서비스 설계와 유지보수 교육 — 3일 과정

서비스 설계의 시작 왜 구조가 중요한가?

Day 1: 집 잘 짓기 — 서비스의 기본 구조 설계

목표

서비스를 제대로 설계하는 법

여러분이 배달의민족을 처음부터 만든다면?

🤔 "어디서부터 시작할 것인가?"

생각해보세요:

- 사용자가 앱을 열면 무슨 일이 일어나야 할까?
- 음식을 주문하면 내부에서 어떤 과정을 거칠까?
- 결제, 배달 추적, 리뷰... 이걸 한 파일에 다 넣을 수 있을까?

→ 답은 "구조(아키텍처)부터!" 입니다

배달의민족도, 쿠팡도, 토스도 모두
"어떻게 코드를 나눌 것인가"를 먼저 설계했습니다.

오늘 우리는 그 "구조를 설계하는 법"을 배웁니다.

오늘의 학습 목표

1. **스파게티 코드의 문제점을 체험하고 설명할 수 있다**
   실제 코드를 보면서 왜 구조가 중요한지 체감합니다
2. **관심사의 분리(SoC)와 단일 책임 원칙(SRP)을 이해한다**
   좋은 설계의 핵심 원칙을 배웁니다
3. **3계층 아키텍처를 설계에 적용할 수 있다**
   Controller → Service → Repository 구조를 직접 만듭니다
4. **의존성 방향 규칙을 이해한다**
   왜 위에서 아래로만 의존해야 하는지 깨닫습니다
5. **3계층과 REST API 설계를 연결할 수 있다**
   HTTP Method, RESTful URI, 버저닝, DTO까지 설계합니다

오프닝

1일차 타임테이블 (8시간)

09:00 – 10:30	아키텍처란? & 스파게티 코드의 공포
10:40 – 12:00	3계층 아키텍처 이론 — Controller / Service / Repository + API 설계 연계
12:00 – 13:00	🍙 점심시간
13:00 – 14:30	[실습1] 스파게티 코드 분석 & 리팩터링 전략
14:40 – 16:00	[실습2] 방 나누기 — 코드를 3계층으로 분리
16:10 – 17:30	[실습3] 화이트보드 설계 — 게시판 서비스
17:30 – 18:00	정리, 과제 안내, Q&A

SECTION 01

왜 서비스 설계가 필요한가?

구조 없이 코딩하는 것 = 설계도 없이 집 짓는 것

SECTION 01

아키텍처란? — 복잡한 것을 단순하게 만드는 "약속"

건축의 설계도

건축에서:

- 방, 거실, 주방의 위치를 정한다
- 전기 배선, 수도 배관 경로를 계획한다
- 설계도 없이 짓는 건 불법!

설계도가 있으면:

✓ 여러 업체가 동시에 시공 가능
✓ 문제 발생 시 어디를 볼지 안다
✓ 증축/수리가 쉽다

소프트웨어 아키텍처

소프트웨어에서:

- 코드를 어떤 단위로 나눌지 정한다
- 모듈 간 통신 방법을 약속한다
- 설계 없이 코딩하면? → 스파게티 코드!

아키텍처가 있으면:

✓ 여러 개발자가 동시 개발 가능
✓ 버그 발생 시 어디를 볼지 안다
✓ 기능 추가/변경이 쉽다

SECTION 01

비유(1): 1인 푸드트럭 — 모놀리식의 한계

🚚 1인 푸드트럭 = 모든 걸 혼자 하는 코드

요리 + 계산 + 청소 + 서빙을 혼자서 처리
→ 손님 3명까지는 OK!
→ 손님 30명이 되면? 🔥 재앙!

- 계산하는 동안 요리가 탄다
- 청소하는 동안 손님이 기다린다
- 메뉴 하나 추가하려면 전체 프로세스를 바꿔야 한다

💡 실제 사례: 스타트업 초기 → 사용자 증가 → 코드 카오스

당근마켓도 초기에는 모놀리식 Ruby on Rails로 시작.
사용자가 폭증하면서 "하나의 코드베이스로는 한계"를 체감.
→ 결국 구조를 나누는 작업(리팩터링)에 수개월 투자!

SECTION 01

비유(2): 대형 레스토랑 — 역할 분담의 힘

"역할이 명확하면, 각자가 최선을 다할 수 있다"

🤵 홀 매니저

손님 안내

주문 접수

불만 처리

→ Controller 역할

👨‍🍳 주방장

레시피대로 요리

재료 조합

품질 관리

→ Service 역할

📦 창고지기

재료 보관

재고 관리

입출고 기록

→ Repository 역할

SECTION 01

스파게티 코드의 공포 — 코드 예시

```
// ❌ 하나의 라우트 핸들러에 모든 것이 섞여있다!
app.post("/api/orders", async (req, res) => {
  // 입력 검증 (Controller 역할)
  if (!req.body.userId) return res.status(400).send("userId 필수");
  if (!req.body.items || req.body.items.length === 0)
    return res.status(400).send("상품을 선택하세요");
  // 비즈니스 로직 (Service 역할)
  let total = 0;
  for (const item of req.body.items) {
    const product = await db.query(
      "SELECT * FROM products WHERE id=?", [item.productId]
    );
    if (product.stock < item.qty) // DB 접근 (Repository 역할)
      return res.status(400).send("재고 부족");
    total += product.price * item.qty;
  }
  // ❌ 하나의 라우트 핸들러에 모든 것이 섞여있다!
  // 또 DB 접근...
  await db.query("INSERT INTO orders...", [total]);
  await db.query("UPDATE products SET stock=stock-?...");
  res.json({ success: true, total });  // 응답
});
```

⚠️ **한 함수에 입력검증 + 비즈니스로직 + DB접근 + 응답이 전부 섞여있다!**

SECTION 01

스파게티 코드의 4가지 재앙

💥 변경의 파급효과

한 줄을 고치면 엉뚱한 곳에서 에러

기능 동작 안 하거나

서비스 정지

🧪 테스트 불가능

단위 테스트 불가

→ "일단 배포하고 기도하기"

♻️ 재사용 불가

비슷한 로직이 필요할 때

동일 한 기능 개발

버그가 있는 함수 사용

→ 같은 버그가 10곳에서 발생.

🤝 협업 충돌

같은 파일을 여러 사람이 수정 → Git 충돌 폭발. 쿠팡 같은 대규모 팀에서는 치명적!

실제 사고 사례 — 스파게티 코드로 인한 장애

🔴 사례 1: 결제 로직 수정 → 회원가입 장애

결제 모듈에서 사용하는 유틸 함수를 수정했더니,
같은 함수를 사용하던 회원가입 로직이 깨짐.
원인: 모든 로직이 하나의 파일에 섞여있어서 의존관계를 파악 불가.
→ 서비스 30분 장애, 매출 손실 수천만원

🔴 사례 2: DB 마이그레이션에 3주 소요

MySQL → PostgreSQL 전환을 시도했으나,
SQL 쿼리가 100개 이상의 파일에 흩어져있었음.
원인: Repository 계층 없이 곳곳에서 직접 DB 접근.
→ 3주 동안 모든 신기능 개발 중단

SECTION 01

관심사의 분리 (Separation of Concerns)

서비스 설계의 첫 번째 원칙: "자기 일만 잘하자!"

