서비스 설계와 유지보수 5일 과정

2일차: 우리 서비스, 옆집 서버와 대화하기

서비스가 커질 때 서버를 나누는 이유와 대화법

목표

서비스 분리와 연결 설계

오늘의 학습 목표 5가지

1. **1. 서비스를 왜 나누는가**
   모놀리식의 한계와 MSA 도입 이유를 이해합니다
2. **2. MSA의 장단점 파악**
   마이크로서비스 아키텍처의 현실적 Trade-off를 학습합니다
3. **3. API Gateway 설계**
   Nginx를 활용한 통합 진입점 구축 방법을 익힙니다
4. **4. REST API 설계 원칙**
   실무에서 통용되는 API 설계 7대 원칙을 적용합니다
5. **5. 로드밸런싱 이해**
   서비스 안정성을 위한 부하 분산 전략을 학습합니다

오프닝

2일차 타임테이블

09:30 ~ 10:30	SECTION 01 서비스 분리의 이유 (MSA)
10:30 ~ 11:30	SECTION 02 API Gateway 설계
11:30 ~ 12:30	SECTION 03 실습4 — 두 개의 서버 띄우기
12:30 ~ 13:30	점심 휴식
13:30 ~ 14:30	SECTION 04 REST API 설계 원칙
14:30 ~ 15:30	SECTION 05 실습5 — Nginx Gateway 구축
15:30 ~ 16:00	SECTION 06 로드밸런싱 설계
16:00 ~ 17:00	SECTION 07 실습6 — 부하 분산 & 마무리

SECTION 01

서비스가 커지면 왜 나눠야 하는가?

모놀리식의 한계와 MSA의 등장 배경

비유: 백화점 푸드코트

🏪 하나의 큰 주방에서 모든 음식을 만들면?

- 한식 요리사가 쉬면 전체 주방 마비
- 중식 주문이 폭주하면 일식까지 느려짐
- 새 메뉴 추가하려면 전체 주방 리모델링 필요

🏬 푸드코트처럼 나누면?

- 한식/중식/일식 각각 독립적으로 운영 = 마이크로서비스
- 중식 가게만 요리사 추가 가능 = 독립 확장
- 한식 가게가 문 닫아도 중식/일식은 정상 영업 = 장애 격리
- 각 가게가 자기만의 조리법 사용 = 기술 다양성

MSA(Microservice Architecture) 정의

큰 서비스를 작은 서비스 여러 개로 쪼개서
각각 독립적으로 개발·배포·확장하는 아키텍처

실제 도입 기업과 이유

- Netflix — 2012년 데이터센터 장애 후 MSA 전환, 1000+ 마이크로서비스 운영
- 쿠팡 — 주문/배송/결제 서비스 분리로 로켓배송 안정성 확보
- 배달의민족 — 주문 폭주 시 주문 서비스만 스케일아웃
- 토스 — 금융 서비스별 독립 배포로 빠른 기능 출시
- 카카오 — 카카오톡/카카오페이/카카오맵 독립 서비스 운영

💡 **서비스 설계 관점: 처음부터 MSA가 아니라, 성장에 따라 분리!**

모놀리식 vs MSA — 상세 비교

모놀리식 (Monolithic)

배포: 전체를 한 번에 배포
확장: 서버 전체를 복제
장애: 하나 터지면 전체 다운
기술: 하나의 언어/프레임워크
팀: 모든 개발자가 같은 코드
DB: 하나의 데이터베이스

👍 **장점: 초기 개발 빠름**
👍 **장점: 배포가 단순함**
👍 **장점: 트랜잭션 관리 쉬움**

MSA (Microservices)

배포: 서비스별 독립 배포
확장: 필요한 서비스만 확장
장애: 해당 서비스만 영향
기술: 서비스별 자유 선택
팀: 서비스별 담당 팀
DB: 서비스별 독립 DB

👍 **장점: 독립 배포·확장**
👍 **장점: 장애 격리**
👍 **장점: 팀 자율성**

왜 쪼개나? — 4가지 핵심 이유

🚀 독립 배포

주문 서비스 수정 시

결제 서비스 재배포 불필요

쿠팡: 하루 수백회 배포

🛡️ 장애 격리

검색 서비스 장애 시

주문/결제는 정상 운영

배민: 리뷰 장애 ≠ 주문 장애

🔧 기술 다양성

주문: Java + Spring

추천: Python + ML

채팅: Node.js + WebSocket

독립 확장(Scale Out) 실무 — K8s HPA

쿠버네티스(Kubernetes)로 자동 수평 확장하기

```yaml
# HPA (Horizontal Pod Autoscaler) 설정 예시
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: user-service-hpa
spec:
  scaleTargetRef:
    name: user-service    # 확장 대상 서비스
  minReplicas: 2          # 최소 Pod 수
  maxReplicas: 20         # 최대 Pod 수
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        averageUtilization: 60  # CPU 60% 초과 시 자동 증설
```

동작 원리: CPU 60% 초과 → Pod 자동 추가 증설 → 트래픽 분산 → CPU 안정화
배민/쿠팡 점심 피크: 주문 서비스 Pod 2개 → 50개로 자동 확장 후 자동 축소

Nginx Round Robin + K8s HPA = 트래픽 증가에도 무중단 서비스! 쿠팡/배민 실제 운영 방식

쪼개면 생기는 고민 — 분산 시스템의 복잡성

MSA 도입 시 반드시 해결해야 할 과제들

❌ 여러 주소 관리: 프론트엔드가 10개 서비스 주소를 다 알아야 할까?
→ 해결: API Gateway로 단일 진입점 제공

❌ CORS 문제: 서비스마다 포트가 다르면 브라우저가 차단
→ 해결: Gateway에서 통합 CORS 설정

❌ 인증 중복: 모든 서비스에서 토큰 검증 로직 반복?
→ 해결: Gateway에서 인증 처리 후 서비스로 전달

❌ 데이터 일관성: 주문은 성공, 결제는 실패하면?
→ 해결: Saga 패턴, 이벤트 소싱 등 분산 트랜잭션 전략

❌ 디버깅 어려움: 요청이 5개 서비스를 거치면 어디서 에러?
→ 해결: 분산 추적(Distributed Tracing) — Jaeger, Zipkin

