NESTJS WEB PROGRAMING

ADVANCE

Integrative-Generative AI Edition

CONTENTS 1

ConfigModule and Envirenment Variables | 87

Utilizing Global Intereptors | 196

Swagger for API Docs 272

Authemtation System with JWT and Passport | 367

Biobligathy

Author: Student Price Book Center

คำนำ

สู่การเป็นผู้เชี่ยวชาญ NestJS ขั้นสูง

ในโลกของการพัฒนาเว็บแอปพลิเคชันยุคใหม่ **NestJS** ได้รับการยอมรับอย่างกว้างขวางว่าเป็นหนึ่งในเฟรมเวิร์ก Node.js ที่ทรงพลังและเป็นระเบียบที่สุด ด้วยการนำแนวคิดของ Angular มาประยุกต์ใช้ ทำให้ NestJS มอบสถาปัตยกรรมที่ชัดเจน มั่นคง และสามารถปรับขนาดได้ง่ายสำหรับแอปพลิเคชันระดับองค์กร (Enterprise-grade applications)

หนังสือเล่มนี้ **"NestJS Web Programming: Advance"** ไม่ได้มุ่งเน้นที่พื้นฐานทั่วไปของ NestJS หากแต่มุ่งพาคุณเจาะลึกไปในประเด็นที่ซับซ้อนและจำเป็นสำหรับการพัฒนาแอปพลิเคชันจริงที่ต้องใช้งานในสภาพแวดล้อมการผลิต (Production Environment) โดยเฉพาะอย่างยิ่งการจัดการการตั้งค่า, การจัดการการทำงานของระบบแบบทั่วถึง, การสร้างเอกสาร API ที่สมบูรณ์แบบ, และการสร้างระบบรักษาความปลอดภัยที่แข็งแกร่ง

เราเริ่มต้นการเดินทางในส่วน **Advance** นี้ด้วยความเข้าใจว่าการตั้งค่าและ Environment Variables เป็นหัวใจสำคัญของแอปพลิเคชันที่ต้องทำงานได้ในหลายสภาพแวดล้อมที่แตกต่างกัน **บทที่ 11: การใช้ ConfigModule และ Environment Variables** จึงเริ่มต้นด้วยการนำเสนอเครื่องมือที่ทรงพลังอย่าง @nestjs/config แบบเจาะลึก ตั้งแต่การสร้างและจัดการไฟล์ .env สำหรับสภาพแวดล้อมที่แตกต่างกัน ไปจนถึงเทคนิคที่สำคัญที่สุดในการจัดการการตั้งค่าแบบปลอดภัย การ Inject Config แบบ Strongly-Typed เพื่อเพิ่มความมั่นคงของโค้ด (Type-safety) และการจัดการ Secrets อย่างปลอดภัย ซึ่งเป็นสิ่งสำคัญอย่างยิ่งในแอปพลิเคชันที่ต้องเชื่อมต่อกับบริการภายนอก เช่น ฐานข้อมูลหรือ Third-party APIs ทั้งหมดนี้ถูกรวบรวมไว้ในตัวอย่างบูรณาการที่เรียกว่า *Ultimate Examples* เพื่อให้เห็นภาพการประยุกต์ใช้ในสถานการณ์จริง

การควบคุมการทำงานด้วย Aspect-Oriented Programming (AOP) ผ่าน Interceptors

จากนั้น เราจะก้าวเข้าสู่แนวคิดการเขียนโปรแกรมเชิงลึกอย่าง **Aspect-Oriented Programming (AOP)** ผ่านกลไกที่โดดเด่นของ NestJS คือ **Interceptors** ใน **บทที่ 12: การใช้งาน Global Interceptors** บทนี้จะพาคุณไปสำรวจความสามารถอันล้ำลึกของ Interceptors ซึ่งไม่ได้เป็นเพียงแค่การดักจับ Request/Response ทั่วไป แต่เป็นการควบคุมและปรับเปลี่ยนกระบวนการทำงานของแอปพลิเคชันอย่างมีประสิทธิภาพ

เราจะเรียนรู้เทคนิคการสร้าง Interceptor เพื่อจัดรูปแบบ Response (Response Format) ให้เป็นมาตรฐานเดียวกันทั่วทั้งระบบ การจัดการ Log และการแปลงข้อมูลก่อนส่งกลับ (Transform Response) ซึ่งช่วยให้โค้ดส่วนของ Controller สะอาดและเน้นไปที่ Business Logic อย่างแท้จริง หัวใจของบทนี้คือการใช้ AOP ในการกำหนดการทำงาน "ก่อน" และ "หลัง" การประมวลผล (Before / After Execution) ซึ่งจะนำไปสู่การจัดการข้อผิดพลาด (Error Handling) และการใช้ Interceptor แบบมีเงื่อนไข (Conditional Role) ที่ซับซ้อนมากขึ้น

หนังสือเล่มนี้จะนำเสนอ *Ultimate Full Project Template* ที่สาธิตการบูรณาการ Interceptor ทุกรูปแบบเข้าด้วยกัน ทั้งแบบ Global และ Scoped พร้อมคุณสมบัติที่พร้อมใช้งานในระดับ production-ready เพื่อให้คุณมั่นใจว่าสามารถนำความรู้นี้ไปใช้กับโปรเจกต์ขนาดใหญ่ได้อย่างไร้กังวล

การสื่อสารกับโลกภายนอกด้วย Swagger สำหรับ API Documentation

การสร้าง API ที่ยอดเยี่ยมนั้นไม่สมบูรณ์หากขาดเอกสารประกอบที่ชัดเจนและเป็นปัจจุบัน **บทที่ 13: การใช้ Swagger สำหรับ API Docs** ได้รับการออกแบบมาเพื่อแก้ปัญหานี้โดยเฉพาะ เราจะแนะนำการติดตั้งและใช้งาน @nestjs/swagger เพื่อสร้างเอกสาร API แบบ Interactive ที่สามารถทดสอบการทำงานได้ทันที โดยแทบไม่ต้องเขียนโค้ดเพิ่มเติมนอกเหนือจากโค้ดหลัก
บทนี้จะเจาะลึกถึงวิธีการเพิ่ม Metadata ที่จำเป็นลงใน Swagger เพื่อให้เอกสารมีความสมบูรณ์ที่สุด เช่น การกำหนดประเภทข้อมูล, การระบุสถานะ HTTP Response, และการจัดการ Models ต่าง ๆ ที่ใช้ในการแลกเปลี่ยนข้อมูล นอกจากนี้ ยังมีส่วนสำคัญที่กล่าวถึง **การ Deploy Swagger ใน Production** ซึ่งต้องคำนึงถึงความปลอดภัยและความพร้อมในการใช้งานจริง เพื่อให้มั่นใจว่าเอกสาร API ของคุณจะไม่เพียงแต่สวยงาม แต่ยังปลอดภัยและสามารถใช้ได้จริงในทุกสภาพแวดล้อม บทสรุปของบทนี้คือ *ตัวอย่าง Ultimate Project NestJS + Swagger ที่ deploy-ready* ที่จะนำทุกองค์ประกอบมาผนวกเข้าด้วยกัน

สร้างปราการป้องกัน: Authentication และ Authorization ที่แข็งแกร่ง

ส่วนที่สำคัญที่สุดของแอปพลิเคชันทุกตัวคือความปลอดภัย เราได้อุทิศสองบทสุดท้ายให้กับระบบรักษาความปลอดภัยแบบเต็มรูปแบบ ซึ่งเป็นสิ่งที่นักพัฒนาทุกคนต้องเข้าใจอย่างลึกซึ้ง