각 모듈은 하나의 관심사만 담당해야 한다.

🏥 병원 비유

접수 → 진료 → 처방 → 약국

각각의 역할 분리

의사가 약을 직접 조제하지 않는다!

💰 은행 비유

창구 → 심사 → 금고

창구 직원이 금고를 직접 열지 않는다.

역할 분리 = 보안 + 효율

🛒 쿠팡 사례

주문/결제/배송/리뷰 각각 별도 팀.

주문팀이 배송 코드를 직접 수정하지 않는다.

→ 수천 명이 동시 개발 가능

SECTION 02

3계층 아키텍처 서비스 구조의 정석

Controller → Service → Repository — 왜 이렇게 나눌까?

SECTION 02

3계층 전체 그림 — 요청의 흐름

🌐 **Client (브라우저/앱)** — 사용자의 요청 시작점

↓ HTTP 요청

📋 **Controller (문지기)** — 요청 접수 · 입력 검증 · 응답 반환

↓ 함수 호출

⚙️ **Service (요리사)** — 비즈니스 로직 · 규칙 판단 · 트랜잭션

↓ 함수 호출

🗄️ **Repository (창고지기)** — DB CRUD · 데이터 접근

↓ SQL/ORM

💾 **Database (저장소)** — 실제 데이터 저장

SECTION 02

왜 이렇게 나누는가? — 의존성 방향 규칙

핵심 규칙: 의존성은 "위에서 아래로만!" (단방향)

🎯 비유로 이해하기:

"냉장고가 바뀌어도
요리법은 안 바뀐다"

냉장고(DB) → Repository
요리법 → Service
손님 응대 → Controller

✅ 아래 계층이 바뀌어도 위 계층은 영향 없음!
✅ 각 계층을 독립적으로 테스트할 수 있음!

SECTION 02

설계 원칙: Thin Controller, Fat Service

✅ 좋은 예 (Thin Controller)

Controller:
- 요청 파라미터 추출 (3줄)
- 입력 검증 (5줄)
- Service 호출 (1줄)
- 응답 반환 (1줄)

→ **총 10줄 이내!**

Service:
- 비즈니스 규칙 (20줄+)
- Repository 조합 (10줄+)
- 에러 처리 (5줄+)

→ **핵심 로직이 여기에!**

❌ 나쁜 예 (Fat Controller)

Controller:
- 입력 검증 (10줄)
- 비즈니스 규칙 (30줄)
- DB 쿼리 (15줄)
- 에러 처리 (10줄)

→ **65줄짜리 거대 핸들러!**

문제점:
- 테스트하려면 HTTP 서버 필요
- 다른 곳에서 재사용 불가
- 유지보수 악몽

→ **취업 면접에서 감점 요인!**

[계층1] Controller — 문지기 역할

Controller의 4가지 책임:

① 요청(Request) 접수
HTTP 메서드, URL, 파라미터 처리

② 입력 검증 (Validation)
필수 필드 확인, 타입 검사

③ Service에 위임
실제 로직은 Service에게 맡긴다

④ 응답(Response) 반환
적절한 HTTP 상태코드와 데이터 반환

👩‍💼 비유: 레스토랑 홀 매니저

- 손님(Client)의 주문을 접수
- 메뉴에 없는 건 거절 (검증)
- 주문서를 주방(Service)에 전달
- 완성된 음식을 손님에게 전달

⚠️ Fat Controller 안티패턴

Controller에 비즈니스 로직을 넣지 마세요!

- 재사용 불가 (HTTP에 종속)
- 테스트 어려움 (req, res 필요)
- 코드 비대화 → 유지보수 악몽

Controller 코드 예시 (Express.js)

```
// controllers/orderController.js
const orderService =
require("../services/orderService");

// ✅ Controller는 요청/응답만 담당!
const createOrder = async (req, res) => {
  try {
    // 1. 입력 검증
    const { userId, items } = req.body;
    if (!userId) return res.status(400)
      .json({ error: "userId는 필수입니다" });
    if (!items?.length) return res.status(400)
      .json({ error: "상품을 선택하세요" });
```

```
// controllers/orderController.js
    // 2. Service에 위임 (비즈니스 로직은 Service에!)
    const order = await orderService.createOrder(userId,
items);

    // 3. 응답 반환
    return res.status(201).json(order);
  } catch (err) {
    return res.status(500).json({ error: err.message });
  }
};
```

💡 **Controller는 얇게! req/res를 다루는 코드만 존재합니다.**

[계층2] Service — 요리사 (가장 중요한 계층!)

Service의 핵심 책임:

① 비즈니스 규칙 구현
"재고가 충분한가?", "할인 조건은?"

② 여러 Repository 조합
상품 조회 + 재고 확인 + 주문 생성

③ 트랜잭션 관리
"전부 성공하거나, 전부 실패하거나"

④ 에러 처리 & 비즈니스 예외
"재고 부족", "중복 주문" 등 비즈니스 에러

👨‍🍳 비유: 레스토랑 주방장

- 레시피(비즈니스 규칙)대로 요리
- 재료를 창고(Repository)에 요청
- 여러 재료를 조합하여 완성
- 손님에 대해서는 모른다! (HTTP 모름)

🚫 Service가 절대 하면 안 되는 것

"HTTP를 모른다!" — 절대 원칙

✗ req, res 객체 사용 금지
✗ HTTP 상태코드 설정 금지
✗ 직접 DB 쿼리 금지 → Repository 사용

SECTION 02

Service 코드 예시 — 주문 생성 비즈니스 로직

```
// services/orderService.js
const productRepo = require("../repositories/productRepo");
const orderRepo = require("../repositories/orderRepo");

// ✅ HTTP를 전혀 모른다! 순수 비즈니스 로직만!
const createOrder = async (userId, items) => {
  let totalPrice = 0;

  // 1. 각 상품의 재고 확인 (비즈니스 규칙)
  for (const item of items) {
    const product = await productRepo.findById(item.productId);
    if (!product) throw new Error("상품을 찾을 수 없습니다");
    if (product.stock < item.quantity)
      throw new Error(`${product.name} 재고가 부족합니다`);
    totalPrice += product.price * item.quantity;
  }
```

```
// services/orderService.js
  // 2. 주문 생성 & 재고 차감 (트랜잭션)
  const order = await orderRepo.create(userId, items, totalPrice);
  for (const item of items) {
    await productRepo.decreaseStock(item.productId, item.quantity);
  }
  return order;
};
```

💡 **req, res가 없다! → 테스트가 쉽다. CLI에서도, API에서도 재사용 가능!**

[계층3] Repository — 창고지기 역할

Repository의 핵심 책임:

① 데이터 CRUD만 담당
Create / Read / Update / Delete

② 비즈니스 로직을 모른다!
"왜 이 데이터를 조회하는지" 관심 없음

③ DB 기술에 종속
MySQL, PostgreSQL, MongoDB 등

핵심 포인트:
DB가 바뀌어도 Repository만 수정하면 된다!
Service와 Controller는 건드릴 필요 없음.

