NestJS Web Programming: Beginner

(Integrative-Generative AI Edition)

Contents:

Author: Student Price Book Center

คำนำ

การพัฒนาแอปพลิเคชันฝั่งเซิร์ฟเวอร์ (Backend) ในปัจจุบันมีความซับซ้อนและหลากหลายมากขึ้นเรื่อย ๆ นักพัฒนาจำเป็นต้องเข้าใจทั้งแนวคิดการออกแบบซอฟต์แวร์สมัยใหม่ การจัดการโครงสร้างโค้ดให้ง่ายต่อการบำรุงรักษา รวมถึงการสร้างระบบที่สามารถขยายตัวได้ในอนาคต **NestJS** จึงเป็นหนึ่งในเฟรมเวิร์กที่ตอบโจทย์เหล่านี้ได้อย่างครบถ้วน ด้วยการผสานแนวคิดจาก **Object-Oriented Programming (OOP)**, **Functional Programming (FP)** และ **Dependency Injection (DI)** เข้ากับภาษา **TypeScript** NestJS ช่วยให้นักพัฒนาสามารถสร้างแอปพลิเคชันที่มีความมั่นคง ปลอดภัย และรองรับระบบขนาดใหญ่ได้

หนังสือเล่มนี้ **"NestJS Web Programming: Beginner"** ถูกออกแบบมาเพื่อเป็นคู่มือสำหรับนักพัฒนาที่ต้องการเริ่มต้นกับ NestJS ตั้งแต่พื้นฐานไปจนถึงการสร้างระบบที่ซับซ้อนและเป็นมาตรฐานอุตสาหกรรม โดยแต่ละบทจะปูพื้นฐานและต่อยอดอย่างเป็นขั้นตอน เริ่มจากการแนะนำ NestJS และโครงสร้างพื้นฐานในบทแรก ซึ่งครอบคลุมตั้งแต่แนวคิด ความเป็นมาของ NestJS ข้อแตกต่างจาก Framework และ Library ยอดนิยมบน Node.js ไปจนถึงการติดตั้ง Nest CLI การสร้างโปรเจกต์ใหม่ และการทดสอบโปรเจกต์บน IDE/Editor ที่หลากหลาย

ต่อมาในบทที่ 2 และบทที่ 3 ผู้อ่านจะได้เรียนรู้การสร้าง **Controller** และการจัดการ **Routing** เบื้องต้น รวมถึงการสร้าง **Service** และการนำแนวคิด **Dependency Injection (DI)** มาใช้เพื่อแยกตรรกะทางธุรกิจออกจาก Controller การเข้าใจวิธีการ inject service เข้า Controller และการจัดการ logic อย่างเป็นระบบ จะช่วยให้โค้ดอ่านง่าย ทดสอบง่าย และขยายระบบได้โดยไม่ซับซ้อน

บทที่ 4 จะเน้นไปที่ระบบ **Modules ของ NestJS** ซึ่งเป็นกลไกสำคัญในการจัดการโปรเจกต์ขนาดใหญ่ การสร้างโมดูลย่อย การ Import และ Export โมดูล ตลอดจนการใช้ **Shared Module** เพื่อรวม logic หรือ Service ที่ใช้ร่วมกันหลายโมดูล ทำให้ระบบมีความเป็นระเบียบ ขยายได้ง่าย และลดการซ้ำซ้อนของโค้ด แนวคิดเรื่องโมดูลยังช่วยให้การทำงานเป็นทีมเป็นไปอย่างมีประสิทธิภาพ และรองรับการพัฒนาระบบที่ซับซ้อน

บทที่ 5 จะสอนเรื่อง **Data Transfer Objects (DTOs) และ Validation** ซึ่งเป็นหัวใจสำคัญในการจัดการข้อมูลที่ผู้ใช้ส่งเข้ามา การใช้ไลบรารี **class-validator** และ **class-transformer** ช่วยให้สามารถตรวจสอบและแปลงข้อมูลให้อยู่ในรูปแบบที่ถูกต้อง การสร้าง DTO พร้อม type-check การตรวจสอบ input และการจัดการ Validation Error จะช่วยให้ API มีความปลอดภัย ใช้งานได้จริง และให้ feedback ที่ชัดเจนต่อผู้ใช้

หนังสือเล่มนี้ยังมีตัวอย่างบูรณาการและ Ultimate Integrated Project ในแต่ละบท เพื่อให้ผู้อ่านได้ฝึกปฏิบัติจริงและเห็นภาพรวมของการพัฒนาแอปพลิเคชัน NestJS ตั้งแต่เริ่มต้นจนถึงระบบที่ซับซ้อน รวมถึง Super Ultimate NestJS Project ที่รวมความรู้จากทุกบทให้ผู้อ่านสามารถประยุกต์สร้างโปรเจกต์จริงได้ทันที

ท้ายที่สุด หนังสือเล่มนี้ถูกเขียนด้วยความตั้งใจให้ผู้อ่าน **เข้าใจแนวคิดเชิงลึก ควบคู่ไปกับการปฏิบัติจริง** ไม่ว่าจะเป็นนักพัฒนาที่มีประสบการณ์ Node.js มาก่อนหรือผู้เริ่มต้นที่เพิ่งรู้จัก NestJS ก็ตาม เราหวังว่าหนังสือเล่มนี้จะเป็นคู่มือที่ช่วยให้คุณพัฒนาทักษะ สร้างระบบที่เป็นมาตรฐาน และเข้าใจแนวทางการออกแบบแอปพลิเคชันในเชิงมืออาชีพ

ด้วยรักและปรารถนาดี

ศูนย์หนังสือราคานักเรียน

สารบัญ

หน้า

บทที่ 1
แนะนำ NestJS และโครงสร้างพื้นฐาน
(Introduction to NestJS and Basic Structure)

เนื้อหา

- แนะนำ NestJS และโครงสร้างพื้นฐาน
- ประวัติและความเป็นมา
- ข้อแตกต่างหลักของ NestJS เทียบกับ Framework/Library ยอดนิยมบน Node.js
- NestJS คืออะไร?
- แนวคิด OOP, FP และ DI ใน NestJS
- โครงสร้างโฟลเดอร์มาตรฐานของโปรเจกต์ NestJS
- การติดตั้ง Nest CLI และการสร้างโปรเจกต์ใหม่
- การติดตั้งและทดสอบ NestJS บน IDE/Editor
- การติดตั้ง NestJS และทดสอบโปรเจกต์บน NetBeans IDE

บทนำบทที่ 1: แนะนำ NestJS และโครงสร้างพื้นฐาน

ในยุคที่แอปพลิเคชันฝั่งเซิร์ฟเวอร์มีความซับซ้อนมากขึ้นเรื่อย ๆ นักพัฒนาจำเป็นต้องใช้เครื่องมือที่ช่วยจัดการโครงสร้างโค้ดให้ชัดเจนและรองรับการขยายในอนาคต **NestJS** จึงกลายเป็นหนึ่งในเฟรมเวิร์กที่โดดเด่น เพราะออกแบบมาให้สอดคล้องกับหลักการเขียนโปรแกรมสมัยใหม่ พร้อมทั้งใช้ **TypeScript** เป็นภาษาหลัก ทำให้นักพัฒนาสามารถสร้างระบบที่ปลอดภัย มีประสิทธิภาพ และบำรุงรักษาได้ง่าย

NestJS คือเฟรมเวิร์กสำหรับสร้าง **Server-Side Application** โดยทำงานบน Node.js และได้รับแรงบันดาลใจจากสถาปัตยกรรมของ Angular ซึ่งใช้แนวคิดแบบโมดูลาร์ (Modular) และสนับสนุนการพัฒนาแบบ **MVC (Model–View–Controller)** และ *MV (เช่น MVVM)** ได้อย่างยืดหยุ่น ความสามารถเหล่านี้ทำให้ NestJS เป็นที่นิยมในการสร้างแอปพลิเคชันระดับองค์กร เช่น ระบบจัดการผู้ใช้ ระบบ API ขนาดใหญ่ หรือระบบ Microservices

หนึ่งในหัวใจสำคัญของ NestJS คือการนำแนวคิดเชิงสถาปัตยกรรมของ **OOP (Object-Oriented Programming)**, **FP (Functional Programming)** และ **DI (Dependency Injection)** มาผสมผสานเข้าด้วยกัน OOP ช่วยให้นักพัฒนาสามารถแยกความรับผิดชอบออกจากกันได้ชัดเจน FP ช่วยให้การประมวลผลข้อมูลมีความยืดหยุ่นและเขียนโค้ดที่อ่านง่ายขึ้น และ DI ทำให้การจัดการการพึ่งพาระหว่าง

คลาสต่าง ๆ มีความเป็นอิสระต่อกัน ส่งผลให้โค้ดทดสอบได้ง่ายและสามารถปรับเปลี่ยนได้อย่างคล่องตัว