บทที่ 14: ระบบ Authentication ด้วย JWT และ Passport จะนำคุณเข้าสู่โลกของการพิสูจน์ตัวตน (Authentication) โดยใช้มาตรฐานอุตสาหกรรมอย่าง **JSON Web Tokens (JWT)** ร่วมกับ **Passport.js** ซึ่งเป็นเครื่องมือจัดการการยืนยันตัวตนที่ได้รับความนิยมสูงสุดในระบบนิเวศของ Node.js เราจะสร้าง AuthService และ LoginController ตั้งแต่เริ่มต้น พร้อมเรียนรู้การใช้ @UseGuards(AuthGuard) เพื่อปกป้อง Endpoint ต่าง ๆ อย่างง่ายดายแต่มีประสิทธิภาพ
ที่สำคัญกว่านั้น เราจะเจาะลึกในระดับ **Advanced Auth Examples** โดยเฉพาะอย่างยิ่ง **การจัดการ Refresh Token ใน NestJS** ซึ่งเป็นแนวทางปฏิบัติที่ดีที่สุดในการรักษาความปลอดภัยและประสบการณ์ของผู้ใช้ (User Experience) ให้คงอยู่ได้อย่างยาวนานโดยไม่ต้อง Log-in ซ้ำบ่อย ๆ บทนี้สรุปด้วย *Full Integration Project NestJS* ที่แสดงให้เห็นถึงการทำงานร่วมกันของ JWT และ Passport ในสถานการณ์จริง

การกำหนดสิทธิ์แบบละเอียด: Role-Based Access Control (RBAC)

เมื่อผู้ใช้ได้รับการยืนยันตัวตนแล้ว ขั้นตอนต่อไปคือการกำหนดสิทธิ์การเข้าถึง (Authorization) **บทที่ 15: Authorization แบบ Role-Based Access Control (RBAC)** จะนำเสนอวิธีการจัดการสิทธิ์แบบยืดหยุ่นและมีประสิทธิภาพที่สุด

เราจะเริ่มจากการกำหนด **Roles บน User Entity** และเรียนรู้การสร้าง **RoleGuard** ที่เป็นหัวใจของการตรวจสอบสิทธิ์การเข้าใช้ โดยใช้แนวคิดที่เรียกว่า **Metadata** ผ่านการใช้ @SetMetadata

และ Reflector ของ NestJS ซึ่งเป็นเทคนิคที่ทำให้การกำหนดสิทธิ์เป็นไปอย่างยืดหยุ่นและสามารถจัดการได้ง่ายผ่าน Decorator

บทที่ 15 ไม่ได้หยุดอยู่แค่การแนะนำแนวคิดเท่านั้น แต่จะนำเสนอ *Full Project NestJS – RoleGuard + JWT + Multi-role + Swagger UI* ซึ่งเป็นการบูรณาการองค์ประกอบทั้งหมดที่เราได้เรียนรู้มาตั้งแต่ต้นเข้าด้วยกันอย่างสมบูรณ์แบบ ทั้งระบบ Authentication, Authorization แบบหลาย Role, และการจัดทำเอกสาร Swagger ที่แสดงข้อมูลการกำหนดสิทธิ์อย่างครบถ้วน ทำให้คุณมีพิมพ์เขียว (Blueprint) สำหรับการสร้างระบบความปลอดภัยที่ซับซ้อนและพร้อมใช้งานจริง

สำหรับใคร?

หนังสือเล่มนี้เหมาะสำหรับนักพัฒนาที่เคยมีประสบการณ์พื้นฐานในการพัฒนาเว็บแอปพลิเคชันด้วย NestJS หรือเฟรมเวิร์กอื่น ๆ ที่มีแนวคิดใกล้เคียงกัน เช่น Angular หรือ Spring Boot และต้องการยกระดับความรู้และทักษะของตนเองไปสู่การพัฒนาแอปพลิเคชันระดับมืออาชีพ (Professional Development) ที่ต้องคำนึงถึงความมั่นคง ความปลอดภัย และการดูแลรักษา (Maintainability)

"NestJS Web Programming: Advance" คือกุญแจสำคัญที่จะปลดล็อกศักยภาพสูงสุดของ NestJS และเปลี่ยนคุณจากนักพัฒนาที่มีประสบการณ์มาเป็นสถาปนิกซอฟต์แวร์ (Software Architect) ที่สามารถสร้างแอปพลิเคชัน Web Programming ที่ซับซ้อน ทรงประสิทธิภาพ และพร้อมรับมือกับความท้าทายในโลกของการผลิตได้อย่างแท้จริง

เราขอเชิญชวนให้คุณเริ่มต้นการเดินทางที่น่าตื่นเต้นนี้ไปพร้อมกับเรา และหวังว่าเนื้อหาในเล่มนี้จะเป็นประโยชน์อย่างยิ่งต่อการพัฒนาอาชีพของคุณในฐานะผู้เชี่ยวชาญด้าน NestJS

ด้วยรักและปรารถนาดี

ศูนย์หนังสือราคานักเรียน

สารบัญ

หน้า

บทที่ 11 การใช้ ConfigModule และ Environment Variables 1

- การใช้ ConfigModule และ Environment Variables
- ConfigModule และ Environment Variables
- การใช้ ConfigModule และ Environment Variables
- การใช้ @nestjs/config (เชิงลึก)
- การสร้างไฟล์ .env และ .env.production ใน NestJS
- การ Inject Config แบบ Strongly-Typed
- การจัดการ Secrets อย่างปลอดภัย
- ตัวอย่างบูรณาการ
- Ultimate Examples

บทที่ 12 การใช้งาน Global Interceptors 87

- การใช้งาน Global Interceptors
- บทที่ 12: การใช้งาน Global Interceptors
- บทที่ 12: การใช้งาน Global Interceptors (Deep Dive)
- การสร้าง Interceptor สำหรับ Response Format (NestJS)
- NestJS Integration Examples
- การจัดการ Log & Transform Response ใน NestJS Interceptors
- NestJS: Logging & Transform Response
- การใช้ AOP กับ Interceptors (Before / After Execution)
- NestJS – AOP Interceptor (Before/After Execution)
- Ultimate Integration Example สำหรับ NestJS Interceptor AOP + Logging + Response Transform + Error Handling + Conditional Role + Global/Scoped
- การใช้ Interceptor เป็น Global / Scoped
- NestJS Interceptor – Global / Scoped
- Ultimate Full Project Template – NestJS Interceptor Integration แบบ production-ready