📦 비유: 레스토랑 창고지기

- 재료 보관 및 관리
- 입출고 기록
- 주방장이 요청한 재료를 가져다준다
- "이 재료로 뭘 만드는지"는 모른다

💡 배민 사례:

배달의민족은 MySQL에서 DynamoDB로 일부 전환 시, Repository 계층만 교체하여 비즈니스 로직 변경 최소화.

SECTION 02

Repository 코드 예시

```
// repositories/productRepo.js
const db = require("../config/database");

// ✅ 순수 DB 접근 코드만! 비즈니스 로직 없음!
const findById = async (productId) => {
  const [rows] = await db.query(
    "SELECT * FROM products WHERE id = ?",
    [productId]
  );
  return rows[0] || null;
};
```

```
// repositories/productRepo.js
const decreaseStock = async (productId, quantity) => {
  await db.query(
    "UPDATE products SET stock = stock - ? WHERE id = ?",
    [quantity, productId]
  );
};

const findAll = async () => {
  const [rows] = await db.query("SELECT * FROM products");
  return rows;
};

module.exports = { findById, decreaseStock, findAll };
```

💡 **DB가 바뀌면? 이 파일만 수정! Service/Controller는 그대로.**

SECTION 03

API 설계와 3계층의 연결

REST API 는 Controller 어떻게 연결되는가?
— HTTP Method, URI, 요청/응답 설계

SECTION 03

REST API란? — Controller와 HTTP의 연결

🌐 REST API 핵심 개념

RE presentational
S tate
T ransfer

→ HTTP를 통해 자원(Resource)의
상태를 주고받는 설계 방식

자원(Resource) = URI로 표현
행위(Action) = HTTP Method
표현(Representation) = JSON

→

📋 Controller 역할

HTTP 요청을 받아서
Service를 호출하고
HTTP 응답을 돌려준다

✓ URI → 어떤 자원?
✓ HTTP Method → 어떤 행위?
✓ Body/Params → 어떤 데이터?

⚠ **HTTP를 아는 건 Controller뿐!**
Service/Repository는 HTTP 모른다

💡 **핵심: API 엔드포인트 1개 = Controller 메서드 1개. REST API 설계 = Controller 설계의 시작!**

SECTION 03

HTTP Method → Controller 메서드 매핑

HTTP Method	URI 패턴	Controller 메서드	역할
GET	/api/users	getUsers(req, res)	목록 조회
GET	/api/users/:id	getUser(req, res)	단건 조회
POST	/api/users	createUser(req, res)	생성
PUT	/api/users/:id	updateUser(req, res)	전체 수정
PATCH	/api/users/:id	patchUser(req, res)	부분 수정
DELETE	/api/users/:id	deleteUser(req, res)	삭제

중첩 자원 (Nested Resource) 예시 — 게시글의 댓글:

```
POST   /api/posts/:postId/comments   →  createComment(req, res)  // 댓글 생성
GET    /api/posts/:postId/comments   →  getComments(req, res)    // 댓글 목록
```

⚠ **안티패턴: GET /api/deleteUser, POST /api/getUserList → HTTP Method를 무시하면 RESTful이 아님!**

SECTION 03

RESTful URI 설계 원칙 — 자원 이름을 잘 짓는 법

✅ 원칙 1: 명사 사용

❌ `/api/getUsers` `/api/createUser`

✅ `/api/users` `/api/users`

행위는 HTTP Method로 표현!

✅ 원칙 2: 복수형 사용

❌ `/api/user/:id` `/api/post`

✅ `/api/users/:id` `/api/posts`

컬렉션은 항상 복수형!

✅ 원칙 3: 계층 구조 표현

`/api/users/:id/orders`

`/api/orders/:id/items`

부모→자식 관계를 URI로 표현

✅ 원칙 4: 소문자 + 하이픈

❌ `/api/UserProfile` `/api/user_list`

✅ `/api/user-profiles` (복합어)

카멜케이스, 언더스코어 금지

실제 설계 예시 — 쇼핑몰 API:

```
GET    /api/products              # 상품 목록
GET    /api/products/:id          # 상품 상세
POST   /api/products/:id/reviews  # 리뷰 작성
POST   /api/cart/items            # 장바구니 담기
```

SECTION 03

HTTP 상태 코드 — API 응답의 언어

✅ 2xx 성공

200 OK — GET/PUT 성공
201 Created — POST 생성
204 No Content — DELETE

⚠️ 4xx 클라이언트 오류

400 Bad Request — 잘못된 요청
401 Unauthorized — 미인증
403 Forbidden — 권한 없음
404 Not Found — 자원 없음
409 Conflict — 중복 충돌

🔴 5xx 서버 오류

500 Internal Error
503 Service Unavailable
→ 클라이언트 잘못 아님!

3계층에서 상태 코드 결정은 Controller의 책임:

```
// Controller
async createUser(req, res) {
  try {
    const user = await userService.create(req.body);   // Service 호출
    res.status(201).json(user);  // ← 201 Created: Controller가 결정!
  } catch(e) {
    if (e.code === "DUPLICATE") res.status(409).json({ error: e.message });
  }
}  // Service는 HTTP 상태코드를 모른다!
```

SECTION 03

요청(Request) · 응답(Response) Body 설계

📥 요청 Body (Request)

```
// POST /api/users (회원가입)
{
  "email":    "alice@test.com",
  "password": "Str0ng!Pass",
  "name":     "Alice"
}
```

📤 응답 Body (Response)

```
// 201 Created 응답 (DTO 적용!)
{
  "id":        1,
  "email":     "alice@test.com",
  "name":      "Alice",
  "createdAt": "2024-01-15T09:00:00Z"
}  // password 없음! ← DTO 덕분에
```

요청 Body 설계 원칙:

① 필수 vs 선택 필드 명확히
② 민감 정보는 Body에 (URL 금지)
③ camelCase 사용 (JSON 표준)

응답 Body 설계 원칙:

① password, 내부 ID 등 제거 (DTO)
② 에러 응답 형식 통일
③ ISO 8601 날짜 형식 사용

에러 응답 표준 형식 (통일이 중요!):

```
{ "error": "이메일이 이미 사용중입니다.", "code": "DUPLICATE_EMAIL", "field": "email" }
// code: 클라이언트가 에러 종류를 구분할 수 있게 / field: 어떤 필드가 잘못됐는지
```

SECTION 03

API 버저닝 — 변경해도 기존 클라이언트를 깨뜨리지 않는 법

서비스를 운영하다 보면 API를 변경해야 할 때가 온다. 이미 앱에 배포된 클라이언트는 어떻게 할까?