Saga 패턴 — 분산 트랜잭션 처리

📋 Choreography (안무) 방식

이벤트 기반 자율 처리
주문생성 → [이벤트] → 결제서비스
결제완료 → [이벤트] → 재고서비스
재고감소 → [이벤트] → 배송서비스

✅ 장점: 중앙 제어 없음, 느슨한 결합
❌ 단점: 흐름 추적 어려움

🎯 Orchestration (오케스트라) 방식

중앙 오케스트레이터가 지휘

```
Saga 오케스트레이터
  ├→ 결제서비스.charge()
  ├→ 재고서비스.decrease()
  └→ 배송서비스.start()
```

✅ 장점: 흐름 명확, 디버깅 쉬움
❌ 단점: 오케스트레이터 복잡도

보상 트랜잭션 (Compensating Transaction) — 실패 시 롤백

```
// 결제 성공 후 재고 감소 실패 시 → 결제 취소(보상) 실행
결제완료 → 재고감소 실패 → 결제취소(compensate) → 주문취소(compensate)
```

핵심: Saga는 ACID 대신 '최종 일관성(Eventual Consistency)'을 보장합니다

🦻 면접 필수: 분산 트랜잭션이 왜 어려운가? → DB 트랜잭션은 단일 DB만 가능, 서비스 간엔 Saga 패턴!

MSA 도입 시점 판단 기준

"모놀리식으로 시작하고, 필요할 때 쪼개라"

— Martin Fowler, MonolithFirst 패턴

✅ MSA 도입 시점

- 팀이 10명 이상으로 성장
- 배포가 주 1회 → 하루 여러 번 필요
- 특정 기능 장애가 전체에 영향
- 특정 기능만 트래픽이 급증
- 서비스별 기술 스택이 달라야 함

❌ 아직 이르다

- 팀이 5명 미만
- 서비스가 MVP/초기 단계
- 도메인 경계가 불분명
- DevOps 인프라가 미비
- 당근마켓도 초기엔 모놀리식!

SECTION 02

API Gateway 서비스의 현관문 설계

클라이언트와 마이크로서비스 사이의 통합 진입점

API Gateway란?

🏪 푸드코트 입구의 통합 키오스크

손님(클라이언트)은 키오스크(Gateway) 하나만 알면 됩니다
한식/중식/일식 어디로 보낼지는 키오스크가 알아서 라우팅!

API Gateway 없이 vs 있을 때

❌ Without Gateway:
프론트엔드 → http://user:3001, http://product:3002, http://order:3003 ...
주소 10개를 다 관리해야 함, CORS 10번 설정, 인증 10번 검증

✅ With Gateway:
프론트엔드 → http://api.myservice.com (하나만!)
Gateway가 /users → user-service, /products → product-service 라우팅

리버스 프록시(Reverse Proxy) 개념

손님은 누가 요리하는지 몰라도 됩니다

프록시(Proxy) = 대리인
포워드 프록시: 클라이언트를 대신 → 회사 방화벽, VPN
리버스 프록시: 서버를 대신 → API Gateway, Nginx, HAProxy

클라이언트 → [Nginx] → 실제 서버 (클라이언트는 Nginx만 봄)

리버스 프록시의 이점 (서비스 유지보수 관점)

- 보안: 내부 서버 IP/포트 노출 방지
- SSL 종료: HTTPS를 Gateway에서만 처리
- 캐싱: 자주 요청되는 응답을 Gateway에서 캐시
- 로드밸런싱: 같은 서비스 여러 인스턴스로 분배
- 로깅: 모든 요청/응답을 한 곳에서 기록

Nginx 기본 구조 — nginx.conf

```
# /etc/nginx/nginx.conf

worker_processes auto;

events {
    worker_connections 1024;
}

http {
    # 로그 형식 정의
    log_format main '$remote_addr - $request';

    server {
        listen 80;              # 80번 포트에서 수신
        server_name api.myservice.com;

        # location 블록에서 라우팅 설정
        location / { ... }
    }
}
```

💡 **worker_processes auto → CPU 코어 수만큼 워커 생성**

location 블록과 proxy_pass — 라우팅 설정

```
server {
    listen 80;

    # /api/users 요청 → user-service (포트 3001)
    location /api/users {
        proxy_pass http://localhost:3001;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
    }

    # /api/products 요청 → product-service (포트 3002)
    location /api/products {
        proxy_pass http://localhost:3002;
        proxy_set_header Host $host;
    }
}
```

💡 **proxy_pass = '이 요청을 이 주소로 보내라' — 핵심 라우팅 지시자**

API Gateway의 핵심 역할 5가지

1. **1. 라우팅 (Routing)**
   URL 경로에 따라 적절한 서비스로 요청 전달
2. **2. 인증/인가 (Authentication)**
   JWT 토큰 검증을 Gateway에서 처리 → 서비스는 비즈니스 로직에 집중
3. **3. CORS 처리**
   브라우저 보안 정책을 Gateway 한 곳에서 관리
4. **4. Rate Limiting**
   초당 요청 수 제한으로 DDoS/악의적 요청 방어
5. **5. 로깅/모니터링**
   모든 요청을 한 곳에서 기록 → 장애 추적 용이

Rate Limiting — 서비스 보호 설계

```
# Nginx Rate Limiting 설정

http {
    # 요청 제한 존 정의: IP당 초당 10개 요청
    limit_req_zone $binary_remote_addr
        zone=api_limit:10m rate=10r/s;

    server {
        location /api/ {
            limit_req zone=api_limit burst=20;
            # burst=20: 순간 20개까지 허용 (대기열)
            proxy_pass http://backend;
        }
    }
}
```

