KISIA S-개발자 4기

5일차: API 테스트 자동화와 최종 프로젝트

완벽한 API를 위한 테스팅 (Final)

Day 5 of 5

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

5일차 학습 목표

자동화 테스트 프레임워크 이해

Jest + Supertest를 활용한 API 테스트 작성법 학습

테스트 피라미드 개념 습득

유닛/통합/E2E 테스트의 역할과 비율 이해

API 퍼징 및 성능 테스트

Schemathesis, Artillery를 이용한 고급 테스트 기법

최종 프로젝트 완성

1~4일차 내용을 종합한 보안 API 서비스 구축

팀별 발표 및 피드백

완성된 API 설계도와 보안 리포트 공유

5일차 교육 일정

09:00~09:30	복습 및 4일차 과제 리뷰
09:30~10:30	Section 1: API 테스트 자동화 기초
10:30~10:45	☕ 휴식
10:45~12:00	Section 2: 고급 테스트 기법 (퍼징/성능)
12:00~13:00	🍚 점심시간
13:00~15:00	Section 3: 최종 프로젝트 구축
15:00~15:15	☕ 휴식
15:15~16:30	Section 4: 팀별 발표 및 피드백
16:30~17:00	총정리 및 수료식

4일차 복습: 공급망 보안과 패키지 관리

- **npm audit / Snyk를 통한 취약점 스캔**

  → 의존성 잠금(lock file)과 버전 관리 전략

- **SBOM(Software Bill of Materials) 개념**

  → 공급망 공격 사례: event-stream, ua-parser-js

- **GitHub Dependabot과 자동 패치**

  → 보안 정책: .npmrc, 허용 목록(Allowlist)

- **CI/CD 파이프라인에 보안 게이트 통합**

Section 1

API 테스트 자동화 기초

반복되는 수동 테스트에서 벗어나기

테스트의 종류

- **유닛 테스트(Unit Test): 함수/모듈 단위 검증**

  → 가장 빠르고 작성 비용이 낮다

- **통합 테스트(Integration Test): 모듈 간 상호작용 검증**

  → DB, 외부 API 연동 포함

- **E2E 테스트(End-to-End): 사용자 시나리오 전체 검증**

  → 가장 느리지만 현실에 가까움

- **보안 테스트(Security Test): 취약점 탐지 자동화**

  → OWASP ZAP, Schemathesis 등 활용

테스트 피라미드 (Test Pyramid)

이상적인 비율

- 유닛 테스트: 70%
- 통합 테스트: 20%
- E2E 테스트: 10%

▲ E2E (느림, 비쌈)

▲▲ 통합 (중간)

▲▲▲ 유닛 (빠름, 저렴)

현실에서 실수

- 아이스크림 콘 패턴 주의
- E2E에만 의존하면 느려짐
- 유닛 테스트 없으면 디버깅 지옥

- 핵심: 하위 레벨 테스트를
- 충분히 작성한 후에
- 상위 레벨을 보완한다

왜 자동화인가?

- **반복되는 수동 테스트는 실수를 만든다**

  → 사람은 피곤하면 실수하지만, 스크립트는 안 쉬어도 됨

- **배포 전 자동 검증 = 장애 예방**

  → CI/CD 파이프라인에 테스트를 필수 게이트로 설정

- **회귀 테스트(Regression): 새 코드가 기존 기능을 안 깨뜨리는지 확인**

  → 매번 전체 기능을 수동으로 확인할 수 없다

- **테스트 커버리지 = 코드 품질의 객관적 지표**

  → '테스트 없는 코드는 레거시' - Michael Feathers

Jest: JavaScript 테스트 프레임워크

- **Meta(Facebook)가 만든 올인원 테스트 러너**

  → 테스트 러너 + 단언(assertion) + 모킹 내장

- **Zero-config: 설정 없이 바로 시작 가능**
- **스냅샷 테스팅, 코드 커버리지 기본 지원**
- **watch 모드로 파일 변경 시 자동 재실행**

  → 개발 중 즉각적인 피드백 루프 형성

- **Node.js API 테스트에 가장 널리 사용**

Jest 설치 및 기본 설정

bash / json

```
# Jest 설치
npm install --save-dev jest

# package.json에 테스트 스크립트 추가
{
  "scripts": {
    "test": "jest",
    "test:watch": "jest --watch",
    "test:coverage": "jest --coverage"
  }
}

# jest.config.js (선택)
module.exports = {
  testEnvironment: 'node',
  coverageThreshold: {
    global: { branches: 80, functions: 80, lines: 80 }
  }
};
```

첫 번째 Jest 테스트 작성

javascript

```
// math.js
function add(a, b) { return a + b; }
function divide(a, b) {
  if (b === 0) throw new Error('Division by zero');
  return a / b;
}
module.exports = { add, divide };

// math.test.js
const { add, divide } = require('./math');

describe('Math functions', () => {
  test('add: 1 + 2 = 3', () => {
    expect(add(1, 2)).toBe(3);
  });

  test('divide: 10 / 2 = 5', () => {
    expect(divide(10, 2)).toBe(5);
  });

  test('divide by zero throws error', () => {
    expect(() => divide(10, 0)).toThrow('Division by zero');
  });
});
```