เมื่อพูดถึงโครงสร้างโค้ด NestJS ได้วาง **โครงสร้างโฟลเดอร์มาตรฐาน** ที่รองรับการทำงานระดับใหญ่ เช่น การแยกโฟลเดอร์สำหรับ Controller ที่จัดการกับการรับส่งคำขอ การแยก Service สำหรับเก็บตรรกะทางธุรกิจ และการใช้ Module เพื่อจัดกลุ่มองค์ประกอบเหล่านี้ให้เป็นเอกเทศ การจัดโครงสร้างเช่นนี้ช่วยให้นักพัฒนาสามารถทำงานร่วมกันเป็นทีมได้อย่างมีระบบ ลดความสับสน และช่วยให้ระบบขยายตัวได้อย่างมีประสิทธิภาพ

นอกจากแนวคิดเชิงสถาปัตยกรรมแล้ว NestJS ยังมาพร้อมเครื่องมืออำนวยความสะดวกอย่าง **Nest CLI (Command Line Interface)** ซึ่งช่วยให้การสร้างโปรเจกต์ใหม่ การเพิ่มโมดูล การสร้าง Controller หรือ Service ทำได้อย่างรวดเร็วเพียงใช้คำสั่งไม่กี่บรรทัด Nest CLI ยังช่วยกำหนดโครงสร้างเริ่มต้นของโปรเจกต์ให้อยู่ในรูปแบบมาตรฐาน ทำให้แม้แต่นักพัฒนามือใหม่ก็สามารถเริ่มต้นได้อย่างถูกต้องตั้งแต่แรก

ในส่วนการปฏิบัติ บทนี้จะสาธิตการ **ติดตั้ง Nest CLI** และการ **สร้างโปรเจกต์ใหม่** โดยเริ่มจากการติดตั้งผ่าน npm หรือ yarn จากนั้นจะแสดงวิธีการใช้คำสั่ง nest new project-name เพื่อสร้างโครงสร้างโปรเจกต์อัตโนมัติ พร้อมไฟล์และโฟลเดอร์ที่ครบถ้วน เช่น app.controller.ts, app.service.ts, และ app.module.ts สิ่งนี้จะช่วยให้คุณได้เห็นภาพรวมของการทำงานใน NestJS ได้อย่างชัดเจน

กล่าวโดยสรุป บทนี้ไม่เพียงแต่ทำให้คุณเข้าใจว่า **NestJS คืออะไรและใช้ทำอะไร** แต่ยังช่วยปูพื้นฐานความเข้าใจเกี่ยวกับ **แนวคิด OOP, FP และ DI**, การจัดโครงสร้างโฟลเดอร์ของโปรเจกต์ และการใช้ Nest CLI เพื่อสร้างแอปพลิเคชันแรกของคุณ ความรู้เหล่านี้จะเป็นรากฐานสำคัญในการต่อยอดไปสู่บทถัดไป ซึ่งเราจะได้ลงลึกในเรื่องการสร้าง API และการเชื่อมต่อกับระบบต่าง ๆ อย่างมีประสิทธิภาพ

แนะนำ NestJS และโครงสร้างพื้นฐาน

- NestJS คืออะไร? ใช้ทำอะไร?
- แนวคิด OOP, FP, และ DI ใน Nest
- โครงสร้างโฟลเดอร์มาตรฐานของโปรเจกต์
- การติดตั้ง Nest CLI และสร้างโปรเจกต์ใหม่

บทที่ 1 — แนะนำ NestJS และโครงสร้างพื้นฐาน (ละเอียดแต่เข้าใจง่าย)

ต่อไปนี้เป็นการอธิบายเชิงลึกแบบเป็นขั้นเป็นตอนสำหรับหัวข้อที่คุณขอ — เหมาะสำหรับเริ่มต้นโครงการจริงและเข้าใจแนวคิดพื้นฐานเชิงสถาปัตยกรรม

---

1) NestJS คืออะไร? ใช้ทำอะไร?

NestJS คือ **ฟราเมิร์กสำหรับพัฒนา Backend บน Node.js** เขียนด้วย **TypeScript** (รองรับ JavaScript) ออกแบบมาให้เขียนโค้ดแบบมีโครงสร้าง เหมาะกับโปรเจกต์ขนาดเล็กถึงระดับองค์กร

(enterprise).

ใช้งานหลัก ได้แก่

- สร้าง REST APIs
- สร้าง GraphQL APIs (code-first / schema-first)
- สร้าง microservices (gRPC, TCP, NATS, Kafka, RabbitMQ ฯลฯ)
- WebSocket / real-time apps
- ใช้เป็น backend service ในสถาปัตยกรรมแบบหลายบริการ

เหตุผลที่ชอบใช้ NestJS

- โครงสร้างชัด (Modules / Controllers / Providers) — เหมาะกับทีมใหญ่
- มี Dependency Injection (DI) ในตัว ทำให้ทดสอบง่าย
- สนับสนุน patterns ยอดนิยม (CQRS, Clean Architecture, Hexagonal)
- รวม integration กับ ORM (TypeORM, MikroORM), Prisma, validation, config ฯลฯ

---

2) แนวคิด OOP, FP, และ DI ใน NestJS (ละเอียด)

NestJS ผสานแนวคิด 3 แบบหลัก — **OOP (Object-Oriented), FP (Functional Programming), DI (Dependency Injection)** — เพื่อความยืดหยุ่นและความเป็นระบบ

OOP (Object-Oriented Programming)

- Nest ใช้ **คลาส** เป็นหน่วยหลัก (@Controller(), @Injectable(), @Module() คือ class-decorators)
- แนวคิด OOP ที่เด่น: **encapsulation, inheritance, polymorphism**
- ตัวอย่าง: สร้าง UsersService เป็น class แล้ว inject ให้ Controller — แยกหน้าที่ (SRP)

```
@Injectable()
export class UsersService {
  private users: User[] = [];
  create(userDto: CreateUserDto) { /* ... */ }
  findAll() { return this.users; }
}
```

FP (Functional Programming) ใน Nest

- Nest สนับสนุน FP โดยการใช้ **ฟังก์ชันบริสุทธิ์**, การไม่เปลี่ยน state, และโดยเฉพาะ **RxJS / Observables** สำหรับการจัดการ asynchronous/stream data
- ตัวอย่าง: controller หรือ service สามารถคืนค่าเป็น Observable<T> แล้ว pipe operators (map, switchMap, catchError) เพื่อจัดการ flow
- FP pattern ช่วยในการเขียน logic ที่ทดสอบง่ายและ composable

```
// ตัวอย่าง service ที่คืน Observable
```

```
import { of } from 'rxjs';
import { delay } from 'rxjs/operators';

getUsers() {
  return of(this.users).pipe(delay(100)); // เป็น stream ของข้อมูล
}
```

DI (Dependency Injection) — หัวใจของ Nest

- Nest มี DI container ในตัว — providers (services, repositories, factories) ถูก **register** ใน module และ **inject** เข้ากับคลาสที่ต้องการผ่าน constructor
- ข้อดี: แยก concerns, รองรับการ mock สำหรับ unit test, ดูแล lifecycle ของ dependencies

```
@Controller('users')
export class UsersController {
  constructor(private readonly usersService: UsersService) {} // injection ที่ constructor
}
```

รูปแบบ provider ที่สำคัญ

- useClass — ให้ใช้อีก class หนึ่ง
- useValue — ให้ value คงที่ (config, mock)
- useFactory — สร้าง provider ด้วย factory function (รองรับ DI ภายใน factory)
- useExisting — alias ให้ token เดิม

```
{
  provide: 'PAYMENT_SERVICE',
  useClass: StripePaymentService,
}
// หรือ
{
  provide: 'CONFIG',
  useValue: { host: 'localhost' }
}
```

Scope ของ provider

- Singleton (default) — instance เดียวตลอดแอป
- Request — ใหม่ทุก request (ใช้กับ request-scoped data)
- Transient — สร้างใหม่ทุกครั้งที่ inject

```
@Injectable({ scope: Scope.REQUEST })
export class RequestScopedService { /* ... */ }
```

3) โครงสร้างโฟลเดอร์มาตรฐานของโปรเจกต์ (Best practice)

โครงสร้างทั่วไปที่รักษา scalability และความชัดเจน:

```
project-root/
├── src/
│   ├── main.ts                 # bootstrap app
│   ├── app.module.ts
│   ├── config/                 # ConfigModule, env schema
│   ├── common/                 # filters, guards, interceptors, pipes
│   │   ├── filters/
│   │   ├── guards/
│   │   ├── interceptors/
│   │   └── pipes/
│   ├── modules/                # ฟีเจอร์เป็น sub-folders
│   │   ├── users/
│   │   │   ├── users.module.ts
│   │   │   ├── users.controller.ts
│   │   │   ├── users.service.ts
│   │   │   ├── dto/
│   │   │   └── entities/
│   │   └── auth/
│   ├── shared/                 # shared utilities, constants, types
│   └── infrastructure/         # DB adapters, repositories, external API clients
├── test/
├── .env
├── nest-cli.json
├── package.json
├── tsconfig.json
└── Dockerfile
```

