Vanilla PHP
API 아키텍쳐

워드프레스식 프론트 컨트롤러로 만드는 실전 PHP API 서버

`/api/{module}/{controller}/{method}`

Controller → DTO → Service → DAO → Response

초판 · 2026.05.01

머리말

이 책은 프레임워크 없이 순수 PHP로 API 서버를 구성하는 실전 아키텍처를 설명한다. 목표는 Laravel이나 Slim 같은 프레임워크를 대체하는 것이 아니다. 오히려 프레임워크가 내부에서 해결하는 문제를 직접 만나보고, 기존 PHP 사이트에도 안전하게 API 계층을 추가할 수 있는 기준 구조를 만드는 것이다.

> **핵심 Vanilla PHP는 구조가 없는 PHP가 아니다. 프레임워크가 대신 정해주던 구조를 개발자가 직접 설계하는 PHP다.**

예제 구조는 Apache .htaccess 기반 프론트 컨트롤러, PHP 내장 서버용 router.php, /api 요청 분기, 모듈형 Controller, DTO 기반 Request 매핑, Service/DAO 계층, PDO DB 연결, .env.local 환경 변수 관리, JSON Response 표준화로 구성된다.

이 책의 대상 독자

- Laravel 없이 작은 API 서버 또는 MVP 백엔드를 빠르게 만들고 싶은 개발자
- 기존 PHP, 워드프레스, 카페24, 사내 레거시 사이트에 API 계층을 붙여야 하는 개발자
- 프레임워크의 Router, Controller, Middleware, Service 구조를 직접 이해하고 싶은 개발자
- 외부 API와 LLM API를 서버 측 Service 계층으로 감싸고 싶은 개발자

기준 아키텍처

```
/api/{module}/{controller}/{method}

GET  /api/test/health
POST /api/translate/translate/do
GET  /api/kakao/daumSearch/web
POST /api/customer/login/excuteLogin
```

목차

1장. 전체 구조와 설계 원칙

이 아키텍처는 "모든 요청은 index.php로 모으되, /api로 시작하는 요청만 API 라우터가 처리한다"는 단순한 원칙에서 출발한다. 이 방식은 기존 웹사이트와 API를 한 서버에서 함께 운영해야 하는 환경에 적합하다.

요청 처리 흐름

```
Client
  ↓
Apache .htaccess
  ↓
index.php
  ↓
Route::init($_SERVER['REQUEST_URI'])
  ↓
/api 요청 여부 판단
  ↓
api/common/route.php
  ↓
module / controller / method 해석
  ↓
Controller → Service → DAO 또는 외부 API
  ↓
Response::json()
```

기준 폴더 구조

```
/
├── .htaccess
├── router.php
├── index.php
└── api/
    ├── common/
    │   ├── route.php
    │   ├── response.php
    │   ├── request.php
    │   ├── service.php
    │   ├── dao.php
    │   ├── dbconn.php
    │   └── constant.php
    ├── test/
    ├── customer/
    ├── kakao/
    ├── translate/
    └── board/
```

설계 원칙

원칙	설명
단일 진입점	실제 파일/디렉터리를 제외한 요청은 index.php로 모아 제어한다.
API 경로 분리	/api로 시작하는 요청만 API 라우터가 처리한다.
규칙 기반 라우팅	/api/{module}/{controller}/{method} 규칙으로 파일과 클래스를 찾는다.
계층 분리	Controller는 흐름, Service는 비즈니스 로직, DAO는 SQL을 담당한다.
공통 응답	모든 API는 JSON 응답과 HTTP 상태 코드를 일관되게 반환한다.
점진적 개선	처음부터 거대한 프레임워크를 만들지 않고 필요한 기능만 단계적으로 보강한다.

핵심 작은 API 서버일수록 복잡한 추상화보다 "URL 규칙, 파일 규칙, 응답 규칙"을 먼저 고정하는 것이 중요하다.