💡 **악의적 요청에서 서비스를 지키는 첫 번째 방어선 — 면접 빈출!**

BFF(Backend for Frontend) 패턴
BFF 패턴 — 클라이언트별 맞춤 Gateway

웹/앱/관리자 각각 다른 Gateway를 두는 설계

🌐 Web BFF

- 웹 브라우저용 Gateway
- SSR 지원
- 풀 사이즈 데이터 반환
- SEO 최적화 응답

📱 Mobile BFF

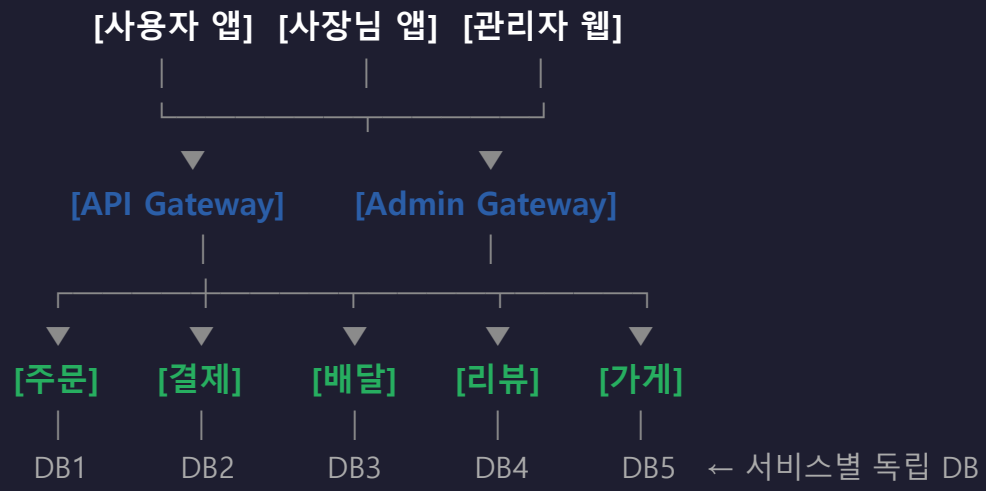
- 모바일 앱용 Gateway
- 경량 데이터 반환
- 이미지 최적화
- 배터리/데이터 절약

🔧 Admin BFF

- 관리자용 Gateway
- 통계/대시보드 API
- 벌크 작업 지원
- 높은 권한 검증

실무 설계 예시: 배달의민족 구조

배달의민족 추정 아키텍처 (텍스트 다이어그램)



💡 **서비스 설계 핵심: 각 서비스가 자신의 DB를 소유하고 독립적으로 배포**

SECTION 03

실습4:
두 개의 서버 띄우기

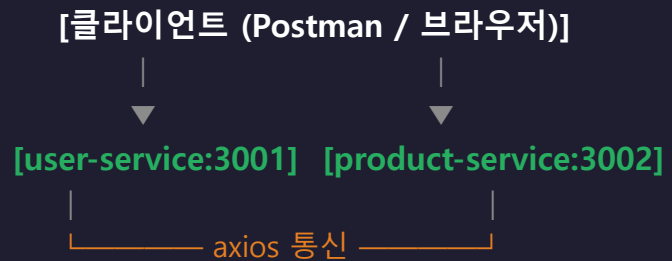
서비스 분리를 직접 체험하고 서비스 간 통신을 구현합니다

실습 목표 & 아키텍처 다이어그램

실습 목표

· user-service (포트 3001)와 product-service (포트 3002) 두 개의 독립 서버 구축
· 각 서비스가 독립적으로 동작하는 것을 확인
· 서비스 간 REST API 호출 구현

아키텍처 다이어그램



💡 나중에 Nginx Gateway를 앞에 추가할 예정입니다!

Step 1: user-service 생성

```
# 프로젝트 생성
mkdir user-service && cd user-service
npm init -y
npm install express cors

// user-service/index.js
const express = require('express');
const cors = require('cors');
const app = express();
const PORT = process.env.PORT || 3001;

app.use(cors());
app.use(express.json());
```

```
# 프로젝트 생성
// 인메모리 데이터 저장소
let users = [
  { id: 1, name: '김개발', email: 'kim@dev.com' },
  { id: 2, name: '이서버', email: 'lee@dev.com' },
];
```

💡 **PORT를 환경변수로 받으면 같은 코드로 여러 인스턴스 실행 가능**

user-service CRUD 엔드포인트

```
// GET /api/users — 전체 조회
app.get('/api/users', (req, res) => {
  res.json({ service: 'user-service', data: users });
});

// GET /api/users/:id — 단건 조회
app.get('/api/users/:id', (req, res) => {
  const user = users.find(u => u.id === +req.params.id);
  if (!user) return res.status(404).json({ error: 'Not found' });
  res.json({ data: user });
});
```

```
// POST /api/users — 생성
app.post('/api/users', (req, res) => {
  const user = { id: users.length + 1, ...req.body };
  users.push(user);
  res.status(201).json({ data: user });
});

app.listen(PORT, () =>
  console.log(`user-service running on port ${PORT}`));
```

💡 각 응답에 service 필드를 넣어 어느 서비스가 응답했는지 확인!