คำอธิบายเพิ่มเติม

- src/modules/<feature> — แยกเป็น feature-based modules (Users, Auth, Products) — เพิ่มความเป็น modular
- common/ — เก็บ cross-cutting concerns (เช่น global exception filter, logging interceptor)

- infrastructure/ — adapter ของ DB หรือ third-party (เช่น repository ที่ติด TypeORM/Prisma) — ช่วยแยก layer ตาม Hexagonal/Clean Architecture
- config/ — ใช้ @nestjs/config เพื่อ load .env และ schema validation
- test/ — unit / e2e tests

ตัวอย่างไฟล์สำคัญ

- main.ts — ตั้งค่า global pipes, global filters, CORS, adapters
- app.module.ts — root module ที่ import feature modules
- nest-cli.json — config ของ CLI (sourceRoot ฯลฯ)

---

4) การติดตั้ง Nest CLI และสร้างโปรเจกต์ใหม่ (ทีละขั้นตอน พร้อมคำสั่งจริง)

มี 2 วิธีหลัก: ติดตั้ง CLI แบบ global หรือใช้ npx (ไม่ต้องติดตั้งถาวร)

ติดตั้งแบบ global (แนะนำถ้าจะใช้บ่อย)

```
npm i -g @nestjs/cli
# หรือ
yarn global add @nestjs/cli
```

จากนั้นสร้างโปรเจกต์

```
nest new my-nest-app
# CLI จะถามให้เลือก package manager: npm / yarn / pnpm
```

หรือใช้ npx (ไม่ติดตั้งถาวร)

```
npx @nestjs/cli new my-nest-app
```

คำสั่งสำคัญหลังสร้างโปรเจกต์

```
cd my-nest-app
npm run start          # รัน production build (dist)
npm run start:dev      # รัน dev แบบ watch (hot reload)
npm run build          # build -> dist/
```

ใช้ CLI เพื่อ generate scaffolding (เร็วและเป็นมาตรฐาน)

```
nest g module users
nest g controller users --no-spec
nest g service users --no-spec
# หรือ full resource (module+controller+service)
nest g resource users
```

ตัวอย่าง main.ts หลัง scaffold — แนะนำการตั้งค่าเริ่มต้น

```
// src/main.ts
import { ValidationPipe } from '@nestjs/common';
```

```
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';

async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  // global validation และ transform DTO -> class instance ได้อัตโนมัติ
  app.useGlobalPipes(new ValidationPipe({ whitelist: true, transform: true }));
  app.enableCors();
  await app.listen(3000);
}
bootstrap();
```

ตัวอย่าง package.json scripts (ที่ CLI ใส่มาให้)

```
"scripts": {
  "start": "nest start",
  "start:dev": "nest start --watch",
  "build": "nest build",
  "format": "prettier --write \"src/**/*.ts\""
}
```

เพิ่มเติม (tips แบบผู้เชี่ยวชาญสั้น ๆ)

- เปิดใช้ ValidationPipe และ class-validator เพื่อความปลอดภัยของ input (whitelist/transform)
- ใช้ @nestjs/config สำหรับจัดการ .env และตั้งค่า isGlobal: true เพื่อเรียกใช้ทั่วแอป
- แยก **feature modules** ให้เล็กและชัด — แต่ละ module ควรมี controller/service/dto/entities ของตัวเอง
- ใช้ repository pattern (interface + provider) เพื่อให้ง่ายต่อการเปลี่ยน ORM/DB และง่ายต่อการ mock ใน unit tests
- เริ่มต้นโครงการด้วย eslint/prettier และตั้ง CI (lint + tests) ตั้งแต่แรก

ประวัติและความเป็นมา

- NestJS ถูกเริ่มพัฒนาโดย **Kamil MyŚliwiec** โดยแรงบันดาลใจจาก Angular (ในแง่ architecture, decorators, modularity) เพื่อสร้าง framework บน Node.js ที่โครงสร้างชัดและรองรับ TypeScript โดยตรง (วิกิพีเดีย)

- เผยแพร่ครั้งแรกในปี **2017** โดยมี version ที่แท็กแรกคือ **v4.4.0** เมื่อวันที่ 23 พฤศจิกายน 2017 (วิกิพีเดีย)
- ปรับปรุงเรื่อยมา เพิ่ม adapter และ driver ต่าง ๆ เช่น Express เป็น default, ต่อมาเพิ่ม Fastify เพื่อประสิทธิภาพสูงขึ้น (วิกิพีเดีย)
- เพิ่ม integration กับ message brokers เช่น RabbitMQ, Kafka เพื่อรองรับ architecture แบบ distributed / microservices (วิกิพีเดีย)

วิวัฒนาการสำคัญ / จุดเปลี่ยน

จุดเวลา	สิ่งที่เกิดขึ้น
2017	NestJS เปิดตัวครั้งแรก — สร้างบน Express + TypeScript โดยเอาแนวคิด Angular มาปรับใช้ในฝั่ง backend (วิกิพีเดีย)
หลังปี 2017	เริ่มมี feature เพิ่ม เช่น support for Fastify adapter, support for message brokers, เพิ่ม libraries/modules ต่างๆ (GraphQL, WebSockets, microservices) (วิกิพีเดีย)
ปัจจุบัน (2024–2025)	NestJS เข้าสู่ version 10.x/11.x — มี migration guide สำหรับการอัปเดต major versions เช่น v11 ที่ drop support Node.js เก่า, เน้น performance, security, tooling ปรับปรุง (NestJS Docs)

สถิติการใช้งาน / ความนิยม

นี่คือข้อมูลที่พอมีให้ ณ ปัจจุบัน (ไม่ทั้งหมด แต่ให้ภาพรวม):

- NestJS มี **GitHub stars** จำนวนมาก — อยู่ในหมวด frameworks/node.js ที่ได้รับความสนใจสูง (วิกิพีเดีย)
- ใน Latin America, มีจำนวน engineer ที่ใช้ NestJS มากขึ้น:
  - Brazil ~ 6,500 คน (matildaexp.com)
  - Argentina ~ 1,200 คน (matildaexp.com)
  - Colombia ~ 1,100 คน (matildaexp.com)
  - ประเทศอื่น ๆ ใน LATAM ก็มีการเติบโตสูง เช่น Mexico, Peru, Chile (matildaexp.com)
- การเติบโตรายปีของ engineers ที่ใช้ NestJS ใน LATAM สูง — บางประเทศเพิ่มขึ้นเกือบ 80–100%+ ต่อปี (matildaexp.com)
- สถานะล่าสุด: NestJS มีเวอร์ชันล่าสุด เช่น v10.x, และ v11 ที่เริ่มมีการเปลี่ยนแปลงสำคัญ (drop support Node.js เก่า, ปรับปรุง features) (NestJS Docs)

ข้อจำกัดของข้อมูล / สิ่งที่ยังไม่ชัด

- ข้อมูลสถิติบางอย่าง เช่น จำนวนผู้ใช้ทั่วโลก, จำนวนดาวน์โหลด npm ที่แม่นยำ, จำนวนบริษัทที่ใช้จริง, market share เปรียบเทียบกับ framework อื่นๆ ยังไม่มีข้อมูล公開อย่างครบถ้วนหรือแม่นยำเสมอ
- มากกว่าเน้น region (เช่น Latin America) มากกว่า global ในหลายแหล่ง

ข้อแตกต่างหลักของ NestJS เทียบกับ Framework/Library ยอดนิยมบน Node.js

ข้อแตกต่างหลักของ NestJS เทียบกับ Framework/Library ยอดนิยมบน Node.js เช่น **Express.js, Koa.js, Fastify** และ **AdonisJS** (รวมทั้งเปรียบเทียบกับการเขียน JS ปกติแบบ raw Node.js)

☐ **ความแตกต่างของ NestJS กับ JS Framework อื่น ๆ**

ประเด็น	NestJS	Express.js	Koa.js	Fastify	AdonisJS
โครงสร้าง (Architecture)	มี **Modular Architecture**, ใช้ **Modules / Controllers / Providers**, ชัดเจน เหมาะกับทีมใหญ่	ไม่มีโครงสร้างตายตัว ต้องจัดการเอง	Minimal, middleware-driven, developer กำหนดเอง	เน้น performance, ใช้ schema validation (JSON Schema)	มีโครงสร้างใกล้เคียง Laravel (MVC), ออกแบบ opinionated
ภาษา (Language)	เน้น **TypeScript** โดยตรง, รองรับ JS	JS เป็นหลัก, TS ต้อง config เอง	JS/TS, แต่ TS integration ไม่ดีเท่า Nest	มี TS support ดี แต่ไม่ opinionated	TS + JS, มี built-in decorators
แนวคิดหลัก	ผสาน **OOP + FP + DI**	Callback / Middleware	Middleware-first	Schema-based validation	MVC + Dependency Injection
Dependency Injection (DI)	มี DI container ในตัว (คล้าย Angular)	ไม่มี ต้องจัดการเอง	ไม่มี ต้องใช้ lib เสริม	มี basic DI (แต่ไม่เทียบเท่า Nest)	มี DI แบบ built-in
Scaffolding /	มี Nest CLI, generate	ไม่มี CLI	ไม่มี CLI	มี CLI	มี CLI