2장. .htaccess와 Front Controller

Apache 환경에서는 .htaccess가 요청의 첫 관문이다. 아래 설정은 워드프레스식 프론트 컨트롤러 구조를 따른다. 실제 파일이나 디렉터리가 있으면 그대로 제공하고, 그 외 요청은 index.php로 전달한다.

```
<IfModule mod_rewrite.c>
RewriteEngine On
RewriteBase /
RewriteRule ^index\.php$ - [L]
RewriteCond %{REQUEST_FILENAME} !-f
RewriteCond %{REQUEST_FILENAME} !-d
RewriteRule . /index.php [L]
</IfModule>
```

API 요청의 캐시 방지

정적 리소스는 적극적으로 캐시해도 되지만 API 응답은 보통 사용자 상태, 인증, 시간, 외부 API 결과에 따라 달라진다. 따라서 /api 요청에는 no-store, no-cache 정책을 부여한다.

```
<IfModule mod_headers.c>
    SetEnvIf Request_URI "^/api(/|$)" IS_API_REQUEST=1
    Header always set Cache-Control "no-store, no-cache, must-revalidate, max-age=0" env=IS_API_REQUEST
    Header always set Pragma "no-cache" env=IS_API_REQUEST
    Header always set Expires "0" env=IS_API_REQUEST
</IfModule>
```

보안 헤더와 .env 보호

```
Header set X-Frame-Options SAMEORIGIN
Header set X-Content-Type-Options nosniff

<FilesMatch "^\.env">
    Require all denied
</FilesMatch>
```

X-Frame-Options는 외부 사이트의 iframe 삽입을 제한하고, X-Content-Type-Options는 브라우저의 MIME 타입 추측을 막는다. .env 접근 차단은 최소한의 안전장치이며, 가능하면 .env 파일은 웹 루트 밖에 두는 것이 더 안전하다.

> **주의** WordPress 블록 사이의 설정은 워드프레스가 덮어쓸 수 있다. 직접 추가한 보안/캐시/API 설정은 WordPress 블록 밖에 두는 것이 안전하다.

3장. 개발 서버용 router.php

로컬 개발 환경에서는 Apache rewrite 설정을 쓰지 않고 PHP 내장 서버로 빠르게 실행할 수 있다. 이때 router.php는 실제 파일과 디렉터리를 직접 제공하고, 나머지 요청은 index.php로 전달한다.

```
<?php
// Development router for PHP built-in server.
$path = parse_url($_SERVER['REQUEST_URI'], PHP_URL_PATH);
if (is_string($path)) {
    $fullPath = __DIR__ . $path;
    if ($path !== '/' && (is_file($fullPath) || is_dir($fullPath))) {
        return false;
    }
}

require __DIR__ . '/index.php';
```

실행 명령

```
php -S localhost:8000 router.php
```

동작 방식

요청	처리
/assets/app.js	파일이 존재하면 PHP 내장 서버가 직접 제공한다.
/api/test/health	실제 파일이 아니므로 index.php로 전달한다.
/some/page	실제 파일이 아니므로 index.php로 전달한다.

4장. /api 라우터의 동작 원리

Route 클래스는 /api 요청만 감지해 API 라우터로 넘긴다. 일반 웹 페이지 요청은 기존 사이트 로직이 처리할 수 있도록 남겨둔다.

```
class Route
{
    public static function init($reqUri)
    {
        if (Route::isAPI($reqUri)) {
            Route::routeApi($reqUri);
            exit;
        }
    }

    private static function isAPI($reqUri)
    {
        $path = parse_url($reqUri, PHP_URL_PATH);
        if (!is_string($path)) {
            return false;
        }
        return strpos($path, '/api') === 0;
    }
}
```

URL 세그먼트 분해

```
$path = parse_url($reqUri, PHP_URL_PATH);
$segments = explode('/', trim($path, '/'));

$module = $segments[1];
$controller = $segments[2];
$method = isset($segments[3]) ? $segments[3] : null;
```

라우팅 예시

URL	module	controller	method
/api/test/health	test	health	defaultMethod
/api/kakao/daumSearch/web	kakao	daumSearch	web
/api/translate/translate/do	translate	translate	do

식별자 검증

동적 파일 로딩 구조에서는 URL 값이 곧 파일 경로와 클래스명에 영향을 준다. 따라서 module, controller, method는 반드시 안전한 식별자 형태로 제한해야 한다.

```
private static function isValidIdentifier($value)
{
    return is_string($value) && preg_match('/^[A-Za-z][A-Za-z0-9_]*$/', $value) === 1;
}
```

핵심 식별자 검증은 디렉터리 트래버설, 의도치 않은 파일 include, 내부 메서드 노출을 줄이기 위한 최소 방어선이다.