Supertest: HTTP 요청 테스트 라이브러리

- **Express 앱을 직접 import해서 HTTP 요청을 시뮬레이션**

  → 실제 서버를 띄우지 않아도 API 테스트 가능

- **체이닝 방식으로 직관적인 테스트 작성**

  → request(app).get('/api/users').expect(200)

- **응답 본문, 헤더, 상태 코드 모두 검증 가능**
- **Jest와 함께 사용하면 완벽한 API 테스트 환경**

Supertest: Express API 테스트

javascript

```
# 설치
npm install --save-dev supertest

// app.js (Express 앱 export)
const express = require('express');
const app = express();
app.use(express.json());

app.get('/api/users', (req, res) => {
  res.json([{ id: 1, name: 'Alice' }]);
});

module.exports = app;  // ← 반드시 export!

// app.test.js
const request = require('supertest');
const app = require('./app');

describe('GET /api/users', () => {
  it('should return 200 and user list', async () => {
    const res = await request(app).get('/api/users');
    expect(res.status).toBe(200);
    expect(res.body).toHaveLength(1);
    expect(res.body[0].name).toBe('Alice');
  });
});
```

JWT 인증 API 테스트 (Supertest)

javascript

```
describe('인증이 필요한 API 테스트', () => {
  let token;

  beforeAll(async () => {
    // 로그인으로 토큰 획득
    const res = await request(app)
      .post('/api/auth/login')
      .send({ email: 'test@test.com', password: 'pass123' });
    token = res.body.token;
  });

  it('토큰 없이 요청하면 401', async () => {
    const res = await request(app).get('/api/users/me');
    expect(res.status).toBe(401);
  });

  it('유효한 토큰으로 요청하면 200', async () => {
    const res = await request(app)
      .get('/api/users/me')
      .set('Authorization', `Bearer ${token}`);
    expect(res.status).toBe(200);
    expect(res.body.email).toBe('test@test.com');
  });
});
```

BOLA 방어 테스트 작성

javascript

```
describe('BOLA 방어 테스트', () => {
  let userAToken, userBToken;

  beforeAll(async () => {
    // 사용자 A 로그인
    const resA = await request(app).post('/api/auth/login')
      .send({ email: 'alice@test.com', password: 'passA' });
    userAToken = resA.body.token;

    // 사용자 B 로그인
    const resB = await request(app).post('/api/auth/login')
      .send({ email: 'bob@test.com', password: 'passB' });
    userBToken = resB.body.token;
  });

  it('사용자 B가 사용자 A의 데이터에 접근하면 403', async () => {
    const res = await request(app)
      .get('/api/users/1/profile')  // 사용자 A의 프로필
      .set('Authorization', `Bearer ${userBToken}`);
    expect(res.status).toBe(403);
    expect(res.body.error).toMatch(/권한/);
  });
});
```

HANDS-ON

실습: Jest + Supertest 테스트 환경 구축

1. 프로젝트에 jest, supertest 설치
2. app.js에서 Express 앱을 module.exports
3. tests/ 폴더 생성 후 auth.test.js 작성
4. 로그인 API 테스트 3개 이상 작성
5. BOLA 방어 테스트 1개 이상 작성
6. npm test 실행하여 결과 확인
7. 실패하는 테스트를 만들어보고 원인 분석

코드 커버리지 (Code Coverage)

- **테스트가 코드의 어느 정도를 실행하는가?**

  → 커버리지 100% ≠ 버그 없음 (필요조건일 뿐)
- **라인 커버리지(Line): 실행된 코드 라인 비율**
- **브랜치 커버리지(Branch): if/else 분기 실행 비율**

  → 브랜치 커버리지가 더 중요!
- **함수 커버리지(Function): 호출된 함수 비율**
- **목표: 최소 80% 이상 유지 권장**

  → CI에서 커버리지 임계값 미달 시 빌드 실패 설정

Jest 코드 커버리지 실행 및 리포트

bash

```
# 커버리지 실행
npx jest --coverage

# 출력 예시
------------|---------|----------|---------|---------|
File        | % Stmts | % Branch | % Funcs | % Lines |
------------|---------|----------|---------|---------|
All files   |   87.5  |    75    |   100   |   87.5  |
 app.js     |   85.7  |    66.7  |   100   |   85.7  |
 auth.js    |   90    |    80    |   100   |   90    |
 users.js   |   83.3  |    75    |   100   |   83.3  |
------------|---------|----------|---------|---------|

# HTML 리포트 확인
open coverage/lcov-report/index.html

# jest.config.js에서 임계값 설정
coverageThreshold: {
  global: { branches: 80, functions: 80, lines: 80 }
}
```