ประเด็น	NestJS	Express.js	Koa.js	Fastify	AdonisJS
CLI	module/controller/service ได้รวดเร็ว	(ต้องใช้ generator เสริม)		เล็กน้อย	ครบถ้วน (เหมือน Laravel)
Microservices	รองรับหลาย transport (gRPC, Kafka, RabbitMQ, NATS)	ต้องต่อ lib เสริม	ต้องต่อ lib เสริม	รองรับบางส่วน แต่ไม่ครบ ecosystem	รองรับบาง microservices แต่ไม่ยืดหยุ่นเท่า Nest
GraphQL Support	มี integration โดยตรง (schema-first & code-first)	ต้องใช้ Apollo/อื่นๆ ต่อเอง	ต้องติดตั้งเอง	ต้องติดตั้งเอง	มี GraphQL module ให้ใช้
Learning Curve	ชันกว่า Express/Koa เพราะต้องเข้าใจ Module/DI	ง่ายมาก เหมาะกับ beginner	ง่าย แต่ต้องออกแบบเองเยอะ	กลาง ๆ, ถนัดเรื่อง performance	ชันเหมือน Nest เพราะ opinionated
Community / Ecosystem	ใหญ่, เติบโตเร็ว, ใช้ในระดับองค์กร (enterprise)	ใหญ่ที่สุด (เป็น base ของหลาย framework)	เล็กกว่า Express	กำลังโต, ชูจุดขายเร็ว	Community เล็กกว่า Nest/Express
Use Cases ที่เหมาะ	ระบบใหญ่, ทีมหลายคน, Enterprise, Microservices, API Gateway	โปรเจกต์เล็ก–กลาง, POC, MVP	โปรเจกต์เล็ก–กลาง, custom APIs	APIs ที่เน้น performance สูง	ระบบ fullstack แบบ MVC, startup apps

☐ **จุดแข็งของ NestJS เมื่อเทียบกับ JS Framework อื่น ๆ**

1. **โครงสร้างมาตรฐาน**
   - Express/Koa ให้อิสระมาก แต่ขาดโครงสร้าง ⟶ โครงการใหญ่จะรก
   - NestJS มี module system ที่ชัดเจน เหมาะกับทีมขนาดใหญ่
2. **TypeScript-first**
   - Express/Koa/JS raw รองรับ TS แบบ optional
   - NestJS ออกแบบโดยคิดถึง TS ตั้งแต่ต้น ⟶ static typing, DTO, validation ใช้งานง่าย
3. **Dependency Injection (DI)**

- จุดเด่นที่ Express/Koa ไม่มีในตัว
- ช่วยให้ test ง่าย, mock ได้, ลด coupling

4. **Microservices-ready**
   - NestJS รองรับ transport protocol หลากหลาย ⟶ RabbitMQ, Kafka, gRPC, MQTT
   - Express/Koa ต้องใช้ library ภายนอก
5. **เครื่องมือครบใน Ecosystem**
   - CLI, config, validation, GraphQL, WebSocket, Microservices, Testing
   - Express/Koa เน้น minimal ต้องหามาเสริมเอง
6. **แนวคิดสถาปัตยกรรมชัดเจน (OOP + FP + DI)**
   - เหมาะกับการนำ Clean Architecture / Hexagonal Architecture มาใช้
   - Express/Koa จะยืดหยุ่นกว่า แต่ไม่มี framework-level support

---

☐ **จุดอ่อนของ NestJS**

- Learning curve สูงกว่า Express/Koa ⟶ ต้องเข้าใจ module system และ DI
- Boilerplate code มากกว่าพวก minimal frameworks
- ถ้าโปรเจกต์เล็กมาก (เช่น API ง่าย ๆ ไม่กี่ route) อาจจะ “หนักเกินไป”

☐ สรุปสั้น:

- ถ้าโปรเจกต์เล็ก/ต้องการเร็ว ⟶ **Express/Koa**
- ถ้าเน้น performance สูง ⟶ **Fastify**
- ถ้าอยากได้ fullstack MVC (Laravel style) ⟶ **AdonisJS**
- ถ้าต้องการ **enterprise-grade, scalable, structured, testable ⟶ NestJS คือตัวเลือกที่ดีที่สุด**

NestJS คืออะไร?

NestJS คือ **Backend Framework บน Node.js** ที่สร้างขึ้นด้วย **TypeScript** (แต่ก็รองรับ JavaScript)

ถูกออกแบบมาเพื่อให้การพัฒนา **Server-side Application** มีโครงสร้างที่ชัดเจน เป็นระบบ และ **เหมาะกับงานขนาดใหญ่หรือ Enterprise**

NestJS ได้แรงบันดาลใจมาจาก **Angular** (ในฝั่ง Frontend) เช่น

- การใช้ **Decorators** (@Controller, @Module, @Injectable)
- การทำงานแบบ **Dependency Injection (DI)**

- การแบ่งโครงสร้างเป็น **Modules / Controllers / Services**

□ NestJS ใช้ทำอะไร?

NestJS ใช้สำหรับสร้างแอปพลิเคชันฝั่งเซิร์ฟเวอร์ที่ต้องการ **โครงสร้างและความยืดหยุ่นสูง** เช่น

1. **RESTful APIs**
   - API แบบดั้งเดิมสำหรับ Mobile / Web Apps
2. **GraphQL APIs**
   - รองรับทั้ง **schema-first** และ **code-first** approach
3. **Microservices**
   - รองรับ protocol เช่น TCP, gRPC, MQTT, RabbitMQ, Kafka ⟶ ทำให้สร้างระบบแบบ distributed ได้ง่าย
4. **Real-time Applications**
   - ใช้ **WebSocket** สำหรับ Chat app, Notification, Live update
5. **Enterprise Applications**
   - ระบบใหญ่ที่ต้องการ **scalability**, **testability**, และ **maintainability**

□ ตัวอย่างโค้ดเปรียบเทียบ (Express vs NestJS)

□ Express (Minimal)

```
// app.js
const express = require('express');
const app = express();
const port = 3000;

app.get('/hello', (req, res) => {
  res.send('Hello from Express!');
});

app.listen(port, () => {
  console.log(`Server running at http://localhost:${port}`);
});
```

□ Express จะเห็นว่า **สั้นและตรงไปตรงมา** แต่ถ้าโปรเจกต์ใหญ่ขึ้น โค้ดจะเริ่มยุ่งเหยิง (ไม่มีโครงสร้างบังคับ)

NestJS (Structured)

```
// main.ts (entry point)
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';

async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  await app.listen(3000);
}
bootstrap();
// app.module.ts (root module)
import { Module } from '@nestjs/common';
import { AppController } from './app.controller';
import { AppService } from './app.service';

@Module({
  imports: [],
  controllers: [AppController],  // จัดการ request
  providers: [AppService],       // จัดการ business logic
})
export class AppModule {}
// app.controller.ts (Controller layer)
import { Controller, Get } from '@nestjs/common';
import { AppService } from './app.service';

@Controller()
export class AppController {
  constructor(private readonly appService: AppService) {}

  @Get('hello')
  getHello(): string {
    return this.appService.getHello();
  }
}
```

```
// app.service.ts (Service layer - Business logic)
import { Injectable } from '@nestjs/common';

@Injectable()
export class AppService {
  getHello(): string {
    return 'Hello from NestJS!';
  }
}
```

□ **สิ่งที่เห็นจากโค้ด**

- NestJS แบ่งโครงสร้างเป็น **Module / Controller / Service** ⟶ ชัดเจนว่าหน้าไหนทำอะไร
- ใช้ **Dependency Injection (DI)** ⟶ AppController ไม่ต้องสร้าง AppService เอง NestJS จัดการให้
- โค้ดมีการขยายต่อได้ง่าย ⟶ ถ้าเพิ่ม UsersModule, AuthModule, ProductsModule ก็จะจัดการเป็นระบบมากขึ้น

□ **สรุปสั้น ๆ**

- **NestJS คือ Backend Framework ที่ใช้ Node.js + TypeScript**
- ใช้ทำ **REST API, GraphQL, Microservices, Real-time apps** ได้ในระดับ **Enterprise**
- จุดเด่นคือ **มีโครงสร้างชัดเจน, มี Dependency Injection, TypeScript-first, เหมาะกับทีมใหญ่**

