계층 관계는 슬래시(/)로 표현**

  → ✓ /api/users/123/orders (123번 사용자의 주문들)

- **소문자 + 하이픈(-) 사용 (카멜케이스, 언더스코어 지양)**

  → ✓ /api/order-items ✗ /api/orderItems

- **동사 대신 HTTP 메서드 활용**

  → ✓ DELETE /api/orders/42 ✗ POST /api/deleteOrder

- **쿼리 파라미터로 필터링/정렬/검색**

  → ✓ /api/menus?category=에스프레소&sort=price

페이지네이션 & 필터링 설계

YAML

```yaml
# 페이지네이션 파라미터 설계
paths:
  /api/menus:
    get:
      parameters:
        - name: page
          in: query
          description: 페이지 번호 (1부터 시작)
          schema:
            type: integer
            minimum: 1
            default: 1
        - name: limit
          in: query
          description: 한 페이지당 항목 수
          schema:
            type: integer
            minimum: 1
            maximum: 100
            default: 20
        - name: category
          in: query
          description: 카테고리 필터
          schema:
            type: string
            enum: [에스프레소, 프라푸치노, 티, 푸드]
        - name: sort
          in: query
          description: 정렬 기준
          schema:
            type: string
            enum: [price_asc, price_desc, name_asc]

# 응답에 메타 정보 포함
# { "data": [...], "meta": { "total": 50, "page": 1 } }
```

에러 응답 설계 Best Practice

- **에러 응답도 일관된 포맷으로 설계**

  → { "error": { "code": "VALIDATION_ERROR", "message": "...", "details": [...] } }

- **사용자 친화적 메시지 + 개발자용 코드 분리**

  → 사용자: '이메일 형식이 올바르지 않습니다'

  → 개발자: code: 'INVALID_EMAIL_FORMAT'

- **상태 코드를 적절히 활용**

  → 400: 입력 오류, 401: 인증 필요, 403: 권한 없음

  → 404: 없음, 409: 충돌, 429: 요청 제한 초과

API 설계 DO vs DON'T

DO (좋은 설계)

- GET /api/users/123
- POST /api/orders
- DELETE /api/orders/42
- /api/v1/menus?page=1
- 복수형 명사 사용
- 일관된 에러 포맷

DON'T (나쁜 설계)

- GET /api/getUser?id=123
- POST /api/createOrder
- POST /api/deleteOrder/42
- /api/menus (버전 없음)
- 동사+명사 혼용
- 에러마다 다른 포맷

내일 배울 내용 미리보기: 오픈소스

- **오픈소스란? — 전 세계 개발자가 공유하는 코드**
- **npm (Node.js 패키지 매니저) 사용법**
- **Express.js 프레임워크로 API 서버 구축**
- **오픈소스 선택 기준: Star, Fork, 최근 커밋**
- **라이선스 주의사항: MIT vs GPL**
- **1일차 설계도를 실제 코드로 구현!**

→ 오늘의 Swagger 설계도 → 내일 동작하는 서버로 변신

OpenAPI 3.0 핵심 구조 정리

- **openapi: 버전 정보 (3.0.3)**
- **info: API 제목, 설명, 버전, 연락처**
- **servers: API 서버 URL 목록**
- **paths: 엔드포인트 경로 + HTTP 메서드 + 요청/응답 정의**
- **components:**
  - → schemas: 데이터 구조 정의 (재사용 가능)
  - → securitySchemes: 인증 방식 정의
  - → responses: 공통 응답 정의 (재사용 가능)
- **$ref: 다른 곳에서 정의한 스키마 참조**
  - → '$ref': '#/components/schemas/Menu'

API 인증 방식 비교 — API Key vs JWT vs OAuth 2.0

- **API Key — 가장 단순한 인증**
  - → 서버가 발급한 고정 키를 헤더에 포함: X-API-Key: abc123
  - → 장점: 구현 쉬움 / 단점: 탈취 시 권한 전체 노출
- **JWT (JSON Web Token) — 실무 표준**
  - → 로그인 → 서버가 JWT 발급 → 이후 요청마다 Bearer 헤더에 포함
  - → 구조: Header.Payload.Signature (Base64 인코딩)
  - → 장점: 서버 DB 조회 없이 토큰 자체로 검증 가능 (Stateless)
- **OAuth 2.0 — 위임 인증의 표준**
  - → '카카오로 로그인' 같은 제3자 인증 위임 프로토콜
  - → Access Token + Refresh Token 구조로 보안 강화

★ 이 강좌 실습에서는 JWT (BearerAuth) 방식 사용!