테스트 더블: Mock, Stub, Spy

개념

- Stub: 미리 정해진 값을 반환
- Mock: 호출 여부/횟수 검증
- Spy: 실제 동작 + 호출 추적

- 외부 의존성 격리에 필수
- DB, 외부 API 호출 대체
- 테스트 속도 향상 효과

Jest에서 사용법

- jest.fn() → Mock 함수 생성
- jest.spyOn(obj, 'method')
- mockReturnValue(value)
- mockResolvedValue(value)
- toHaveBeenCalledWith(arg)
- toHaveBeenCalledTimes(n)
- jest.mock('./module')

Jest Mock 실전: DB 호출 모킹

javascript

```
// userService.js
const db = require('./db');
async function getUser(id) {
  const user = await db.findById(id);
  if (!user) throw new Error('User not found');
  return { ...user, displayName: user.name.toUpperCase() };
}

// userService.test.js
jest.mock('./db');
const db = require('./db');
const { getUser } = require('./userService');

describe('getUser', () => {
  it('사용자를 찾으면 displayName 반환', async () => {
    db.findById.mockResolvedValue({ id: 1, name: 'alice' });
    const user = await getUser(1);
    expect(user.displayName).toBe('ALICE');
    expect(db.findById).toHaveBeenCalledWith(1);
  });

  it('사용자가 없으면 에러 발생', async () => {
    db.findById.mockResolvedValue(null);
    await expect(getUser(999)).rejects.toThrow('User not found');
  });
});
```

Section 2

고급 테스트 기법

퍼징, 성능 테스트, Postman 자동화

API 퍼징(Fuzzing) 테스트

- **무작위/비정상 데이터를 API에 대량 전송하여 취약점 탐지**

  → 예상치 못한 입력에 서버가 어떻게 반응하는가?

- **일반 테스트로는 잡을 수 없는 엣지 케이스 발견**

  → 정수 오버플로, SQL 인젝션, 버퍼 오버플로 등

- **자동화된 퍼저(fuzzer)가 수천~수만 건을 시도**

- **OpenAPI 스펙 기반 퍼징 = 스마트 퍼징**

  → Schemathesis: OpenAPI 스펙을 분석해 자동으로 테스트 케이스 생성

Schemathesis: OpenAPI 기반 퍼징 도구

bash

```
# 설치
pip install schemathesis

# 기본 실행 (OpenAPI 스펙 URL 지정)
schemathesis run http://localhost:3000/api-docs/swagger.json

# 출력 예시
================ Schemathesis Test Session ================
Schema: http://localhost:3000/api-docs/swagger.json
Base URL: http://localhost:3000

GET /api/users ................... SUCCESS
POST /api/users
  → 500 Internal Server Error
    Input: {"name": null, "email": "x" * 10000}
    → 서버가 입력 길이를 검증하지 않음!

POST /api/auth/login
  → 500 Internal Server Error
    Input: {"email": "' OR 1=1 --", "password": 0}
    → SQL Injection 가능성 발견!

Results: 2 failures, 15 passed
```

Schemathesis 고급 옵션

bash

```
# 특정 엔드포인트만 테스트
schemathesis run --endpoint /api/users \
  http://localhost:3000/swagger.json

# 인증 헤더 추가
schemathesis run \
  --header "Authorization: Bearer eyJhbG..." \
  http://localhost:3000/swagger.json

# 최대 테스트 케이스 수 제한
schemathesis run --hypothesis-max-examples=500 \
  http://localhost:3000/swagger.json

# 상태 코드별 검증 (5xx는 항상 실패로 간주)
schemathesis run --validate-schema=true \
  --checks all \
  http://localhost:3000/swagger.json

# JUnit XML 리포트 생성 (CI 연동)
schemathesis run --junit-xml=report.xml \
  http://localhost:3000/swagger.json
```

Postman을 이용한 테스트 자동화

- **Postman Tests 탭에서 JavaScript로 검증 스크립트 작성**

  → pm.test(), pm.response, pm.expect() 활용

- **Collection Runner로 전체 API를 한번에 테스트**
- **Newman: Postman CLI 도구로 CI/CD 파이프라인 연동**

  → npm install -g newman

- **환경 변수로 개발/스테이징/프로덕션 분리**
- **Pre-request Script로 토큰 자동 갱신**

Postman 테스트 스크립트 예제

javascript

```javascript
// Tests 탭에 작성

// 1. 상태 코드 검증
pm.test("Status code is 200", () => {
  pm.response.to.have.status(200);
});

// 2. 응답 시간 검증
pm.test("Response time < 500ms", () => {
  pm.expect(pm.response.responseTime).to.be.below(500);
});

// 3. 응답 본문 검증
pm.test("Response has users array", () => {
  const json = pm.response.json();
  pm.expect(json).to.be.an('array');
  pm.expect(json.length).to.be.above(0);
  pm.expect(json[0]).to.have.property('id');
  pm.expect(json[0]).to.have.property('name');
});

// 4. 헤더 검증
pm.test("Content-Type is JSON", () => {
  pm.response.to.have.header("Content-Type", /json/);
});

// 5. 환경 변수 저장 (다음 요청에서 사용)
const token = pm.response.json().token;
pm.environment.set("auth_token", token);
```