แนวคิด OOP, FP และ DI ใน NestJS

---

1) แนวคิด OOP (Object-Oriented Programming) ใน NestJS

NestJS ถูกออกแบบโดยใช้ **Class และ Decorator** เป็นหลัก

ซึ่งเป็นการนำแนวคิด OOP มาใช้ เช่น **Encapsulation, Inheritance, Polymorphism**

□ ตัวอย่าง **Service class** ที่ห่อหุ้ม (Encapsulation) logic การทำงาน

```
// user.service.ts
import { Injectable } from '@nestjs/common';

@Injectable()
export class UserService {
```

```
  private users = ['Alice', 'Bob', 'Charlie'];

  // Encapsulation: users ถูกซ่อน (private) และจัดการผ่าน method
  getAllUsers(): string[] {
    return this.users;
  }

  addUser(name: string) {
    this.users.push(name);
  }
}
```

□ ตัวอย่าง **Inheritance** (การสืบทอดคลาส)

```
// base.entity.ts
export class BaseEntity {
  id: number;
  createdAt: Date;
  updatedAt: Date;
}

// user.entity.ts
import { BaseEntity } from './base.entity';

export class UserEntity extends BaseEntity {
  name: string;
  email: string;
}
```

➡□ จุดแข็งของ OOP ใน Nest คือช่วยให้เราแยก **Business Logic** ออกมาในรูปแบบ class ที่อ่านง่ายและขยายต่อได้

---

2) แนวคิด FP (Functional Programming) ใน NestJS

แม้ว่า NestJS จะขับเคลื่อนด้วย OOP เป็นหลัก แต่ก็รองรับ **แนวคิด Functional Programming (FP)** โดยเฉพาะการใช้ **pure functions, immutability, และ higher-order functions**

□ ตัวอย่าง **Pure Function** ใน Utility

```
// math.utils.ts
```

```
export function sum(a: number, b: number): number {
  return a + b; // ไม่พึ่ง state ภายนอก ⟶ pure function
}
```

 การใช้ FP กับ **RxJS (Reactive Extensions for JavaScript)**

NestJS มี integration กับ **Observables** (ซึ่งเป็นแนวทาง functional)

```
// user.controller.ts
import { Controller, Get } from '@nestjs/common';
import { of, map } from 'rxjs';

@Controller('user')
export class UserController {
  @Get()
  getUserNames() {
    return of(['Alice', 'Bob', 'Charlie']).pipe(
      map(users => users.map(u => u.toUpperCase())) // Functional style
    );
  }
}
```

➡ FP ช่วยให้โค้ดใน NestJS มีความ **declarative** และเหมาะกับการจัดการ async streams (เช่น HTTP requests, WebSockets)

---

3) DI (Dependency Injection) ใน NestJS

หนึ่งใน **หัวใจหลัก** ของ NestJS คือ **Dependency Injection (DI)**

ซึ่งช่วยให้เราสามารถแยกความรับผิดชอบของแต่ละ class และจัดการ dependency ได้ง่ายขึ้น

 ตัวอย่างการใช้ DI

```
// user.module.ts
import { Module } from '@nestjs/common';
import { UserService } from './user.service';
import { UserController } from './user.controller';

@Module({
  controllers: [UserController],
  providers: [UserService], // Register Service as a provider
})
```

```
export class UserModule {}
// user.controller.ts
import { Controller, Get, Post, Body } from '@nestjs/common';
import { UserService } from './user.service';

@Controller('users')
export class UserController {
  // NestJS จะ Inject UserService มาให้อัตโนมัติ
  constructor(private readonly userService: UserService) {}

  @Get()
  getAllUsers() {
    return this.userService.getAllUsers();
  }

  @Post()
  addUser(@Body('name') name: string) {
    this.userService.addUser(name);
    return { message: 'User added!' };
  }
}
```

➡ □ จุดเด่นของ **DI ใน NestJS** คือ

- ไม่ต้อง new class เอง ⟶ Nest จัดการ lifecycle ให้
- สามารถ mock service ได้ง่ายเวลา **เขียน unit test**
- ช่วยให้โค้ด **หลวม (loosely coupled)** และเปลี่ยน implementation ได้ง่าย

□ **สรุป**

- **OOP** ⟶ ใช้ class + decorator ⟶ โค้ดมีโครงสร้างและจัดการง่าย
- **FP** ⟶ ใช้ pure functions + RxJS ⟶ เหมาะกับ async และ reactive programming
- **DI** ⟶ Nest จัดการ dependency lifecycle ให้ ⟶ ลดความซับซ้อนและเพิ่มความ testable

□ NestJS ไม่ได้บังคับให้ใช้ OOP หรือ FP เพียงอย่างเดียว แต่ **บูรณาการทั้งสองแนวคิด** แล้วเสริมด้วย **DI** เพื่อให้ได้ framework ที่ทั้งยืดหยุ่นและ maintainable

---

โครงสร้างโฟลเดอร์มาตรฐานของโปรเจกต์ NestJS

□ โครงสร้างโฟลเดอร์มาตรฐานใน NestJS

เมื่อเราสร้างโปรเจกต์ด้วย Nest CLI (nest new project-name)
จะได้โครงสร้างเริ่มต้นประมาณนี้ □

```
my-nest-project/
│
├── src/                        # Source code ทั้งหมดอยู่ในนี้
│   ├── app.controller.ts       # Controller หลัก (ตัวอย่าง)
│   ├── app.controller.spec.ts  # Unit test ของ Controller
│   ├── app.module.ts           # Root Module (entry point ของ app)
│   ├── app.service.ts          # Service หลัก (ตัวอย่าง)
│   └── main.ts                 # Entry point ของโปรแกรม (bootstrap)
│
├── test/                       # Integration/E2E tests
│   └── app.e2e-spec.ts
│
├── node_modules/               # Dependencies
├── package.json                # รายการ dependencies และ scripts
├── tsconfig.json               # TypeScript config
└── nest-cli.json               # Config ของ Nest CLI
```

1) ไฟล์หลัก

- **main.ts** ⟶ จุดเริ่มต้นของแอป ใช้ NestFactory bootstrap application
- import { NestFactory } from '@nestjs/core';
- import { AppModule } from './app.module';
-
- async function bootstrap() {
- const app = await NestFactory.create(AppModule);
- await app.listen(3000);
- console.log(` □ Application is running on: http://localhost:3000`);
- }
- bootstrap();

- **app.module.ts** ⟶ Root module ของโปรเจกต์ (บอกว่าใช้ controller/service ไหนบ้าง)
- import { Module } from '@nestjs/common';
- import { AppController } from './app.controller';
- import { AppService } from './app.service';
-
- @Module({
- imports: [], // import module อื่นๆ
- controllers: [AppController],
- providers: [AppService], // register service
- })
- export class AppModule {}
- **app.controller.ts** ⟶ จัดการ HTTP request/response
- import { Controller, Get } from '@nestjs/common';
- import { AppService } from './app.service';
-
- @Controller()
- export class AppController {
- constructor(private readonly appService: AppService) {}
-
- @Get()
- getHello(): string {
- return this.appService.getHello();
- }
- }
- **app.service.ts** ⟶ Business logic
- import { Injectable } from '@nestjs/common';
-
- @Injectable()
- export class AppService {
- getHello(): string {
- return 'Hello NestJS!';
- }
- }

2) เมื่อมี Feature/Module เพิ่มขึ้น

NestJS แนะนำให้จัดโฟลเดอร์แยกตาม **Domain / Feature-based structure**

เช่น ถ้ามี **Users module**

```
src/
├── users/
│   ├── users.controller.ts
│   ├── users.service.ts
│   ├── users.module.ts
│   └── dto/
│       └── create-user.dto.ts
```

- users.module.ts ⟶ module สำหรับจัดการ Users
- users.controller.ts ⟶ API endpoint เช่น GET /users, POST /users
- users.service.ts ⟶ logic การทำงาน เช่น จัดการ user list
- dto/ ⟶ Data Transfer Object (แยกโครงสร้างข้อมูล request/response)

ตัวอย่าง **users.module.ts**

```
import { Module } from '@nestjs/common';
import { UsersController } from './users.controller';
import { UsersService } from './users.service';

@Module({
  controllers: [UsersController],
  providers: [UsersService],
})
export class UsersModule {}
```

---

3) โครงสร้างโปรเจกต์ในระดับกลางถึงใหญ่

เมื่อโปรเจกต์ใหญ่ขึ้น เราจะเจอโฟลเดอร์ประมาณนี้

```
src/
├── common/          # utilities, filters, interceptors, guards
├── config/          # configuration (database, env)
├── database/        # entities, repositories
├── users/           # feature module: users
├── auth/            # feature module: authentication
```

```
├── products/          # feature module: products
├── app.module.ts      # root module
└── main.ts            # entry point
```

☐ **หลักการ NestJS ⟶ Everything is a Module**

ทุกฟีเจอร์ควรแยกออกเป็น **Module** เพื่อให้ maintain ง่าย, test ง่าย, และ reuse ได้

