KISIA S-개발자 4기

2일차: 효율적인 개발과 오픈소스 프레임워크

거인의 어깨 위에 올라타기: 오픈소스 활용 (Build)

Day 2 of 5

Day 1 **Day 2** Day 3 Day 4 Day 5

2일차 학습 목표

좋은 오픈소스를 판별하는 눈을 기르자

GitHub Star, Fork, 라이선스, 커밋 활성도 등 핵심 지표 학습

패키지 매니저를 자유자재로 사용하자

npm, pip 등을 활용한 의존성 관리 방법

나만의 API 서버를 직접 구동하자

Express.js 기반 Hello World 서버부터 JSON API까지

1일차 설계도를 실제 코드로 구현하자

RESTful 엔드포인트를 실제 동작하는 서버로 변환

2일차 교육 일정

09:00-09:30	1일차 복습 및 오늘 학습 목표 안내
09:30-10:00	오픈소스의 힘과 패키지 매니저
10:00-10:30	오픈소스 선택 기준: 인기도와 신뢰도
10:30-10:40	🧉 쉬는 시간
10:40-11:10	라이선스와 프레임워크 이해
11:10-12:00	[실습 3] 헬로 월드 서버 띄우기
12:00-13:00	🍙 점심시간
13:00-14:30	[실습 4] 1일차 설계도를 코드로 구현
14:30-14:40	🧉 쉬는 시간
14:40-15:20	API 문서 자동화 (Swagger)
15:20-16:00	2일차 요약 및 Q&A

1일차 복습: API 설계의 기본

- **API = 소프트웨어 간의 약속된 대화 방법**

  → REST: 자원(URL) + 행위(HTTP Method) + 표현(JSON)

- **HTTP 메서드: GET(조회), POST(생성), PUT(수정), DELETE(삭제)**

  → 상태 코드: 200(성공), 201(생성), 404(없음), 500(서버에러)

- **1일차에서 우리는 '설계도'를 만들었다**

  → 오늘은 이 설계도를 '실제 코드'로 구현한다!

Section 1

오픈소스의 세계

전 세계 코드의 90%는 이미 공개되어 있다

오픈소스의 힘

- **전 세계 소프트웨어의 90% 이상이 오픈소스 컴포넌트 포함**

  → GitHub에 공개된 저장소: 3억 개 이상

- **혼자 모든 것을 만들 필요가 없다!**

  → 바퀴를 재발명하지 말라 (Don't Reinvent the Wheel)

- **오픈소스 = 수천 명의 개발자가 함께 만든 검증된 코드**

  → Linux, React, Node.js, Python 모두 오픈소스

- **우리가 오늘 사용할 Express.js도 오픈소스!**

오픈소스 성공 사례

웹 프레임워크

- Express.js - 6,400만 주간 다운로드
- React - Meta가 만든 UI 라이브러리
- Django - Python 웹 프레임워크
- Spring Boot - Java 엔터프라이즈
- FastAPI - 초고속 Python API

인프라/도구

- Linux - 서버의 96% 사용
- Docker - 컨테이너 혁명
- Kubernetes - 오케스트레이션
- VS Code - 에디터 1위
- Git - 버전 관리의 표준

패키지 매니저의 이해

- **패키지 매니저 = 오픈소스 설치/관리 도구**

  → 앱스토어처럼 명령어 하나로 라이브러리 설치

- **npm (Node Package Manager) - JavaScript/Node.js**

  → npm install express ← 이 한 줄이면 Express 설치 완료

- **pip - Python 패키지 매니저**

  → pip install fastapi ← Python에서는 이렇게

- **Maven/Gradle - Java, NuGet - C#, Cargo - Rust**

npm 핵심 명령어

bash

```
# 프로젝트 초기화 (package.json 생성)
npm init -y

# 패키지 설치 (node_modules 폴더에 저장)
npm install express

# 개발용 패키지 설치 (배포 시 미포함)
npm install --save-dev nodemon

# 설치된 패키지 목록 확인
npm list

# package.json의 모든 의존성 한번에 설치
npm install

# 스크립트 실행 (package.json에 정의된)
npm run start
```

package.json - 프로젝트의 신분증

json

```
{
  "name": "my-api-server",
  "version": "1.0.0",
  "description": "나의 첫 API 서버",
  "main": "index.js",
  "scripts": {
    "start": "node index.js",
    "dev": "nodemon index.js"
  },
  "dependencies": {
    "express": "^4.18.2"     ← 실행에 필요한 패키지
  },
  "devDependencies": {
    "nodemon": "^3.0.0"      ← 개발 중에만 필요한 패키지
  }
}
```