5장. Controller 자동 로딩과 allowedMethods

라우터는 module과 controller 이름을 조합해 Controller 파일을 찾는다. 파일이 없으면 404, 클래스가 없으면 500을 반환한다.

```
$controllerFile = 'api/' . $module . '/controller/' . $controller . 'Controller.php';
if (!file_exists($controllerFile)) {
    Response::json([
        'result' => false,
        'message' => 'Controller not found.',
        'requestPath' => $path,
    ], 404);
    return;
}

require($controllerFile);
$ctrName = $controller . 'Controller';
```

allowedMethods가 필요한 이유

URL로 클래스의 public method를 호출하는 구조는 편리하지만, 실수로 공개하면 안 되는 public method까지 노출될 수 있다. allowedMethods는 외부에서 호출 가능한 메서드를 명시적으로 제한한다.

```
class healthController {
    public $allowedMethods = ['defaultMethod'];

    public function defaultMethod(){
        $service = new HealthService();
        $service->check();
    }
}
```

defaultMethod 규칙

URL에 method가 없으면 defaultMethod를 실행할 수 있다. 이 규칙을 사용하면 /api/test/health 같은 간결한 엔드포인트를 만들 수 있다.

```
if ($method === null || $method === '') {
    if (method_exists($ctr, 'defaultMethod')) {
        $ctr->defaultMethod();
        return;
    }
}
```

6장. Request, DTO, Response 표준화

API 서버에서 가장 먼저 흔들리는 부분은 요청 데이터와 응답 데이터다. 컨트롤러마다 $_GET, $_POST, php://input을 직접 읽기 시작하면 구조가 금방 흩어진다. 따라서 DTO와 Response 클래스로 경계를 만든다.

JSON Body를 DTO로 매핑

```
class Request {
    public static function setJsonParam(&$class)
    {
        $reflect = new ReflectionClass($class);
        $props = $reflect->getProperties(ReflectionProperty::IS_PUBLIC);

        $json = file_get_contents('php://input');
        $data = json_decode($json);

        if (!is_object($data)) {
            return;
        }

        foreach ($props as $prop) {
            $name = $prop->getName();
            if (property_exists($data, $name)) {
                $class->{$name} = $data->{$name};
```

```
            }
        }
    }
}
```

DTO 예시

```
class TranslateRequestDTO {
    public $text;
    public $source_lang;
    public $target_lang;
}
```

Response 클래스

```
class Response {
    public static function jsonheader(){
        header('Access-Control-Allow-Origin: *');
        header('Content-Type: application/json; charset=UTF-8');
    }

    public static function json($payload, $statusCode = 200){
        http_response_code($statusCode);
        echo json_encode($payload);
    }
}
```

추천 응답 포맷

상황	예시
성공	{"result_code":200,"result":"success","data":{...}}
검증 실패	{"result_code":400,"result":"fail","message":"no param: text"}
인증 실패	{"result_code":401,"result":"fail","message":"unauthorized"}
서버 오류	{"result_code":500,"result":"fail","message":"internal server error"}

> **개선** 일부 코드가 echo json_encode를 직접 사용하더라도, 책의 최종 구조에서는 Response::json으로 통일하는 것이 좋다.

7장. Service, DAO, DB 연결 계층

Controller가 모든 일을 처리하면 API가 커질수록 유지보수가 어려워진다. 이 구조에서는 Controller는 요청 흐름을 담당하고, Service는 비즈니스 로직, DAO는 SQL 실행을 담당한다.

Service 기본 클래스

```
class service {
    protected $pdo;

    public function __construct($isDbCon = 'N')
    {
        if($isDbCon == Constants::DB_CONN){
            require_once('api/common/dbconn.php');
            $this->pdo = new dbconn();
        }
    }
}
```

DAO 기본 클래스

```
class dao {
    protected $pdo;

    public function __construct($pdo)
```

```
    {
        $this->pdo = $pdo;
    }
}
```

PDO 기반 DB 연결

```
class dbconn extends PDO
{
    public function __construct()
    {
        self::loadEnvFromFile();
        $dbHost = self::readEnv('DB_HOST', 'localhost');
        $dbName = self::readEnv('DB_NAME');
        $dbUser = self::readEnv('DB_USER');
        $dbPass = self::readEnv('DB_PASS');
        $dbCharset = self::readEnv('DB_CHARSET', 'utf8mb4');

        $dsn = sprintf('mysql:host=%s;dbname=%s;charset=%s', $dbHost, $dbName, $dbCharset);
        parent::__construct($dsn, $dbUser, $dbPass);
        $this->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
    }
}
```