☐ **สรุป**

- **ระดับเล็ก** ⟶ Nest จะมี app.controller.ts, app.service.ts, app.module.ts, main.ts
- **ระดับกลาง/ใหญ่** ⟶ แยกโฟลเดอร์ตาม **feature/domain** (เช่น users, auth, products)
- **ข้อดีของโครงสร้าง Nest**
  - อ่านง่าย ⟶ แยกตามฟีเจอร์ชัดเจน
  - ขยายได้ ⟶ เพิ่ม module ได้เรื่อยๆ
  - รองรับ **OOP + DI** โดยธรรมชาติ

การติดตั้ง Nest CLI และการสร้างโปรเจกต์ใหม่

☐ **การติดตั้ง Nest CLI และสร้างโปรเจกต์ใหม่**

1) ติดตั้ง Node.js

- ก่อนอื่นต้องมี **Node.js** และ **npm (Node Package Manager)** อยู่ในเครื่อง
- แนะนำ **Node.js เวอร์ชัน LTS (Long Term Support)**

  ตรวจสอบเวอร์ชันด้วยคำสั่ง:

```
node -v
npm -v
```

---

2) ติดตั้ง Nest CLI

NestJS มี **Command Line Interface (CLI)** ที่ช่วยให้การสร้างโปรเจกต์สะดวกขึ้น

ติดตั้งแบบ global:

```
npm install -g @nestjs/cli
```

ตรวจสอบว่า CLI ใช้งานได้:

```
nest --version
```

---

3) สร้างโปรเจกต์ใหม่

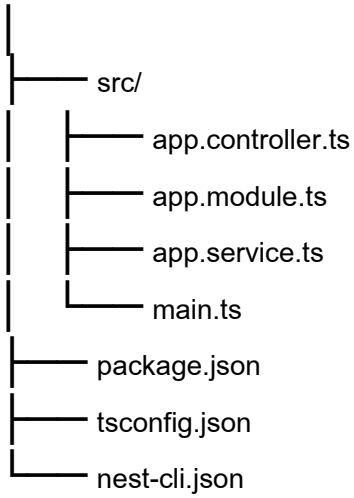
ใช้คำสั่ง:

nest new my-nest-app

- my-nest-app คือชื่อโปรเจกต์ (ตั้งชื่ออะไรก็ได้)
- CLI จะถามว่าให้ใช้ **แพ็กเกจจัดการ dependency อะไร** ⟶ เลือกได้ เช่น npm หรือ yarn

เมื่อสร้างเสร็จจะได้โครงสร้างโปรเจกต์แบบนี้:

my-nest-app/



---

4) รันโปรเจกต์

เข้าไปที่โฟลเดอร์โปรเจกต์ แล้วรัน:

cd my-nest-app

npm run start

ถ้าต้องการให้รีโหลดอัตโนมัติเมื่อแก้โค้ด:

npm run start:dev

- เมื่อรันสำเร็จ จะเห็นข้อความ:
- Application is running on: http://localhost:3000

ลองเข้าเว็บเบราว์เซอร์ ⟶ ไปที่ http://localhost:3000

จะได้ข้อความตอบกลับ:

Hello World!

---

5) ตัวอย่างโค้ดเริ่มต้น (default)

โค้ดที่ NestJS สร้างมาให้จะประมาณนี้

main.ts

```
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';

async function bootstrap() {
```

```
  const app = await NestFactory.create(AppModule);
  await app.listen(3000);
  console.log('Application is running on: http://localhost:3000');
}
bootstrap();
```

app.module.ts

```
import { Module } from '@nestjs/common';
import { AppController } from './app.controller';
import { AppService } from './app.service';

@Module({
  imports: [],
  controllers: [AppController],
  providers: [AppService],
})
export class AppModule {}
```

app.controller.ts

```
import { Controller, Get } from '@nestjs/common';
import { AppService } from './app.service';

@Controller()
export class AppController {
  constructor(private readonly appService: AppService) {}

  @Get()
  getHello(): string {
    return this.appService.getHello();
  }
}
```

app.service.ts

```
import { Injectable } from '@nestjs/common';

@Injectable()
export class AppService {
```

```
  getHello(): string {
    return 'Hello World!';
  }
}
```

 เพียงเท่านี้ก็ได้ **NestJS Project แรก** พร้อมรันแล้ว

การติดตั้งและทดสอบ NestJS บน IDE/Editor

เราจะพูดถึง 3 Tools ดังนี้:

1. **Visual Studio Code (VS Code)**
2. **WebStorm (JetBrains)**
3. **IntelliJ IDEA / Node.js plugin**

---

1 Visual Studio Code (VS Code)

ขั้นตอนติดตั้ง

1. ดาวน์โหลดและติดตั้ง VS Code
2. ติดตั้ง **Node.js LTS** และ **npm**
3. เปิด VS Code ⟶ เปิด **Terminal** (Ctrl+`)
4. ติดตั้ง Nest CLI global:
5. npm install -g @nestjs/cli
6. สร้างโปรเจกต์ใหม่:
7. nest new my-nest-app
8. เลือก **npm** หรือ **yarn** ตาม preference

การรันโปรเจกต์

- เปิด terminal ใน VS Code ⟶ รัน:
- npm run start:dev
- เปิดเว็บเบราว์เซอร์ ⟶ http://localhost:3000
- จะเห็นข้อความ:
- Hello World!

Extension แนะนำ

- **ESLint** ⟶ ตรวจ syntax/format
- **Prettier** ⟶ จัด format code
- **NestJS Snippets** ⟶ ช่วย generate code ของ NestJS

2 WebStorm (JetBrains)

ขั้นตอนติดตั้ง

1. ดาวน์โหลด WebStorm
2. เปิด WebStorm ⟶ New Project ⟶ **Node.js Express** (เลือก TypeScript)
3. เปิด **Terminal** ภายใน WebStorm
4. ติดตั้ง Nest CLI:
5. npm install -g @nestjs/cli
6. สร้างโปรเจกต์:
7. nest new my-nest-app
8. WebStorm จะ recognize project เป็น Node/TypeScript project อัตโนมัติ

การรันโปรเจกต์

- ไปที่ **Terminal** ⟶ รัน:
- npm run start:dev
- หรือสร้าง **Run Configuration** ⟶ เลือก npm start เพื่อรันจาก IDE

Features ที่ช่วย

- Auto-import decorators (@Controller, @Injectable)
- Code navigation ระหว่าง Module/Service/Controller
- Integrated terminal และ debugger

---

3 IntelliJ IDEA (Community หรือ Ultimate) + Node.js plugin

ขั้นตอนติดตั้ง

1. ดาวน์โหลด IntelliJ IDEA
2. ติดตั้ง **Node.js plugin** (Settings ⟶ Plugins ⟶ Marketplace ⟶ Node.js)
3. เปิด IntelliJ ⟶ New Project ⟶ **Node.js** + **TypeScript**
4. เปิด Terminal ⟶ ติดตั้ง Nest CLI:
5. npm install -g @nestjs/cli
6. สร้างโปรเจกต์ Nest ใหม่:
7. nest new my-nest-app

การรันโปรเจกต์

- เปิด Terminal ⟶ npm run start:dev
- หรือสร้าง **Run/Debug Configuration** เลือก Node.js script ⟶ main.ts

Features ที่ช่วย

- Advanced debugging (breakpoints, step-over, variable watch)
- TypeScript auto-completion
- Project structure management สำหรับหลาย Module

Tips ทั่วไปสำหรับทุก IDE

1. ใช้ **start:dev** สำหรับ hot-reload ⟶ พัฒนาเร็ว
2. ใช้ **Decorators + DI** ⟶ IDE จะช่วย auto-import
3. ติดตั้ง **ESLint + Prettier** ⟶ รักษามาตรฐานโค้ด
4. สร้าง **Launch/Debug configuration** ⟶ รันและ debug ใน IDE ได้โดยตรง

สรุป:

IDE	เหมาะกับ	จุดเด่น
VS Code	ฟรี, เบา, plugin เยอะ	Terminal + Snippets + Hot reload
WebStorm	Professional, ทีมใหญ่	Code navigation, debugger, auto-import
IntelliJ IDEA	ทีมที่ใช้ JetBrains ecosystem	Advanced debug, multi-language support

---

การติดตั้ง NestJS และทดสอบโปรเจกต์บน NetBeans IDE

การติดตั้ง NestJS และทดสอบโปรเจกต์บน NetBeans IDE อย่างละเอียดทีละขั้นตอน

หมายเหตุ: **NetBeans** เป็น IDE ที่เน้น Java เป็นหลัก แต่ก็รองรับ **Node.js / JavaScript / TypeScript** ผ่าน plugin ที่ติดตั้งเพิ่มได้

ดังนั้นการพัฒนา NestJS บน NetBeans ยังไม่เป็นที่นิยมเท่า VS Code หรือ WebStorm แต่สามารถทำงานได้