Section 2

오픈소스 선택 기준

좋은 오픈소스를 고르는 5가지 기준

선택 기준 ① 인기도 - GitHub Star & Fork

- ⭐ **Star 수 = 얼마나 많은 개발자가 관심을 가졌는가**

  → Express.js: ★ 63,000+ / React: ★ 220,000+

- 🍴 **Fork 수 = 얼마나 많은 개발자가 기여/활용하는가**

  → Fork가 많을수록 커뮤니티가 활발한 증거

- 📦 **주간 다운로드 수 (npmjs.com에서 확인)**

  → Express: 주간 6,400만 다운로드 = 신뢰할 수 있는 수준

- **비교: Star 100개 미만 라이브러리는 신중하게 선택**

선택 기준 ② 신뢰도 - 커밋 & 이슈 활성도

- **마지막 커밋 날짜가 중요하다!**

  → 최근 3개월 내 커밋이 있는가? → 유지보수 중

  → 1년 이상 커밋 없음 → 사실상 폐기된 프로젝트

- **Issue 답변 속도와 해결률**

  → 이슈 등록 후 24시간 내 응답 → 우수한 프로젝트

  → Open Issue가 1,000개 이상인데 답변 없음 → 위험 신호

- **기여자(Contributors) 수**

  → 1인 프로젝트 vs 100인 이상 → 지속 가능성 차이

선택 기준 ③ 문서화 수준

- **README.md가 잘 작성되어 있는가?**

  → 설치 방법, 빠른 시작 가이드, 예제 코드 포함 여부

- **공식 문서 사이트가 있는가?**

  → Express: expressjs.com - 체계적인 문서

  → FastAPI: fastapi.tiangolo.com - 대화형 문서

- **변경 이력(CHANGELOG)이 관리되는가?**

  → 버전별 변경사항을 투명하게 공개하는 프로젝트 = 신뢰

선택 기준 ④ 라이선스 주의사항

- **MIT License (가장 자유로운 라이선스)**

  → 자유롭게 사용, 수정, 배포 가능 → Express.js가 MIT

- **Apache 2.0 (기업 친화적)**

  → 특허 관련 보호 조항 포함 → 기업에서 선호

- **GPL (GNU General Public License) ⚠️ 주의!**

  → GPL 코드 사용 시 내 코드도 GPL로 공개해야 함!

  → 상업적 프로젝트에서 GPL 라이브러리 사용 → 법적 위험

- **ISC, BSD → MIT와 유사, 자유롭게 사용 가능**

라이선스 비교: MIT vs GPL

MIT License ✅ 안전

- 자유롭게 사용 가능
- 수정 후 비공개 배포 OK
- 상업적 사용 완전 자유
- Express.js, React, jQuery
- → 대부분의 프로젝트에 적합

GPL License ⚠️ 주의

- 사용은 자유로우나...
- 수정 시 소스코드 공개 의무!
- 상업적 사용 시 코드 공개
- Linux Kernel, WordPress
- → 사내 프로젝트에 부적합할 수 있음

선택 기준 ⑤ 보안 취약점 확인

- **npm audit - 설치된 패키지의 보안 취약점 검사**

  → npm audit 명령어로 취약점 자동 스캔

- **Snyk, Dependabot 등 보안 모니터링 도구**

  → GitHub에서 자동으로 보안 알림을 보내줌

- **알려진 취약점(CVE)이 있는 버전 피하기**

  → 최신 안정 버전 사용이 보안의 기본

- **node_modules 폴더는 절대 Git에 올리지 않는다!**

  → .gitignore에 반드시 추가

Section 3

프레임워크 vs 라이브러리

주도권의 차이를 이해하자

프레임워크 vs 라이브러리

프레임워크 (Framework)

- 프레임워크가 나를 호출한다
- 정해진 구조와 규칙을 따른다
- 뼈대를 제공 → 내가 살을 붙임
- 예: Express.js, Django, Spring
- → app.get('/', handler)

라이브러리 (Library)

- 내가 라이브러리를 호출한다
- 자유로운 구조, 필요할 때 사용
- 도구를 제공 → 내가 조립
- 예: Lodash, Axios, Moment
- → axios.get(url)

현대 API 프레임워크 비교

Express.js (Node.js)

- 가장 인기 있는 Node.js 프레임워크
- 미니멀리즘: 핵심만 제공
- 미들웨어 기반 확장
- ★ 63,000+ / 주간 6,400만 DL
- 학습 곡선이 완만함 → 오늘의 선택!

FastAPI (Python)

- 가장 빠른 Python API 프레임워크
- 자동 문서 생성 (Swagger)
- 타입 힌트 기반 검증
- ★ 72,000+ / 급성장 중
- Python 선호 시 최고의 선택

오늘 Express.js를 선택하는 이유

- **1일차에 배운 REST API 설계를 바로 구현 가능**

  → HTTP 메서드와 라우팅이 직관적

- **JavaScript 하나로 프론트엔드 + 백엔드 가능**

  → 풀스택 개발의 기초 → 가장 범용적인 언어

- **NPM 생태계: 200만 개 이상의 패키지**

  → 필요한 기능 대부분이 이미 패키지로 존재

- **실무에서 가장 많이 사용되는 조합**

  → Node.js + Express = 전 세계 API 서버의 표준

Section 4

개발 환경 구축

코드를 작성할 준비를 하자

환경 구축 ① VS Code 설치

- **Visual Studio Code = 전 세계 개발자 1위 에디터**

  → https://code.visualstudio.com 에서 다운로드

- **추천 확장 프로그램 (Extensions)**

  → ESLint - JavaScript 코드 품질 검사

  → Prettier - 코드 자동 포맷팅

  → REST Client - VS Code에서 API 테스트

  → Thunder Client - Postman 대안 (VS Code 내장)

환경 구축 ② Node.js 설치

- **Node.js = JavaScript를 서버에서 실행하는 런타임**

  → https://nodejs.org → LTS 버전 다운로드

- **설치 확인 명령어:**

  → node --version → v20.x.x 이상 권장

  → npm --version → 10.x.x 이상 권장

- **Node.js를 설치하면 npm이 자동으로 포함됨**

  → npm = Node Package Manager (패키지 설치 도구)

환경 구축 ③ 프로젝트 폴더 생성

bash

```
# 1. 바탕화면에 프로젝트 폴더 만들기
mkdir my-api-server
cd my-api-server

# 2. npm 프로젝트 초기화 (package.json 생성)
npm init -y

# 3. Express.js 설치
npm install express

# 4. 설치 확인
npm list

# 결과:
# my-api-server@1.0.0
# └── express@4.18.2
```

Section 5

[실습 3] 헬로 월드 서버 띄우기

첫 API 엔드포인트를 만들어 보자!