방법 1

URI 버저닝

```
/api/v1/users
/api/v2/users
```

✅ 가장 직관적
✅ 브라우저 캐시 용이
⚠️ URL이 지저분해짐

→ 가장 많이 사용!
카카오, 네이버, 토스 채택

방법 2

헤더 버저닝

```
Accept:
application/vnd.api.v2+json
```

✅ URL 깔끔
✅ REST 원칙에 충실
⚠️ 브라우저 테스트 어려움

→ GitHub API 채택

방법 3

쿼리 파라미터

```
/api/users?version=2
```

✅ 구현 간단
⚠️ 캐시 어려움
⚠️ 표준 아님

→ 잘 쓰지 않음

💡 **실무 버저닝 전략**

① 처음부터 /api/v1/ 으로 시작 ② v1은 최소 6~12개월 유지 ③ Deprecation Notice(예고) 후 삭제
④ CHANGELOG에 변경 이유 상세 기록 ← 3계층에서 Controller 레벨만 바꾸면 Service는 그대로!

SECTION 03

OpenAPI(Swagger) — API 문서를 코드로 관리하는 법

"API 스펙이 문서에만 있으면? 코드가 바뀌어도 문서는 그대로 → 프론트엔드 팀과 매일 싸움!"

📄 OpenAPI Spec이란?

API 엔드포인트, 요청/응답 형식을
YAML/JSON으로 정의하는 표준

- ✅ Swagger UI 자동 생성
- ✅ 클라이언트 코드 자동 생성
- ✅ API 변경 시 문서 자동 갱신

실무 채택: 카카오, 토스, 쿠팡

```
// Express + swagger-jsdoc 예시

/**
 * @swagger
 * /api/v1/users:
 *   post:
 *     summary: 회원가입
 *     requestBody:
 *       $ref: #/components/UserDTO
 *     responses:
 *       '201': { description: Created }
 */
router.post("/users", createUser);  // ← Controller
```

API First 개발 프로세스 (3계층과 연동):

① OpenAPI Spec 작성 → ② Swagger UI 확인 (프론트/백 협의) → ③ Controller 구현 (스펙 일치)
④ Service/Repository 구현 → ⑤ API 변경 시 Spec 먼저 수정 → Controller 자동 문서화
→ 스펙이 진실의 원천(Single Source of Truth)! 코드와 문서가 항상 동기화됨

SECTION 03

DTO와 API 스펙 — 계층별 데이터 변환 패턴

```
Client → [RequestDTO] → Controller → Service → Repository → DB → Service → [ResponseDTO] → Controller → Client
```

📩 Request DTO

```
// createUserDto.js
class CreateUserDto {
  email;      // @IsEmail()
  password;   // @MinLength(8)
  name;       // @IsString()
}
```

→ 입력 검증 + 허용 필드 제한

📤 Response DTO

```
// userResponseDto.js
class UserResponseDto {
  id;         // 공개 OK
  email;      // 공개 OK
  name;       // 공개 OK
  // password ← 제외!
}
```

→ 민감정보 제거 + 필요 필드만

Controller에서의 DTO 변환 패턴:

```
async createUser(req, res) {
  const dto = new CreateUserDto(req.body);  // ① Request DTO 생성 (검증 포함)
  if (!dto.isValid()) return res.status(400).json({ error: dto.errors });
  const entity = await userService.create(dto); // ② Service에 DTO 전달
  const response = new UserResponseDto(entity); // ③ Response DTO로 변환
  res.status(201).json(response);              // ④ 응답 (password 없음!)
}
```

SECTION 03

API 설계 Before vs After — 실제 개선 사례

❌ Before (나쁜 설계)

```
GET  /api/getUser?id=1
POST /api/deleteUser
GET  /api/getAllPosts
POST /api/createNewPost
GET  /api/UserComment
```

문제점:

- HTTP Method 의미 무시
- 동사+명사 혼재
- 대소문자 불일치
- 단수/복수 혼재

✅ After (좋은 설계)

```
GET    /api/v1/users/:id
DELETE /api/v1/users/:id
GET    /api/v1/posts
POST   /api/v1/posts
GET    /api/v1/users/:id/comments
```

개선 포인트:

- 동사 → HTTP Method로 이동
- 명사 복수형으로 통일
- 소문자 + v1 버저닝 추가
- 중첩으로 관계 표현

💡 **체크리스트: URI에 동사가 있다? → HTTP Method로 이동 / 대문자가 있다? → 소문자로 / 단수형? → 복수형으로**

SECTION 03

실습 4: API 스펙 먼저 설계하기 (API First)

"코드보다 API 설계를 먼저! 스펙이 확정되면 프론트/백이 동시에 개발할 수 있다."

📋 실습 목표 (25분)

오늘 화이트보드 실습에서 설계한 게시판 서비스를 RESTful API 스펙으로 변환하세요.

Controller 설계 → HTTP Method + URI + 상태코드 + 요청/응답 Body 까지 완성!

단계별 가이드

Step 1	엔드포인트 목록 → Method + URI 형식으로 변환 예: "게시글 목록" → GET /api/v1/posts	(5분)
Step 2	각 API의 성공/실패 상태코드 정의 예: 게시글 생성 성공 → 201, 미인증 → 401, 없음 → 404	(5분)
Step 3	요청/응답 Body JSON 구조 설계 (DTO 적용) 예: POST /posts Body, GET /posts Response DTO	(10분)
Step 4	팀 발표: 설계 결정 이유 설명	(5분)

✅ 완성 체크리스트

☐ URI에 동사 없음 ☐ HTTP Method 적절 ☐ 상태코드 완비 ☐ DTO에 password 없음 ☐ 버저닝 적용

SECTION 03

API 설계 면접 질문 TOP 5

카카오, 토스, 네이버, 쿠팡 면접에서 자주 나오는 API 설계 질문 — 오늘 배운 것으로 답할 수 있다!

Q1. RESTful API란 무엇이고, REST 원칙 중 가장 중요한 것은?
→ 자원(명사 URI), 행위(HTTP Method), 표현(JSON). 핵심은 Stateless + 자원 중심 설계

Q2. GET과 POST의 차이? 언제 각각 사용하나요?
→ GET: 조회(멱등성 보장, 캐시 가능, Body 없음) / POST: 생성(비멱등, Body 있음, 201 반환)

Q3. PUT과 PATCH의 차이는?
→ PUT: 전체 교체(없으면 생성 가능) / PATCH: 부분 수정. 비밀번호 변경은 PATCH /users/:id

Q4. API 버저닝을 왜 하고 어떻게 하나요?
→ 기존 클라이언트를 깨뜨리지 않기 위해. URI 버저닝(/v1/, /v2/)이 가장 직관적이고 널리 사용됨

Q5. 왜 Entity를 그대로 API 응답으로 내보내면 안 되나요?
→ password, 내부 id 등 민감정보 노출 위험. DTO(Response DTO)로 필요한 필드만 선택해서 응답해야 함

SECTION 04

실습1: 엉망진창 코드 구경하기

스파게티 코드를 직접 보고, 문제점을 찾아보자!