Newman: Postman CLI로 CI 연동

bash

```
# Newman 설치
npm install -g newman

# 컬렉션 실행
newman run my-api-tests.postman_collection.json \
  -e production.postman_environment.json

# HTML 리포트 생성
npm install -g newman-reporter-html
newman run collection.json \
  -r html --reporter-html-export report.html

# CI/CD (GitHub Actions) 예시
name: API Tests
on: push
jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - run: npm install -g newman
      - run: newman run tests/api-collection.json
            -e tests/env.json
            --bail  # 첫 실패 시 중단
```

성능 테스트 기초

- **동시 접속자가 많을 때 내 API가 버틸 수 있는가?**
  - → 부하 테스트(Load Test): 예상 트래픽에서의 성능 확인
  - → 스트레스 테스트(Stress Test): 한계점 찾기
  - → 스파이크 테스트(Spike Test): 급격한 트래픽 증가 대응
- **핵심 지표:**
  - → 응답 시간(Response Time): p50, p95, p99
  - → 처리량(Throughput): 초당 요청 수 (RPS)
  - → 에러율(Error Rate): 5xx 응답 비율

Artillery: 성능 테스트 도구

yaml / bash

```yaml
# 설치
npm install -g artillery

# 시나리오 파일 (load-test.yml)
config:
  target: "http://localhost:3000"
  phases:
    - duration: 60      # 60초 동안
      arrivalRate: 10   # 초당 10명 접속
    - duration: 30
      arrivalRate: 50   # 점진적 증가

scenarios:
  - name: "사용자 조회 시나리오"
    flow:
      - post:
          url: "/api/auth/login"
          json: { "email": "test@test.com", "password": "pass" }
          capture:
            - json: "$.token"
              as: "token"
      - get:
          url: "/api/users"
          headers:
            Authorization: "Bearer {{token}}"

# 실행
artillery run load-test.yml --output report.json
artillery report report.json  # HTML 리포트
```

k6: 개발자 친화적 성능 테스트

javascript

```
// load-test.js
import http from 'k6/http';
import { check, sleep } from 'k6';

export const options = {
  stages: [
    { duration: '30s', target: 20 },   // Ramp up
    { duration: '1m',  target: 20 },   // Sustain
    { duration: '10s', target: 0 },    // Ramp down
  ],
  thresholds: {
    http_req_duration: ['p(95)<500'],   // 95% < 500ms
    http_req_failed: ['rate<0.01'],     // 에러율 < 1%
  },
};

export default function () {
  const res = http.get('http://localhost:3000/api/users');
  check(res, {
    'status is 200': (r) => r.status === 200,
    'response time < 200ms': (r) => r.timings.duration < 200,
  });
  sleep(1);
}

// 실행: k6 run load-test.js
```

성능 테스트 결과 읽는 법

주요 지표

- p50 (중간값): 일반적 응답 시간
- p95: 95%의 요청이 이 시간 내
- p99: 최악의 1%를 제외한 시간
- RPS: 초당 처리 요청 수
- 에러율: 5xx 응답 비율
- p95 < 500ms가 일반적 목표

문제 진단

- p50은 좋은데 p99가 나쁘면?
  → 간헐적 느린 쿼리/GC 의심
- RPS가 낮으면?
  → CPU/메모리 병목 확인
- 에러율 급증 시?
  → 커넥션 풀 고갈 의심
  → DB 커넥션 수 확인

HANDS-ON

실습: API 퍼징 및 성능 테스트

1. Schemathesis로 자신의 API를 퍼징 테스트
2. 발견된 취약점을 기록하고 수정
3. Postman에서 테스트 스크립트 3개 이상 작성
4. Newman으로 CLI 실행 확인
5. Artillery 시나리오 작성 후 부하 테스트 실행
6. p95 응답 시간 500ms 이하 달성
7. 결과를 정리하여 팀원들과 공유

Section 3

최종 프로젝트: 보안 API 서비스 구축

1~5일차 내용을 종합한 실전 프로젝트

최종 프로젝트 개요

- **주제: 보안이 강화된 사용자 관리 REST API 서비스**
- **Day 1 ~ Day 5의 모든 내용을 적용하는 종합 실습**

  → API 설계 → 구현 → 보안 → 공급망 → 테스트
- **팀 단위(2~3인)로 진행**
- **결과물: 동작하는 API + 보안 리포트 + 테스트 결과**
- **발표 시간: 팀당 10~15분**

  → 코드 리뷰 + 라이브 데모 포함

[Day 1] API 설계 요구사항

- **OpenAPI 3.0 스펙 문서 작성 (YAML/JSON)**
  → 최소 5개 이상의 엔드포인트 정의
- **RESTful 규칙 준수 (명사형 URI, 적절한 HTTP 메서드)**
  → GET, POST, PUT, DELETE 모두 사용
- **요청/응답 스키마 정의 (JSON Schema)**
  → 필수/선택 필드, 데이터 타입 명시
- **에러 응답 형식 통일**
  → RFC 7807 Problem Details 권장
- **Swagger UI로 문서 자동 생성**

