서비스 설계와 유지보수 교육

3일차: 서비스를 세상에 내보내고, 건강하게 관리하기

"만드는 것보다 운영하는 것이 더 어렵다"

목표

서비스를 안정적으로 운영하는 법

1~2일차에 우리가 배운 것

설계 → 분리 → 연결, 이제 "운영"할 차례!

Day 1: 기본 구조

3계층 아키텍처

Controller → Service → Repository

관심사 분리로 정리 정돈

"코드를 체계적으로 정리하기"

Day 2: 확장 구조

MSA 마이크로서비스

API Gateway 설계

서비스 간 통신 (REST/Event)

"서비스를 독립적으로 분리"

Day 3: 운영과 유지보수

클라우드 배포 전략

Docker 컨테이너화

모니터링 & CI/CD

"서비스를 세상에 내보내기"

정리

오늘의 학습 목표 5가지

1. **클라우드 배포 전략 이해**
   AWS, GCP, Azure — 서비스를 어디에 배포할 것인가?
2. **Docker 컨테이너화**
   코드 + 환경을 하나의 상자에 담아 어디서든 실행
3. **모니터링 & 로깅 설계**
   고객보다 먼저 문제를 발견하는 시스템
4. **CI/CD 파이프라인 구축**
   코드 변경 → 자동 테스트 → 자동 배포
5. **유지보수 전략과 기술부채 관리**
   오래 살아남는 서비스를 만드는 전략

3일차 타임테이블

오전 (09:00 ~ 12:00)

09:00-09:20 오프닝 & 1~2일차 복습
09:20-10:20 SECTION 01: 클라우드와 배포 설계
10:20-10:30 휴식 10분
10:30-12:00 SECTION 02: Docker 컨테이너 설계

오후 (13:00 ~ 18:00)

13:00-14:30 SECTION 03: 모니터링 설계
14:30-16:00 SECTION 04: CI/CD 파이프라인 설계
16:00-17:00 SECTION 05: 유지보수 설계
17:00-18:00 SECTION 06: 최종 프로젝트 & 마무리

SECTION 01

서비스를 어디에 배포할 것인가?

클라우드 배포 전략 — 서비스 설계의 첫 번째 결정

On-Premise vs Cloud — 자가용 vs 렌트카

On-Premise (자가용)

서버를 직접 구매하고 관리

장점:
- 완전한 통제권
- 데이터 보안 직접 관리

단점:
- 초기 비용 수천만원~수억원
- 서버실 관리, 냉각, 전기세
- 트래픽 증가 시 서버 추가 구매

Cloud (렌트카)

"서버를 직접 사지 않고, 빌려 쓴다"

장점:
- 초기 비용 0원, 쓴 만큼만 과금
- 클릭 몇 번으로 서버 증설
- 전 세계 리전에 배포 가능

단점:
- 장기적으로 비용 증가 가능
- 클라우드 종속(Vendor Lock-in)

클라우드 3대장: AWS / GCP / Azure

한국 기업들은 어디를 선택했을까?

AWS (Amazon)

- 시장 점유율 1위 (32%)
- 가장 많은 서비스 종류
- 쿠팡, 배달의민족 사용
- 한국 리전: 서울
- 취업 시 가장 많이 요구됨

GCP (Google)

- 데이터/AI 분야 강점
- Kubernetes 원조
- 당근마켓, 토스 사용
- BigQuery 데이터 분석
- 스타트업 크레딧 지원

Azure (Microsoft)

- 기업용 솔루션 강점
- MS 생태계 통합
- 삼성SDS, LG CNS 사용
- Active Directory 연동
- 대기업 프로젝트 다수

IaaS / PaaS / SaaS — 피자 비유

클라우드 서비스의 3가지 모델

IaaS — 재료만 줌

피자 반죽+치즈+오븐 제공

직접 만들어 먹어야 함

예: AWS EC2, GCP Compute

자유도 높음 / 관리 많음

"직접 요리하는 셰프"

PaaS — 주방까지 제공

주방+레시피+재료 제공

요리만 하면 됨

예: Heroku, AWS Elastic Beanstalk

자유도 중간 / 관리 적음

"밀키트 요리"

SaaS — 배달 완제품

완성된 피자 배달

먹기만 하면 됨

예: Gmail, Slack, Notion

자유도 낮음 / 관리 없음

"배달의민족 주문"

Serverless — 서버를 관리하지 않는 아키텍처

Serverless = "서버가 없다"가 아니라 "서버 관리를 안 해도 된다"

핵심 개념:

- 이벤트가 발생할 때만 코드가 실행됨 (이벤트 기반)
- 실행 시간만큼만 과금 (초 단위 과금)
- 서버 프로비저닝, 패치, 스케일링 모두 자동

대표 서비스: AWS Lambda, GCP Cloud Functions, Azure Functions

비유로 이해하기:

자가용 = On-Premise (항상 유지비 발생)
렌트카 = Cloud EC2 (빌려 쓰지만 직접 운전)
택시 = Serverless (탈 때만 비용, 관리 불필요)

Serverless 장단점 & 활용 사례

장점

- 서버 관리 불필요 (NoOps)
- 자동 스케일링 (0 → 무한)
- 실행한 만큼만 비용 (비용 효율적)
- 빠른 개발 & 배포

활용 사례:

- 이미지 리사이징 (S3 업로드 → 자동 변환)
- 푸시 알림 발송 (이벤트 트리거)
- 정기 작업 스케줄링 (매일 오전 9시)

단점

- Cold Start (첫 실행 시 지연)
- 실행 시간 제한 (15분)
- 디버깅 어려움
- Vendor Lock-in 위험

부적합 사례:

- 장시간 실행 작업 (영상 인코딩)
- WebSocket 실시간 통신
- 고성능 컴퓨팅

"우리 서비스는 어디에 배포해야 할까?"

프로젝트 규모별 배포 전략 추천

학생 프로젝트

추천: Vercel / Railway

비용: 무료 티어 충분

Git push만으로 자동 배포

장점: 설정 거의 불필요

포트폴리오에 바로 활용!