SECTION 04

실습1: 목표 & 진행 방법

스파게티 코드의 문제점을 직접 발견하고, 개선 방향을 논의한다

1. **코드 읽기**
   스파게티 코드 3개를 순서대로 분석합니다 (15분)
2. **색칠하기**
   코드에서 Controller/Service/Repository 역할을 색으로 구분합니다 (10분)
3. **문제점 찾기**
   "DB를 바꾸면?", "테스트하려면?" 질문에 답해봅니다 (10분)
4. **토론하기**
   팀별로 문제점과 해결방안을 공유합니다 (10분)

스파게티 코드 예시 1: 회원가입 (Express.js)

```
// ❌ 모든 것이 하나의 라우트에! (스파게티)
app.post("/api/users/register", async (req, res) => {
  const { email, password, name } = req.body;
  if (!email || !email.includes("@"))
    return res.status(400).json({ error: "이메일 형식 오류" });
  if (!password || password.length < 8)
    return res.status(400).json({ error: "비밀번호 8자 이상" });
  const [existing] = await db.query(
    "SELECT id FROM users WHERE email = ?", [email]
  );
```

```
// ❌ 모든 것이 하나의 라우트에! (스파게티)
if (existing.length > 0)
    return res.status(409).json({ error: "이미 가입된 이메일" });
  const hashedPw = await bcrypt.hash(password, 10);
  const [result] = await db.query(
    "INSERT INTO users (email, password, name) VALUES
(?,?,?)",
    [email, hashedPw, name]
  );
  const token = jwt.sign({ userId: result.insertId }, SECRET);
  res.status(201).json({ token, userId: result.insertId });
});
```

🔍 **질문: 이 코드에서 Controller/Service/Repository 역할을 찾아보세요!**

SECTION 04

분석: 역할별 색칠하기 — 회원가입 코드

Controller 역할 (파란색):
- req.body에서 데이터 추출
- 입력 검증 (이메일 형식, 비밀번호 길이)
- res.status().json() 응답 반환

Service 역할 (초록색):
- 이메일 중복 체크 (비즈니스 규칙)
- 비밀번호 해싱 (보안 로직)
- JWT 토큰 생성 (인증 로직)

Repository 역할 (보라색):
- SELECT 쿼리 (이메일 존재 여부 조회)
- INSERT 쿼리 (사용자 데이터 저장)

→ 3가지 역할이 하나의 함수에 뒤섞여 있다!

스파게티 코드 예시 2: 게시글 작성

```
// ❌ 게시글 작성 — 하나의 핸들러에 모든 것
app.post("/api/posts", async (req, res) => {
  const token = req.headers.authorization?.split(" ")[1];
  if (!token) return res.status(401).json({ error: "로그인 필요" });
  let decoded;
  try { decoded = jwt.verify(token, SECRET); }
  catch { return res.status(401).json({ error: "토큰 만료" }); }
  const { title, content, category } = req.body;
  if (!title || title.length > 100)
    return res.status(400).json({ error: "제목 오류" });
```

```
// ❌ 게시글 작성 — 하나의 핸들러에 모든 것
// 금칙어 필터링 (비즈니스 로직)
  const banned = ["욕설1", "욕설2"];
  if (banned.some(w => content.includes(w)))
    return res.status(400).json({ error: "금칙어 포함" });
  const [result] = await db.query(
    "INSERT INTO posts (user_id,title,content,category) VALUES(?,?,?,?)",
    [decoded.userId, title, content, category]
  );
  // 포인트 적립 (비즈니스 로직)
  await db.query("UPDATE users SET point=point+10 WHERE
id=?",[decoded.userId]);
  res.status(201).json({ postId: result.insertId });
});
```

🔍 **문제: DB를 MongoDB로 바꾸면, 이 코드에서 몇 줄을 수정해야 할까?**

SECTION 04

분석: "DB를 바꾸면 어디를 수정해야 할까?"

❌ 스파게티 코드의 경우

MySQL → MongoDB 전환 시:

- 회원가입 핸들러: 2곳 수정
- 게시글 작성 핸들러: 3곳 수정
- 주문 처리 핸들러: 4곳 수정
- 리뷰 핸들러: 2곳 수정
- 결제 핸들러: 3곳 수정

→ 총 14곳 이상 수정 필요!

→ 수정 중 실수 확률 ↑↑↑

✅ 3계층 구조의 경우

MySQL → MongoDB 전환 시:

- userRepository.js: 1파일 수정
- postRepository.js: 1파일 수정
- orderRepository.js: 1파일 수정

→ Repository 파일만 수정!

→ Controller, Service는 그대로!

→ 테스트도 Repository만 다시!

💡 **쿠팡이 실제로 이렇게 함**

SECTION 04

스파게티 코드 예시 3: 주문 처리 (가장 복잡)

```
app.post("/api/orders", async (req, res) => {
  // 인증 체크
  const user = await db.query("SELECT * FROM users WHERE id=?",[req.body.userId]);
  if (!user[0].length) return res.status(404).json({error:"사용자 없음"});
  // 상품 검증 & 가격 계산
  let total = 0;
  for (const item of req.body.items) {
    const [p] = await db.query("SELECT * FROM products WHERE id=?",[item.id]);
    if (p[0].stock < item.qty) return res.status(400).json({error:"재고부족"});
    total += p[0].price * item.qty;
    // 할인 로직 (비즈니스)
    if (user[0][0].grade === "VIP") total *= 0.9;
  }
  // DB 저장
  await db.query("INSERT INTO orders VALUES(...)", [total]);
  for (const item of req.body.items)
    await db.query("UPDATE products SET stock=stock-? WHERE id=?",[item.qty,item.id]);
  // 알림 발송
  await fetch("https://slack.com/api/chat.postMessage",{method:"POST",...});
  res.json({ success: true, total });
});
```

🔍 **질문: "이 코드를 테스트하려면?" → DB, Slack API, HTTP 서버가 전부 필요!**

SECTION 04

분석: "이 코드를 테스트하려면?"

❌ 스파게티 코드 테스트

테스트에 필요한 것:

1. Express 서버 실행
2. MySQL 데이터베이스 연결
3. 테스트용 사용자 데이터 준비
4. 테스트용 상품 데이터 준비
5. Slack API 모킹
6. HTTP 클라이언트 (supertest 등)

→ 테스트 준비만 100줄!

→ "일단 배포하고 기도하기" 전략

✅ 3계층 구조 테스트

Service만 단위 테스트:

1. Mock Repository 주입
2. orderService.createOrder() 호출
3. 결과 검증

→ DB 없이, 서버 없이 테스트 가능!