HANDS-ON

[실습 3] 목표: 첫 번째 API 서버 구동

1. Express.js를 사용해 웹 서버를 만든다
2. GET / 요청에 'Hello World!' 응답
3. 브라우저에서 http://localhost:3000 접속 확인
4. 다양한 라우트(경로)를 추가해 본다
5. JSON 형식으로 응답하는 방법을 배운다
6. 서버 재시작 없이 코드 변경 반영 (nodemon)

[실습 3-1] 첫 Express 서버 코드

javascript

```
// index.js - 첫 번째 Express 서버

const express = require('express');  // Express 불러오기
const app = express();               // Express 앱 생성
const PORT = 3000;                   // 포트 번호 설정

// GET / 요청 처리
app.get('/', (req, res) => {
  res.send('Hello World! 🎉');
});

// 서버 시작
app.listen(PORT, () => {
  console.log(`서버가 http://localhost:${PORT} 에서 실행 중`);
});
```

[실습 3-2] 서버 실행하기

bash

```
# 터미널에서 실행
node index.js

# 출력:
# 서버가 http://localhost:3000 에서 실행 중

# 브라우저를 열고 접속:
# http://localhost:3000
# → "Hello World! 🎉" 표시됨!

# 서버 종료: Ctrl + C

# 코드 수정할 때마다 재시작해야 하는 불편함...
# → 해결책: nodemon!
```

[실습 3-3] nodemon으로 자동 재시작

bash

```
# nodemon 설치 (개발용)
npm install --save-dev nodemon

# package.json의 scripts 수정:
# "scripts": {
#   "start": "node index.js",
#   "dev": "nodemon index.js"    ← 이 줄 추가
# }

# nodemon으로 실행 (코드 변경 시 자동 재시작)
npm run dev

# 출력:
# [nodemon] watching path(s): *.*
# [nodemon] starting `node index.js`
# 서버가 http://localhost:3000 에서 실행 중
```

[실습 3-4] 코드 구조 한 줄씩 이해하기

- **const express = require('express')**

  → node_modules에서 express 패키지를 불러온다

- **const app = express()**

  → Express 애플리케이션 인스턴스를 생성한다

- **app.get('/', (req, res) => { ... })**

  → GET 메서드 + '/' 경로로 요청이 오면 실행

- **res.send('Hello World!')**

  → 클라이언트에게 텍스트 응답을 보낸다

- **app.listen(3000, callback)**

  → 3000번 포트에서 요청을 기다린다 (서버 시작)

[실습 3-5] 여러 개의 라우트 추가하기

javascript

```
// GET /about - 소개 페이지
app.get('/about', (req, res) => {
  res.send('이것은 나의 첫 API 서버입니다!');
});

// GET /time - 현재 시간 반환
app.get('/time', (req, res) => {
  const now = new Date().toLocaleString('ko-KR');
  res.send(`현재 시간: ${now}`);
});

// GET /random - 랜덤 숫자 반환
app.get('/random', (req, res) => {
  const num = Math.floor(Math.random() * 100);
  res.send(`랜덤 숫자: ${num}`);
});
```

[실습 3-6] JSON 형식으로 응답하기

javascript

```
// res.send() → 텍스트 응답
// res.json() → JSON 응답  ← API에서는 이것을 사용!

// GET /api/status - 서버 상태 (JSON)
app.get('/api/status', (req, res) => {
  res.json({
    status: 'running',
    server: 'my-api-server',
    version: '1.0.0',
    timestamp: new Date().toISOString()
  });
});

// 응답 예시:
// {
//   "status": "running",
//   "server": "my-api-server",
//   "version": "1.0.0",
//   "timestamp": "2024-01-15T09:30:00.000Z"
// }
```

[실습 3-7] HTTP 상태 코드 설정하기

javascript

```
// 200 OK (기본값)
app.get('/api/health', (req, res) => {
  res.status(200).json({ message: '서버 정상' });
});

// 201 Created
app.post('/api/data', (req, res) => {
  res.status(201).json({ message: '데이터 생성됨' });
});

// 404 Not Found
app.get('/api/notfound', (req, res) => {
  res.status(404).json({ error: '리소스를 찾을 수 없습니다' });
});

// 모든 정의되지 않은 경로 처리 (맨 마지막에 추가)
app.use('*', (req, res) => {
  res.status(404).json({ error: '존재하지 않는 경로입니다' });
});
```

[실습 3-8] 쿼리 파라미터 받기

javascript

```
// GET /api/greet?name=홍길동
app.get('/api/greet', (req, res) => {
  const name = req.query.name || '익명';
  res.json({
    message: `안녕하세요, ${name}님!`,
    timestamp: new Date().toISOString()
  });
});

// 테스트:
// http://localhost:3000/api/greet?name=홍길동
// → { "message": "안녕하세요, 홍길동님!", ... }

// http://localhost:3000/api/greet
// → { "message": "안녕하세요, 익명님!", ... }
```