스타트업 (소규모)

추천: AWS EC2 / DigitalOcean

비용: 월 5~50만원

Docker + 직접 관리

장점: 자유도 높음

토스 초기에 EC2 사용

대규모 서비스

추천: AWS EKS / GKE

비용: 월 수백만원~

Kubernetes 오케스트레이션

장점: 자동 스케일링

쿠팡, 배달의민족 사용

배포 환경 설계: 개발 / 스테이징 / 프로덕션

왜 환경을 나누는가? — "실수를 고객에게 보여주지 않기 위해"

Development (개발)

- 개발자가 자유롭게 실험
- 에러가 나도 괜찮음
- 가짜 데이터 사용
- DB: 로컬 / Docker
- DEBUG=true

Staging (검증)

- 프로덕션과 동일한 환경
- 배포 전 최종 테스트
- QA팀이 검증하는 곳
- 실제 데이터와 유사한 데이터
- "리허설 무대"

Production (운영)

- 실제 고객이 사용하는 환경
- 장애 = 매출 손실
- 모니터링 필수
- 환경변수로 설정 관리
- DEBUG=false

SECTION 02

Docker — 서비스를 상자에 담는 기술

"내 컴퓨터에서는 되는데요?" 문제를 해결하는 컨테이너 기술

"내 컴퓨터에서는 되는데요?" — 환경 차이 문제

개발자라면 누구나 한 번쯤 겪는 상황...

A 개발자: Node.js 18 + Mac + M1 칩
B 개발자: Node.js 16 + Windows + Intel
서버: Node.js 14 + Ubuntu Linux

발생하는 문제들:

- npm install 결과가 OS마다 다름
- bcrypt 같은 네이티브 모듈 호환성 문제
- 환경 변수 설정 방법이 OS마다 다름
- Python 버전, 라이브러리 버전 충돌

해결책: 환경 자체를 통째로 패키징하자! → Docker

Docker란? — 코드 + 환경을 하나의 상자에

비유: 이사할 때 짐을 컨테이너에 담아서 통째로 옮기기

일반 이사:
가구 하나하나 포장 → 트럭에 싣기 → 내리기 → 다시 배치
매번 "이거 어디 놓지?" 고민 필요

컨테이너 이사 (Docker):
방 전체를 컨테이너에 담음 → 트럭 → 새 집에 놓기만 하면 끝
어디서든 동일한 환경 보장!

Docker = "코드 + OS + 라이브러리 + 설정"을 하나의 이미지로 패키징
→ 어떤 환경에서든 docker run 한 줄이면 동일하게 실행!

Docker vs 가상머신(VM) 비교

가상머신 (VM)

개념: 하드웨어를 소프트웨어로 시뮬레이션

- 전체 OS를 포함 (Windows, Linux 등)
- 이미지 크기: 수 GB
- 시작 시간: 수 분
- 메모리: 많이 소비
- 격리 수준: 매우 높음

비유: 집 전체를 통째로 옮기기

Docker 컨테이너

개념: OS 커널을 공유하고 앱만 격리

- 앱 + 라이브러리만 포함
- 이미지 크기: 수십 MB
- 시작 시간: 수 초
- 메모리: 적게 소비
- 격리 수준: 적절히 높음

비유: 방만 컨테이너에 담아 옮기기

Docker 핵심 3요소

Image / Container / Registry

Image (이미지)

"레시피"

실행에 필요한 모든 것 포함

읽기 전용 (변경 불가)

Dockerfile로 생성

예: node:18-alpine

Container (컨테이너)

"실제 요리"

이미지를 실행한 인스턴스

읽기/쓰기 가능

여러 개 동시 실행 가능

격리된 프로세스

Registry (저장소)

"레시피 보관함"

이미지를 저장/공유하는 곳

Docker Hub (공개)

AWS ECR (비공개)

npm처럼 push/pull

Dockerfile 작성법 — 기본 구조

```
# Node.js Express API 서버 Dockerfile

# 1. 베이스 이미지 선택 (경량 Alpine Linux)
FROM node:18-alpine

# 2. 작업 디렉토리 설정
WORKDIR /app

# 3. 의존성 파일 먼저 복사 (캐시 활용)
COPY package*.json ./
RUN npm ci --only=production

# 4. 소스 코드 복사
COPY . .

# 5. 포트 노출 & 실행 명령
EXPOSE 3000
CMD ["node", "src/index.js"]
```

TIP: package.json을 먼저 복사하면 코드만 바뀌었을 때 npm install을 다시 안 해도 됨!

Dockerfile 고급 — 멀티스테이지 빌드

```
# 멀티스테이지 빌드: 이미지 크기 최적화 (300MB → 80MB)

# Stage 1: 빌드 단계
FROM node:18-alpine AS builder
WORKDIR /app
COPY package*.json ./
RUN npm ci
COPY . .
RUN npm run build

# Stage 2: 실행 단계 (빌드 결과물만 복사)
FROM node:18-alpine
WORKDIR /app
COPY --from=builder /app/dist ./dist
COPY --from=builder /app/node_modules ./node_modules
EXPOSE 3000
CMD ["node", "dist/index.js"]
```

TIP: .dockerignore에 node_modules, .git, .env 추가 필수! (보안 + 크기 최적화)

Docker 핵심 명령어 정리

이미지 관련

```
docker build -t my-api .          # Dockerfile로 이미지 빌드
docker images                     # 이미지 목록 확인
docker pull node:18-alpine        # 이미지 다운로드
```

컨테이너 관련

```
docker run -d -p 3000:3000 my-api  # 컨테이너 실행
docker ps                          # 실행 중인 컨테이너 확인
docker logs <container-id>         # 로그 확인
docker exec -it <id> /bin/sh       # 컨테이너 내부 접속
docker stop <container-id>         # 컨테이너 정지
```

정리

```
docker system prune                # 미사용 리소스 정리
```

실습: Dockerfile 작성해보기

Express API 서버를 Docker 이미지로 빌드하고 실행

1. **Dockerfile 작성**
   FROM node:18-alpine / WORKDIR / COPY / RUN / EXPOSE / CMD
2. **.dockerignore 작성**
   node_modules, .git, .env, *.log 제외
3. **이미지 빌드**
   docker build -t my-express-api .
4. **컨테이너 실행**
   docker run -d -p 3000:3000 --name api my-express-api
5. **동작 확인**
   curl http://localhost:3000/health → {"status":"ok"}

docker-compose란? — 여러 컨테이너를 한 번에

"API 서버 + DB + Redis를 명령어 한 줄로!"