💡 **토스의 개발 문화:**

테스트 커버리지 80% 이상 유지.

이건 좋은 구조 없이는 불가능!

SECTION 04

토론: 실무에서 스파게티 코드를 만나면?

💬 팀별 토론 주제 (10분)

Q1. 이미 스파게티 코드인 프로젝트에 투입되었습니다.
어디서부터 정리를 시작할 것인가?

Q2. 팀장이 "일단 돌아가니까 리팩터링은 나중에" 라고 합니다.
어떻게 설득할 것인가?

Q3. 당근마켓은 초기 모놀리식에서 시작해 성장 후 구조를 개선했습니다.
처음부터 완벽한 구조가 필요할까?

💡 **힌트: "완벽한 설계는 없다. 하지만 최소한의 구조는 필요하다"**

네이버 면접 단골 질문: "레거시 코드를 어떻게 개선하시겠습니까?"

SECTION 04

리팩터링 전략 — 어디서부터 시작할 것인가?

실무에서 스파게티 코드를 점진적으로 개선하는 4단계 전략

1. **Repository부터 분리하라**
   DB 접근 코드를 먼저 뽑아낸다. 가장 안전하고 효과가 크다.
2. **Service를 만들어라**
   비즈니스 로직을 Controller에서 Service로 이동. 재사용성 확보.
3. **Controller를 정리하라**
   요청/응답 처리만 남긴다. Thin Controller 완성.
4. **테스트를 추가하라**
   Service 단위 테스트를 작성. "리팩터링해도 안전하다"는 확신.

SECTION 05

실습2: 방 나누기 코드를 정리하자!

스파게티 코드를 3계층 구조로 리팩터링하는 실습

실습2 목표: 스파게티 코드를 3계층으로 분리하기

🎯 실습 목표

Slide 28의 "회원가입 스파게티 코드"를 3계층으로 리팩터링합니다.

완료 후 기대 결과:

- ✅ controllers/userController.js — 요청/응답만 처리 (15줄 이내)
- ✅ services/userService.js — 비즈니스 로직 담당
- ✅ repositories/userRepository.js — DB 접근만 담당

⏰ 진행 방법 (50분)

Step 1: 폴더 구조 만들기 (5분)
Step 2: Repository 분리 (15분) ← DB 접근 코드 먼저!
Step 3: Service 분리 (15분) ← 비즈니스 로직 이동
Step 4: Controller 정리 (10분) ← 요청/응답만 남기기
마무리: Before vs After 비교 (5분)

SECTION 05

Step 1: 폴더 구조 만들기

📁 프로젝트 구조:

```
my-service/
├── controllers/
│   └── userController.js
├── services/
│   └── userService.js
├── repositories/
│   └── userRepository.js
├── config/
│   └── database.js
├── routes/
│   └── userRoutes.js
├── app.js
└── package.json
```

💡 폴더명이 곧 아키텍처!

controllers/ = 문지기 방
- → HTTP 요청/응답 처리
- → req, res 사용 OK

services/ = 요리사 방
- → 비즈니스 로직 구현
- → req, res 사용 X

repositories/ = 창고지기 방
- → DB 접근 코드
- → SQL/ORM 쿼리

🏢 실제 회사의 구조:

토스, 배민, 카카오 모두
이 폴더 구조를 사용합니다.
면접에서도 이 구조를 물어봅니다!

SECTION 05

Step 2: Repository 분리 — DB 접근 코드를 뽑아내자

원칙: "DB와 직접 대화하는 코드"를 모두 Repository로 이동

찾아야 할 코드 패턴:

- db.query("SELECT ...") → findByEmail(email)
- db.query("INSERT ...") → create(userData)
- db.query("UPDATE ...") → update(id, data)
- db.query("DELETE ...") → remove(id)

💡 **핵심: Repository 함수 이름은 "무엇을 하는지" 명확하게! (findByEmail, createUser 등)**

SECTION 05

Repository 리팩터링 코드 예시

```
// repositories/userRepository.js
const db = require("../config/database");

const findByEmail = async (email) => {
  const [rows] = await db.query(
    "SELECT * FROM users WHERE email = ?", [email]
  );
  return rows[0] || null;  // 없으면 null 반환
};
```

```
// repositories/userRepository.js
const create = async ({ email, password, name }) => {
  const [result] = await db.query(
    "INSERT INTO users (email, password, name) VALUES (?, ?, ?)",
    [email, password, name]
  );
  return { id: result.insertId, email, name };
};

module.exports = { findByEmail, create };
```

✅ **SQL 쿼리는 이 파일에만 존재! DB가 바뀌면 이 파일만 수정.**

SECTION 05

Step 3: Service 분리 — 비즈니스 로직을 뽑아내자

원칙: "왜/어떻게 해야 하는가?"를 판단하는 코드를 Service로 이동

찾아야 할 코드 패턴:

- "이미 가입된 이메일인지 체크" → 비즈니스 규칙!
- "비밀번호를 해싱" → 보안 로직!
- "JWT 토큰 생성" → 인증 로직!
- "VIP면 10% 할인" → 비즈니스 규칙!

🚫 **Service에서 절대 하면 안 되는 것: req, res 사용, HTTP 상태코드 설정, 직접 DB 쿼리**

SECTION 05

Service 리팩터링 코드 예시

```
// services/userService.js
const userRepo = require("../repositories/userRepository");
const bcrypt = require("bcrypt");
const jwt = require("jsonwebtoken");

// ✅ HTTP를 전혀 모른다! 순수 비즈니스 로직만!
const register = async ({ email, password, name }) => {
  // 1. 이메일 중복 체크 (비즈니스 규칙)
  const existing = await userRepo.findByEmail(email);
  if (existing) throw new Error("이미 가입된 이메일입니다");
```

```
// services/userService.js
  // 2. 비밀번호 해싱 (보안 로직)
  const hashedPw = await bcrypt.hash(password, 10);

  // 3. 사용자 생성
  const user = await userRepo.create({email,
password:hashedPw, name});

  // 4. 토큰 발급
  const token = jwt.sign({ userId: user.id },
process.env.JWT_SECRET);
  return { token, userId: user.id };
};

module.exports = { register };
```

✅ **req, res가 없다! CLI, 테스트, 다른 서비스에서도 재사용 가능!**

Step 4: Controller 정리 — 요청/응답만 남기기

원칙: Controller에는 "요청 받기 → Service 호출 → 응답 반환"만!