Step 2: product-service 생성

```
# 프로젝트 생성
mkdir product-service && cd product-service
npm init -y
npm install express cors

// product-service/index.js
const express = require('express');
const cors = require('cors');
const app = express();
const PORT = process.env.PORT || 3002;
```

```
# 프로젝트 생성
app.use(cors());
app.use(express.json());

let products = [
  { id: 1, name: '노트북', price: 1200000, sellerId: 1
},
  { id: 2, name: '키보드', price: 89000, sellerId: 2 },
  { id: 3, name: '모니터', price: 450000, sellerId: 1 },
];
```

💡 **sellerId 필드 — 나중에 user-service와 연동할 때 사용합니다**

product-service CRUD 엔드포인트

```
// GET /api/products — 전체 조회
app.get('/api/products', (req, res) => {
  res.json({ service: 'product-service', data: products });
});

// GET /api/products/:id — 단건 조회
app.get('/api/products/:id', (req, res) => {
  const product = products.find(p => p.id === +req.params.id);
  if (!product) return res.status(404).json({ error: 'Not found' });
  res.json({ data: product });
});
```

```
// POST /api/products — 생성
app.post('/api/products', (req, res) => {
  const product = { id: products.length + 1, ...req.body };
  products.push(product);
  res.status(201).json({ data: product });
});

app.listen(PORT, () =>
  console.log(`product-service running on port ${PORT}`));
```

💡 **user-service와 동일한 패턴 — RESTful 일관성이 핵심입니다**

Step 3: 두 서버 동시 실행 & 테스트

두 개의 독립 서비스를 동시에 실행하고 각각 테스트합니다

1. **터미널 1에서 user-service 실행**
   cd user-service && node index.js → 포트 3001
2. **터미널 2에서 product-service 실행**
   cd product-service && node index.js → 포트 3002
3. **user-service 테스트**
   curl http://localhost:3001/api/users
4. **product-service 테스트**
   curl http://localhost:3002/api/products
5. **독립성 확인: user-service를 종료해도 product-service 정상 동작!**
   장애 격리 확인 — MSA 핵심 장점 체험

서비스 간 통신 — 왜 필요한가?

시나리오: 상품 상세 페이지에서 판매자 정보도 보여주고 싶다

product-service는 sellerId만 갖고 있음
판매자 이름/이메일은 user-service가 갖고 있음
→ product-service가 user-service에게 요청해야 함!

서비스 간 통신 방법 2가지

1. 동기(Synchronous) — REST API 호출 (HTTP)
전화 통화처럼 즉시 응답을 기다림
axios, fetch, http 모듈 사용

2. 비동기(Asynchronous) — 메시지 큐 (RabbitMQ, Kafka)
카카오톡처럼 보내고 나중에 확인
이벤트 기반 통신, 느슨한 결합

axios로 서비스 간 REST 호출

```
// product-service/index.js 에 추가
const axios = require('axios');  // npm install axios

// 상품 + 판매자 정보 조합 API
app.get('/api/products/:id/detail', async (req, res) => {
  try {
    const product = products.find(p => p.id ===
+req.params.id);
    if (!product) return res.status(404).json({ error: 'Not found' });
```

```
// product-service/index.js 에 추가
 // user-service에게 판매자 정보 요청
    const { data } = await axios.get(
      `http://localhost:3001/api/users/${product.sellerId}`
    );
    res.json({
      product,
      seller: data.data  // 서비스 간 통신 결과
    });
  } catch (err) {
    res.status(503).json({ error: 'user-service 응답 없음' });
  }
});
```

💡 **user-service가 다운되면 503 반환 — 장애 전파 방지가 중요!**

토론: 서비스 경계 설계

"어떤 기능을 어떤 서비스에 넣을 것인가?"

서비스 경계(Service Boundary) 설계는 MSA의 가장 어려운 결정입니다

토론 시나리오: 쇼핑몰 서비스를 설계하세요

다음 기능들을 어떤 서비스로 나눌 것인가?

- 회원가입/로그인/프로필 관리
- 상품 등록/조회/검색
- 장바구니/위시리스트
- 주문/결제/환불
- 배송 추적
- 리뷰/평점

힌트: DDD(Domain-Driven Design)의 Bounded Context 개념
쿠팡 조직 구조를 참고 — 팀 구조가 서비스 구조를 결정 (Conway's Law)

SECTION 04

REST API 설계 서비스의 대화 규칙

실무에서 통용되는 API 설계 원칙과 표준

REST API 설계 7대 원칙

1. URI는 명사로 — 리소스를 표현

/api/users (O) /api/getUsers (X)

2. HTTP 메서드로 행위 표현

GET=조회, POST=생성, PUT=수정, DELETE=삭제

3. 복수형 사용

/api/users (O) /api/user (X)

4. 계층 구조 표현

/api/users/1/orders — 1번 유저의 주문목록

5. 상태코드로 결과 전달

200=성공, 201=생성, 404=없음, 500=서버에러

6. 버전 관리

/api/v1/users — 호환성 유지

7. 필터/정렬/페이징

/api/products?sort=price&page=1&limit=20

URI 설계 — 좋은 예 vs 나쁜 예

❌ 나쁜 예

GET /api/getUsers
GET /api/getUserById?id=1
POST /api/createUser
POST /api/deleteUser
GET /api/user (단수형)
GET /api/User (대문자)

문제점:
· 동사가 URI에 포함됨
· HTTP 메서드 역할 무시
· 일관성 없는 네이밍

✅ 좋은 예

GET /api/users
GET /api/users/1
POST /api/users
DELETE /api/users/1
GET /api/users (복수형)
GET /api/users/1/orders

장점:
· URI = 리소스(명사)
· HTTP 메서드 = 행위(동사)
· 직관적이고 예측 가능

HTTP 메서드 ↔ CRUD 매핑

HTTP 메서드	CRUD	URI 예시	설명
GET	Read	/api/users	리소스 조회 (목록/단건)
POST	Create	/api/users	리소스 생성
PUT	Update	/api/users/1	리소스 전체 수정
PATCH	Partial	/api/users/1	리소스 부분 수정
DELETE	Delete	/api/users/1	리소스 삭제

💡 **면접 TIP: PUT vs PATCH 차이를 설명할 수 있어야 합니다!**

PUT vs PATCH — 면접 단골 질문

PUT — 전체 교체 (Replace All)

```
// PUT /api/users/1
// Request Body (전체 필드 필수!)
{
  "name": "김개발",
  "email": "new@dev.com",  // 수정
  "age": 30               // 필수 포함!
}
```

⚠️ age 빠지면 null로 덮어씌워짐!
→ 전체 리소스를 새 값으로 완전 교체

PATCH — 부분 수정 (Partial Update)

```
// PATCH /api/users/1
// Request Body (변경 필드만!)
{
  "email": "new@dev.com"  // 이것만!
}
```