[Day 2] API 구현 요구사항

- **Express.js로 REST API 서버 구현**

  → 라우터 분리, 미들웨어 패턴 적용

- **CRUD 기능 완성 (Create, Read, Update, Delete)**
- **입력 검증(Validation) 미들웨어 적용**

  → express-validator 또는 Joi 사용

- **에러 핸들링 미들웨어 구현**

  → 400, 401, 403, 404, 500 모두 처리

- **환경 변수 관리 (.env + dotenv)**
- **로깅 미들웨어 적용 (morgan/winston)**

[Day 3] 보안 요구사항

- **JWT 기반 인증 구현 (로그인/회원가입)**

  → Access Token + Refresh Token 구조

- **RBAC(역할 기반 접근제어) 적용**

  → admin, user 역할 구분

- **BOLA(Broken Object Level Authorization) 방어**

  → 리소스 소유자 검증 미들웨어

- **Rate Limiting 적용 (express-rate-limit)**
- **CORS 설정 (허용 도메인 제한)**
- **Helmet.js로 HTTP 보안 헤더 설정**

[Day 4] 공급망 보안 요구사항

- **npm audit으로 취약점 스캔 실행 및 결과 보고**

  → critical/high 취약점 0개 달성

- **package-lock.json 커밋 (의존성 잠금)**
- **.npmrc 보안 설정 적용**

  → audit=true, save-exact=true

- **최소 권한 의존성 원칙 적용**

  → 불필요한 패키지 제거

- **GitHub Dependabot 또는 Snyk 연동 (선택)**

[Day 5] 테스트 요구사항

- **Jest + Supertest로 API 테스트 작성**
  - → 최소 15개 이상의 테스트 케이스
- **코드 커버리지 80% 이상 달성**
- **테스트 종류 포함:**
  - → 유닛 테스트: 비즈니스 로직 검증
  - → 통합 테스트: API 엔드포인트 검증
  - → 보안 테스트: 인증/인가 우회 시도 검증
- **Postman 컬렉션 작성 및 Newman 실행**
- **성능 테스트 결과 포함 (Artillery/k6)**

최종 프로젝트 디렉토리 구조

```
secure-api-project/
├── package.json           # 의존성 및 스크립트
├── package-lock.json      # 의존성 잠금
├── .env                   # 환경 변수 (git 제외)
├── .npmrc                 # npm 보안 설정
├── jest.config.js         # 테스트 설정
├── openapi.yaml           # API 스펙 문서 [Day 1]
├── src/
│   ├── app.js             # Express 앱 [Day 2]
│   ├── server.js          # 서버 시작점
│   ├── routes/            # 라우터 [Day 2]
│   ├── middleware/         # 미들웨어 [Day 3]
│   │   ├── auth.js        # JWT 인증
│   │   ├── rbac.js        # 역할 기반 접근제어
│   │   └── validate.js    # 입력 검증
│   ├── models/            # 데이터 모델
│   └── utils/             # 유틸리티
├── tests/                 # 테스트 [Day 5]
│   ├── unit/
│   ├── integration/
│   └── security/
├── postman/               # Postman 컬렉션
└── docs/                  # 보안 리포트 [Day 4]
```

Step 1: 프로젝트 초기화

bash

```
# 프로젝트 생성
mkdir secure-api-project && cd secure-api-project
npm init -y

# 핵심 의존성 설치
npm install express jsonwebtoken bcryptjs dotenv cors \
  helmet express-rate-limit express-validator morgan \
  swagger-ui-express yamljs

# 개발 의존성 설치
npm install --save-dev jest supertest nodemon

# .npmrc 보안 설정
echo "audit=true" >> .npmrc
echo "save-exact=true" >> .npmrc

# package.json scripts
{
  "scripts": {
    "start": "node src/server.js",
    "dev": "nodemon src/server.js",
    "test": "jest --coverage",
    "audit": "npm audit --audit-level=high"
  }
}
```

Step 2: OpenAPI 3.0 스펙 작성

yaml

```yaml
# openapi.yaml
openapi: 3.0.3
info:
  title: Secure User API
  version: 1.0.0
  description: 보안이 강화된 사용자 관리 API

paths:
  /api/auth/register:
    post:
      summary: 회원가입
      requestBody:
        content:
          application/json:
            schema:
              $ref:
'#/components/schemas/RegisterRequest'
      responses:
        '201':
          description: 가입 성공
        '400':
          description: 유효성 검증 실패
```

```yaml
# openapi.yaml
openapi: 3.0.3

paths:
  /api/auth/login:
    post:
      summary: 로그인
      responses:
        '200':
          description: 로그인 성공, JWT 토큰 반환

  /api/users/{id}:
    get:
      summary: 사용자 조회 (BOLA 방어 적용)
      security:
        - bearerAuth: []
```