문제 상황:

1. MongoDB 컨테이너 먼저 실행
2. Redis 컨테이너 실행
3. API 서버 컨테이너 실행 (DB 주소 설정 필요)
4. 프론트엔드 컨테이너 실행

→ 매번 4개 명령어를 순서대로 실행? 비효율적!

해결: docker-compose

docker compose up -d ← 이 한 줄이면 전부 실행!

- YAML 파일 하나로 여러 컨테이너 정의
- 네트워크, 볼륨, 환경변수 한 곳에서 관리
- 개발 환경 셋업 시간: 2시간 → 5분

docker-compose.yml 코드 예시 (API + MongoDB)

```yaml
version: "3.8"
services:
  api:
    build: .
    ports:
      - "3000:3000"
    environment:
      - MONGO_URL=mongodb://mongo:27017/mydb
      - NODE_ENV=development
    depends_on:
      - mongo

  mongo:
    image: mongo:7
    ports:
      - "27017:27017"
    volumes:
      - mongo-data:/data/db

volumes:
  mongo-data:    # 데이터 영속성 보장
```

TIP: depends_on으로 시작 순서 지정, volumes로 데이터 유지!

docker-compose 핵심 명령어

시작 & 종료

```
docker compose up -d          # 모든 서비스 백그라운드 실행
docker compose down           # 모든 서비스 정지 & 삭제
docker compose up --build     # 이미지 다시 빌드 후 실행
```

모니터링

```
docker compose ps             # 서비스 상태 확인
docker compose logs -f api    # api 서비스 로그 실시간 확인
docker compose top            # 프로세스 목록 확인
```

개별 제어

```
docker compose restart api    # api 서비스만 재시작
docker compose exec api sh    # api 컨테이너 접속
docker compose stop mongo     # mongo만 정지
```

실습: docker-compose로 서비스 배포

Express API + MongoDB를 docker-compose로 한 번에 실행

1. **docker-compose.yml 작성**
   api 서비스 + mongo 서비스 + volume 정의
2. **docker compose up -d**
   모든 서비스 백그라운드 실행
3. **docker compose ps**
   서비스 상태 확인 (모두 running인지)
4. **API 동작 확인**
   curl http://localhost:3000/users → MongoDB 연동 확인
5. **docker compose down**
   모든 서비스 정리하기

Docker 네트워킹 — 컨테이너 간 통신

컨테이너들은 어떻게 서로 통신할까?

bridge 네트워크 (기본)

- docker-compose의 서비스들은 자동으로 같은 네트워크에 연결
- 서비스 이름으로 통신 가능: mongo:27017 (IP 주소 불필요!)

host 네트워크

- 컨테이너가 호스트 네트워크를 직접 사용
- 포트 매핑 불필요, 성능 우수

핵심 포인트:

docker-compose에서 서비스 이름 = DNS 이름

api 서비스에서 MongoDB 접속 시:

mongodb://mongo:27017/mydb (IP 대신 "mongo" 사용)

Docker로 개발 환경 통일하기 — 팀 협업의 핵심

"docker compose up 한 줄로 모든 팀원이 동일한 환경"

Docker 없는 팀 (Before):
- 신입: "환경 세팅 3일째인데 아직 안 돼요..."
- 리드: "Node 버전 뭐 쓰세요? npm 버전은요?"
- QA: "이 버그 재현이 안 되는데요?"

Docker 쓰는 팀 (After):
- 신입: "git clone → docker compose up → 바로 개발 시작!"
- 리드: "README에 docker compose up 적어두면 끝"
- QA: "같은 환경이니 버그 재현 100%"

쿠팡, 배달의민족, 토스 모두 Docker 기반 개발 환경 사용!

면접 TIP: "Docker로 팀 개발환경 통일 경험" → 강력한 어필 포인트

Kubernetes 기초 — Docker에서 K8s로

컨테이너가 많아지면? 오케스트레이션이 필요하다

⚠ Docker만의 한계

컨테이너 100개를 수동 관리?
컨테이너 죽으면 자동 재시작 없음
트래픽 증가 시 수동 확장만 가능
서버 간 로드밸런싱 직접 설정 필요

✅ Kubernetes 해결

Auto Healing: 죽으면 자동 재시작
Auto Scaling: 트래픽 따라 자동 확장
Load Balancing: 자동 트래픽 분산
Rolling Deploy: 무중단 배포 기본 제공

K8s 핵심 개념: Pod(컨테이너 단위) > Deployment(배포 단위) > Service(네트워크) > Ingress(외부 트래픽)
취업 현장: 쿠팡, 배달의민족, 토스 모두 K8s(EKS/GKE) 운영 중 — 기본 개념 숙지 필수

면접 TIP: "Docker로 컨테이너화하고 K8s로 오케스트레이션" — 이 흐름을 말할 수 있으면 큰 가산점!