계층별 책임

계층	책임	하지 말아야 할 일
Controller	DTO 생성, Service 호출, 응답 반환	SQL 직접 실행, 복잡한 비즈니스 로직
Service	업무 규칙, 외부 API 호출, DAO 조합	HTTP 세부 처리 남발
DAO	SQL 작성, DB 조회/수정	응답 JSON 생성
DTO/VO	데이터 전달	로직 수행

8장. Health Check API 구현

Health Check API는 구조를 검증하는 가장 작은 API다. DB나 외부 API 없이 라우터, 컨트롤러, 서비스, 응답 흐름이 정상인지 확인할 수 있다.

요청

```
GET /api/test/health
```

healthController.php

```
<?php
require('api/test/service/HealthService.php');

class healthController {
    public $allowedMethods = ['defaultMethod'];

    public function defaultMethod(){
        $service = new HealthService();
        $service->check();
    }
}
```

HealthService.php

```
<?php
class HealthService extends service {
    public function check(){
        Response::json([
            'result' => true,
            'message' => 'API is healthy',
        ], 200);
    }
}
```

응답

```
{
  "result": true,
  "message": "API is healthy"
}
```

핵심 새 모듈을 추가할 때는 Health Check 수준의 가장 작은 API부터 만든 뒤, DTO, Service, DAO를 차례로 붙이는 방식이 안전하다.

9장. Customer/Login API 구조

customer 모듈은 DB 기반 업무 API 구조를 설명하기 좋다. 로그인 요청은 DTO로 받고, Transformer를 거쳐 내부 VO로 변환한 뒤 DAO를 통해 DB를 조회한다.

```
customer/
├── controller/
│   ├── loginController.php
│   ├── signupController.php
│   └── appController.php
├── service/
│   ├── LoginService.php
│   ├── SignUpService.php
│   └── AppService.php
├── dao/
│   └── LoginDAO.php
├── dto/
│   └── LoginRequestDTO.php
├── vo/
│   └── LoginRequestVO.php
└── transformer/
    └── LoginTransformer.php
```

DTO와 VO를 분리하는 이유

DTO는 외부 요청 형식에 가깝고, VO는 내부 처리 형식에 가깝다. API 입력 필드명과 DB/업무 처리 필드명이 달라질 수 있으므로 둘을 분리하면 변경에 강해진다.

```
Client JSON
  ↓
LoginRequestDTO
  ↓
LoginTransformer
  ↓
LoginRequestVO
  ↓
LoginDAO
  ↓
DB
```

로그인 API에서 점검할 것

- 비밀번호는 반드시 password_hash/password_verify 기반으로 처리한다.
- 실패 사유를 과도하게 자세히 노출하지 않는다.
- 로그인 실패 횟수와 IP를 기록해 비정상 시도를 감지한다.
- 토큰을 발급한다면 만료 시간과 재발급 정책을 분리한다.

10장. Kakao API Proxy 구조

kakao 모듈은 외부 API를 서버 측 Service 계층으로 감싸는 예제다. 클라이언트가 외부 API 키를 직접 들고 호출하지 않고, 서버가 API 호출을 대행한다.

```
kakao/
├── controller/
│   ├── addressSearchController.php
│   ├── channelController.php
│   ├── daumSearchController.php
│   └── loginController.php
```

```
├──── service/
│     ├──── AddressSearchService.php
│     ├──── ChannelService.php
│     ├──── DaumSearchService.php
│     ├──── KakaoAPIService.php
│     └──── LoginService.php
└──── dto/
    ├──── AddressSearchRequestDTO.php
    ├──── DaumSearchRequestDTO.php
    └──── LoginCallBackRequestDTO.php
```

Proxy 패턴의 장점

장점	설명
키 보호	REST API Key, Admin Key, Client Secret을 브라우저에 노출하지 않는다.
응답 표준화	외부 API 응답을 내부 API 포맷으로 변환할 수 있다.
오류 통제	외부 API 오류 메시지를 사용자에게 그대로 노출하지 않고 정리할 수 있다.
로깅	요청/응답, 실패 코드, 지연 시간을 서버에서 기록할 수 있다.