●ตัวอย่างบูรณาการ (Integration Examples)
●Ultimate Super Integration Project
บทที่ 13 การใช้ Swagger สำหรับ API Docs 196
●การใช้ Swagger สำหรับ API Docs
●การใช้ Swagger สำหรับ API Docs – รายละเอียดเชิงลึก
●การติดตั้งและใช้ @nestjs/swagger ใน NestJS
●การเพิ่ม Metadata ใน Swagger (NestJS)
●ตัวอย่าง Ultimate Super Project NestJS + Swagger Integration
●การสร้าง UI Docs อัตโนมัติใน NestJS
●การ Deploy Swagger ใน Production
●ตัวอย่าง Ultimate Project NestJS + Swagger ที่ deploy-ready
●ตัวอย่างบูรณาการ
●ตัวอย่าง Ultimate NestJS + Swagger
บทที่ 14 ระบบ Authentication ด้วย JWT และ Passport 272
●ระบบ Authentication ด้วย JWT และ Passport
●ระบบ Authentication ด้วย JWT และ Passport – รายละเอียดเชิงลึก
●การใช้ Passport + JWT
●การสร้าง AuthService และ LoginController
●การใช้ @UseGuards(AuthGuard) ใน NestJS
●NestJS Advanced Auth Examples – Overview
●การจัดการ Refresh Token ใน NestJS
●ตัวอย่างบูรณาการ
●Full Integration Project NestJS
บทที่ 15 Authorization แบบ Role-Based Access Control (RBAC).....367
●Authorization แบบ Role-Based Access Control (RBAC)
●Authorization แบบ Role-Based Access Control (RBAC) – รายละเอียดเชิงลึก
●การกำหนด Roles บน User Entity
●การสร้าง RoleGuard ตรวจสอบการเข้าใช้ (RBAC)
●Full Project NestJS – RoleGuard + JWT + Multi-role + Swagger UI

- การใช้ @SetMetadata และ Reflector
- ตัวอย่างบูรณาการ
- Ultimate Example

บรรณานุกรม 451

บทที่ 11

การใช้ ConfigModule และ Environment Variables

(ConfigModule and Environment Variables)

เนื้อหา

- การใช้ ConfigModule และ Environment Variables
- ConfigModule และ Environment Variables
- การใช้ ConfigModule และ Environment Variables
- การใช้ @nestjs/config (เชิงลึก)
- การสร้างไฟล์ .env และ .env.production ใน NestJS
- การ Inject Config แบบ Strongly-Typed
- การจัดการ Secrets อย่างปลอดภัย
- ตัวอย่างบูรณาการ
- Ultimate Examples

บทนำบทที่ 11: การใช้ ConfigModule และ Environment Variables

ในยุคปัจจุบันของการพัฒนาแอปพลิเคชันด้วย NestJS การจัดการการตั้งค่า (configuration) อย่างมีประสิทธิภาพถือเป็นสิ่งสำคัญ ConfigModule และ environment variables จึงกลายเป็นเครื่องมือหลักที่ช่วยให้โค้ดของเรามีความยืดหยุ่นและปลอดภัยมากยิ่งขึ้น ในบทนี้ เราจะเริ่มต้นด้วยการทำความรู้จักกับ **@nestjs/config** ซึ่งเป็นโมดูลหลักที่ NestJS จัดเตรียมไว้เพื่อรองรับการโหลดค่าการตั้งค่าต่าง ๆ จาก environment variables

หนึ่งในแนวปฏิบัติที่ดีคือการสร้างไฟล์ **.env** สำหรับ environment พื้นฐาน และ **.env.production** สำหรับ environment การผลิต เพื่อแยกความแตกต่างของค่าการตั้งค่าที่อาจเปลี่ยนแปลงระหว่างการพัฒนา การทดสอบ และการใช้งานจริง การจัดการแบบนี้ช่วยให้เราไม่ต้องแก้ไขโค้ดซ้ำ ๆ เพียงแค่เปลี่ยน environment variables ก็สามารถปรับ behavior ของแอปได้อย่างง่ายดาย

ใน NestJS การ inject ค่าจาก ConfigModule สามารถทำได้หลายวิธี แต่ในบทนี้เราจะเน้นการทำแบบ **strongly-typed** ซึ่งหมายถึงการสร้าง interface หรือ class สำหรับ type ของ configuration ที่เราต้องการ การทำแบบนี้ช่วยลดข้อผิดพลาดจากการพิมพ์ผิด และทำให้ IDE สามารถช่วยตรวจสอบและ autocomplete ค่าต่าง ๆ ได้อย่างแม่นยำ

อีกหัวข้อสำคัญที่เราจะกล่าวถึงคือ **การจัดการความลับ (secrets)** เช่น API keys, database passwords หรือ tokens การเก็บความลับเหล่านี้อย่างปลอดภัยเป็นสิ่งจำเป็น การใช้ environment variables ร่วมกับ ConfigModule ทำให้เราสามารถแยกความลับออกจากโค้ดและ repository ได้อย่างชัดเจน

นอกจากนี้ เรายังจะแนะนำ **เคล็ดลับและ best practices** ในการจัดการ secrets เช่น การใช้ dotenv-safe, การตั้งค่า default values, และการตรวจสอบว่าค่าที่จำเป็นถูกกำหนดหรือไม่ เทคนิคเหล่านี้ช่วยเพิ่มความปลอดภัยและลดปัญหาที่อาจเกิดขึ้นใน production environment

บทนี้ยังครอบคลุมตัวอย่างการโหลด configuration แบบ modular ซึ่งทำให้โค้ดสามารถจัดการ environment variables ของแต่ละโมดูลได้อย่างยืดหยุ่น การแยก responsibility แบบนี้ทำให้ระบบสามารถ scale ได้ง่ายและ maintainable มากขึ้น

ท้ายที่สุด คุณจะได้เห็น **แนวทางการใช้งาน ConfigModule ในโปรเจกต์จริง** ซึ่งสามารถปรับใช้ได้ทั้งแอปขนาดเล็กและขนาดใหญ่ พร้อมทั้งเข้าใจหลักการของ environment-driven configuration อย่างถ่องแท้ ทำให้การพัฒนาแอป NestJS ของคุณมีความปลอดภัย ยืดหยุ่น และมืออาชีพมากยิ่งขึ้น

การใช้ ConfigModule และ Environment Variables

- การใช้ @nestjs/config
- การสร้างไฟล์ .env และ .env.production
- การ Inject config แบบ strongly-typed
- เคล็ดลับจัดการความลับ/secret อย่างปลอดภัย

บทที่ 11 — การใช้ ConfigModule และ Environment Variables (ละเอียด)

บทนี้จะพาคุณตั้งค่า configuration ให้ **ปลอดภัย**, **เป็นระบบ**, และ **strongly-typed** ในแอป NestJS ของจริง โดยครอบคลุม:

- การติดตั้งและใช้งาน @nestjs/config
- การสร้าง .env และ .env.production (และการเลือกไฟล์ตาม NODE_ENV)
- การทำ configuration ให้ **strongly-typed** (ใช้ registerAs + ConfigType + @Inject)
- การตรวจสอบค่า env (validation with Joi)
- เคล็ดลับจัดการความลับ / secrets อย่างปลอดภัยใน production

ผมจะยกตัวอย่างโค้ดจริง (TypeScript) พร้อมไฟล์ .env ตัวอย่าง และ checklist ปฏิบัติได้ทันที

---

1) ติดตั้งเบื้องต้น

npm install @nestjs/config joi

หรือ yarn add @nestjs/config joi

joi เอาไว้ทำ validation ของ env (optional แต่แนะนำ)

---

2) เปิด ConfigModule ใน AppModule — แบบพื้นฐาน + เลือกไฟล์ตาม NODE_ENV

```
// src/app.module.ts
import { Module } from '@nestjs/common';
import { ConfigModule } from '@nestjs/config';
import configuration from './config/configuration'; // optional loader

@Module({
  imports: [
    ConfigModule.forRoot({
      isGlobal: true, // แนะนำให้ global เพื่อไม่ต้อง import ซ้ำ
      envFilePath: [`.env.${process.env.NODE_ENV}`, '.env'], // โหลดตาม NODE_ENV ก่อน แล้ว fallback ไป .env
      load: [configuration], // optional: load config object(s)
      ignoreEnvFile: process.env.NODE_ENV === 'production', // ใน production อาจไม่ใช้ไฟล์ .env
      validationOptions: {
        abortEarly: true,
      },
      // validationSchema: ... // เราจะใส่ Joi schema ต่อไป
    }),
    // ... โมดูลอื่น ๆ
  ],
})
export class AppModule {}
```