[실습 3-9] URL 파라미터 (경로 변수)

javascript

```
// GET /api/users/:id  → :id 가 변수!
app.get('/api/users/:id', (req, res) => {
  const userId = req.params.id;
  res.json({
    userId: userId,
    name: `사용자 ${userId}`,
    role: 'member'
  });
});

// 테스트:
// http://localhost:3000/api/users/42
// → { "userId": "42", "name": "사용자 42", "role": "member" }

// http://localhost:3000/api/users/100
// → { "userId": "100", "name": "사용자 100", "role": "member" }
```

[실습 3-10] 완성된 index.js 전체 코드

javascript

```
const express = require('express');
const app = express();
const PORT = 3000;

app.get('/',          (req, res) => res.send('Hello World!'));
app.get('/about',     (req, res) => res.send('My First API Server'));
app.get('/api/status', (req, res) => res.json({ status: 'ok' }));
app.get('/api/time',   (req, res) => res.json({
  time: new Date().toLocaleString('ko-KR')
}));
app.get('/api/greet',  (req, res) => {
  const name = req.query.name || 'World';
  res.json({ message: `Hello, ${name}!` });
});
app.get('/api/users/:id', (req, res) => {
  res.json({ userId: req.params.id, name: `User ${req.params.id}` });
});
app.use('*', (req, res) => res.status(404).json({ error: 'Not Found' }));
app.listen(PORT, () => console.log(`Running on port ${PORT}`));
```

HANDS-ON

[실습 3] 확인 체크리스트

1. ✅ node index.js 실행 시 포트 3000 메시지 출력
2. ✅ 브라우저에서 localhost:3000 접속 → Hello World!
3. ✅ localhost:3000/api/status → JSON 응답 확인
4. ✅ localhost:3000/api/greet?name=내이름 → 이름 포함 응답
5. ✅ localhost:3000/api/users/1 → userId 파라미터 동작
6. ✅ 존재하지 않는 경로 → 404 에러 JSON 응답
7. ✅ nodemon 실행 시 코드 변경 후 자동 재시작

Section 6

[실습 4] 1일차 설계도를 코드로 구현하기

JSON 데이터를 반환하는 RESTful API 서버

HANDS-ON

[실습 4] 목표: 할 일(Todo) API 서버 만들기

1. 1일차에 설계한 Todo API를 Express로 구현
2. GET /api/todos - 전체 목록 조회
3. GET /api/todos/:id - 특정 항목 조회
4. POST /api/todos - 새 항목 생성
5. PUT /api/todos/:id - 항목 수정
6. DELETE /api/todos/:id - 항목 삭제
7. 메모리(배열)에 데이터 저장 (DB 없이 시작)

[실습 4-1] 프로젝트 구조 설계

bash

```
my-api-server/
 ├── package.json       ← 프로젝트 설정
 ├── node_modules/      ← 설치된 패키지들 (git 제외)
 ├── index.js           ← 서버 진입점
 ├── routes/
 │   └── todos.js       ← Todo API 라우터
 ├── data/
 │   └── todos.js       ← 초기 데이터
 └── .gitignore         ← node_modules 제외

# 폴더 생성
mkdir routes data
```

[실습 4-2] 초기 데이터 준비 (data/todos.js)

javascript

```
// data/todos.js - 메모리에 저장할 할 일 목록

let todos = [
  {
    id: 1,
    title: 'Express.js 배우기',
    completed: false,
    createdAt: '2024-01-15T09:00:00Z'
  },
  {
    id: 2,
    title: 'REST API 설계하기',
    completed: true,
    createdAt: '2024-01-15T10:00:00Z'
  },
  {
    id: 3,
    title: '오픈소스 프로젝트 찾아보기',
    completed: false,
    createdAt: '2024-01-15T11:00:00Z'
  }
];

module.exports = todos;
```

[실습 4-3] JSON 요청 본문(body) 파싱 설정

javascript

```
// index.js 에 미들웨어 추가

const express = require('express');
const app = express();
const PORT = 3000;

// ★ JSON 요청 본문을 파싱하는 미들웨어
// POST, PUT 요청에서 req.body를 사용하려면 필수!
app.use(express.json());

// 로깅 미들웨어 (모든 요청을 콘솔에 출력)
app.use((req, res, next) => {
  console.log(`[${new Date().toISOString()}] ${req.method} ${req.url}`);
  next();  // 다음 미들웨어/라우터로 이동
});
```

[실습 4-4] 미들웨어(Middleware) 이해하기

- **미들웨어 = 요청(req)과 응답(res) 사이에서 처리하는 함수**

  → 요청 → [미들웨어1] → [미들웨어2] → 라우터 → 응답

- **app.use(express.json()) → JSON 본문 파싱 미들웨어**

  → POST/PUT 요청의 body를 req.body로 접근 가능하게 해줌

- **app.use(로깅 함수) → 모든 요청을 로깅하는 미들웨어**

  → next() 호출 필수! 안 하면 요청이 멈춤

- **미들웨어 순서가 중요하다!**

  → 위에서 아래로 순서대로 실행됨

[실습 4-5] GET /api/todos - 전체 목록 조회

javascript

```
// routes/todos.js
const express = require('express');
const router = express.Router();
let todos = require('../data/todos');

// GET /api/todos - 모든 할 일 조회
router.get('/', (req, res) => {
  res.json({
    success: true,
    count: todos.length,
    data: todos
  });
});

// 응답 예시:
// {
//   "success": true,
//   "count": 3,
//   "data": [ { "id": 1, "title": "Express.js 배우기", ... }, ... ]
// }

module.exports = router;
```