Controller에 남겨야 할 것:

- ✅ req.body / req.params에서 데이터 추출
- ✅ 입력 검증 (필수 필드, 형식 체크)
- ✅ Service 함수 호출 (딱 1줄!)
- ✅ res.status().json()으로 응답 반환

SECTION 05

Controller 리팩터링 코드 예시

```
// controllers/userController.js
const userService = require("../services/userService");

// ✅ Thin Controller — 깔끔하고 명확!
const register = async (req, res) => {
  try {
    // 1. 입력 검증
    const { email, password, name } = req.body;
    if (!email || !email.includes("@"))
      return res.status(400).json({ error: "이메일 형식이 올바르지 않습니다" });
    if (!password || password.length < 8)
      return res.status(400).json({ error: "비밀번호는 8자 이상이어야 합니다" });
```

```
// controllers/userController.js
    // 2. Service에 위임 (핵심! 딱 1줄!)
    const result = await userService.register({ email, password, name });

    // 3. 응답 반환
    return res.status(201).json(result);
  } catch (err) {
    if (err.message.includes("이미 가입된"))
      return res.status(409).json({ error: err.message });
    return res.status(500).json({ error: "서버 오류" });
  }
};
```

✅ **Controller는 15줄 이내! 비즈니스 로직이 하나도 없다!**

Before vs After — 리팩터링 결과 비교

❌ Before (스파게티)

파일 1개에 모든 것:

routes/users.js:

- 입력 검증 (5줄)
- DB 조회 — SELECT (3줄)
- 비밀번호 해싱 (1줄)
- DB 삽입 — INSERT (3줄)
- JWT 생성 (1줄)
- 응답 반환 (1줄)

→ 총 20줄, 뒤섞인 관심사

문제: 수정 시 전체 영향
문제: 테스트 불가능

✅ After (3계층)

3개 파일로 명확한 분리:

Controller (7줄):
입력 검증 + Service 호출 + 응답

Service (10줄):
중복체크 + 해싱 + 생성 + 토큰

Repository (6줄):
findByEmail + create

장점: 각 파일 독립 수정 가능
장점: Service 단위 테스트 가능
장점: 포트폴리오에 좋은 구조!

SECTION 05

에러 처리 패턴 — 각 계층에서의 에러 처리

"에러도 각 계층의 역할에 맞게 처리한다"

📋 Controller

- 입력 검증 에러 → 400
- Service 에러 → 적절한 상태코드
- 예상치 못한 에러 → 500

"HTTP 에러 코드를 설정"

⚙️ Service

- 비즈니스 규칙 위반
  → throw new Error()
- "이메일 중복", "재고 부족"

"비즈니스 예외를 발생"

🗄️ Repository

- DB 연결 에러
- 쿼리 실행 에러
- 데이터 무결성 에러

"DB 관련 에러를 전파"

SECTION 06

서비스 설계 원칙 심화

의존성, DI, DTO, SOLID — 취업 면접에서 반드시 나오는 주제들

SECTION 06

의존성(Dependency)이란?

"요리사가 창고지기에게 재료 달라는 관계"

A가 B를 사용한다 = A는 B에 의존한다 (A → B)

3계층의 의존 관계:

Controller → Service (의존)
- Controller가 Service의 함수를 호출한다
- Service가 바뀌면 Controller에 영향 있음

Service → Repository (의존)
- Service가 Repository의 함수를 호출한다
- Repository가 바뀌면 Service에 영향 있음

⚠️ **절대 금지: Repository → Service (역방향 의존!)**
→ "순환 의존"이 생기면 변경의 파급효과가 무한 루프!

💡 **카카오 면접: "의존성 방향을 설명해보세요"**

SECTION 06

의존성 주입(DI) — 테스트가 쉬워지는 마법

❌ 하드코딩 의존성

```
// Service가 직접 Repository 생성
class OrderService {
  constructor() {
    // 직접 생성 (하드코딩)
    this.repo =
      new MySQLOrderRepo();
  }
}
```

문제점:

- MySQL 없이 테스트 불가
- DB 교체 시 Service 수정 필요

✅ 의존성 주입 (DI)

```
// 외부에서 Repository를 주입
class OrderService {
  constructor(orderRepo) {
    // 외부에서 받음 (주입)
    this.repo = orderRepo;
  }
}
```

장점:

- Mock 주입으로 테스트 쉬움!
- DB 교체 시 Service 변경 불필요
- **네이버/카카오 면접 단골 주제**

SECTION 06

DTO vs Entity — 데이터를 넘기는 올바른 방법

Entity (엔티티)

정의:
DB 테이블과 1:1 매핑되는 객체

예시:
{ id, email, password,
name, createdAt,
updatedAt, deletedAt }

특징:
- DB의 모든 컬럼을 포함
- password 등 민감 정보 포함
- Repository ↔ Service 사이에서 사용

DTO (Data Transfer Object)

정의:
계층 간 데이터 전달용 객체

예시 (응답 DTO):
{ id, email, name }
→ password 제외!

특징:
- 필요한 데이터만 선택적으로 포함
- 민감 정보 제거
- Controller ↔ Client 사이에서 사용

💡 **토스 면접: "왜 Entity를 그대로 API 응답으로 내보내면 안 되나요?"**

유지보수를 위한 설계 — "6개월 후의 나도 다른 사람이다"

🔑 유지보수가 쉬운 코드의 4가지 조건:

① 읽기 쉬운 코드 (Readable)
→ 변수명, 함수명이 역할을 설명한다
→ "createOrder"는 이해되지만 "process1"은 의미 불명

② 바꾸기 쉬운 코드 (Changeable)
→ 한 곳을 수정해도 다른 곳에 영향이 없다
→ 3계층 분리의 핵심 이유!

③ 테스트 가능한 코드 (Testable)
→ DB 없이도 비즈니스 로직을 검증할 수 있다
→ 의존성 주입(DI)의 핵심 이유!

④ 재사용 가능한 코드 (Reusable)
→ 같은 Service를 API, CLI, 배치잡에서 사용
→ 배민: 같은 주문 Service를 앱/웹/POS에서 공유

SECTION 06

SOLID 원칙 — 유지보수하기 좋은 코드의 5가지 원칙

S **Single Responsibility (단일 책임 원칙)**
하나의 클래스는 하나의 이유로만 변경되어야 한다

O **Open/Closed (개방-폐쇄 원칙)**
확장에는 열려있고, 수정에는 닫혀있어야 한다

L **Liskov Substitution (리스코프 치환 원칙)**
자식 클래스는 부모 클래스를 대체할 수 있어야 한다

I **Interface Segregation (인터페이스 분리 원칙)**
사용하지 않는 인터페이스에 의존하지 않아야 한다

D **Dependency Inversion (의존성 역전 원칙)**
추상화에 의존하라, 구체적인 것에 의존하지 마라

SECTION 06

단일 책임 원칙(SRP) 심화 — 오늘의 핵심!

❌ SRP 위반 예시

```
class UserService {
  register(user) { ... }
  login(email, pw) { ... }
  sendEmail(to, body) { ... }
  generatePDF(user) { ... }
  uploadImage(file) { ... }
}
```

문제:
이메일 라이브러리가 바뀌면
UserService를 수정해야 한다!
→ 변경 이유가 5가지!