Step 3: Express 앱 기본 구조

javascript

```
// src/app.js
const express = require('express');
const helmet = require('helmet');
const cors = require('cors');
const rateLimit = require('express-rate-limit');
const morgan = require('morgan');
const swaggerUi = require('swagger-ui-express');
const YAML = require('yamljs');

const app = express();

// 보안 미들웨어 [Day 3]
app.use(helmet());
app.use(cors({ origin: 'https://myapp.com' }));
app.use(rateLimit({ windowMs: 15*60*1000, max: 100 }));

// 기본 미들웨어
app.use(express.json({ limit: '10kb' }));
app.use(morgan('combined'));

// API 문서 [Day 1]
const spec = YAML.load('./openapi.yaml');
app.use('/api-docs', swaggerUi.serve, swaggerUi.setup(spec));
```

```
// src/app.js
// 라우터 [Day 2]
app.use('/api/auth', require('./routes/auth'));
app.use('/api/users', require('./routes/users'));

// 에러 핸들러
app.use((err, req, res, next) => {
  res.status(err.status || 500).json({ error: err.message });
});

module.exports = app;
```

Step 4: JWT 인증 미들웨어 구현

javascript

```
// src/middleware/auth.js
const jwt = require('jsonwebtoken');

function authenticate(req, res, next) {
  const header = req.headers.authorization;
  if (!header || !header.startsWith('Bearer ')) {
    return res.status(401).json({ error: '인증 토큰이 필요합니다' });
  }

  try {
    const token = header.split(' ')[1];
    const decoded = jwt.verify(token, process.env.JWT_SECRET);
    req.user = decoded;
    next();
  } catch (err) {
    if (err.name === 'TokenExpiredError') {
      return res.status(401).json({ error: '토큰이 만료되었습니다' });
    }
    return res.status(401).json({ error: '유효하지 않은 토큰' });
  }
}
```

```
// src/middleware/auth.js
// BOLA 방어 미들웨어 [Day 3]
function authorizeOwner(req, res, next) {
  if (req.user.id !== parseInt(req.params.id)
      && req.user.role !== 'admin') {
    return res.status(403).json({ error: '접근 권한이 없습니다' });
  }
  next();
}

module.exports = { authenticate, authorizeOwner };
```

Step 5: 사용자 라우터 구현

javascript

```
// src/routes/users.js
const router = require('express').Router();
const { authenticate, authorizeOwner } = require('../middleware/auth');
const { body, param, validationResult } = require('express-validator');

// 입력 검증 미들웨어
const validate = (req, res, next) => {
  const errors = validationResult(req);
  if (!errors.isEmpty()) {
    return res.status(400).json({ errors: errors.array() });
  }
  next();
};

// GET /api/users - 전체 사용자 목록 (admin만)
router.get('/', authenticate, authorize('admin'), (req, res) => {
  res.json(users);
});
```

```
// src/routes/users.js

// GET /api/users/:id - 사용자 조회 (본인 또는 admin)
router.get('/:id',
  authenticate,
  authorizeOwner,  // BOLA 방어!
  param('id').isInt(),
  validate,
  (req, res) => {
    const user = users.find(u => u.id === parseInt(req.params.id));
    if (!user) return res.status(404).json({ error: 'Not found' });
    res.json(user);
  }
);
```

Step 6: 통합 테스트 작성

javascript

```
// tests/integration/users.test.js
const request = require('supertest');
const app = require('../../src/app');

describe('Users API', () => {
  let adminToken, userToken;

  beforeAll(async () => {
    // 테스트 사용자 생성 및 토큰 획득
    await request(app).post('/api/auth/register')
      .send({ name: 'Admin', email: 'admin@t.com', password: 'P@ss1234', role: 'admin' });
    const adminRes = await request(app).post('/api/auth/login')
      .send({ email: 'admin@t.com', password: 'P@ss1234' });
    adminToken = adminRes.body.token;

    await request(app).post('/api/auth/register')
      .send({ name: 'User', email: 'user@t.com', password: 'P@ss1234' });
    const userRes = await request(app).post('/api/auth/login')
      .send({ email: 'user@t.com', password: 'P@ss1234' });
    userToken = userRes.body.token;
  });
```

```
// tests/integration/users.test.js
  it('인증 없이 요청 → 401', async () => {
    await request(app).get('/api/users').expect(401);
  });

  it('admin은 전체 목록 조회 가능', async () => {
    const res = await request(app).get('/api/users')
      .set('Authorization', `Bearer ${adminToken}`);
    expect(res.status).toBe(200);
  });
});
```

Step 7: 보안 테스트 작성

javascript

```
// tests/security/auth.test.js
describe('보안 테스트', () => {
  it('SQL Injection 시도 → 400', async () => {
    const res = await request(app).post('/api/auth/login')
      .send({ email: "' OR 1=1 --", password: 'anything'
});
    expect(res.status).toBe(400);
  });

  it('JWT 조작 시도 → 401', async () => {
    const res = await request(app).get('/api/users/1')
      .set('Authorization', 'Bearer fake.token.here');
    expect(res.status).toBe(401);
  });

  it('Rate Limit 초과 → 429', async () => {
    const promises = Array(101).fill().map(() =>
      request(app).post('/api/auth/login')
        .send({ email: 'x@x.com', password: 'x' })
    );
    const results = await Promise.all(promises);
    const tooMany = results.filter(r => r.status === 429);
    expect(tooMany.length).toBeGreaterThan(0);
  });
```

```
// tests/security/auth.test.js
  it('XSS 페이로드 → 이스케이프 처리', async () => {
    const res = await request(app).post('/api/users')
      .set('Authorization', `Bearer ${adminToken}`)
      .send({ name: '<script>alert(1)</script>' });
    expect(res.body.name).not.toContain('<script>');
  });
});
```