[실습 4-6] GET /api/todos/:id - 단일 항목 조회

javascript

```
// GET /api/todos/:id - 특정 할 일 조회
router.get('/:id', (req, res) => {
  const id = parseInt(req.params.id);
  const todo = todos.find(t => t.id === id);

  if (!todo) {
    return res.status(404).json({
      success: false,
      error: `ID ${id}에 해당하는 할 일을 찾을 수 없습니다`
    });
  }

  res.json({
    success: true,
    data: todo
  });
});

// 테스트: GET /api/todos/1 → 성공
// 테스트: GET /api/todos/999 → 404 에러
```

[실습 4-7] POST /api/todos - 새 항목 생성

javascript

```
// POST /api/todos - 새 할 일 생성
router.post('/', (req, res) => {
  const { title } = req.body;

  // 유효성 검사
  if (!title) {
    return res.status(400).json({
      success: false,
      error: 'title 필드는 필수입니다'
    });
  }

  const newTodo = {
    id: todos.length > 0 ? Math.max(...todos.map(t => t.id)) + 1 : 1,
    title: title,
    completed: false,
    createdAt: new Date().toISOString()
  };

  todos.push(newTodo);
  res.status(201).json({ success: true, data: newTodo });
});
```

[실습 4-8] PUT /api/todos/:id - 항목 수정

javascript

```
// PUT /api/todos/:id - 할 일 수정
router.put('/:id', (req, res) => {
  const id = parseInt(req.params.id);
  const todo = todos.find(t => t.id === id);

  if (!todo) {
    return res.status(404).json({
      success: false,
      error: `ID ${id}에 해당하는 할 일을 찾을 수 없습니다`
    });
  }

  // 전달된 필드만 업데이트
  if (req.body.title !== undefined) todo.title = req.body.title;
  if (req.body.completed !== undefined) todo.completed = req.body.completed;

  res.json({ success: true, data: todo });
});

// 테스트: PUT /api/todos/1  Body: { "completed": true }
```

[실습 4-9] DELETE /api/todos/:id - 항목 삭제

javascript

```
// DELETE /api/todos/:id - 할 일 삭제
router.delete('/:id', (req, res) => {
  const id = parseInt(req.params.id);
  const index = todos.findIndex(t => t.id === id);

  if (index === -1) {
    return res.status(404).json({
      success: false,
      error: `ID ${id}에 해당하는 할 일을 찾을 수 없습니다`
    });
  }

  const deleted = todos.splice(index, 1)[0];

  res.json({
    success: true,
    message: '삭제되었습니다',
    data: deleted
  });
});

module.exports = router;
```

[실습 4-10] index.js에 라우터 연결하기

javascript

```
// index.js - 최종 서버 코드

const express = require('express');
const app = express();
const PORT = 3000;

// 미들웨어 설정
app.use(express.json());
app.use((req, res, next) => {
  console.log(`[${req.method}] ${req.url}`);
  next();
});

// 라우터 연결
const todosRouter = require('./routes/todos');
app.use('/api/todos', todosRouter);
//     ^^^^^^^^^^^^  ^^^^^^^^^^^^
//     URL 접두사     라우터 파일

// 기본 라우트
app.get('/', (req, res) => res.json({ message: 'Todo API Server' }));

// 서버 시작
app.listen(PORT, () => console.log(`Server running on port ${PORT}`));
```

[실습 4-11] API 테스트하기 (REST Client)

http

```
### test.http (VS Code REST Client 확장 사용)

### 전체 목록 조회
GET http://localhost:3000/api/todos

### 특정 항목 조회
GET http://localhost:3000/api/todos/1

### 새 항목 생성
POST http://localhost:3000/api/todos
Content-Type: application/json

{
  "title": "새로운 할 일 추가하기"
}

### 항목 수정 (완료 처리)
PUT http://localhost:3000/api/todos/1
Content-Type: application/json

{ "completed": true }

### 항목 삭제
DELETE http://localhost:3000/api/todos/1
```

[실습 4-12] 터미널에서 curl로 테스트

bash

```
# GET - 전체 목록 조회
curl http://localhost:3000/api/todos

# GET - 특정 항목 조회
curl http://localhost:3000/api/todos/1

# POST - 새 항목 생성
curl -X POST http://localhost:3000/api/todos \
  -H "Content-Type: application/json" \
  -d '{"title": "curl로 추가한 할 일"}'

# PUT - 항목 수정
curl -X PUT http://localhost:3000/api/todos/1 \
  -H "Content-Type: application/json" \
  -d '{"completed": true}'

# DELETE - 항목 삭제
curl -X DELETE http://localhost:3000/api/todos/1
```

[실습 4-13] 에러 핸들링 미들웨어 추가

javascript

```
// index.js 맨 아래에 추가 (app.listen 전에)

// 404 처리 - 정의되지 않은 라우트
app.use((req, res) => {
  res.status(404).json({
    success: false,
    error: `${req.method} ${req.url} - 존재하지 않는 경로입니다`
  });
});

// 500 처리 - 서버 에러
app.use((err, req, res, next) => {
  console.error('서버 에러:', err.stack);
  res.status(500).json({
    success: false,
    error: '서버 내부 오류가 발생했습니다'
  });
});

// 에러 핸들링 미들웨어는 인자가 4개! (err, req, res, next)
```