SECTION 03

서비스 모니터링 — 고객보다 먼저 문제를 발견하기

운영의 핵심: 보이지 않으면 관리할 수 없다

왜 모니터링인가? — 실제 장애 시나리오

실제 장애 시나리오 3가지

1. 서버 다운

고객: "사이트 접속이 안 돼요!" → 트위터에 장애 공유
개발팀: 고객 문의 보고 30분 뒤에야 인지 → 매출 손실

2. 응답 느려짐 (레이턴시 증가)

평소 200ms → 갑자기 5000ms → 사용자 이탈율 급증
DB 쿼리가 느려진 건지, 서버 CPU가 부족한 건지 모름

3. 에러 급증

배포 후 500 에러가 100배 증가 → 원인 모르고 롤백
어떤 API에서, 어떤 사용자에게, 왜 에러가 나는지 로그가 없음

모니터링이 있었다면? → 고객 문의 전에 자동 감지 & 알림!

모니터링의 3가지 기둥

Observability(관측 가능성)의 핵심 요소

Metrics (지표)

"서비스 체온/혈압"

CPU, 메모리, 응답시간

숫자로 된 시계열 데이터

이상 수치 감지 → 알림

Prometheus로 수집

Logs (로그)

"서비스의 블랙박스"

무슨 일이 일어났는지 기록

에러 원인 추적

구조화된 JSON 로그

Winston/Morgan 활용

Traces (추적)

"요청의 여행 경로"

MSA에서 요청 흐름 추적

A서비스→B서비스→DB

어디서 느려지는지 파악

X-Request-Id 활용

Metrics — 서비스 건강 지표

서비스의 "체온/혈압"을 측정하는 숫자 지표

시스템 지표:

- CPU 사용률: 80% 이상 → 경고, 95% → 위험
- 메모리 사용량: 메모리 누수 감지
- 디스크 사용량: 로그 파일로 디스크 풀(full) 방지

애플리케이션 지표 — RED 방법론:

R (Rate): 초당 요청 수 — 트래픽 변화 감지
E (Errors): 에러율 — 배포 후 에러 급증 감지
D (Duration): 응답 시간 — 성능 저하 감지

쿠팡: "주문 API 응답시간 P99 < 500ms" 목표로 모니터링
배달의민족: "에러율 0.1% 초과 시 즉시 알림" 설정

Prometheus + Grafana — 지표 수집 & 시각화

Prometheus (수집기)

"지표를 수집하는 데이터 수집가"

- 서버의 /metrics 엔드포인트를 주기적으로 수집
- 시계열 데이터베이스(TSDB) 저장
- PromQL로 쿼리 가능

수집 예시:

```
http_requests_total{method="GET"}
http_request_duration_seconds
```

Grafana (시각화)

"지표를 예쁜 대시보드로"

- Prometheus 데이터를 그래프로 시각화
- 실시간 대시보드 구성
- 알림(Alert) 규칙 설정 가능

대시보드 패널:

요청수/초, 에러율, 응답시간
CPU/메모리/디스크 사용량

Logs — 서비스의 블랙박스

"비행기에 블랙박스가 있듯, 서비스에는 로그가 있어야 한다"

비행기 블랙박스:
→ 사고 발생 시 원인을 추적하기 위한 기록 장치
→ 모든 대화, 계기판 데이터, 조작 기록 저장

서비스 로그:
→ 장애 발생 시 원인을 추적하기 위한 기록
→ 누가, 언제, 어떤 API를, 어떤 데이터로 호출했는지

나쁜 로그:
console.log("여기 왔음") ← 쓸모없는 정보

좋은 로그 (구조화된 JSON):

```
{"level":"error","message":"결제 실패",
 "userId":"U123","orderId":"O456","reason":"잔액부족"}
```

로그 레벨: debug / info / warn / error / fatal

로그 레벨 = "얼마나 심각한가?" 에 따라 분류

DEBUG — 개발 중 디버깅 정보
예: "쿼리 실행: SELECT * FROM users WHERE id=1"
프로덕션에서는 비활성화 (성능 이슈)

INFO — 정상 동작 기록
예: "사용자 U123 로그인 성공" / "주문 O456 생성됨"

WARN — 잠재적 문제 (아직 에러는 아님)
예: "API 응답시간 3초 초과" / "디스크 80% 사용"

ERROR — 에러 발생 (서비스는 계속 동작)
예: "결제 API 호출 실패" / "DB 연결 타임아웃"

FATAL — 서비스 중단 수준의 치명적 에러
예: "DB 연결 불가" / "필수 환경변수 누락"

Winston + Morgan으로 구조화된 로그

```
// logger.js — Winston 설정
const winston = require("winston");

const logger = winston.createLogger({
  level: process.env.LOG_LEVEL || "info",
  format: winston.format.combine(
    winston.format.timestamp(),
    winston.format.json()
  ),
  transports: [
    new winston.transports.File({ filename: "error.log", level: "error" }),
    new winston.transports.File({ filename: "combined.log" }),
  ]
});

// 사용 예시
logger.info("주문 생성", { userId: "U123", orderId: "O456" });
logger.error("결제 실패", { orderId: "O456", reason: "잔액 부족" });
```

TIP: Morgan은 HTTP 요청 로그 자동 기록, Winston은 애플리케이션 로그 관리

Traces — 분산 추적 (MSA에서 요청 흐름)

하나의 요청이 여러 서비스를 거칠 때 — 어디서 느려졌나?

배달의민족 주문 흐름:

사용자 → API Gateway → 주문서비스 → 결제서비스 → 알림서비스

총 응답시간: 3초 — 어디서 병목인지 어떻게 알지?