การติดตั้งและรัน NestJS บน NetBeans IDE

1 เตรียมเครื่องมือพื้นฐาน

1. ติดตั้ง **Node.js** เวอร์ชัน LTS
    - ดาวน์โหลดจาก https://nodejs.org
    - ตรวจสอบเวอร์ชัน:
    - node -v
    - npm -v
2. ติดตั้ง **Nest CLI** แบบ global:
3. npm install -g @nestjs/cli

4. ติดตั้ง **TypeScript** (ถ้าโปรเจกต์ Nest ยังไม่ได้ติดตั้ง)
5. npm install -g typescript

---

2 ติดตั้ง NetBeans และ Plugin Node.js

1. ดาวน์โหลด Apache NetBeans (เวอร์ชันล่าสุด แนะนำ 17+)
2. เปิด NetBeans ⟶ ไปที่

   Tools ⟶ Plugins ⟶ Available Plugins ⟶ ค้นหา "Node.js" ⟶ ติดตั้ง
3. หลังติดตั้ง ให้ restart IDE

หลังจากติดตั้ง plugin Node.js แล้ว NetBeans จะสามารถ:

- สร้าง Node.js project
- รัน / debug script JS/TS
- รองรับ npm commands

---

3 สร้างโปรเจกต์ NestJS ใหม่ใน NetBeans

วิธีที่ 1: ใช้ Nest CLI ผ่าน Terminal

1. เปิด **Terminal** ใน NetBeans (Tools ⟶ Terminal)
2. รันคำสั่ง:
3. nest new my-nest-app
4. เลือก **npm** หรือ **yarn**
5. Nest CLI จะสร้างโปรเจกต์พร้อมโครงสร้างมาตรฐาน:

```
my-nest-app/
├── src/
│   ├── main.ts
│   ├── app.module.ts
│   ├── app.controller.ts
│   └── app.service.ts
├── package.json
├── tsconfig.json
└── nest-cli.json
```

วิธีที่ 2: สร้าง Node.js Project ใน NetBeans แล้วติดตั้ง NestJS

1. **File ⟶ New Project ⟶ Node.js Application**
2. กำหนด **Project Folder**
3. เปิด Terminal ⟶ รัน:

4. npm install @nestjs/core @nestjs/common rxjs reflect-metadata
5. npm install --save-dev @nestjs/cli typescript ts-node @types/node
6. npx nest new .

---

4 การรันโปรเจกต์ NestJS ใน NetBeans

1. เปิด **Terminal** ใน NetBeans ⟶ ไปที่โฟลเดอร์โปรเจกต์
2. รันคำสั่ง:
3. npm run start:dev
   - ใช้ start:dev เพื่อ **hot reload** เวลาแก้ไขโค้ด
4. เปิดเบราว์เซอร์ ⟶ เข้า http://localhost:3000
   จะเห็นข้อความ:
5. Hello World!

---

5 การตั้งค่า Debug ใน NetBeans

1. **Run ⟶ Set Project Configuration ⟶ Customize ⟶ Node.js**
2. กำหนด **Main File** เป็น src/main.ts
3. กด **Debug Project** ⟶ NetBeans จะเปิด debugger
   - สามารถตั้ง **breakpoints** ใน Controller / Service
   - Inspect variable / call stack ได้

---

6 การติดตั้ง Extensions / Tools เสริม

- **ESLint** ⟶ ตรวจ syntax / linting
- npm install --save-dev eslint @typescript-eslint/parser @typescript-eslint/eslint-plugin
- **Prettier** ⟶ จัด format code
- npm install --save-dev prettier
- **NestJS Snippets** ⟶ สามารถใช้ผ่าน VS Code/Editor แยกเพื่อ copy code (NetBeans ไม่มี plugin เฉพาะ NestJS)

ข้อจำกัดบน NetBeans

1. Plugin Node.js ยังไม่ครบ feature เท่า VS Code หรือ WebStorm
2. Auto-import decorators (@Controller, @Injectable) ยังไม่สะดวก
3. Hot reload ใช้ได้ผ่าน npm run start:dev แต่ไม่มี integration ใน GUI IDE
4. เหมาะสำหรับ **ผู้ที่ชำนาญ Node.js + NetBeans อยู่แล้ว**

สรุปขั้นตอน

1. ติดตั้ง Node.js + npm + TypeScript
2. ติดตั้ง Nest CLI
3. ติดตั้ง NetBeans + Node.js plugin
4. สร้างโปรเจกต์ NestJS (CLI หรือ Node.js project)
5. รันโปรเจกต์ผ่าน Terminal (npm run start:dev)
6. ตั้ง Debug Configuration ⟶ Debug controller/service ได้

สรุป

บทที่ 1 จะปูพื้นฐานเกี่ยวกับ **NestJS** โดยอธิบายว่า NestJS คือเฟรมเวิร์กบน Node.js ที่ใช้ TypeScript เพื่อสร้างแอปพลิเคชันฝั่งเซิร์ฟเวอร์ได้อย่างมีประสิทธิภาพ รองรับงานระดับองค์กร พร้อมผสานแนวคิด **OOP, FP และ Dependency Injection (DI)** เพื่อทำให้โค้ดมีความเป็นระบบและดูแลรักษาง่าย นอกจากนี้ยังแนะนำ **โครงสร้างโฟลเดอร์มาตรฐานของโปรเจกต์** ที่แยก Controller, Service และ Module ออกจากกันอย่างชัดเจน และแนะนำการใช้ **Nest CLI** ในการติดตั้งและสร้างโปรเจกต์ใหม่ เพื่อเตรียมความพร้อมสำหรับการพัฒนาในบทถัดไป.

บทที่ 2

Controllers และ Routing เบื้องต้น

(Basic of Controllers and Routing)

เนื้อหา

- Controllers และ Routing เบื้องต้น
- เจาะลึกเรื่อง Controllers และ Routing เบื้องต้นใน NestJS แบบ เชิงลึก
- การสร้าง Controller ด้วย CLI
- การสร้าง Route ด้วย @Get และ @Post
- การรับค่าจาก Route Params, Query, และ Body
- DTO เบื้องต้นใน NestJS
- ตัวอย่างบูรณาการ
- Ultimate Integrated Project

บทนำบทที่ 2: Controllers และ Routing เบื้องต้น

ในการพัฒนาแอปพลิเคชันฝั่งเซิร์ฟเวอร์ หน้าที่สำคัญที่สุดประการหนึ่งคือการ **จัดการคำขอ (Request) และส่งคำตอบ (Response)** กลับไปยังผู้ใช้งาน NestJS ออกแบบให้กระบวนการนี้เป็นเรื่องง่ายผ่าน **Controller** และระบบ **Routing** ซึ่งเป็นหัวใจหลักในการสื่อสารระหว่าง Client และ Server หากในบทแรกเราได้ทำความเข้าใจกับโครงสร้างพื้นฐานของโปรเจกต์แล้ว บทนี้จะพาไปสู่การสร้าง Controller และเส้นทาง (Route) แรกของคุณใน NestJS

NestJS มีเครื่องมือ CLI ที่ช่วยให้นักพัฒนาสามารถสร้าง Controller ได้อย่างสะดวก โดยเพียงใช้คำสั่งไม่กี่บรรทัดก็จะได้ไฟล์ Controller พร้อมโครงร่างเบื้องต้น ซึ่งสามารถปรับแต่งให้รองรับการทำงานที่ซับซ้อนขึ้นได้ การใช้ CLI ไม่เพียงช่วยลดเวลาในการตั้งค่า แต่ยังคงมาตรฐานโครงสร้างโค้ดให้เป็นรูปแบบเดียวกันตลอดทั้งโปรเจกต์

ในส่วนของการสร้างเส้นทาง (Routing) NestJS ใช้ **Decorator** เช่น @Get() และ @Post() เพื่อกำหนดว่าฟังก์ชันภายใน Controller จะตอบสนองต่อคำขอแบบใด @Get() จะทำงานเมื่อมีการส่งคำขอแบบ HTTP GET เข้ามา ส่วน @Post() จะใช้สำหรับคำขอแบบ HTTP POST การทำงานเช่นนี้ช่วยให้นักพัฒนากำหนดพฤติกรรมของ API แต่ละเส้นทางได้อย่างชัดเจนและสั้นกระชับ

นอกจากนี้ Routing ใน NestJS ยังรองรับการ **รับค่าเพิ่มเติมจากผู้ใช้** ไม่ว่าจะเป็น **Route Parameters**, **Query Parameters** หรือ **Request Body** เช่น การดึงค่า id จาก URL, การอ่านค่า

ค้นหาจาก query string หรือการรับข้อมูลแบบ JSON จาก body การรองรับรูปแบบที่หลากหลายนี้ทำให้ API ที่สร้างขึ้นมีความยืดหยุ่นและสามารถตอบสนองต่อความต้องการที่ซับซ้อนได้