Step 8: 공급망 보안 점검 및 리포트

markdown

```
# npm audit 실행
npm audit

# 취약점 수정
npm audit fix

# 보안 리포트 작성 (docs/security-report.md)
# ==================================
# 보안 점검 리포트
# 프로젝트: Secure User API
# 점검일: 2026-02-22
#
# ## 1. 의존성 보안
# - npm audit 결과: 취약점 0개 (critical/high)
# - 총 패키지 수: 45개
# - 직접 의존성: 12개
#
# ## 2. 인증/인가
# - JWT 기반 인증 구현 완료
# - BOLA 방어 미들웨어 적용
# - Rate Limiting 적용 (15분당 100회)
#
# ## 3. 테스트 결과
# - 전체 테스트: 25개 통과
# - 코드 커버리지: 85%
# - 보안 테스트: 8개 통과
#
# ## 4. 개선 필요 사항
# - Refresh Token 구현 필요
# - 로그 모니터링 연동 필요
```

Step 9: GitHub Actions CI/CD 구성

yaml

```yaml
# .github/workflows/ci.yml
name: Secure API CI

on: [push, pull_request]

jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

      - name: Setup Node.js
        uses: actions/setup-node@v4
        with:
          node-version: '20'

      - name: Install dependencies
        run: npm ci  # lock file 기반 정확한 설치

      - name: Security audit
        run: npm audit --audit-level=high

      - name: Run tests with coverage
        run: npm test -- --coverage
        env:
          JWT_SECRET: test-secret-key
```

```yaml
# .github/workflows/ci.yml
      - name: Check coverage threshold
        run: |
          COVERAGE=$(npx jest --coverage --coverageReporters=json-summary \
            | grep -o '"pct":[0-9.]*' | head -1 | cut -d: -f2)
          if (( $(echo "$COVERAGE < 80" | bc -l) )); then
            echo "Coverage $COVERAGE% is below 80%"
            exit 1
          fi
```

최종 프로젝트 평가 기준

기술 점수 (70%)

- API 설계 완성도: 15점
- 구현 품질 (코드 구조): 15점
- 보안 적용 수준: 20점
- 테스트 커버리지: 10점
- 공급망 보안 점검: 10점
- * 코드 리뷰 기반 채점

발표 점수 (30%)

- 아키텍처 설명 명확성: 10점
- 라이브 데모 완성도: 10점
- 보안 리포트 품질: 5점
- Q&A 대응력: 5점
- 총점: 100점
- * 80점 이상 수료 인정

최종 프로젝트 체크리스트

- ☐ OpenAPI 스펙 문서 (openapi.yaml)
- ☐ Express.js 서버 정상 실행
- ☐ 5개 이상 엔드포인트 동작
- ☐ JWT 인증/인가 구현
- ☐ BOLA 방어 미들웨어 적용
- ☐ Rate Limiting, Helmet, CORS 설정
- ☐ npm audit critical/high = 0
- ☐ 15개 이상 테스트 케이스 통과
- ☐ 코드 커버리지 80% 이상
- ☐ 보안 리포트 작성 완료

HANDS-ON

최종 프로젝트 실습 시작!

1. 팀 구성 (2~3인) 및 역할 분담
2. GitHub 레포지토리 생성 및 협업 환경 구축
3. Step 1~9에 따라 순서대로 진행
4. 중간 중간 팀원과 코드 리뷰
5. 막히는 부분은 바로 질문!
6. 목표: 2시간 내 동작하는 API 완성
7. 발표 자료 준비 (간단한 슬라이드 또는 터미널 데모)

Section 4

팀별 발표 및 피드백

완성된 API 설계도와 보안 리포트 공유

팀별 발표 형식

- **발표 시간: 팀당 10~15분 (발표 10분 + Q&A 5분)**
- **발표 내용:**
  - → 1. 프로젝트 소개 및 아키텍처 설명 (2분)
  - → 2. 보안 설계 결정사항 설명 (3분)
  - → 3. 라이브 데모 (3분)
  - → 4. 테스트 결과 및 커버리지 공유 (1분)
  - → 5. 어려웠던 점 및 배운 점 (1분)
- **라이브 데모 팁:**
  - → Postman 또는 curl로 실제 API 호출 시연
  - → 의도적으로 실패하는 케이스도 보여주기 (401, 403 등)

동료 피드백 가이드

좋은 피드백

- 구체적으로 칭찬하기
- 'JWT 에러 처리가 세밀하네요'
- 개선 제안 시 대안 함께 제시
- 'Refresh Token도 추가하면 좋겠어요'
- 코드 구조에 대한 의견 공유
- 보안 관점에서 놓친 부분 지적

피드백 체크리스트

- API 설계가 RESTful한가?
- 에러 처리가 일관적인가?
- 보안 미들웨어가 적절한가?
- 테스트가 의미 있는 시나리오를 커버하는가?
- 코드 가독성은 어떤가?
- 문서화는 충분한가?