X-Request-Id로 추적:

모든 서비스가 동일한 요청 ID를 공유

X-Request-Id: "abc-123-def"

```
Gateway:  [abc-123-def] 요청 수신        +0ms
주문서비스: [abc-123-def] 주문 생성       +50ms
결제서비스: [abc-123-def] PG사 호출       +2500ms  ← 병목!
알림서비스: [abc-123-def] 푸시 발송       +200ms
```

도구: Jaeger, Zipkin, AWS X-Ray

Alerts — 이상 감지 시 자동 알림

"CPU 90% 초과 → Slack 알림 → 즉시 대응"

알림 설계 원칙:

1. 중요한 것만 알림 (알림 피로 방지)
2. 단계별 알림 (Warning → Critical → Emergency)
3. 알림에 충분한 정보 포함 (무엇이, 어디서, 얼마나)

알림 채널 설계:

Warning: Slack 채널 알림
Critical: Slack + 이메일 + PagerDuty
Emergency: Slack + 이메일 + 전화 (당직자)

Slack 웹훅 활용:

POST https://hooks.slack.com/services/...
{ "text": "[ALERT] 주문서비스 에러율 5% 초과!" }

Health Check 엔드포인트 설계

```
// health.js — 서비스 건강 상태 체크
const express = require("express");
const mongoose = require("mongoose");
const router = express.Router();

router.get("/health", async (req, res) => {
  const health = {
    status: "ok",
    timestamp: new Date().toISOString(),
    uptime: process.uptime(),
```

```
// health.js — 서비스 건강 상태 체크
checks: {
      database: mongoose.connection.readyState === 1
        ? "connected" : "disconnected",
      memory: process.memoryUsage().heapUsed
    }
  };
  const statusCode = health.checks.database === "connected" ? 200 : 503;
  res.status(statusCode).json(health);
});
```

TIP: Load Balancer가 /health를 주기적으로 호출하여 서버 상태를 판단합니다

실습: Morgan + Winston 로그 설정

Express 서버에 구조화된 로깅 시스템 추가하기

1. **Winston 설치 & 설정**
   npm install winston → logger.js 작성 (JSON 포맷, 파일 출력)
2. **Morgan 설치 & 연동**
   npm install morgan → HTTP 요청 로그를 Winston으로 전달
3. **로그 레벨별 테스트**
   logger.info(), logger.warn(), logger.error() 호출해보기
4. **로그 파일 확인**
   combined.log, error.log 파일 내용 확인
5. **환경별 로그 레벨 설정**
   개발: debug / 프로덕션: info (환경변수로 제어)

실습: Health Check 엔드포인트 작성

```
// 1. 기본 Health Check
app.get("/health", (req, res) => {
  res.json({ status: "ok", timestamp: new Date() });
});

// 2. 상세 Health Check (DB 연결 포함)
app.get("/health/detailed", async (req, res) => {
  try {
    await mongoose.connection.db.admin().ping();
    res.json({
      status: "healthy",
      database: "connected",
      uptime: process.uptime() + "s",
     memory: Math.round(process.memoryUsage().heapUsed/1024/1024) + "MB"
    });
  } catch (err) {
    res.status(503).json({ status: "unhealthy", error: err.message });
  }
});
```

TIP: /health는 200만 반환하는 단순 체크, /health/detailed은 의존성까지 확인

ELK Stack — 로그 중앙화 아키텍처

수백 개 서버 로그를 한 곳에서 검색하고 분석하기

E

Elasticsearch

로그 저장 및 검색엔진
"에러" 검색 → 즉시 결과
분산 저장, 실시간 색인

L

Logstash

로그 수집 및 변환(ETL)
다양한 포맷 → 통일 형식
필터링, 파싱, 강화

K

Kibana

시각화 & 대시보드 UI
로그 검색 UI 제공
알림, 이상 감지 기능

서비스 로그 (Winston/Morgan) → Logstash 수집 → Elasticsearch 저장 → Kibana 검색/시각화

대안 (경량): Loki + Grafana — K8s 환경에서 더 가볍고 인기 상승 중

면접 TIP: "ELK Stack으로 로그 중앙화 → 장애 원인 추적 시간 2시간 → 5분으로 단축" — 수치로 말하면 강력!

SLI / SLO / SLA — 운영 품질 지표 설계

"우리 서비스 몇 % 가용성을 보장하나요?" — 숫자로 답할 수 있어야 한다

SLI

Service Level Indicator

측정하는 실제 지표

예: 응답시간 200ms 이하
예: 에러율 0.1% 미만
예: 가용성 99.9% 이상

SLO

Service Level Objective

내부 목표값

예: 월 가용성 99.9%
예: P99 응답시간 500ms
Error Budget으로 관리

SLA

Service Level Agreement

고객과의 계약

위반 시 환불/페널티
AWS: 99.99% 보장
SLO보다 느슨하게 설정

가용성 해석: 99.9%(Three Nines) = 연간 8.7시간 다운타임 허용 | 99.99% = 52분 | 99.999%(Five Nines) = 5분
Error Budget = 1 - SLO. SLO 99.9% = 월 43.8분 허용 오류 예산 — 소진 시 새 기능 배포 중단

면접 TIP: "SLI 측정 → SLO 목표 → SLA 계약" 구조 설명 시, Google SRE Book의 개념으로 어필 가능

SECTION 04

CI/CD — 자동으로 배포하는 파이프라인 설계

"코드를 push하면 자동으로 테스트하고 배포한다"

CI(Continuous Integration) — 지속적 통합

"코드를 push하면 자동으로 테스트가 돌아간다"

CI가 하는 일:

1. 코드 변경 감지 (git push)
2. 자동 빌드 (npm run build)
3. 자동 테스트 (npm test)
4. 코드 품질 검사 (ESLint, 타입 체크)
5. 결과 알림 (성공/실패 → PR에 표시)

왜 필요한가?

- 버그를 빨리 발견: 코드 push 후 5분 내 피드백
- 코드 품질 유지: 테스트 통과해야 merge 가능
- 팀 신뢰: "이 코드는 테스트를 통과했으니 안전해요"

CD — Continuous Delivery vs Deployment

Continuous Delivery (지속적 전달)

"배포할 준비까지 자동화"

코드 push
→ 자동 빌드
→ 자동 테스트
→ 스테이징 배포
→ 수동 승인 후 프로덕션 배포

대부분의 기업이 채택
안전한 배포를 위해 최종 승인 필요

Continuous Deployment (지속적 배포)

"프로덕션까지 완전 자동화"

코드 push
→ 자동 빌드
→ 자동 테스트
→ 자동 프로덕션 배포

매우 높은 테스트 커버리지 필요
넷플릭스, 아마존이 사용
하루 수십~수백 번 배포!