เพื่อให้โค้ดมีความเป็นระบบและตรวจสอบข้อมูลได้ง่าย NestJS แนะนำให้ใช้ **DTO (Data Transfer Object)** ในการจัดการข้อมูลที่ส่งเข้ามาจาก Client DTO คือคลาสที่ทำหน้าที่กำหนดรูปแบบข้อมูลที่คาดหวัง เช่น ฟิลด์ที่ต้องมีหรือประเภทของข้อมูล วิธีนี้ช่วยให้สามารถควบคุมและตรวจสอบความถูกต้องของข้อมูลได้ตั้งแต่ระดับโครงสร้างโค้ด ทำให้ระบบมีความปลอดภัยและน่าเชื่อถือมากขึ้น

ในเชิงปฏิบัติ บทนี้จะนำเสนอวิธีการสร้าง Controller เบื้องต้นด้วย CLI จากนั้นจะอธิบายการเพิ่มเส้นทางด้วย @Get และ @Post รวมถึงการใช้งานตัวอย่างการรับค่าจาก route params, query และ body อย่างเป็นขั้นตอน สุดท้ายจะอธิบายวิธีสร้าง DTO อย่างง่ายเพื่อใช้ตรวจสอบข้อมูลที่รับเข้ามาจากผู้ใช้งาน

กล่าวโดยสรุป บทนี้จะปูพื้นฐานการทำงานของ **Controller และ Routing ใน NestJS** อย่างครบถ้วน เพื่อให้คุณสามารถสร้าง API ที่รับคำขอจาก Client ได้จริง โดยมีทั้งโครงสร้างที่เป็นระบบ ความปลอดภัย และความยืดหยุ่น ความรู้เหล่านี้จะเป็นเครื่องมือสำคัญในการพัฒนาฟีเจอร์ที่ซับซ้อนยิ่งขึ้นในบทต่อไป

Controllers และ Routing เบื้องต้น

- สร้าง Controller ด้วย CLI
- การสร้าง route ด้วย @Get, @Post
- การรับค่าจาก route params, query, body
- การใช้ DTO เบื้องต้น

□ บทที่ 2: Controllers และ Routing เบื้องต้น

ใน NestJS **Controller** คือส่วนที่ **รับ HTTP request และส่ง response** เป็นหลัก

Controller จะเป็นตัวกลางระหว่าง **client** กับ **business logic (Service)**

1□ สร้าง Controller ด้วย CLI

NestJS CLI ช่วยสร้าง controller แบบอัตโนมัติ

```
nest generate controller users
# หรือสั้น
nest g co users
```

- users ⟶ ชื่อ controller
- CLI จะสร้างไฟล์ในโฟลเดอร์ src/users/
- สร้างตัวอย่างไฟล์:

- src/users/users.controller.ts
- src/users/users.controller.spec.ts

ตัวอย่าง users.controller.ts เริ่มต้น

```
import { Controller, Get } from '@nestjs/common';

@Controller('users') // route prefix
export class UsersController {
  @Get()
  findAll(): string {
    return 'This action returns all users';
  }
}
```

- @Controller('users') ⟶ prefix ของ route /users
- @Get() ⟶ mapping HTTP GET method

---

2 การสร้าง Route ด้วย @Get และ @Post

NestJS รองรับ decorator สำหรับ HTTP methods เช่น:

HTTP Method	Decorator
GET	@Get()
POST	@Post()
PUT	@Put()
DELETE	@Delete()
PATCH	@Patch()

ตัวอย่าง controller แบบ GET และ POST

```
import { Controller, Get, Post, Body } from '@nestjs/common';

@Controller('users')
export class UsersController {
  private users = ['Alice', 'Bob'];

  @Get()
  getAllUsers(): string[] {
```

```
    return this.users;
  }

  @Post()
  addUser(@Body('name') name: string): string {
    this.users.push(name);
    return `User ${name} added successfully`;
  }
}
```

- @Body('name') ⟶ ดึงค่า name จาก request body

---

3 การรับค่าจาก Route Params, Query, Body

a) Route Params

```
import { Controller, Get, Param } from '@nestjs/common';

@Controller('users')
export class UsersController {
  private users = ['Alice', 'Bob'];

  @Get(':id')
  getUser(@Param('id') id: number): string {
    return this.users[id];
  }
}
```

- URL: /users/1 ⟶ id = 1
- ใช้ @Param('paramName') เพื่อดึงค่า

b) Query Params

```
import { Controller, Get, Query } from '@nestjs/common';

@Controller('users')
export class UsersController {
  @Get('search')
  searchUser(@Query('name') name: string): string {
    return `Search result for user: ${name}`;
```

```
  }
}
```

- URL: /users/search?name=Alice ⟶ name = Alice
- ใช้ @Query('queryParam')

c) Request Body

```
import { Controller, Post, Body } from '@nestjs/common';

@Controller('users')
export class UsersController {
  @Post()
  createUser(@Body() body: { name: string; age: number }): string {
    return `User ${body.name} aged ${body.age} added`;
  }
}
```

- Body JSON:

```
{
  "name": "Charlie",
  "age": 25
}
```

---

4 การใช้ DTO (Data Transfer Object) เบื้องต้น

DTO คือ **object ที่กำหนดรูปแบบข้อมูลที่ส่งเข้าหรือออก**

ช่วย validate, type-safe และอ่านโค้ดง่ายขึ้น

ตัวอย่าง DTO

```
// src/users/dto/create-user.dto.ts
export class CreateUserDto {
  name: string;
  age: number;
}
```

ใช้ DTO ใน Controller

```
import { Controller, Post, Body } from '@nestjs/common';
import { CreateUserDto } from './dto/create-user.dto';

@Controller('users')
```

```
export class UsersController {
  @Post()
  createUser(@Body() createUserDto: CreateUserDto): string {
    return `User ${createUserDto.name} aged ${createUserDto.age} added`;
  }
}
```

☐ ข้อดีของ DTO

- ชัดเจนว่าข้อมูล body ต้องมีอะไรบ้าง
- สามารถต่อยอดกับ **class-validator** เพื่อ validate อัตโนมัติ

```
import { IsString, IsInt } from 'class-validator';

export class CreateUserDto {
  @IsString()
  name: string;

  @IsInt()
  age: number;
}
```

☐ **สรุปบทนี้**

1. **Controller** ⟶ รับ request และ return response
2. สร้าง Controller ด้วย CLI: nest g co users
3. **Route** ⟶ ใช้ decorator เช่น @Get(), @Post(), @Put(), @Delete()
4. รับค่า **Route Params** ⟶ @Param()
5. รับค่า **Query Params** ⟶ @Query()
6. รับค่า **Body** ⟶ @Body()
7. ใช้ **DTO** ⟶ กำหนด structure ข้อมูล, เพิ่ม type-safety, ต่อยอด validation

---

เจาะลึกเรื่อง Controllers และ Routing เบื้องต้นใน NestJS แบบ เชิงลึก

☐ **1) ความเข้าใจเชิงลึกของ Controller**

- **Controller** คือ class ที่ **รับผิดชอบเฉพาะเรื่องการจัดการ HTTP request และ response**
- NestJS แนะนำให้ **Controller เป็น thin layer**

- **ไม่ควรมี business logic หรือ database query**
- แค่ call Service แล้ว return response

- **Decorator @Controller()**:
  - เป็น **metadata** สำหรับ NestJS เพื่อบอกว่า class นี้เป็น Controller
  - สามารถใส่ **prefix ของ route** เช่น @Controller('users') ⟶ routes ทั้งหมดจะเริ่มด้วย /users

การทำงานภายใน NestJS

1. เมื่อ HTTP request เข้ามา NestJS **Router Explorer** จะหา Controller ที่ match กับ route และ HTTP method
2. NestJS **inject dependency** ให้ Controller อัตโนมัติ (DI)
3. Controller ส่ง response กลับ client ผ่าน **return value หรือ Observable / Promise**

☐ **2) Routing ด้วย HTTP method decorator**

NestJS ใช้ **decorator** แทนการเขียน routing แบบดั้งเดิม เช่น Express

Decorator	HTTP Method	Usage
@Get()	GET	ดึงข้อมูล
@Post()	POST	สร้าง resource
@Put()	PUT	แก้ไข resource
@Patch()	PATCH	แก้ไขบางส่วน
@Delete()	DELETE	ลบ resource

เชิงลึก: decorator ไม่ใช่แค่กำหนด route แต่ **เพิ่ม metadata ให้ NestJS สร้าง router map** อัตโนมัติ

```
@Controller('users')
export class UsersController {
  @Get(':id')
  findOne(@Param('id') id: string) {
    // NestJS แปลง route param เป็น argument ให้
    return `User ID: ${id}`;
  }

  @Post()
  create(@Body() body: any) {
    // body ถูก parse จาก JSON ผ่าน built-in body parser
```