✅ name, age는 기존 값 유지!
→ 지정한 필드만 선택적으로 수정

Express.js 구현 비교

```
// PUT — Object.assign으로 전체 교체
app.put('/api/users/:id', (req, res) => { users[idx] = { id, ...req.body }; });
// PATCH — spread로 부분 병합
app.patch('/api/users/:id', (req, res) => { users[idx] = { ...users[idx], ...req.body }; });
```

🦻 *면접 답변: PUT은 전체 교체(없는 필드는 null), PATCH는 부분 수정(명시한 필드만 변경)*

HTTP 상태 코드 전략

서비스 응답의 명확한 상태 전달

2xx 성공

200 OK — 조회 성공

201 Created — 생성 성공

204 No Content — 삭제 성공

요청이 정상 처리됨을
명확히 전달

4xx 클라이언트 에러

400 Bad Request — 잘못된 요청

401 Unauthorized — 인증 필요

403 Forbidden — 권한 없음

404 Not Found — 리소스 없음

429 Too Many Requests

5xx 서버 에러

500 Internal Server Error

502 Bad Gateway

503 Service Unavailable

504 Gateway Timeout

서버 측 문제임을 전달

에러 응답 표준화 (RFC 7807)

```
// 표준화된 에러 응답 형식 (RFC 7807 Problem Details)

// Express 미들웨어로 에러 핸들링 표준화
app.use((err, req, res, next) => {
  const statusCode = err.statusCode || 500;
  res.status(statusCode).json({
    type: "https://api.myservice.com/errors/not-found",
    title: "리소스를 찾을 수 없습니다",
    status: statusCode,
    detail: err.message || "요청한 리소스가 존재하지 않습니다",
    instance: req.originalUrl,
    timestamp: new Date().toISOString()
  });
});
```

```
// 표준화된 에러 응답 형식 (RFC 7807 Problem Details)
// 응답 예시:
// { "type": "...errors/not-found",
//   "title": "리소스를 찾을 수 없습니다",
//   "status": 404,
//   "detail": "ID 999에 해당하는 사용자가 없습니다" }
```

💡 **네이버/카카오 API도 표준화된 에러 형식을 사용합니다 — 실무 필수!**

동기(REST) vs 비동기(Message Queue)

📞 동기 방식 (REST)

= 전화 통화

· 요청하면 즉시 응답을 기다림
· HTTP 요청/응답
· 간단하고 직관적

사용 예:
· 사용자 정보 조회
· 상품 목록 조회
· 실시간 재고 확인

단점: 호출 대상 서비스 다운 시 실패

💬 비동기 방식 (MQ)

= 카카오톡 메시지

· 메시지 보내고 나중에 처리
· RabbitMQ, Kafka 등 사용
· 느슨한 결합, 높은 내구성

사용 예:
· 주문 완료 → 알림 발송
· 결제 완료 → 배송 시작
· 로그 수집 → 분석

장점: 서비스 다운 시 큐에 보관

RabbitMQ — 비동기 메시지 큐 실습

주문 완료 시 '이메일 알림'을 비동기로 처리하는 패턴 — 주문 서비스 응답 속도 유지!

📤 Publisher (주문 서비스)

```
// npm install amqplib
const amqp = require('amqplib');

async function publishOrder(order) {
  const conn = await amqp.connect(
    'amqp://localhost');
  const ch = await conn.createChannel();
  await ch.assertQueue('order_queue');
  ch.sendToQueue('order_queue',
    Buffer.from(JSON.stringify(order)));
  // 응답 즉시 반환 – 큐에만 넣고 끝!
}
```

📥 Subscriber (알림 서비스)

```
async function startConsumer() {
  const conn = await amqp.connect(
    'amqp://localhost');
  const ch = await conn.createChannel();
  await ch.assertQueue('order_queue');
  ch.consume('order_queue', (msg) => {
    const order = JSON.parse(
      msg.content.toString());
    sendEmail(order.userEmail);  // 이메일 발송
    ch.ack(msg);  // 처리 완료 확인
  });
}
```

흐름: 주문 서비스 → [order_queue] → 알림 서비스 | 주문 서비스가 느린 이메일 전송을 기다리지 않음!

알림 서비스가 다운돼도 큐에 메시지 보존 → 복구 후 자동 처리! 토스/쿠팡 주문 알림 방식

SECTION 05

실습5: Nginx로 통합 입구 만들기

API Gateway를 직접 구축하여 두 서비스를 하나로 연결합니다

실습 목표 & 최종 구조

최종 아키텍처

실습 목표:

✅ Nginx 설치 → 설정 작성 → 라우팅 테스트 → CORS 설정

✅ 클라이언트는 localhost:80 하나만 알면 됨!

Step 1: Nginx 설치

```
# === 방법 1: apt로 설치 (Ubuntu/Debian) ===
sudo apt update
sudo apt install nginx -y
nginx -v   # 버전 확인

# === 방법 2: Docker로 설치 (추천) ===
docker pull nginx:latest

# 설정 파일 마운트하여 실행
docker run -d ₩
  --name api-gateway ₩
  -p 80:80 ₩
  -v $(pwd)/nginx.conf:/etc/nginx/nginx.conf ₩
  nginx:latest

# === 방법 3: macOS ===
brew install nginx
```

💡 **Docker 방식을 추천합니다 — 환경에 영향 없이 깔끔하게 실습!**

Step 2: nginx.conf 작성 — 라우팅 설정

```
# nginx.conf
events {
    worker_connections 1024;
}

http {
    server {
        listen 80;
        server_name localhost;
```

```
# nginx.conf
        # user-service로 라우팅
        location /api/users {
            proxy_pass
http://host.docker.internal:3001;
        }

        # product-service로 라우팅
        location /api/products {
            proxy_pass
http://host.docker.internal:3002;
        }
    }
}
```