คำอธิบาย:

- envFilePath เป็น array — Nest จะพยายามโหลดไฟล์ตัวแรกก่อน เช่น NODE_ENV=production -> .env.production ⟶ ถ้าไม่พบจะโหลด .env
- ignoreEnvFile: true ใน production เป็น pattern ที่ปลอดภัย: ใน production ควรเอา secret จาก secret manager / container env variables แทนการพึ่งไฟล์ .env ที่เก็บใน repo

---

3) ตัวอย่างไฟล์ .env และ .env.production

.env (สำหรับ dev)

```
NODE_ENV=development
PORT=3000
```

DATABASE_HOST=localhost

DATABASE_PORT=5432

DATABASE_USER=dev_user

DATABASE_PASSWORD=dev_pass

DATABASE_NAME=app_dev

JWT_SECRET=dev_jwt_secret

JWT_EXPIRES_IN=3600

.env.production (ตัวอย่าง — **ห้าม commit ขึ้น repo**)

NODE_ENV=production

PORT=8080

DATABASE_HOST=db.prod.internal

DATABASE_PORT=5432

DATABASE_USER=prod_user

DATABASE_PASSWORD=ProdStrongPassword!@#

JWT_SECRET=SuperSecretProductionKey

JWT_EXPIRES_IN=3600

ข้อสำคัญ: อย่า commit ไฟล์ .env.production ที่มี secrets จริง ขึ้น VCS — ใช้ตัวอย่าง .env.example แทน (มีแต่ชื่อ keys แต่ไม่ใส่ค่า)

.env.example:

NODE_ENV=

PORT=

DATABASE_HOST=

DATABASE_PORT=

DATABASE_USER=

DATABASE_PASSWORD=

DATABASE_NAME=

JWT_SECRET=

JWT_EXPIRES_IN=

4) การทำ validation ของ env ด้วย Joi (แนะนำ)

การตรวจ schema ของ env จะช่วยจับ error ตั้งแต่เริ่ม:

```
// src/config/env.validation.ts
import * as Joi from 'joi';

export const envValidationSchema = Joi.object({
  NODE_ENV: Joi.string().valid('development','production','test').default('development'),
  PORT: Joi.number().default(3000),

  DATABASE_HOST: Joi.string().required(),
  DATABASE_PORT: Joi.number().default(5432),
  DATABASE_USER: Joi.string().required(),
  DATABASE_PASSWORD: Joi.string().required(),
  DATABASE_NAME: Joi.string().required(),

  JWT_SECRET: Joi.string().min(10).required(),
  JWT_EXPIRES_IN: Joi.number().default(3600),
});
```

แล้วเชื่อมกับ ConfigModule.forRoot:

```
ConfigModule.forRoot({
  isGlobal: true,
  envFilePath: [`.env.${process.env.NODE_ENV}`, '.env'],
  validationSchema: envValidationSchema,
  ignoreEnvFile: process.env.NODE_ENV === 'production',
});
```

ผลลัพธ์: ถ้าค่า env ขาดหรือชนิดไม่ถูกต้อง แอปจะไม่ start — ป้องกันปัญหา runtime

---

5) การทำ Config แบบ modular & strongly-typed ด้วย registerAs + ConfigType

วิธีนี้ช่วยให้เรา **ได้ type-safe config** (ไม่ต้องเรียก configService.get('foo') เป็น string แบบไม่ sure)

ตัวอย่าง: แยกไฟล์ config หลายไฟล์ — database, auth, app

```
// src/config/database.config.ts
import { registerAs } from '@nestjs/config';
```

```
export default registerAs('database', () => ({
  host: process.env.DATABASE_HOST,
  port: parseInt(process.env.DATABASE_PORT, 10) || 5432,
  user: process.env.DATABASE_USER,
  password: process.env.DATABASE_PASSWORD,
  name: process.env.DATABASE_NAME,
}));
// src/config/auth.config.ts
import { registerAs } from '@nestjs/config';

export default registerAs('auth', () => ({
  jwtSecret: process.env.JWT_SECRET,
  jwtExpiresIn: parseInt(process.env.JWT_EXPIRES_IN, 10) || 3600,
}));
```

รวมแล้วโหลดใน ConfigModule.forRoot:

```
import databaseConfig from './config/database.config';
import authConfig from './config/auth.config';

ConfigModule.forRoot({
  isGlobal: true,
  load: [databaseConfig, authConfig],
  validationSchema: envValidationSchema,
});
```

การใช้งานแบบ strongly-typed ใน Service/Module:

```
// src/database/database.module.ts
import { Module } from '@nestjs/common';
import { ConfigModule } from '@nestjs/config';
import { DatabaseService } from './database.service';
import databaseConfig from '../config/database.config';

@Module({
  imports: [ConfigModule],
  providers: [DatabaseService],
  exports: [DatabaseService],
```

```
})
export class DatabaseModule {}
```

และใน service:

```
// src/database/database.service.ts
import { Inject, Injectable } from '@nestjs/common';
import { ConfigType } from '@nestjs/config';
import databaseConfig from '../config/database.config';

@Injectable()
export class DatabaseService {
  constructor(@Inject(databaseConfig.KEY) private dbConfig: ConfigType<typeof
databaseConfig>) {}

  getConnectionString() {
    const { user, password, host, port, name } = this.dbConfig;
    return `postgresql://${user}:${password}@${host}:${port}/${name}`;
  }
}
```

ข้อดี:

- dbConfig มี type ชัดเจน (จาก ConfigType<typeof databaseConfig>) — editor ช่วย autocomplete
- ปลอดภัยกว่า configService.get<string>('DATABASE_HOST') เพราะไม่ต้องพึ่ง string key

อีกตัวอย่างการ inject auth config:

```
import authConfig from '../config/auth.config';
constructor(@Inject(authConfig.KEY) private auth: ConfigType<typeof authConfig>) {}
```

---

6) ตัวอย่างการใช้ ConfigService ทั่วไป

บางครั้งยังอยากใช้ ConfigService โดยตรง:

```
import { ConfigService } from '@nestjs/config';

@Injectable()
export class SomeService {
  constructor(private configService: ConfigService) {}
```

```
  getJwtSecret() {
    return this.configService.get<string>('auth.jwtSecret'); // ถ้าใช้ registerAs + load
    // หรือ get('JWT_SECRET') ถ้าไม่ใช้ registerAs
  }
}
```

ถ้าใช้ registerAs('auth', ...) คุณสามารถเข้าถึงเป็น configService.get('auth.jwtSecret')

---

7) การจัดการ Secrets อย่างปลอดภัย — แนวปฏิบัติ (practical tips)

สรุป: อย่าเก็บ secrets ใน repo, ใช้ secret manager/secret store, rotate, limit access, audit

รายละเอียดและวิธีปฏิบัติที่แนะนำ:

1. **อย่า commit .env ที่มีค่า secret ขึ้น VCS**
   - เก็บ .env.example สำหรับคนพัฒนา แต่ไม่เก็บค่า secret จริง
2. **ใน production ใช้ Secret Manager หรือ Secret Store**
   - Cloud providers: **AWS Secrets Manager / SSM Parameter Store**, **GCP Secret Manager**, **Azure Key Vault**
   - On-prem / self-hosted: **HashiCorp Vault** (ยอดนิยม)
   - Nest app ควรดึง secret ที่ startup (หรือ runtime) ผ่าน SDK ของ cloud provider แล้ว inject เข้า ConfigService (หรือใช้ sidecar)
3. **ใน container / Docker ใช้ environment variables หรือ Docker secrets**
   - Docker run: docker run -e DB_PASSWORD=... my-app
   - Docker secrets (swarm): docker secret create db_password secret.txt และ reference ใน service
   - Kubernetes: ใช้ Secret (base64-encoded) และ mount เป็น env var หรือ volume:
   - kubectl create secret generic db-secret --from-literal=DB_PASSWORD='ProdSecret'

     ใน Deployment:

     ```
     env:
       - name: DATABASE_PASSWORD
         valueFrom:
           secretKeyRef:
             name: db-secret
             key: DB_PASSWORD
     ```
4. **ให้บริการที่เข้าถึง secrets ใช้ least privilege**
   - IAM role / service account มีสิทธิน้อยที่สุดที่จำเป็น

- ห้ามใช้งาน accounts ที่มีสิทธิสูงเกินจำเป็น

5. **Encryption at rest / in transit**
   - Secret stores มักเข้ารหัสอยู่แล้ว แต่ตรวจสอบเสมอ
   - การสื่อสารระหว่างแอปและ secret manager ต้องใช้ TLS
6. **Rotate secrets เป็นประจำ**
   - ใช้นโยบายหมุนรหัสผ่านและ keys อัตโนมัติ
   - ตรวจสอบ secret rotation compatibility ในแอป (เช่น reload config / restart หรือ dynamic refresh)
7. **ไม่ log ค่า secret**
   - ห้าม print env values ที่เป็น secret ใน logs / error messages
   - ถ้าจำเป็นต้อง log ให้ mask (เช่น password: ****)
8. **ใช้ runtime injection แทนการเขียนค่าในไฟล์ config ที่ถูก build เข้า image**
   - ใน Docker multi-stage build อย่าเอา .env เข้า image อีกทีหนึ่ง
   - ให้แอปอ่านค่า env ที่ระบบส่งให้ตอน runtime (container orchestration จะ supply)
9. **ใช้ Vault or Secret Manager integration ถ้าต้องการ dynamic secrets**
   - ตัวอย่าง: Vault ให้ dynamic db credentials ที่ auto rotate
10. **CI/CD: อย่าเก็บ secret ใน CI config**

- เก็บ secret ใน CI secret store (GitHub Actions secrets, GitLab CI variables) และ inject ใน runtime step เท่านั้น

11. **การตรวจสอบความปลอดภัย (audit)**

- เปิด logging / audit ใน secret managers เพื่อ trace ใครดึง secret เมื่อไหร่

---

8) ตัวอย่าง: ผสมกับ Docker + Kubernetes (แนวปฏิบัติ)

- Dockerfile: ไม่รวมไฟล์ .env ใน image
- Deployment: ใช้ k8s Secret (valueFrom secretKeyRef) เพื่อส่งค่า DATABASE_PASSWORD, JWT_SECRET เป็นต้น
- ใน ConfigModule.forRoot ตั้ง ignoreEnvFile: true ใน production เพื่อบังคับให้อ่านจาก process.env (ที่ orchestration มอบให้)

---

9) ตัวอย่าง: โหลด secret จาก AWS Secrets Manager (แนวทาง)

(โค้ดย่อให้เห็นแนวคิด — ต้องติดตั้ง AWS SDK และ configure IAM role)

```
// src/config/aws-secrets.loader.ts
import { registerAs } from '@nestjs/config';
import AWS from 'aws-sdk';
```

```
export default registerAs('awsSecrets', () => {
  // synchronous registerAs ไม่เหมาะกับ async fetch — แต่แนวทาง: fetch ก่อน bootstrap แล้ว set process.env
  return {
    // ค่า default หรือ placeholder
  }
});
```

แนวทางปฏิบัติจริง: ก่อน bootstrapping Nest, ให้ process ที่ run container ดึง secrets มาใส่เป็น ENV หรือเขียนไฟล์ tmp แล้วให้ Nest อ่าน — หรือใช้ runtime provider ที่ async (ดู advanced patterns ของ Nest + custom provider)

---

10) Checklist: การใช้งาน ConfigModule แบบ Production-ready

- ☐ ใช้ ConfigModule.forRoot({ isGlobal: true, load: [...], ignoreEnvFile: process.env.NODE_ENV === 'production', validationSchema })
- ☐ แยก config เป็นไฟล์ (database, auth, app) และใช้ registerAs
- ☐ ใช้ Joi / validation schema เพื่อตรวจ env ใน startup
- ☐ เก็บตัวอย่าง .env.example ไว้ใน repo — **ไม่**เก็บ .env ที่มีค่า secret
- ☐ ใน production: ใช้ secret manager / k8s secrets / cloud provider secrets
- ☐ ไม่ log secrets และให้การเข้าถึง secrets แบบ least privilege
- ☐ วางแผนการ rotate และ monitoring/audit
- ☐ ตรวจสอบว่า ignoreEnvFile: true ถ้าใช้ external secret manager

---

11) ตัวอย่างโค้ดรวม (รวบรัด) — AppModule + config files + DB service + Auth usage

```
// src/config/app.config.ts
import { registerAs } from '@nestjs/config';
export default registerAs('app', () => ({
  nodeEnv: process.env.NODE_ENV || 'development',
  port: parseInt(process.env.PORT, 10) || 3000,
}));
// src/config/index.ts
import databaseConfig from './database.config';
import authConfig from './auth.config';
```

```
import appConfig from './app.config';
export default [databaseConfig, authConfig, appConfig];
// src/main.ts
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';
import { ValidationPipe } from '@nestjs/common';

async function bootstrap(){
  const app = await NestFactory.create(AppModule);
  app.useGlobalPipes(new ValidationPipe({ whitelist: true, transform: true }));
  await app.listen(process.env.PORT || 3000);
}
bootstrap();
// src/app.module.ts
import { Module } from '@nestjs/common';
import { ConfigModule } from '@nestjs/config';
import configs from './config';

@Module({
  imports: [
    ConfigModule.forRoot({
      isGlobal: true,
      load: configs,
      envFilePath: [`.env.${process.env.NODE_ENV}`, '.env'],
      ignoreEnvFile: process.env.NODE_ENV === 'production',
      validationSchema: envValidationSchema, // import Joi schema
    }),
    // rest modules...
  ],
})
export class AppModule {}
```

ตัวอย่างการ inject แบบ strongly-typed ใน service:

```
// src/auth/auth.service.ts
import { Injectable, Inject } from '@nestjs/common';
```

```
import authConfig from '../config/auth.config';
import { ConfigType } from '@nestjs/config';

@Injectable()
export class AuthService {
  constructor(@Inject(authConfig.KEY) private auth: ConfigType<typeof authConfig>) {}

  getJwtSecret() {
    return this.auth.jwtSecret;
  }
}
```