우수 사례에서 배우는 포인트

- **깔끔한 프로젝트 구조와 네이밍 컨벤션**

  → routes/, middleware/, models/ 분리가 잘 된 팀

- **테스트 시나리오의 다양성**

  → 해피 패스뿐 아니라 엣지 케이스도 테스트한 팀

- **보안 리포트의 완성도**

  → 발견된 취약점과 대응 조치를 명확히 기술한 팀

- **CI/CD 파이프라인 구성**

  → GitHub Actions까지 연동한 팀에 보너스 점수!

Section 5

5일 과정 총정리

API 전문가로 거듭나기 위한 여정의 시작

5일간의 학습 여정 요약

✓ Day 1: API 설계 원칙 — REST, OpenAPI 스펙, 문서화

✓ Day 2: API 구현 — Express.js, 미들웨어, CRUD

✓ Day 3: API 보안 — JWT, RBAC, BOLA, OWASP Top 10

✓ Day 4: 공급망 보안 — npm audit, Snyk, SBOM, Dependabot

✓ Day 5: 테스트 자동화 — Jest, 퍼징, 성능, 최종 프로젝트

Day 1 핵심: API 설계의 기초

- **REST 아키텍처 스타일의 6가지 원칙**

  → Client-Server, Stateless, Cacheable, Uniform Interface...

- **좋은 API URI 설계: 명사형, 복수형, 계층적**

  → GET /api/users/{id}/orders (O)

- **OpenAPI 3.0 스펙으로 API 문서화**

  → 스펙 먼저(Design-First) vs 코드 먼저(Code-First)

- **HTTP 상태 코드의 올바른 사용**

  → 200, 201, 400, 401, 403, 404, 500

Day 2 핵심: Express.js로 API 구현

- **Express.js의 미들웨어 체인 패턴 이해**

  → 요청 → 미들웨어1 → 미들웨어2 → 핸들러 → 응답

- **라우터 모듈화와 관심사 분리**
- **입력 검증의 중요성 (Never Trust User Input)**

  → express-validator, Joi로 서버사이드 검증

- **에러 핸들링 미들웨어 패턴**

  → app.use((err, req, res, next) => { ... })

- **환경별 설정 관리 (.env + dotenv)**

Day 3 핵심: API 보안 실전

- **OWASP API Security Top 10 이해**
  → BOLA가 1위인 이유: 가장 흔하고 쉽게 악용됨
- **JWT 인증의 원리와 구현**
  → Header.Payload.Signature 구조
- **역할 기반 접근제어 (RBAC) 설계**
- **Rate Limiting으로 무차별 공격 방어**
- **보안 헤더 설정 (Helmet.js)**
  → X-Frame-Options, CSP, HSTS 등

Day 4 핵심: 공급망 보안

- **공급망 공격의 위험성과 실제 사례**

  → event-stream: 1,500만 다운로드의 패키지가 악성코드 포함

- **npm audit, Snyk를 통한 자동 취약점 스캔**
- **의존성 잠금(lock file)의 중요성**

  → CI에서 npm ci 사용 (npm install 대신)

- **SBOM(Software Bill of Materials)으로 투명성 확보**
- **Dependabot으로 자동 보안 패치**

Day 5 핵심: 테스트 자동화

- **테스트 피라미드: 유닛(70%) > 통합(20%) > E2E(10%)**
- **Jest + Supertest로 API 테스트 작성법**

  → describe/it 구조, expect 매처, async/await
- **코드 커버리지: 80% 이상 목표**
- **API 퍼징으로 숨겨진 취약점 탐지**

  → Schemathesis: OpenAPI 기반 자동 퍼징
- **성능 테스트: p95 응답 시간 목표 설정**

  → Artillery, k6로 부하 테스트

API 보안 최종 체크리스트

- ☑ 인증: JWT + Refresh Token 구현
- ☑ 인가: RBAC + BOLA 방어
- ☑ 입력 검증: 모든 사용자 입력 서버사이드 검증
- ☑ Rate Limiting: 무차별 공격 방어
- ☑ 보안 헤더: Helmet.js 적용
- ☑ CORS: 허용 도메인 제한
- ☑ 의존성: npm audit clean
- ☑ 테스트: 보안 테스트 케이스 포함
- ☑ 로깅: 보안 이벤트 로깅
- ☑ CI/CD: 보안 게이트 통합

Q & A

5일간의 교육에 대한 질문을 자유롭게 해주세요!

궁금한 점, 더 알고 싶은 주제, 현업 적용 질문 등

S-개발자 4기 과정