💡 **Docker 사용 시 host.docker.internal, 직접 설치 시 localhost 사용**

Step 3: proxy_pass 상세 설정

```
location /api/users {
    proxy_pass http://localhost:3001;

    # 원본 요청 정보를 백엔드에 전달
    proxy_set_header Host $host;
    proxy_set_header X-Real-IP $remote_addr;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_set_header X-Forwarded-Proto $scheme;

    # 타임아웃 설정 (서비스 유지보수 핵심)
    proxy_connect_timeout 5s;
    proxy_read_timeout 30s;
    proxy_send_timeout 30s;

    # 버퍼 설정
    proxy_buffering on;
    proxy_buffer_size 4k;
}
```

💡 **X-Real-IP: 실제 클라이언트 IP를 백엔드에 전달 — 로깅/보안에 필수**

Step 4: CORS 설정 추가

```
location /api/ {
    # CORS 헤더 설정 — Gateway에서 통합 관리!
    add_header 'Access-Control-Allow-Origin' '*';
    add_header 'Access-Control-Allow-Methods'
               'GET, POST, PUT, DELETE, OPTIONS';
    add_header 'Access-Control-Allow-Headers'
               'Content-Type, Authorization';

    # Preflight 요청(OPTIONS) 처리
    if ($request_method = 'OPTIONS') {
        add_header 'Access-Control-Max-Age' 86400;
        add_header 'Content-Length' 0;
        return 204;
    }

    proxy_pass http://localhost:3001;
}
```

💡 **각 서비스에서 CORS 설정 불필요! Gateway에서 한 번만 설정하면 끝**

Step 5: Nginx 시작 & 테스트

Nginx를 시작하고 라우팅이 정상 작동하는지 확인합니다

1. **Nginx 설정 검증**
   sudo nginx -t → 'syntax is ok, test is successful' 확인
2. **Nginx 시작**
   sudo systemctl start nginx (또는 docker start api-gateway)
3. **user-service 라우팅 테스트**
   curl http://localhost/api/users → user-service 응답 확인
4. **product-service 라우팅 테스트**
   curl http://localhost/api/products → product-service 응답 확인
5. **설정 변경 후 재시작**
   sudo nginx -s reload → 무중단 설정 반영

테스트: localhost/api/users → user-service 응답

```
# 1. user-service 테스트
$ curl http://localhost/api/users

{ "service": "user-service",
  "data": [
    { "id": 1, "name": "김개발", "email": "kim@dev.com" },
    { "id": 2, "name": "이서버", "email": "lee@dev.com" }
  ]
}

# 2. product-service 테스트
$ curl http://localhost/api/products

{ "service": "product-service",
  "data": [
    { "id": 1, "name": "노트북", "price": 1200000 },
    ...
  ]
}
```

💡 **포트 번호 없이 localhost만으로 접근! — Gateway가 라우팅 처리**

디버깅 가이드 — 자주 만나는 에러와 해결법

자주 발생하는 에러 TOP 5

1. 502 Bad Gateway
→ 백엔드 서비스가 실행되지 않음. user-service/product-service 확인!

2. 403 Forbidden
→ Nginx 설정에 권한 문제. nginx.conf 경로 확인!

3. 404 Not Found
→ location 블록의 경로 매칭 실패. 트레일링 슬래시(/) 확인!

4. CORS 에러 (브라우저)
→ add_header 설정 누락. OPTIONS preflight 처리 확인!

5. Connection Refused
→ 포트 충돌 또는 방화벽. netstat -tlnp 로 포트 확인!

디버깅 명령어: tail -f /var/log/nginx/error.log

SECTION 06

로드밸런싱 서비스 안정성 설계

트래픽을 분산하여 서비스를 안정적으로 운영하는 방법

로드밸런싱이란?

🍳 손님이 많으면 요리사를 늘린다!

푸드코트 한식 가게에 손님이 100명 몰리면?
→ 한식 가게를 2개로 늘리고, 입구에서 골고루 안내!
→ 이것이 로드밸런싱(Load Balancing)입니다

실제 적용 사례

쿠팡: 로켓배송 주문 서비스 — 동시 접속 수십만 처리
배달의민족: 점심시간 주문 폭주 — 주문 서비스 인스턴스 자동 증가
네이버: 실시간 검색어 — 검색 서비스 수백 대 서버로 분산
카카오: 설날 카카오톡 트래픽 — 메시지 서비스 수평 확장

💡 **서비스 유지보수 관점: 로드밸런싱은 '안 죽는 서비스'의 핵심!**

로드밸런싱 알고리즘 3가지

서버에 요청을 어떤 기준으로 분배할 것인가?

Round Robin

돌아가며 균등 분배

A → B → C → A → B ...

가장 단순하고 기본적

Nginx 기본 방식

서버 성능이 동일할 때

Least Connections

연결 수 가장 적은 서버로

현재 처리 중인 요청이

적은 서버에 우선 배분

서버 처리 시간이

다를 때 효과적

IP Hash

같은 IP → 같은 서버

클라이언트 IP 기반

해시값으로 서버 결정

세션 유지가 필요할 때

스티키 세션 대안

Nginx upstream 설정 — 로드밸런싱 코드

```
http {
    # upstream: 같은 서비스의 여러 인스턴스 그룹
    upstream user_service {
        # Round Robin (기본값, 별도 설정 불필요)
        server localhost:3001;
        server localhost:3003;  # 같은 서비스 2번째 인스턴스
    }

    upstream product_service {
        least_conn;  # Least Connections 방식
        server localhost:3002;
        server localhost:3004;
    }

    server {
        listen 80;
        location /api/users {
            proxy_pass http://user_service;
        }
        location /api/products {
            proxy_pass http://product_service;
        }
    }
}
```

💡 **proxy_pass에 upstream 그룹명을 사용 → 자동으로 부하 분산!**

헬스체크 — 죽은 서버에 요청 보내지 않기

Passive Health Check (Nginx 기본 내장)

```
upstream user_service {
    server localhost:3001 max_fails=3 fail_timeout=30s;
    server localhost:3003 max_fails=3 fail_timeout=30s;
}
```