CI/CD가 없으면? — 수동 배포의 공포

"금요일 밤 수동 배포 후 장애... 주말이 사라진다"

수동 배포 과정:

1. 로컬에서 빌드 (npm run build)
2. FTP로 서버에 파일 업로드
3. SSH로 서버 접속해서 서비스 재시작
4. 브라우저로 직접 확인
5. 문제 발생 → 다시 이전 버전 수동 복구...

실제 발생하는 문제들:

- "테스트 안 돌리고 배포했다가 서비스 다운"
- "A 개발자가 빌드하면 되는데 B가 하면 안 됨"
- "배포 방법을 A만 알고 있는데 A가 퇴사..."
- 배포 담당자 = 팀에서 가장 불행한 사람

CI/CD = 배포를 코드로 자동화 → 누구나 안전하게 배포!

CI/CD 도구 비교

어떤 도구를 선택할까?

GitHub Actions

추천: 학생/스타트업

GitHub 저장소와 완벽 통합

YAML로 쉬운 설정

무료 2000분/월

마켓플레이스 액션 풍부

Jenkins

추천: 대기업/레거시

가장 오래된 CI/CD 도구

플러그인 1800+개

자체 서버 필요 (관리 부담)

삼성, LG 등 사용

GitLab CI

추천: GitLab 사용 팀

GitLab 내장 CI/CD

.gitlab-ci.yml 설정

Docker 기반 실행

네이버 일부 팀 사용

GitHub Actions 기본 구조

```yaml
# .github/workflows/ci.yml
name: CI Pipeline          # 워크플로우 이름

on:                        # 트리거 조건
  push:
    branches: [main, develop]  # main/develop push 시
  pull_request:
    branches: [main]           # main PR 시

jobs:                      # 실행할 작업들
  test:
    runs-on: ubuntu-latest  # 실행 환경
    steps:                  # 단계별 실행
      - uses: actions/checkout@v4  # 코드 체크아웃
      - uses: actions/setup-node@v4
        with:
          node-version: 18
      - run: npm ci             # 의존성 설치
      - run: npm test           # 테스트 실행
```

TIP: .github/workflows/ 디렉토리에 YAML 파일을 추가하면 자동으로 인식됩니다!

GitHub Actions 실전: test → build → deploy

```yaml
jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - run: npm ci && npm test

  build:
    needs: test              # test 성공 후 실행
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - run: docker build -t my-api:${{ github.sha }} .
      - run: docker push my-registry/my-api:${{ github.sha }}

  deploy:
    needs: build             # build 성공 후 실행
    if: github.ref == 'refs/heads/main'   # main만 배포
    runs-on: ubuntu-latest
    steps:
      - run: kubectl set image deployment/api api=my-api:${{ github.sha }}
```

TIP: needs로 작업 순서 지정, if로 조건부 실행 (main 브랜치만 배포)

배포 전략 3가지

서비스 중단 없이 새 버전을 배포하는 방법

Rolling Deploy

"하나씩 교체"

서버 1대씩 순차 업데이트

장점: 리소스 효율적

단점: 신/구 버전 공존 시간

적합: 소규모 서비스

쿠버네티스 기본 전략

Blue-Green Deploy

"한 번에 전환"

구버전(Blue) + 신버전(Green)

테스트 후 트래픽 전환

장점: 즉시 롤백 가능

단점: 2배 리소스 필요

배달의민족 사용

Canary Deploy

"소수에게 먼저"

5% 사용자에게 신버전

문제 없으면 점차 확대

장점: 위험 최소화

단점: 복잡한 설정

네이버/카카오 사용

배포 전략 상세 비교 — 언제 어떤 전략을 쓸까?

상황별 배포 전략 선택 가이드

학생 프로젝트 / 사이드 프로젝트:
→ Rolling Deploy (간단, Vercel/Railway 기본 제공)
→ 또는 그냥 재배포 (다운타임 허용)

스타트업 (사용자 수천~수만):
→ Blue-Green Deploy (빠른 롤백이 중요)
→ 토스 초기: Blue-Green으로 안정성 확보

대규모 서비스 (사용자 수백만):
→ Canary Deploy (위험 최소화)
→ 카카오: 1% → 5% → 25% → 50% → 100% 점진 배포
→ 네이버: A/B 테스트와 결합한 Canary 배포

면접 TIP: "프로젝트에서 어떤 배포 전략을 사용했고 왜 선택했는지"

실습: GitHub Actions 워크플로우 작성

Node.js 프로젝트에 CI 파이프라인 구축하기

1. **.github/workflows/ci.yml 생성**
   on: push/pull_request 트리거 설정
2. **test job 작성**
   checkout → setup-node → npm ci → npm test
3. **lint job 추가**
   npm run lint (ESLint 코드 품질 검사)
4. **build job 추가**
   needs: [test, lint] → Docker 이미지 빌드
5. **PR에서 확인**
   GitHub PR 페이지에서 CI 결과 확인 (초록 체크마크)

Graceful Shutdown — 무중단 재시작 설계

배포 중 진행 중인 요청을 안전하게 완료하고 종료하기

⚠ 강제 종료 시 문제

처리 중인 HTTP 요청 끊김
DB 트랜잭션 롤백 미처리
사용자: 500 에러 또는 응답없음
메시지 큐 메시지 유실 위험

발생 시나리오

K8s 배포 → 구 Pod 즉시 kill
결제 처리 중 → 중단 → 이중 결제

// Graceful Shutdown 구현

```
const server = app.listen(3000);

// SIGTERM: K8s/Docker 종료 신호
process.on('SIGTERM', () => {
  // 새 요청 수락 중단
  server.close(async () => {
    // DB 연결 정리
    await db.disconnect();
    process.exit(0); // 정상 종료
  });
});
```

K8s terminationGracePeriodSeconds: 30 — 30초 내에 종료 완료해야 함 (기본값)