---

12) ข้อผิดพลาด/สาเหตุที่พบบ่อยกับ config และวิธีแก้

- **ลืม set NODE_ENV ⟶ โหลดผิดไฟล์ .env**
  แก้: ตั้งค่า NODE_ENV ใน environment / container / CI และใช้ envFilePath แบบ array
- **Hardcode secret ใน source**
  แก้: ย้ายไป secret manager / env vars
- **ไม่ได้ validation ⟶ app crash ที่ runtime**
  แก้: ใช้ Joi validation schema
- **ใช้ process.env กระจัดกระจาย ⟶ ยากต่อ testing**
  แก้: centralize config ด้วย registerAs + ConfigService / typed injection
- **Logging เผลอแสดงค่า secret**
  แก้: mask secret ใน logs / ปิด debug logs ใน production

---

13) สรุป (best practices สั้น)

1. ทำให้ ConfigModule เป็น global และโหลด config แยกตาม domain (database, auth, app)
2. ใช้ registerAs + ConfigType<typeof config> เพื่อ strongly-typed injection
3. ทำ validation (Joi) เพื่อ fail-fast เมื่อ config ขาดหรือผิดประเภท
4. ใน production อย่าใช้ไฟล์ .env แต่ให้ใช้ secret manager / k8s secret / cloud provider secret store
5. ห้าม commit secrets ขึ้น VCS, ใช้ .env.example แทน
6. จำกัดการเข้าถึง secrets, เปิด audit, หมุน secrets เป็นประจำ และอย่า log ค่า secret

---

บทที่ 11: การใช้ ConfigModule และ Environment Variables

ภาพรวม

ในระบบขนาดใหญ่ การ hard-code ค่าต่าง ๆ เช่น API keys, Database URLs, หรือ JWT secret ลงในโค้ดเป็นสิ่งที่ควรหลีกเลี่ยงอย่างยิ่ง เพราะ:

- ทำให้ไม่สามารถเปลี่ยนค่าตามสภาพแวดล้อมได้ง่าย (Dev/Test/Production)
- มีความเสี่ยงสูงหากรั่วไหล (security breach)
- ไม่สามารถ scale หรือ deploy ได้สะดวกใน CI/CD pipelines

ดังนั้น NestJS จึงมี **@nestjs/config** ซึ่งช่วยจัดการค่า configuration ผ่าน **environment variables (.env)** ได้อย่างเป็นระบบ, ปลอดภัย และรองรับ **type safety** เต็มรูปแบบ

---

1 การใช้ @nestjs/config

ติดตั้ง

```
npm install @nestjs/config
```

การตั้งค่าเบื้องต้น

ใน app.module.ts:

```
import { Module } from '@nestjs/common';
import { ConfigModule } from '@nestjs/config';
import { DatabaseModule } from './database.module';
import configuration from './config/configuration';

@Module({
  imports: [
    ConfigModule.forRoot({
      isGlobal: true,          // ให้ทุก Module เข้าถึง Config ได้โดยไม่ต้อง import ซ้ำ
      load: [configuration],   // โหลด config จากไฟล์ TypeScript
      envFilePath: ['.env'],   // ระบุ path ของไฟล์ environment
    }),
    DatabaseModule,
  ],
})
export class AppModule {}
```

---

2 การสร้างไฟล์ .env และ .env.production

NestJS ใช้ **dotenv** ภายในเพื่อโหลดค่าตัวแปรจากไฟล์ .env

ตัวอย่าง:

.env (ใช้ใน local development)

```
APP_PORT=3000
DATABASE_HOST=localhost
DATABASE_PORT=5432
DATABASE_USER=postgres
DATABASE_PASSWORD=123456
JWT_SECRET=dev_secret_key
```

.env.production (ใช้ใน production)

```
APP_PORT=8080
DATABASE_HOST=prod-db.cluster.amazonaws.com
DATABASE_PORT=5432
DATABASE_USER=prod_admin
DATABASE_PASSWORD=${PROD_DB_PASSWORD}  # โหลดจาก secret store หรือ environment จริง
JWT_SECRET=${JWT_SECRET}               # ไม่เขียนค่าจริงลงไฟล์
```

Tip: ใช้ environment variable ของระบบจริง (เช่น Docker, GitHub Actions, Kubernetes Secret) แทนการเก็บ secret ในไฟล์ .env เพื่อความปลอดภัยสูงสุด

---

3 การ Inject config แบบ strongly-typed

เพื่อให้โค้ด type-safe และ autocomplete ได้ใน VS Code,

คุณสามารถสร้างไฟล์ config แบบแยกโมดูล เช่น:

src/config/configuration.ts

```
export default () => ({
  app: {
    port: parseInt(process.env.APP_PORT, 10) || 3000,
  },
  database: {
    host: process.env.DATABASE_HOST,
    port: parseInt(process.env.DATABASE_PORT, 10) || 5432,
    user: process.env.DATABASE_USER,
    password: process.env.DATABASE_PASSWORD,
```

```
  },
  jwt: {
    secret: process.env.JWT_SECRET,
  },
});
```

การใช้งานใน Service อย่าง strongly-typed

1. สร้าง interface type

```
export interface AppConfig {
  app: { port: number };
  database: { host: string; port: number; user: string; password: string };
  jwt: { secret: string };
}
```

2. Inject ใน service

```
import { Injectable } from '@nestjs/common';
import { ConfigService } from '@nestjs/config';
import { AppConfig } from './config.interface';

@Injectable()
export class AppService {
  constructor(private configService: ConfigService<AppConfig>) {}

  getHello(): string {
    const port = this.configService.get<number>('app.port', { infer: true });
    const dbHost = this.configService.get<string>('database.host', { infer: true });
    return `Running on port ${port}, connected to DB at ${dbHost}`;
  }
}
```

การใช้ { infer: true } ช่วยให้ TypeScript infer ชนิดข้อมูลจาก generic ของ ConfigService ได้อัตโนมัติ
ทำให้ IDE แนะนำ key ได้แบบ autocomplete และ type-safe

4 เคล็ดลับจัดการความลับ (Secrets) อย่างปลอดภัย

แนวทางปฏิบัติที่ดี (Best Practices)

1. ไม่เก็บ secrets ใน repo

อย่า commit ไฟล์ .env ขึ้น GitHub

ใช้ .gitignore:

```
.env
.env.*
```

2. ใช้เครื่องมือจัดการ secrets

- **Docker / Kubernetes Secrets**
- **AWS Secrets Manager / Parameter Store**
- **Vault (HashiCorp)**
- **GitHub Actions Encrypted Secrets**

จากนั้นใน production ให้ NestJS ดึงค่า environment โดยตรง เช่น:

```
JWT_SECRET=$(aws secretsmanager get-secret-value --secret-id MyAppJWT)
```

3. แยกไฟล์ตาม environment

คุณสามารถตั้งค่าโหลดหลายไฟล์ได้ เช่น:

```
ConfigModule.forRoot({
  envFilePath: process.env.NODE_ENV === 'production' ? '.env.production' : '.env',
});
```

4. Validate ค่าที่โหลดจาก environment

NestJS รองรับการใช้ **Joi validation** เพื่อป้องกัน config ผิดพลาด:

```
import * as Joi from 'joi';

ConfigModule.forRoot({
  validationSchema: Joi.object({
    APP_PORT: Joi.number().default(3000),
    DATABASE_HOST: Joi.string().required(),
    DATABASE_PORT: Joi.number().default(5432),
    JWT_SECRET: Joi.string().required(),
  }),
});
```