[실습 4-14] CORS 설정 (프론트엔드 연동 준비)

javascript

```
# cors 패키지 설치
npm install cors

// index.js 상단에 추가
const cors = require('cors');

// CORS 미들웨어 적용 (모든 도메인 허용)
app.use(cors());

// 또는 특정 도메인만 허용
app.use(cors({
  origin: 'http://localhost:5173',  // Vite 기본 포트
  methods: ['GET', 'POST', 'PUT', 'DELETE']
}));

// CORS = Cross-Origin Resource Sharing
// → 다른 도메인에서 이 API에 접근할 수 있게 허용
// → 프론트엔드(React 등)와 연동할 때 필수!
```

HANDS-ON

[실습 4] 확인 체크리스트

1. ✅ GET /api/todos → 전체 목록 JSON 응답
2. ✅ GET /api/todos/1 → 단일 항목 조회
3. ✅ GET /api/todos/999 → 404 에러 응답
4. ✅ POST /api/todos → 새 항목 생성 (201 응답)
5. ✅ POST body 없이 요청 → 400 에러 (유효성 검사)
6. ✅ PUT /api/todos/1 → 항목 수정
7. ✅ DELETE /api/todos/1 → 항목 삭제
8. ✅ 존재하지 않는 경로 → 404 에러 핸들링

Section 7

API 문서 자동화

Swagger/OpenAPI로 자동 생성하는 API 문서

Express에 Swagger 추가하기

javascript

```
# 패키지 설치
npm install swagger-jsdoc swagger-ui-express

// swagger.js 설정 파일
const swaggerJsdoc = require('swagger-jsdoc');
const swaggerUi = require('swagger-ui-express');

const options = {
  definition: {
    openapi: '3.0.0',
    info: {
      title: 'Todo API',
      version: '1.0.0',
      description: '할 일 관리 API 문서',
    },
    servers: [{ url: 'http://localhost:3000' }],
  },
  apis: ['./routes/*.js'],  // JSDoc 주석에서 스펙 읽기
};

const specs = swaggerJsdoc(options);
// app.use('/api-docs', swaggerUi.serve, swaggerUi.setup(specs));
```

Express에서 API 버전 관리 구현

javascript

```
// routes/v1/todos.js - v1 API
const v1Router = require('express').Router();

v1Router.get('/', (req, res) => {
  res.json({ version: 'v1', data: todos });
});

// routes/v2/todos.js - v2 API (새로운 필드 추가)
const v2Router = require('express').Router();

v2Router.get('/', (req, res) => {
  res.json({
    version: 'v2',
    data: todos,
    pagination: { page: 1, limit: 10, total: todos.length }
  });
});

// index.js에서 연결
app.use('/api/v1/todos', v1Router);
app.use('/api/v2/todos', v2Router);
```

실무 팁: Express 프로젝트 구조 (Best Practice)

- **controllers/ - 비즈니스 로직 분리**
  → 라우터에서 로직을 분리하면 테스트가 쉬워짐
- **middleware/ - 인증, 로깅, 에러 처리**
  → 재사용 가능한 미들웨어를 별도 파일로 관리
- **models/ - 데이터 모델 정의**
  → Mongoose(MongoDB) 또는 Sequelize(SQL) 사용
- **config/ - 환경 설정 파일**
  → dotenv로 환경 변수 관리 (.env 파일)
- **tests/ - 테스트 코드**
  → Jest 또는 Mocha로 API 테스트 작성

실무 Express 프로젝트 폴더 구조

bash

```
my-api-server/
├── package.json
├── .env              ← 환경 변수 (절대 Git에 올리지 않음!)
├── .gitignore        ← node_modules, .env 제외
├── src/
│   ├── index.js      ← 서버 진입점
│   ├── config/
│   │   └── db.js     ← 데이터베이스 연결 설정
│   ├── routes/
│   │   ├── todos.js  ← 라우트 정의
│   │   └── users.js
│   ├── controllers/
│   │   ├── todoCtrl.js ← 비즈니스 로직
│   │   └── userCtrl.js
│   ├── middleware/
│   │   ├── auth.js   ← 인증 미들웨어
│   │   └── errorHandler.js
│   └── models/
│       └── Todo.js   ← 데이터 모델
└── tests/
    └── todos.test.js ← 테스트 코드
```

API 서버 보안 기초 체크리스트

- **helmet 미들웨어 사용 - HTTP 보안 헤더 자동 설정**

  → npm install helmet → app.use(helmet())

- **rate limiting - API 호출 횟수 제한**

  → npm install express-rate-limit → DDoS 방어 기본

- **입력값 검증 - 사용자 입력을 절대 신뢰하지 마라**

  → SQL Injection, XSS 방지의 첫 걸음

- **.env로 민감 정보 관리 - 코드에 비밀번호 하드코딩 금지**

  → DB 비밀번호, API 키 등은 환경 변수로 관리

- **HTTPS 사용 - 실서비스에서는 반드시 SSL 적용**

보안 미들웨어 적용 예시

javascript

```
const helmet = require('helmet');
const rateLimit = require('express-rate-limit');

// 보안 헤더 자동 설정
app.use(helmet());

// API 호출 횟수 제한 (15분에 100회)
const limiter = rateLimit({
  windowMs: 15 * 60 * 1000,  // 15분
  max: 100,                  // 최대 100회
  message: {
    error: '너무 많은 요청입니다. 잠시 후 다시 시도하세요.'
  }
});
app.use('/api/', limiter);

# 설치 명령어:
# npm install helmet express-rate-limit
```