Rollback 전략 — 배포 실패 시 빠른 복구

"빠른 롤백이 빠른 개발보다 중요하다" — 장애 복구 MTTR 최소화

📦 Git Revert

이전 커밋으로 복구

`git revert HEAD~1`

장점: 기록 보존

단점: CI/CD 재실행 필요

소규모 프로젝트 적합

🐳 Docker 이미지 롤백

이전 이미지 태그로 배포

`my-api:v1.2.3 (이전)`

장점: 빠른 롤백 (분 단위)

태그 관리 전략 필요

스타트업 권장 방식

☁ K8s Rollout

K8s 내장 롤백 기능

`kubectl rollout undo`

장점: 명령어 한 줄

히스토리 10개 보관

대규모 서비스 표준

롤백의 전제조건: DB 스키마 하위 호환성 유지 + 이전 이미지 태그 보존 + 롤백 판단 기준(에러율 임계값) 사전 정의

배달의민족: 배포 후 5분 이내 에러율 2% 초과 시 자동 롤백 트리거

항상 이전 버전 이미지 태그를 보존할 것 — latest만 쓰면 롤백 불가!

SECTION 05

서비스 유지보수 — 오래 살아남는 서비스 만들기

기술 부채 관리, 코드 리뷰, 보안 — 운영의 지속 가능성

기술 부채(Technical Debt) — "나중에 고치자"의 대가

"신용카드 빚처럼 이자가 쌓인다"

기술 부채란?

빠른 개발을 위해 품질을 희생한 코드 → 나중에 갚아야 할 "빚"

흔한 기술 부채 5가지:

1. 하드코딩된 설정값 (DB URL, API 키를 코드에 직접 작성)
2. 테스트 없는 코드 (수정할 때마다 불안함)
3. 복붙된 코드 (비슷한 로직이 5곳에 흩어져 있음)
4. 문서 없는 API (만든 사람만 사용법을 앎)
5. 오래된 의존성 (보안 취약점 방치)

관리 방법:

- 매 스프린트 20%는 기술 부채 해소에 투자
- TODO 주석 → Issue 등록 → 스프린트에 반영
- "완벽하게 만들기" 보다 "관리 가능하게 만들기"

코드 리뷰 문화 — "혼자 짠 코드는 위험하다"

좋은 코드 리뷰

PR(Pull Request) 기반 프로세스:

1. feature 브랜치에서 작업
2. PR 생성 (변경 설명 작성)
3. 팀원 2명 이상 리뷰
4. CI 통과 확인
5. Approve 후 merge

좋은 리뷰 코멘트:

"이 함수가 너무 길어서 분리하면
테스트하기 쉬울 것 같아요"

나쁜 코드 리뷰

흔한 안티패턴:

"왜 이렇게 짰어요?"
→ 비판만, 대안 없음

"LGTM" (한 줄 승인)
→ 실질적 리뷰 없음

"전부 다 고쳐주세요"
→ 한 번에 너무 많은 지적

카카오: PR 크기 300줄 이내 권장
토스: "작은 PR, 빠른 리뷰" 문화

API 버전 관리 — 변경에 유연한 설계

API를 수정할 때, 기존 사용자는 어떡하지?

버전 관리 방법 3가지:

1. URI Path (가장 많이 사용)

GET /api/v1/users → GET /api/v2/users

장점: 직관적, 캐싱 쉬움 / 카카오, 네이버 사용

2. Header 방식

Accept: application/vnd.myapi.v2+json

장점: URL 깔끔 / GitHub API 사용

3. Query Parameter

GET /api/users?version=2

장점: 간단 / 단점: 캐싱 어려움

Deprecation 전략: Sunset 헤더

Sunset: Sat, 01 Jan 2027 00:00:00 GMT

→ "이 API는 2027년에 종료됩니다" 사전 공지

보안 설계 체크리스트

"보안은 선택이 아닌 필수 — 해킹당하면 서비스 끝"

1. CORS 설정 — 허용된 도메인만 API 접근

```
app.use(cors({ origin: "https://mysite.com" }))
```

2. Helmet — HTTP 보안 헤더 자동 설정

```
app.use(helmet())  // XSS, Clickjacking 방지
```

3. Rate Limiting — 무차별 공격 방지

```
app.use(rateLimit({ windowMs: 15*60*1000, max: 100 }))
```

4. JWT — 안전한 인증 토큰 (만료시간 설정 필수)

5. 환경변수 — DB 비밀번호, API 키는 .env로 관리

절대 코드에 하드코딩하지 말 것!

6. SQL Injection 방지 — ORM 사용 또는 파라미터 바인딩

서비스 운영 체크리스트 총정리

배포 전 확인사항

코드 관련:

- 모든 테스트 통과 확인
- 코드 리뷰 완료 (Approve)
- 환경변수 설정 확인
- DB 마이그레이션 준비

인프라 관련:

- 스테이징 테스트 완료
- 롤백 계획 수립
- 배포 시간대 공유 (팀)

배포 후 확인사항

즉시 확인:

- Health Check 정상 응답
- 핵심 API 동작 확인
- 에러 로그 모니터링
- 응답 시간 정상 범위 확인

30분 후 확인:

- 에러율 변화 없는지
- 사용자 문의 확인
- 배포 완료 공유 (팀)

환경변수 & Secret 관리 전략

DB 비밀번호가 GitHub에 올라가면? — 실제로 일어나는 보안 사고

❌ 절대 하면 안 되는 것

```
// app.js
const DB_PASS = "mypassword123"
const API_KEY = "sk-abc123xyz"
```

GitHub 공개 저장소에 push 시 즉시 해킹

✅ 올바른 방법 (.env)

```
# .env (git에 포함 금지!)
DB_PASS=mypassword123
API_KEY=sk-abc123xyz
```

.gitignore에 .env 추가 필수

환경별 Secret 관리 전략

개발: .env 파일 (로컬) | **스테이징/프로덕션:** GitHub Actions Secrets, AWS Parameter Store | **엔터프라이즈:** HashiCorp Vault, AWS Secrets Manager

```
# GitHub Actions에서 Secret 사용
env:
  DB_PASS: ${{ secrets.DB_PASSWORD }}  # Settings > Secrets에 등록
```

보안 TIP: git-secrets, truffleHog 도구로 커밋 전 자동 시크릿 탐지 가능 — pre-commit hook에 추가!