✅ SRP 준수 예시

```
class UserService {
  register(user) { ... }
  login(email, pw) { ... }
}
class EmailService {
  send(to, body) { ... }
}
class PDFService { ... }
class ImageService { ... }
```

장점:
각 클래스는 하나의 이유로만 변경!
→ 변경 범위가 명확!

SECTION 07

실습3: 서비스를 직접 설계하자!

게시판 서비스를 3계층으로 설계하는 화이트보드 실습

SECTION 07

화이트보드 설계 실습 — 게시판 서비스

📋 요구사항:

1. 게시글 CRUD (작성/목록/상세/수정/삭제) | 2. 댓글 작성/삭제 | 3. 좋아요/좋아요 취소
4. 인증 (로그인한 사용자만 작성 가능) | 5. 게시글 검색 (제목/내용) | 6. 페이지네이션

🎯 실습 과제 (팀별 30분):

1. 필요한 API 엔드포인트를 나열하세요 (10분)
   예: POST /api/posts, GET /api/posts, GET /api/posts/:id ...
2. 각 엔드포인트의 Controller/Service/Repository 역할을 정의하세요 (15분)
   Controller: 어떤 입력을 검증? / Service: 어떤 비즈니스 규칙? / Repository: 어떤 쿼리?
3. 팀별 화이트보드에 그림으로 표현하세요 (5분)
   → 발표 준비!

SECTION 07

설계 가이드: Controller 설계

```
// 게시판 Controller 설계 힌트

PostController:
  POST   /api/posts        → createPost(req, res)
    입력 검증: title(필수, 100자), content(필수)
  GET    /api/posts        → getPosts(req, res)
    쿼리 파라미터: page, limit, search
  GET    /api/posts/:id    → getPost(req, res)
  PUT    /api/posts/:id    → updatePost(req, res)
    검증: 본인 작성 게시글인지 확인은 Controller? Service?
  DELETE /api/posts/:id    → deletePost(req, res)

CommentController:
  POST   /api/posts/:id/comments → createComment(req, res)
  DELETE /api/comments/:id       → deleteComment(req, res)

LikeController:
  POST   /api/posts/:id/like     → toggleLike(req, res)
```

💡 **질문: "본인 확인"은 Controller에서? Service에서? → 토론해보세요!**

설계 가이드: Service & Repository 설계

⚙️ Service 설계 힌트

PostService:

createPost(userId, title, content)
→ 금칙어 필터링
→ 작성자 포인트 적립
getPosts(page, limit, search)
getPost(postId)
→ 조회수 증가
updatePost(userId, postId, data)
→ 본인 확인 (비즈니스 규칙)
deletePost(userId, postId)
→ 본인 확인 + 댓글도 삭제

CommentService:

create(userId, postId, content)
delete(userId, commentId)

🗄️ Repository 설계 힌트

PostRepository:

findAll(offset, limit)
findById(id)
findBySearch(keyword, offset, limit)
create(postData)
update(id, data)
remove(id)
incrementViewCount(id)

CommentRepository:

findByPostId(postId)
create(commentData)
remove(id)

LikeRepository:

findByUserAndPost(userId, postId)
create(userId, postId)
remove(userId, postId)

SECTION 07

팀별 발표 & 피드백

📢 발표 가이드 (팀당 5분):

1. 설계한 API 엔드포인트 소개
2. Controller / Service / Repository 역할 설명
3. 팀에서 가장 고민했던 포인트 공유

🔍 피드백 체크리스트:

☐ Controller에 비즈니스 로직이 섞여있지 않은가?
☐ Service가 HTTP(req, res)를 알고 있지 않은가?
☐ Repository에 비즈니스 판단이 포함되지 않았는가?
☐ 의존 방향이 Controller → Service → Repository 인가?
☐ 각 함수의 이름이 역할을 잘 설명하는가?

💡 **"완벽하지 않아도 괜찮습니다. 중요한 건 왜 이렇게 나눴는가를 설명할 수 있는 것!"**

정리

1일차 핵심 요약 — 오늘 배운 것

1. **스파게티 코드의 위험성**
   모든 로직이 섞이면: 변경 파급효과, 테스트 불가, 재사용 불가, 협업 충돌
2. **관심사의 분리 (SoC)**
   각 모듈은 하나의 관심사만! 병원 비유: 접수→진료→처방→약국
3. **3계층 아키텍처**
   Controller(문지기) → Service(요리사) → Repository(창고지기)
4. **의존성 방향 규칙**
   위에서 아래로만! 냉장고가 바뀌어도 요리법은 안 바뀐다
5. **Thin Controller, Fat Service**
   Controller는 얇게(요청/응답만), Service는 두껍게(비즈니스 로직)

취업 면접에서 나오는 아키텍처 질문 TOP 5

1. **Q1. 3계층 아키텍처란 무엇인가요?**
   → 오늘 배운 Controller/Service/Repository 구조를 설명
2. **Q2. 왜 비즈니스 로직을 Controller에 넣으면 안 되나요?**
   → 재사용성, 테스트 용이성, Fat Controller 안티패턴
3. **Q3. 의존성 주입(DI)이 뭔가요?**
   → 외부에서 의존 객체를 주입, Mock으로 테스트 가능
4. **Q4. DTO와 Entity의 차이는?**
   → Entity는 DB 매핑, DTO는 계층 간 데이터 전달용
5. **Q5. 레거시 코드를 어떻게 개선하시겠습니까?**
   → Repository부터 분리, 점진적 리팩터링, 테스트 추가

내일 예고: Day 2 — 서비스가 커지면?

🏗️ Day 2: 서비스가 커지면? — MSA와 API Gateway

오늘: 하나의 서비스 내부 구조 (3계층 아키텍처)
내일: 여러 서비스를 어떻게 연결하고 관리할 것인가?

Day 2 주요 주제:

- 모놀리식 vs 마이크로서비스 — 언제 MSA가 필요한가?
- API Gateway — 수백 개의 서비스를 하나의 입구로
- 서비스 간 통신 — REST vs 메시지 큐
- 실습: MSA 설계 워크샵

💡 **배달의민족은 모놀리식에서 MSA로 전환하는데 2년이 걸렸습니다.**
내일은 "왜 그렇게 했는지, 어떻게 했는지"를 배웁니다!

과제: 자신의 프로젝트를 3계층으로 설계해오기

📝 과제 안내

자신이 만들고 싶은 서비스(또는 기존 프로젝트)를 3계층으로 설계하세요.

제출 내용:

1. 서비스 개요 (한 줄 설명)
   예: "중고 도서 거래 플랫폼" (당근마켓 스타일)
2. API 엔드포인트 목록 (최소 5개)
   POST /api/books, GET /api/books, GET /api/books/:id, ...
3. 각 엔드포인트의 3계층 역할 정의
   Controller: 어떤 검증? / Service: 어떤 비즈니스 규칙? / Repository: 어떤 쿼리?
4. 폴더 구조도
   controllers/, services/, repositories/ 안에 어떤 파일?

💡 **이 과제가 곧 취업 포트폴리오의 README가 됩니다!**