환경 변수로 설정 관리하기 (.env)

javascript

```
# .env 파일 (절대 Git에 올리지 않음!)
PORT=3000
NODE_ENV=development
DB_HOST=localhost
DB_PASSWORD=mysecretpassword
API_KEY=sk-1234567890abcdef

# .gitignore에 추가
node_modules/
.env

// index.js에서 사용
require('dotenv').config();  // npm install dotenv

const PORT = process.env.PORT || 3000;
const isProduction = process.env.NODE_ENV === 'production';

app.listen(PORT, () => {
  console.log(`Server running in ${process.env.NODE_ENV} mode on port ${PORT}`);
});
```

초보자를 위한 디버깅 팁

- **console.log()는 가장 기본적인 디버깅 도구**

  → req.body, req.params, req.query를 출력해 보자

- **에러 메시지를 꼼꼼히 읽자**

  → 'Cannot read property of undefined' = 변수가 비어있음

  → 'EADDRINUSE' = 이미 같은 포트를 사용 중

- **Postman 또는 Thunder Client로 요청 테스트**

  → 브라우저는 GET만 가능, POST/PUT/DELETE 테스트 불가

- **nodemon 사용 시 종료하고 다시 시작 (Ctrl+C → npm run dev)**

2일차 핵심 요약

✓ 오픈소스의 힘: 전 세계 개발자가 만든 코드를 활용하라

✓ 오픈소스 선택: Star, 커밋 활성도, 라이선스 확인 필수

✓ 패키지 매니저: npm install로 한 줄 설치

✓ Express.js: Node.js 최고의 API 프레임워크

✓ 라우팅: app.get/post/put/delete로 엔드포인트 생성

✓ 미들웨어: 요청-응답 사이에서 공통 로직 처리

✓ 에러 핸들링: 상태 코드와 에러 메시지를 명확하게

✓ 보안: helmet, cors, rate-limit, dotenv 필수

✓ API 문서: Swagger로 자동 문서화

오늘의 한 마디

- **"잘 고른 오픈소스 하나가 열 명의 개발자보다 낫다"**

  → 바퀴를 재발명하지 말고, 거인의 어깨 위에 올라타라!

- **오늘 우리가 한 것:**

  → 1. 오픈소스 선택 기준을 배웠다 (인기도, 신뢰도, 라이선스)

  → 2. Express.js로 첫 API 서버를 만들었다 (Hello World)

  → 3. 1일차 설계도를 실제 CRUD API로 구현했다

  → 4. API 문서화와 보안 기초를 학습했다

내일 예고: 3일차

- **3일차: API 보안과 취약점 진단**

  → OWASP API Security Top 10 학습

- **인증과 인가 구현**

  → JWT 토큰 기반 인증 시스템 구축

- **API 취약점 진단 실습**

  → Broken Authentication, Injection 등 실제 공격 시연

- **보안 테스트 도구 사용**

  → Burp Suite, OWASP ZAP으로 취약점 발견

HANDS-ON

2일차 과제 (선택 사항)

1. Todo API에 검색 기능 추가: GET /api/todos?search=키워드
2. Todo API에 필터 기능 추가: GET /api/todos?completed=true
3. 새로운 API 추가: /api/users (사용자 CRUD)
4. Thunder Client 또는 Postman으로 모든 엔드포인트 테스트
5. Swagger 문서 생성해보기 (swagger-jsdoc 패키지)
6. GitHub에 프로젝트 올려보기 (.gitignore 확인!)

Q & A

잘 고른 오픈소스 하나가 열 명의 개발자보다 낫다

내일 예고: 3일차: API 보안과 취약점 진단

S-개발자 4기 교육

Node.js 비동기 프로그래밍 — 왜 async/await인가?

- **동기(Synchronous) vs 비동기(Asynchronous)**
  → 동기: 요청A 완료 후 → 요청B 시작 (줄 서기)
  → 비동기: 요청A 기다리는 동안 → 요청B, C 동시 처리
- **Node.js는 싱글 스레드 + 이벤트 루프 = 비동기가 기본**
  → DB 조회, 파일 읽기, API 호출 → 기다리는 동안 다른 요청 처리
- **3가지 비동기 방식 (진화 순서)**
  → ① Callback (구식): 콜백 지옥 문제
  → ② Promise: .then().catch() 체이닝
  → ③ async/await (현대): 동기처럼 읽히는 비동기 코드 ✅

★ **Express 라우터에서 async/await 없이 DB 연동은 불가능!**

async/await — Express 라우터에서 사용하기

javascript

```
// ❌ 콜백 지옥 (Callback Hell)
app.get('/users', (req, res) => {
  db.query('SELECT * FROM users', (err, rows) => {
    if (err) { return res.status(500).json({ error: err.message }); }
    res.json(rows);  // 중첩이 깊어질수록 읽기 불가능
  });
});

// ✅ async/await (권장 방식)
app.get('/users', async (req, res) => {
  try {
    const rows = await db.query('SELECT * FROM users');
    res.json(rows);  // 동기 코드처럼 읽힘
  } catch (err) {
    res.status(500).json({ error: err.message });
  }
});
```

Express 미들웨어 동작 원리 — next() 흐름

- **미들웨어 = 요청(req)과 응답(res) 사이에서 실행되는 함수**

  → (req, res, next) => { ... next() } 형태

- **요청 처리 흐름 (체이닝)**

  → 요청 → MW1(로깅) → MW2(인증) → MW3(CORS) → 라우터 → 응답

  → next() 호출 → 다음 미들웨어로 이동

  → next() 없이 res.json() → 체인 종료 (이후 미들웨어 실행 안 됨)