예상 요청 흐름

```
Client
  ↓
GET /api/kakao/daumSearch/web?query=...
  ↓
daumSearchController
  ↓
DaumSearchService
  ↓
KakaoAPIService
  ↓
Kakao/Daum Search API
  ↓
Response::json()
```

> **핵심** 외부 API 연동은 Controller에 직접 넣지 말고, Service 또는 전용 API Client 클래스로 분리하는 것이 유지보수에 유리하다.

11장. OpenAI Translate API 구조

translate 모듈은 LLM API를 Vanilla PHP API 구조에 붙이는 예제다. JSON 요청을 DTO로 받고, Service에서 API Key를 확인한 뒤 OpenAI API를 호출한다.

```
translate/
├──── controller/
│     └──── translateController.php
├──── service/
│     └──── TranslateService.php
├──── dto/
│     └──── TranslateRequestDTO.php
├──── config.php
└──── validate-config.php
```

요청 DTO

```
class TranslateRequestDTO {
    public $text;
    public $source_lang;
    public $target_lang;
}
```

Controller

```
class TranslateController {
```

```
    public function defaultMethod()
    {
        (new TranslateService())->translate(new TranslateRequestDTO());
    }

    public function do()
    {
        (new TranslateService())->translate(new TranslateRequestDTO());
    }
}
```

Service의 핵심 책임

- 필수 파라미터 text, target_lang을 검증한다.
- OPENAI_API_KEY를 환경 변수 또는 config.php에서 읽는다.
- 번역 지시문을 구성한다.
- curl로 OpenAI API를 호출한다.
- 성공/실패 응답을 JSON으로 반환한다.

개선 포인트

현재 구조	개선 방향
echo json_encode 직접 사용	Response::json으로 통일
temperature 0.3 고정	설정 파일 또는 상수로 분리
API URL과 모델 상수	환경별 config로 분리 가능
60초 timeout	서비스 성격에 따라 10~30초로 조정 검토

12장. 보안, 캐시, 운영 체크리스트

Vanilla PHP API는 프레임워크가 제공하는 보호 장치가 적기 때문에, 보안과 운영 기준을 명시적으로 갖고 있어야 한다.

보안 체크리스트

- 동적 include 경로에 들어가는 module/controller/method는 정규식으로 검증한다.
- 외부 호출 가능한 method는 allowedMethods로 제한한다.
- DB 쿼리는 PDO Prepared Statement를 사용한다.
- 비밀번호는 password_hash/password_verify로 처리한다.
- API Key, DB 계정, Client Secret은 코드에 직접 쓰지 않는다.
- 운영 환경에서는 display_errors를 끄고 로그 파일에만 기록한다.
- CORS는 개발 중에는 *를 쓰더라도 운영에서는 허용 Origin을 제한한다.
- 외부 API 호출에는 timeout을 반드시 둔다.

캐시 정책

대상	권장 정책
/api/*	no-store, no-cache, must-revalidate
이미지	public, max-age 2주~1년
CSS/JS	파일명 버전 해시를 쓰고 장기 캐시
HTML/PHP 페이지	짧은 private 캐시 또는 must-revalidate

운영 로그

로그	내용
access log	요청 경로, IP, User-Agent, 응답 코드, 처리 시간
error log	예외 메시지, stack trace, 발생 위치
api log	module/controller/method, result_code, elapsed_ms
security log	인증 실패, 권한 실패, rate limit 초과

13장. 리팩터링 로드맵

현재 구조는 학습용이면서 실무 적용 가능한 기본 뼈대를 갖고 있다. 다만 운영 품질을 높이려면 다음 순서로 점진적으로 개선하는 것이 좋다.

1단계: 응답 포맷 통일

```
Response::json([
    'result_code' => 200,
    'result' => 'success',
    'data' => $data,
], 200);
```

2단계: HTTP Method 검증

```
public $allowedMethods = [
    'defaultMethod' => ['GET'],
    'do' => ['POST'],
];
```

3단계: Autoload 도입

초기에는 require_once로 충분하지만, 파일이 늘어나면 Composer autoload 또는 간단한 PSR-4 스타일 autoloader를 도입하는 것이 좋다.

```
spl_autoload_register(function ($class) {
    $base = __DIR__ . '/../';
    $file = $base . str_replace('\', '/', $class) . '.php';
    if (is_file($file)) {
        require_once $file;
    }
});
```

4단계: 예외 처리기 추가

Controller와 Service마다 try-catch를 반복하는 대신 전역 예외 처리기를 두면 에러 응답이 일관된다.

```
set_exception_handler(function (Throwable $e) {
    error_log($e);
    Response::json([
        'result_code' => 500,
        'result' => 'fail',
        'message' => 'internal server error',
    ], 500);
});
```

5단계: 테스트와 문서화

- Health Check는 배포 후 가장 먼저 검증한다.
- Postman 또는 Bruno 컬렉션으로 엔드포인트를 문서화한다.
- Service 단위 테스트와 DAO 통합 테스트를 분리한다.
- OpenAPI 문서는 후반에 자동화해도 늦지 않다.