หากค่าที่จำเป็นไม่มีหรือผิด type ⟶ NestJS จะ throw error ทันทีตอนเริ่มต้นระบบ

□ สรุปภาพรวม

หัวข้อ	จุดประสงค์	ตัวอย่างสำคัญ
@nestjs/config	โหลดและจัดการ environment variables	ConfigModule.forRoot()
.env / .env.production	เก็บค่าการตั้งค่าแบบ environment-specific	APP_PORT, JWT_SECRET
Strongly-typed Config	ป้องกัน error และช่วย autocomplete	ConfigService<AppConfig>
Secrets Management	ปลอดภัย, ไม่ commit secret	ใช้ Secrets Manager, .gitignore
Validation Schema	ตรวจสอบค่าที่โหลดจาก env	Joi

□ สรุปแนวคิดเชิงสถาปัตยกรรม

- NestJS สนับสนุน **12-Factor App Principle** ข้อที่ 3: *"Store config in the environment"*
- ConfigModule ทำให้ระบบรองรับ **multi-environment deployment** ได้ง่าย (เช่น dev ⟶ staging ⟶ prod)
- เมื่อรวมกับ **DI (Dependency Injection)** และ **typed config**, โค้ดจะ maintain ได้ดีและปลอดภัยยิ่งขึ้น

การใช้ @nestjs/config (เชิงลึก)

□ ภาพรวม

@nestjs/config คือ **official module ของ NestJS** ที่ช่วยให้คุณ:

- โหลดค่าต่าง ๆ จาก environment variables (.env, system env)
- ใช้งานค่าพวกนี้ในทุก service/module ผ่าน Dependency Injection
- มีระบบ type safety, default values, และ schema validation (ผ่าน Joi)
- รองรับหลาย environment (development, staging, production)

⚙□ 1. ติดตั้งและเริ่มต้นใช้งาน

npm install @nestjs/config

จากนั้น import module ใน app.module.ts:

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

@Module({
```

```
  imports: [
    ConfigModule.forRoot({
      isGlobal: true, // □ ทำให้ทุก module ใช้ config ได้โดยไม่ต้อง import ซ้ำ
    }),
  ],
  controllers: [AppController],
  providers: [AppService],
})
export class AppModule {}
```

□ isGlobal: true คือการประกาศว่า ConfigModule เป็น global module
ดังนั้น service หรือ controller ใด ๆ ก็ inject ConfigService ได้เลยโดยไม่ต้อง import อีกใน module ของมัน

□ 2. สร้างไฟล์ .env

สร้างไฟล์ .env ไว้ที่ root ของโปรเจกต์:

```
APP_NAME=MyNestApp
APP_PORT=3000
DATABASE_URL=postgres://user:password@localhost:5432/mydb
JWT_SECRET=mysecretkey
```

ตอนที่ NestJS รัน มันจะโหลดค่าพวกนี้เข้าสู่ process.env โดยอัตโนมัติ

□ 3. Inject ค่า Config ผ่าน ConfigService

ConfigService คือ class ที่ใช้เข้าถึงค่าใน .env
ตัวอย่างการใช้งานใน service:

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

@Injectable()
export class AppService {
  constructor(private readonly configService: ConfigService) {}

  getHello(): string {
    const appName = this.configService.get<string>('APP_NAME');
    const port = this.configService.get<number>('APP_PORT');
```

```
    return `Welcome to ${appName}! Running on port ${port}.`;
  }
}
```

□ การใช้ generic <string> หรือ <number> จะช่วยให้ TypeScript รู้ชนิดข้อมูล
และ VS Code autocomplete ได้ถูกต้อง

□ 4. ใช้ Default Values

คุณสามารถตั้งค่า default value ได้ในกรณีที่ environment variable ไม่มีใน .env:

```
const port = this.configService.get<number>('APP_PORT', 8080);
```

ถ้า .env ไม่มี APP_PORT ⟶ จะใช้ค่า 8080 แทน

□ 5. ใช้ร่วมกับ TypeScript Configuration Object

เพื่อจัดระเบียบ config และหลีกเลี่ยงการเรียก process.env โดยตรง
สามารถสร้าง config file ที่รวมค่าทั้งหมดไว้ในที่เดียว

□ src/config/configuration.ts

```
export default () => ({
  appName: process.env.APP_NAME || 'NestApp',
  port: parseInt(process.env.APP_PORT, 10) || 3000,
  database: {
    url: process.env.DATABASE_URL,
  },
  jwtSecret: process.env.JWT_SECRET,
});
```

แล้ว import ไฟล์นี้ใน app.module.ts:

```
ConfigModule.forRoot({
  isGlobal: true,
  load: [configuration], // □ โหลดค่าจากไฟล์ config
});
```

ใช้งานใน service ได้แบบนี้:

```
const dbUrl = this.configService.get<string>('database.url');
const jwtSecret = this.configService.get<string>('jwtSecret');
```

□ จุดเด่นคือคุณสามารถใช้ **nested key** (database.url) ได้อย่างสะดวก
และยังสามารถขยายหรือแยก config ตามโมดูลได้ (เช่น auth.config.ts, db.config.ts)

🧩 6. Validation Schema (ตรวจสอบค่าที่โหลดจาก .env)

หากต้องการให้ NestJS ตรวจสอบว่า environment variable ครบหรือไม่

สามารถใช้ **Joi Schema Validation** ได้:

npm install joi

จากนั้นแก้ใน app.module.ts:

```
import * as Joi from 'joi';

ConfigModule.forRoot({
  isGlobal: true,
  validationSchema: Joi.object({
    APP_NAME: Joi.string().required(),
    APP_PORT: Joi.number().default(3000),
    DATABASE_URL: Joi.string().uri().required(),
    JWT_SECRET: Joi.string().required(),
  }),
});
```

🧩 ถ้าค่าจาก .env ขาดหรือผิด type ⟶ NestJS จะ throw error ทันทีตอน startup

ช่วยลดโอกาส deploy ระบบผิด environment

⚡ 7. การแยก environment หลายชุด

เราสามารถกำหนดไฟล์ .env ตามสภาพแวดล้อมได้ เช่น:

- .env.development
- .env.production
- .env.test

ตัวอย่างใน app.module.ts:

```
ConfigModule.forRoot({
  isGlobal: true,
  envFilePath: [`.env.${process.env.NODE_ENV || 'development'}`],
});
```

เมื่อรัน:

NODE_ENV=production npm run start:prod

NestJS จะโหลดค่าจาก .env.production โดยอัตโนมัติ

---

8. ตัวอย่างโปรเจกต์ ConfigModule แบบสมบูรณ์

```
// app.module.ts
import { Module } from '@nestjs/common';
import { ConfigModule } from '@nestjs/config';
import configuration from './config/configuration';
import { AppService } from './app.service';
import { AppController } from './app.controller';
import * as Joi from 'joi';

@Module({
  imports: [
    ConfigModule.forRoot({
      isGlobal: true,
      envFilePath: ['.env'],
      load: [configuration],
      validationSchema: Joi.object({
        APP_NAME: Joi.string().default('NestApp'),
        APP_PORT: Joi.number().default(3000),
        DATABASE_URL: Joi.string().required(),
        JWT_SECRET: Joi.string().required(),
      }),
    }),
  ],
  controllers: [AppController],
  providers: [AppService],
})
export class AppModule {}
// config/configuration.ts
export default () => ({
  appName: process.env.APP_NAME,
  port: parseInt(process.env.APP_PORT, 10),
  database: {
    url: process.env.DATABASE_URL,
  },
```

```
  jwtSecret: process.env.JWT_SECRET,
});
// app.service.ts
import { Injectable } from '@nestjs/common';
import { ConfigService } from '@nestjs/config';

@Injectable()
export class AppService {
  constructor(private config: ConfigService) {}

  getInfo() {
    return {
      name: this.config.get<string>('appName'),
      port: this.config.get<number>('port'),
      db: this.config.get<string>('database.url'),
    };
  }
}
// app.controller.ts
import { Controller, Get } from '@nestjs/common';
import { AppService } from './app.service';

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

  @Get('info')
  getInfo() {
    return this.appService.getInfo();
  }
}
# .env
APP_NAME=MyNestApp
APP_PORT=4000
```

```
DATABASE_URL=postgres://postgres:1234@localhost:5432/mydb
JWT_SECRET=my_jwt_secret
```