- **미들웨어 종류**

  → 전역: app.use(fn) — 모든 요청에 적용

  → 경로: app.use('/api', fn) — 해당 경로만 적용

  → 에러: (err, req, res, next) => { } — 4개 인자로 에러 미들웨어 구분

★ **미들웨어 순서가 곧 실행 순서 — app.use() 위치가 매우 중요!**

입력값 검증 — express-validator 활용

- 사용자 입력을 절대 신뢰하지 마라 (보안의 기본)

javascript

```
npm install express-validator

const { body, validationResult } = require('express-validator');

app.post('/api/todos',
  body('title').notEmpty().withMessage('제목은 필수입니다')
               .isLength({ max: 100 }).withMessage('100자 이하로 입력하세요'),
  body('priority').isIn(['low','medium','high']).withMessage('잘못된 우선순위'),
  async (req, res) => {
    const errors = validationResult(req);
    if (!errors.isEmpty()) {
      return res.status(400).json({ errors: errors.array() });
    }
    // 검증 통과 후 실제 처리
    const todo = { id: Date.now(), ...req.body };
    todos.push(todo);
    res.status(201).json(todo);
  }
);
```

API 표준 응답 포맷 설계 — 일관성이 핵심

- **왜 표준 응답 포맷이 필요한가?**

  → 프론트엔드가 모든 API 응답을 같은 방식으로 처리 가능

  → 에러 처리, 페이지네이션, 메타데이터 일관성 확보

javascript

```
// 성공 응답 표준 형식
res.json({
  success: true,
  data: { id: 1, title: '공부하기', completed: false },
  meta: { timestamp: new Date().toISOString() }
});

// 에러 응답 표준 형식
res.status(400).json({
  success: false,
  error: {
    code: 'VALIDATION_ERROR',
    message: '제목은 필수입니다',
    details: [{ field: 'title', message: '비어있음' }]
  }
});
```

GitHub 기본 사용법 — 코드를 팀과 공유하는 법

- **Git = 코드 버전 관리 시스템 / GitHub = 원격 저장소 호스팅 서비스**

bash

```
# 1. 로컬 Git 초기화
git init
git config --global user.name  'Kevin'
git config --global user.email 'kevin@example.com'

# 2. 변경사항 추적 & 커밋
git add .                          # 모든 파일 스테이징
git add index.js                   # 특정 파일만
git commit -m 'feat: add todo API'  # 커밋 메시지 (conventional commit)

# 3. GitHub에 올리기
git remote add origin https://github.com/your-id/my-api.git
git push origin main

# 4. 팀원 코드 받기
git pull origin main
git clone https://github.com/팀원/프로젝트.git
```

.gitignore & Git 협업 베스트 프랙티스

- **.gitignore — Git에 올리지 않을 파일 목록**

gitignore

```
# .gitignore (프로젝트 루트에 생성)
node_modules/          # npm 패키지 (용량 거대, npm install로 재생성 가능)
.env                   # 환경 변수 (비밀번호, API 키 포함 → 절대 금지!)
.DS_Store              # macOS 시스템 파일
*.log                  # 로그 파일
dist/                  # 빌드 결과물

# 협업 브랜치 전략 (GitHub Flow)
# main         → 배포용 (항상 동작하는 코드만)
# feature/...  → 기능 개발
# fix/...      → 버그 수정

# PR(Pull Request) 기반 코드 리뷰 워크플로우
# 1. feature 브랜치 생성  → 2. 코드 작성  → 3. PR 생성
# 4. 팀원 리뷰           → 5. main 병합  → 6. 자동 배포
```

PM2 — 서버를 24시간 살아있게 만들기

- **문제: node index.js → 터미널 닫으면 서버 죽음!**
- **해결: PM2 = Node.js 프로세스 매니저 (백그라운드 실행, 자동 재시작)**

bash

```
# PM2 설치 (전역)
npm install -g pm2

# 서버 시작 (백그라운드, 이름 지정)
pm2 start index.js --name my-api

# 자주 쓰는 PM2 명령어
pm2 list              # 실행 중인 프로세스 목록
pm2 logs my-api       # 실시간 로그 보기
pm2 restart my-api    # 서버 재시작
pm2 stop my-api       # 서버 중지
pm2 delete my-api     # 프로세스 제거

# 서버 재부팅 후 자동 시작 설정
pm2 startup           # 시스템 시작 시 자동 실행 설정
pm2 save              # 현재 프로세스 목록 저장
```

Jest + Supertest — API 테스트 자동화

- 테스트 없이 배포 = 폭탄을 안고 비행기 타기
- Jest = JavaScript 테스트 프레임워크 / Supertest = HTTP 요청 테스트

javascript

```
npm install --save-dev jest supertest

// tests/todos.test.js
const request = require('supertest');
const app = require('../index');  // Express 앱 가져오기

describe('Todos API', () => {
  test('GET /api/todos → 200 & 배열 반환', async () => {
    const res = await request(app).get('/api/todos');
    expect(res.status).toBe(200);
    expect(Array.isArray(res.body.data)).toBe(true);
  });

  test('POST /api/todos → 201 & 새 항목 생성', async () => {
    const res = await request(app).post('/api/todos')
      .send({ title: '테스트 항목', priority: 'high' });
    expect(res.status).toBe(201);
    expect(res.body.data.title).toBe('테스트 항목');
  });
});
```