부록 A. Starter Kit 폴더 구조

```
vanilla-api-core/
```

```
├──── .htaccess
├──── router.php
├──── index.php
├──── .env.local.example
├──── api/
│    ├──── common/
│    │    ├──── route.php
│    │    ├──── request.php
│    │    ├──── response.php
│    │    ├──── service.php
│    │    ├──── dao.php
│    │    ├──── dbconn.php
│    │    └──── exception.php
│    ├──── test/
│    │    ├──── controller/healthController.php
│    │    └──── service/HealthService.php
│    ├──── customer/
│    ├──── kakao/
│    └──── translate/
└──── storage/
     └──── logs/
```

새 모듈 추가 절차

1. api/{module}/controller 디렉터리를 만든다.
2. api/{module}/service 디렉터리를 만든다.
3. 요청 데이터가 있으면 dto 디렉터리와 DTO 클래스를 만든다.
4. DB가 필요하면 dao 디렉터리와 DAO 클래스를 만든다.
5. Controller에서 allowedMethods를 정의한다.
6. Service에서 비즈니스 로직을 작성한다.
7. curl 또는 브라우저로 /api/{module}/{controller}/{method}를 테스트한다.

부록 B. Cursor/Codex 개발 지시 프롬프트

아래 프롬프트는 이 구조를 기반으로 Vanilla PHP API Starter Kit을 생성하거나 개선할 때 사용할 수 있다.

```
당신은 Vanilla PHP 기반 API 서버를 구현하는 시니어 백엔드 개발자다.
프레임워크는 사용하지 않는다. Laravel, Slim, CodeIgniter를 사용하지 않는다.

목표:
- Apache .htaccess와 PHP 내장 서버 router.php를 모두 지원한다.
- 모든 /api 요청은 index.php에서 Route::init으로 분기한다.
- API URL 규칙은 /api/{module}/{controller}/{method} 이다.
- Controller, Service, DAO, DTO 계층을 분리한다.
- Response::json으로 모든 JSON 응답을 통일한다.
- module/controller/method는 안전한 식별자 정규식으로 검증한다.
- Controller에는 allowedMethods를 두어 외부 호출 가능 메서드를 제한한다.
- DB 연결은 PDO와 .env.local 기반으로 처리한다.
- 예제 모듈로 test/health, translate, customer/login을 구현한다.

산출물:
1. 전체 폴더 구조
2. 각 파일의 완성 코드
3. 실행 방법
4. curl 테스트 예시
5. 보안 체크리스트

주의:
- API Key와 DB 비밀번호는 코드에 직접 쓰지 않는다.
- 운영 환경에서 display_errors를 켜지 않는다.
- 외부 API 호출에는 timeout을 설정한다.
- echo json_encode 직접 사용을 피하고 Response::json을 사용한다.
```

맺음말

프레임워크 없이 API 서버를 만든다는 것은 모든 것을 직접 구현하겠다는 뜻이 아니다. 핵심은 요청 흐름, 라우팅 규칙, 계층 분리, 응답 표준, 보안 기준을 스스로 이해하고 필요한 만큼만 구현하는 것이다.

이 책의 구조는 작은 MVP, 내부 도구, 테스트베드, 외부 API 프록시, LLM API 실험 서버에 특히 잘 맞는다. 프로젝트가 커지면 Laravel이나 Slim 같은 프레임워크로 넘어가도 된다. 중요한 것은 그때 프레임워크를 "마법"이 아니라 "이미 이해한 구조의 확장"으로 볼 수 있다는 점이다.

마지막 문장 좋은 아키텍처는 거창한 이름에서 나오지 않는다. 요청을 어디서 받고, 어떤 규칙으로 나누고, 어디에서 검증하고, 어떻게 응답할지 명확히 정하는 데서 시작한다.