DB 마이그레이션 전략 — 무중단 스키마 변경

서비스 중단 없이 DB 스키마를 안전하게 변경하는 법

컬럼명 변경 시 무중단 마이그레이션 — 3단계 전략 (Expand-Contract)

1단계 — Expand (확장): 새 컬럼 추가

```
ALTER TABLE users ADD COLUMN full_name VARCHAR(100);
```

앱 코드: 구 컬럼(name)과 신 컬럼(full_name) 동시 지원 배포

2단계 — Migrate (데이터 이전): 기존 데이터 복사

```
UPDATE users SET full_name = name WHERE full_name IS NULL;
```

3단계 — Contract (축소): 구 컬럼 제거

```
ALTER TABLE users DROP COLUMN name;
```

마이그레이션 도구: Flyway (Java) | Liquibase | Knex.js migrate (Node.js) | Prisma migrate | Sequelize migrations

마이그레이션 파일은 버전 관리 — 한 번 실행된 파일 수정 금지! 새 마이그레이션 파일 추가

NOT NULL 컬럼 추가 시 반드시 DEFAULT 값 지정 — 기존 데이터 없으면 마이그레이션 실패!

의존성 보안 관리 — npm audit & Dependabot

오래된 패키지 = 알려진 취약점 = 해킹 문 열어두기

🔍 npm audit — 취약점 스캔

```
$ npm audit
found 3 vulnerabilities
  1 critical, 2 high
$ npm audit fix  # 자동 수정
```

심각도 분류

Critical: 즉시 업데이트 필수
High: 빠른 시일 내 업데이트
Moderate/Low: 일정 조율

🤖 Dependabot — 자동 PR

GitHub 내장 자동 업데이트 봇
취약점 발견 시 자동 PR 생성
무료로 활성화 가능

설정 방법

```
# .github/dependabot.yml
version: 2
updates:
  - package-ecosystem: "npm"
    directory: "/"
    schedule:
      interval: "weekly"
```

CI 파이프라인에 npm audit --audit-level=high 추가 → High 이상 취약점 있으면 배포 자동 차단!

SECTION 06

최종 프로젝트 — 3일간 배운 모든 것을 하나로!

설계 → 구현 → 운영까지 풀사이클 프로젝트

최종 프로젝트 소개 (팀 3~4인)

3일간 배운 모든 개념을 적용하는 팀 프로젝트

주제 (자유 선택):

A. 쇼핑몰 서비스 — 상품/주문/결제
B. SNS 서비스 — 게시글/팔로우/피드
C. 예약 시스템 — 식당/병원/미용실 예약
D. 자유 주제 — 팀이 정한 서비스

팀 구성:

- 3~4인 1팀 (역할 분담 자유)
- 아키텍트 / API 설계자 / DevOps 담당 / 문서화 담당

산출물: 아키텍처 설계도 + API 설계서 + Docker 설정 + 발표 자료
(코드 구현은 선택, 설계 문서가 핵심!)

필수 요구사항 상세

1. **3계층 설계 (Day 1)**
   Controller → Service → Repository 구조 / 각 계층 역할 명확히 분리
2. **REST API 5개 이상 설계 (Day 1-2)**
   CRUD + 추가 기능 / Swagger 또는 API 문서 작성
3. **MSA 분리 포인트 제시 (Day 2)**
   어떤 서비스를 왜 분리하는지 근거 / 서비스 간 통신 방식 설계
4. **Docker 설정 (Day 3)**
   Dockerfile + docker-compose.yml 작성 / 환경변수 관리
5. **모니터링 & 운영 전략 (Day 3)**
   Health Check, 로그 설계, 알림 설정 / 배포 전략 선택 근거

발표 평가 기준

4가지 관점에서 평가합니다

아키텍처 설계 (30%)

- 3계층 구조 적용 여부
- MSA 분리 근거의 타당성
- 데이터 흐름 설명
- ERD / 시스템 다이어그램

API 설계 (25%)

- RESTful 원칙 준수
- URL/Method 설계 적절성
- 에러 처리 설계
- API 문서 완성도

운영 전략 (25%+20%)

- Docker 설정 완성도
- 모니터링 설계
- 배포 전략 선택 근거
- CI/CD 파이프라인
- 확장성 고려 (스케일링)

3일 과정 종합 정리

"서비스를 어떻게 설계하고, 어떻게 유지보수해야 하는가?"

Day 1: 기초 설계

- 3계층 아키텍처
- Controller/Service/Repository
- MVC 패턴
- 관심사 분리 원칙
- Express.js 실습
- "코드의 정리 정돈"

Day 2: 확장 설계

- MSA 마이크로서비스
- API Gateway 패턴
- REST API 설계
- 서비스 간 통신
- DB 분리 전략
- "서비스의 독립과 연결"

Day 3: 운영 설계

- 클라우드 배포 전략
- Docker 컨테이너화
- 모니터링 & 로깅
- CI/CD 파이프라인
- 유지보수 전략
- "서비스의 안정과 지속"

추천 학습 자료

더 깊이 공부하고 싶다면?

아키텍처 학습

"클린 아키텍처" - 로버트 마틴

"도메인 주도 설계" - 에릭 에반스

Martin Fowler 블로그

System Design Primer

(GitHub 무료 자료)

백엔드 실습

"Node.js 교과서" - 조현영

Express.js 공식 문서

MongoDB University (무료)

Prisma 공식 튜토리얼

Inflearn 강의 (한국어)

DevOps 입문

Docker 공식 튜토리얼

GitHub Actions 공식 문서

Kubernetes 기초 (44bits)

AWS 프리티어 실습

"DevOps와 SE를 위한
리눅스 커맨드라인"