3번 연속 실패하면 30초간 해당 서버를 비활성화

헬스체크 엔드포인트 만들기 (서비스 측)

```
// user-service에 추가
app.get('/health', (req, res) => {
  res.json({ status: 'UP', service: 'user-service',
    timestamp: new Date().toISOString() });
});
```

💡 **쿠팡/토스: Kubernetes의 liveness/readiness probe 활용**

Circuit Breaker — 장애 전파 차단 패턴

3가지 상태 전환 (전기 차단기와 동일한 원리)

CLOSED (정상) → 연속 실패 5회 → OPEN (차단) → 30초 후 → HALF-OPEN (탐색) → 성공 시 → CLOSED
실패 시 → OPEN 유지 실패 시 → OPEN

opossum 라이브러리로 Circuit Breaker 구현 (Node.js)

```
const CircuitBreaker = require('opossum');  // npm install opossum

const options = {
  errorThresholdPercentage: 50,  // 50% 실패 시 OPEN
  timeout: 3000,                 // 3초 초과 시 실패로 처리
  resetTimeout: 30000,           // 30초 후 HALF-OPEN 시도
};

const breaker = new CircuitBreaker(callUserService, options);
breaker.fallback(() => ({ name: '알 수 없음', status: 'fallback' }));
// OPEN 상태에서 fallback 반환 → 서비스 전체 장애 방지!
```

실제 사례: 넷플릭스 Hystrix, 쿠팡 추천 서비스 장애 시 기본 추천 목록 fallback 반환

Circuit Breaker = 서비스 A의 장애가 서비스 B, C, D로 전파되는 연쇄 장애(Cascading Failure) 방지!

고급 로드밸런싱 — Weighted & Least Conn

⚖️ Weighted Round Robin

서버 성능에 따라 비율 다르게 분배

```
upstream user_service {
  # 고성능 서버: 요청의 67%
  server 192.168.1.1:3001 weight=2;
  # 일반 서버: 요청의 33%
  server 192.168.1.2:3001 weight=1;
}
```

결과: A→A→B→A→A→B 순으로 분배

적용: 스펙 다른 서버 혼용 시

🔗 Least Connections

현재 연결 수 가장 적은 서버로 라우팅

```
upstream product_service {
  least_conn;  # 핵심 설정!
  server 192.168.1.3:3002;
  server 192.168.1.4:3002;
}
```

결과: A(3개)→B(1개) 상황 → B 선택

적용: 처리시간 다른 요청 혼재 시

알고리즘 선택 기준

Round Robin: 서버 스펙 동일, 요청 처리시간 유사 | Weighted: 서버 스펙 상이 | Least Conn: 처리시간 편차 큼 | IP Hash: 세션 유지 필요

실무 Tip: 대부분의 MSA는 Kubernetes Ingress + Least Conn 조합을 기본으로 사용합니다

수평 확장 vs 수직 확장

📈 수직 확장 (Scale Up)

= 요리사에게 더 큰 칼을 준다

· 서버의 CPU/RAM/SSD 업그레이드
· 하나의 서버를 더 강력하게

장점:
· 구조 변경 불필요
· 관리가 단순함

단점:
· 비용이 기하급수적 증가
· 물리적 한계 존재
· 단일 장애점(SPOF)

📊 수평 확장 (Scale Out)

= 요리사를 더 고용한다

· 같은 서비스를 여러 대 실행
· 로드밸런서로 트래픽 분배

장점:
· 이론상 무한 확장 가능
· 비용 효율적
· 장애 내성(Fault Tolerance)

단점:
· 로드밸런서 필요
· 세션/상태 관리 고민

실습6: 부하 분산 체험 (개요)

같은 서비스를 두 개 실행하고 로드밸런싱을 직접 체험합니다

1. **user-service 인스턴스 2개 실행**
   PORT=3001 node index.js & PORT=3003 node index.js
2. **Nginx upstream 설정**
   nginx.conf에 upstream 블록 추가 후 reload
3. **요청 분배 확인**
   반복 요청하여 3001/3003 번갈아 응답하는지 확인
4. **인스턴스 하나 중지**
   3001 서버 종료 후에도 3003이 처리하는지 확인
5. **결과 분석**
   로드밸런싱 + 장애 대응 = 서비스 안정성의 핵심

실습: 같은 서비스 2개 실행 (PORT 환경변수)

```
# 터미널 1: user-service 인스턴스 1 (포트 3001)
PORT=3001 node index.js
# 출력: user-service running on port 3001

# 터미널 2: user-service 인스턴스 2 (포트 3003)
PORT=3003 node index.js
# 출력: user-service running on port 3003

// index.js에서 응답에 포트 정보 추가 (어느 인스턴스가 응답했는지 확인)
app.get('/api/users', (req, res) => {
  res.json({
    service: 'user-service',
    port: PORT,          // 어느 인스턴스인지 표시!
    data: users
  });
});
```

💡 **같은 코드를 다른 포트로 실행 — 이것이 수평 확장의 기본 원리!**

Nginx upstream 설정 & 테스트

```
# nginx.conf 수정
upstream user_service {
    server localhost:3001;
    server localhost:3003;
}

server {
    listen 80;
    location /api/users {
        proxy_pass http://user_service;  # upstream 그룹 사용
    }
}

# 설정 반영 후 테스트
$ sudo nginx -s reload

$ curl localhost/api/users  # → port: 3001 응답
$ curl localhost/api/users  # → port: 3003 응답
$ curl localhost/api/users  # → port: 3001 응답 (반복)
```

💡 **Round Robin: 요청이 3001 → 3003 → 3001 → 3003 순서로 분배!**

서버 하나 중지 시 Failover 확인

로드밸런싱의 핵심: 서버 하나가 죽어도 서비스는 계속!