API 테스트 도구 — Postman & curl

- **Postman — GUI 기반 API 테스트 도구**
  - → https://www.postman.com (무료 사용 가능)
  - → 주요 기능: HTTP 요청 전송, 응답 확인, 테스트 자동화, Collection 관리
  - → OpenAPI YAML 파일을 Postman에 임포트하면 Collection 자동 생성!
- **curl — CLI 기반 HTTP 요청 도구**
  - → curl -X GET https://api.example.com/api/menus
  - → curl -X POST -H "Content-Type: application/json" ₩
    -d '{"email":"a@b.com"}' https://api.example.com/api/login
- **Swagger UI 내장 'Try it out' 기능**
  - → OpenAPI 명세 작성 후 Swagger UI에서 바로 API 호출 테스트 가능

★ 실무 팁: Postman Collection을 팀 전체와 공유하면 테스트 비용 감소

Mock 서버 활용 — 백엔드 없이 개발하기

- **Mock 서버란?**
  - → 실제 서버 없이 OpenAPI 명세만으로 가짜 응답을 돌려주는 서버
  - → 프론트엔드가 백엔드 완성 전에 선행 개발 가능 (API-First의 핵심!)
- **Prism — 가장 유명한 OpenAPI Mock 서버**
  - → npx @stoplight/prism-cli mock openapi.yaml
  - → 실행 즉시 localhost:4010 에서 모든 엔드포인트 응답
  - → example 값이 있으면 그대로, 없으면 스키마 기반으로 자동 생성
- **Mockoon — GUI 기반 Mock 서버 도구**
  - → https://mockoon.com — 설치형, 팀 공유 기능 제공
- **Mock 서버 활용 흐름**
  - → OpenAPI 명세 완성 → Prism Mock 서버 → 프론트 개발 → 백엔드 연결

API 패러다임 비교 — REST vs GraphQL vs gRPC

- **REST API — 현재 업계 표준 (이 강좌에서 배우는 방식)**
  - → URL + HTTP 메서드 조합으로 리소스 표현
  - → 장점: 단순, 범용 / 단점: Over-fetching (필요 이상 데이터 수신)
- **GraphQL — 클라이언트가 원하는 데이터만 요청**
  - → 단일 엔드포인트 /graphql 에서 쿼리로 원하는 필드만 지정
  - → Meta(Facebook) 개발, GitHub / Twitter API에서 사용
  - → 장점: 유연한 데이터 요청 / 단점: 복잡한 캐싱, 학습 비용
- **gRPC — Google이 만든 고성능 RPC 프레임워크**
  - → Protocol Buffers(바이너리) 사용 → REST보다 10배 빠름
  - → 마이크로서비스 내부 통신에 주로 사용 (외부 API엔 잘 안 씀)

★ 일반 웹/앱 서비스는 REST, 복잡한 데이터 UI는 GraphQL 고려

API 설계 실무 체크리스트

☐ **URL 설계**

→ 복수형 명사 사용 /api/users / 동사 금지 /api/getUser ✗

☐ **HTTP 메서드 & 상태코드**

→ CRUD → POST/GET/PUT/DELETE 매핑 / 201 Created 정확히 반환

☐ **요청/응답 설계**

→ 일관된 응답 포맷: { data, error, meta } / required 필드 명시

☐ **인증 & 보안**

→ 민감 데이터는 Body 전송 / URL에 token, password 절대 금지

☐ **문서화**

→ OpenAPI 명세 작성 / example 값 반드시 포함 / 에러 케이스 정의

☐ **버전 관리 → /api/v1/ 로 시작하는 습관 들이기!**

OpenAPI 생태계 & 다음 단계 로드맵

- **OpenAPI 기반 주요 도구 생태계**
  - → 설계: Swagger Editor / Stoplight Studio / Insomnia
  - → 문서화: Swagger UI / Redoc / Scalar
  - → 코드 생성: OpenAPI Generator (50+ 언어/프레임워크 지원)
  - → 테스트: Prism (Mock) / Dredd / Schemathesis
- **2일차 예고 — 오픈소스 활용 (Build)**
  - → 오늘의 OpenAPI 명세 → Express.js 서버로 실제 구현
  - → npm 생태계 / 오픈소스 라이선스 / 의존성 관리
- **5일 과정 전체 흐름**
  - → Day1 설계 → Day2 구현 → Day3 DB → Day4 배포 → Day5 보안

★ 오늘 작성한 Swagger YAML이 5일 과정의 설계도가 됩니다!