1. **현재 상태 확인**
   curl 반복 → 3001/3003 번갈아 응답 확인
2. **3001 인스턴스 강제 종료**
   터미널 1에서 Ctrl+C → user-service:3001 중지
3. **요청 계속 보내기**
   curl localhost/api/users → 모든 응답이 port: 3003!
4. **서비스 무중단 확인**
   사용자 입장에서는 아무 문제 없이 동작!
5. **3001 재시작 → 자동 복귀**
   PORT=3001 node index.js → 다시 분산 시작

스티키 세션(Sticky Session) 개념

문제: 로그인 세션이 서버 A에 있는데, 다음 요청이 서버 B로 가면?

→ 서버 B에는 세션이 없으므로 '로그인이 풀린 것처럼' 보임!
→ 이를 해결하기 위해 같은 사용자는 같은 서버로 보내는 것 = 스티키 세션

Nginx IP Hash 방식

```
upstream user_service {
    ip_hash;  # IP 기반 고정
    server localhost:3001;
    server localhost:3003;
}
```

더 나은 방법: 외부 세션 저장소

Redis/Memcached에 세션 저장
→ 어느 서버로 가든 세션 공유!

토스/쿠팡 방식:
JWT 토큰 사용으로 서버 무상태
→ 스티키 세션 자체가 불필요!

JWT Stateless vs Redis 세션 공유

🔑 JWT Stateless 방식 (토스/쿠팡)

```
// 로그인 → JWT 토큰 발급
const token = jwt.sign(
  { userId: 1, role: 'user' },  // payload
  process.env.JWT_SECRET,       // 서명키
  { expiresIn: '1h' }           // 만료
);  // 토큰을 클라이언트에 저장!

// 요청마다 토큰 검증 (서버 무상태!)
const payload = jwt.verify(token, SECRET);
```

✅ 서버에 세션 저장 없음 → Scale Out 자유로움
❌ 토큰 만료 전 강제 로그아웃 불가

🗄️ Redis 세션 공유 방식

```
// express-session + connect-redis
const session = require('express-session');
const RedisStore = require('connect-redis');

app.use(session({
  store: new RedisStore({ client }),
  secret: process.env.SESSION_SECRET,
  resave: false,
  saveUninitialized: false,
}));  // 어느 서버든 Redis에서 세션 조회!
```

✅ 즉시 로그아웃/세션 무효화 가능
❌ Redis 장애 = 전체 로그인 불가

선택 기준: 금융/결제(즉시 무효화 필요) → Redis 세션 | 일반 서비스(Scale Out 중요) → JWT | 토스뱅크: JWT + 별도 토큰 블랙리스트 Redis

🦻 *면접 답변: JWT는 서버 무상태, Redis는 즉시 제어 가능 — 두 방식의 trade-off를 설명하면 합격!*

API 설계 체크리스트 (실무용)

- ☐ URI에 동사 대신 명사를 사용했는가?
- ☐ HTTP 메서드를 올바르게 사용했는가? (GET=조회, POST=생성...)
- ☐ 복수형을 사용했는가? (/users, /products)
- ☐ 적절한 HTTP 상태 코드를 반환하는가?
- ☐ 에러 응답이 표준 형식을 따르는가?
- ☐ API 버전 관리를 하고 있는가? (/v1/, /v2/)
- ☐ 페이징/정렬/필터링을 지원하는가?
- ☐ 인증(Authentication)이 적용되어 있는가?
- ☐ Rate Limiting이 설정되어 있는가?
- ☐ API 문서(Swagger/OpenAPI)가 작성되어 있는가?

정리

2일차 핵심 요약 5가지

1. **MSA는 서비스를 독립적으로 분리하는 아키텍처**
   독립 배포, 장애 격리, 기술 다양성이 핵심 장점. 단, 모놀리식 먼저!
2. **API Gateway는 서비스의 통합 현관문**
   Nginx로 라우팅, 인증, CORS, Rate Limiting을 한 곳에서 관리
3. **REST API 설계 원칙을 지켜야 유지보수가 쉽다**
   URI=명사, 메서드=동사, 상태코드=결과 전달이 핵심
4. **로드밸런싱으로 서비스 안정성을 확보한다**
   수평 확장 + 헬스체크 + Failover로 무중단 서비스 구현
5. **서비스 간 통신은 동기/비동기를 적절히 선택**
   즉시 응답 필요 → REST, 느슨한 결합 → 메시지 큐

Q & A

궁금한 점이 있으면 질문해주세요!

수고하셨습니다! 내일은 Docker와 Cloud 배포를 배웁니다 🚀

참고 자료 & 추천 학습 링크

도서

· Building Microservices (Sam Newman) — MSA 바이블
· 가상 면접 사례로 배우는 대규모 시스템 설계 기초 (Alex Xu)

온라인 자료

· Nginx 공식 문서: nginx.org/en/docs/
· Martin Fowler 블로그: martinfowler.com/microservices
· Microsoft MSA 가이드: docs.microsoft.com/azure/architecture/microservices

한국어 자료

· 우아한형제들 기술블로그: techblog.woowahan.com
· 토스 기술블로그: toss.tech
· 쿠팡 기술블로그: medium.com/coupang-engineering
· 카카오 기술블로그: tech.kakao.com
· 네이버 D2: d2.naver.com

2일차 핵심 용어 정리

MSA	큰 서비스를 독립된 작은 서비스들로 분리하는 아키텍처
API Gateway	클라이언트와 마이크로서비스 사이의 통합 진입점
Reverse Proxy	서버를 대신하여 클라이언트 요청을 받는 중간 서버
Load Balancing	트래픽을 여러 서버 인스턴스에 분배하는 기술
Upstream	Nginx에서 백엔드 서버 그룹을 정의하는 설정
Health Check	서버의 생존 여부를 주기적으로 확인하는 메커니즘
Scale Out/Up	수평 확장(서버 추가) / 수직 확장(성능 업그레이드)
Sticky Session	같은 사용자의 요청을 같은 서버로 보내는 방식
BFF	Backend for Frontend — 클라이언트별 맞춤 Gateway
Saga Pattern	분산 트랜잭션을 관리하기 위한 패턴