ผลการรัน

เมื่อเรียก GET /info

จะได้:

```
{
  "name": "MyNestApp",
  "port": 4000,
  "db": "postgres://postgres:1234@localhost:5432/mydb"
}
```

สรุปเชิงสถาปัตยกรรม

ฟีเจอร์	ประโยชน์
ConfigModule	จัดการ environment variables อย่างเป็นระบบ
ConfigService	เข้าถึง config ผ่าน DI (Dependency Injection)
load	รวม config หลายไฟล์เป็น module-based
validationSchema	ตรวจสอบค่า config ก่อน runtime
envFilePath	รองรับ multi-environment setup
isGlobal	ใช้ config ได้ในทุกโมดูลโดยไม่ต้อง import ซ้ำ

---

10 โปรแกรมเต็ม (ครบโครงสร้าง + อธิบาย + ผลการรัน)

แบ่งเป็น:

- **5 โปรแกรมพื้นฐาน** ⟶ เน้นเข้าใจการใช้ ConfigModule
- **5 โปรแกรมแนวประยุกต์** ⟶ ใช้ config ในระบบจริง เช่น DB, JWT, API Key, Cloud Service

---

หมวดที่ 1: โปรแกรมพื้นฐาน (Basic Level)

โปรแกรมที่ 1: แสดงค่าจาก .env ผ่าน ConfigService

โครงสร้างโปรเจกต์

```
config-demo-basic1/
├── src/
```

```
│   ├── app.controller.ts
│   ├── app.module.ts
│   ├── app.service.ts
│   └── main.ts
├── .env
├── package.json
└── tsconfig.json
```

.env

```
APP_NAME=NestConfigDemo
APP_PORT=3000
```

app.module.ts

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

@Module({
  imports: [ConfigModule.forRoot({ isGlobal: true })],
  controllers: [AppController],
  providers: [AppService],
})
export class AppModule {}
```

app.service.ts

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

@Injectable()
export class AppService {
  constructor(private config: ConfigService) {}

  getConfig() {
    return {
      appName: this.config.get('APP_NAME'),
      port: this.config.get('APP_PORT'),
```

```
    };
  }
}
```

app.controller.ts

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

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

  @Get()
  getEnvValues() {
    return this.appService.getConfig();
  }
}
```

⚡ ผลการรัน

```
GET http://localhost:3000
ผลลัพธ์:
{
  "appName": "NestConfigDemo",
  "port": "3000"
}
```

โปรแกรมที่ 2: ใช้ Default Values เมื่อไม่มีใน .env

.env

```
APP_NAME=MyApp
```

app.service.ts

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

@Injectable()
export class AppService {
  constructor(private config: ConfigService) {}
```

```
  getPort() {
    const port = this.config.get<number>('APP_PORT', 8080);
    return { port };
  }
}
```

⚡ ผลการรัน

```
{ "port": 8080 }
```

☐ Default value ถูกใช้เพราะไม่มี APP_PORT ใน .env

☐ โปรแกรมที่ 3: ใช้ Configuration Function (แยก config ออกเป็นไฟล์)

☐ src/config/configuration.ts

```
export default () => ({
  app: {
    name: process.env.APP_NAME || 'DefaultApp',
    port: parseInt(process.env.APP_PORT, 10) || 3000,
  },
});
```

☐ app.module.ts

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

@Module({
  imports: [ConfigModule.forRoot({ isGlobal: true, load: [configuration] })],
  controllers: [AppController],
  providers: [AppService],
})
export class AppModule {}
```

☐ app.service.ts

```
@Injectable()
export class AppService {
```

```
  constructor(private config: ConfigService) {}

  getAppInfo() {
    return {
      name: this.config.get('app.name'),
      port: this.config.get('app.port'),
    };
  }
}
```

⚡ ผลลัพธ์

```
{ "name": "DefaultApp", "port": 3000 }
```

☐ โปรแกรมที่ 4: ใช้ Validation Schema (Joi)

☐ ติดตั้ง

```
npm install joi
```

☐ app.module.ts

```
import * as Joi from 'joi';
ConfigModule.forRoot({
  isGlobal: true,
  validationSchema: Joi.object({
    APP_PORT: Joi.number().default(3000),
    APP_NAME: Joi.string().required(),
  }),
});
```

ถ้า .env ไม่มี APP_NAME ⟶ ระบบจะ throw error:

```
Error: "APP_NAME" is required
```

☐ โปรแกรมที่ 5: แยก environment (development / production)

☐ app.module.ts

```
ConfigModule.forRoot({
  envFilePath: [`.env.${process.env.NODE_ENV || 'development'}`],
});
```

.env.development

```
APP_ENV=dev
```

APP_PORT=3000

.env.production

APP_ENV=prod

APP_PORT=8080

⚡ ผลลัพธ์

NODE_ENV=production npm run start

ผลที่ได้:

{ "APP_ENV": "prod", "APP_PORT": "8080" }

☐ หมวดที่ 2: โปรแกรมแนวประยุกต์ (Applied Level)

☐ โปรแกรมที่ 6: Database Config + TypeORM

☐ .env

DB_HOST=localhost

DB_PORT=5432

DB_USER=postgres

DB_PASS=123456

DB_NAME=mydb

☐ app.module.ts

```
import { TypeOrmModule } from '@nestjs/typeorm';
import { ConfigModule, ConfigService } from '@nestjs/config';

@Module({
  imports: [
    ConfigModule.forRoot({ isGlobal: true }),
    TypeOrmModule.forRootAsync({
      imports: [ConfigModule],
      inject: [ConfigService],
      useFactory: (config: ConfigService) => ({
        type: 'postgres',
        host: config.get('DB_HOST'),
        port: config.get<number>('DB_PORT'),
        username: config.get('DB_USER'),
        password: config.get('DB_PASS'),
```

```
        database: config.get('DB_NAME'),
        autoLoadEntities: true,
        synchronize: true,
      }),
    }),
  ],
})
export class AppModule {}
```

⚡ ผลลัพธ์

NestJS เชื่อมต่อฐานข้อมูลสำเร็จโดยใช้ค่าจาก .env

□ โปรแกรมที่ 7: JWT Secret Config

□ .env

```
JWT_SECRET=mySuperSecretKey
JWT_EXPIRE=3600
```

□ auth.module.ts

```
JwtModule.registerAsync({
  imports: [ConfigModule],
  useFactory: (config: ConfigService) => ({
    secret: config.get('JWT_SECRET'),
    signOptions: { expiresIn: `${config.get('JWT_EXPIRE')}s` },
  }),
  inject: [ConfigService],
});
```

⚡ ผลการรัน

เมื่อสร้าง token ⟶ จะใช้ secret key จาก .env

JWT_SECRET=mySuperSecretKey

□ โปรแกรมที่ 8: API Keys Config (เชื่อมต่อ External API)

.env

```
OPENAI_API_KEY=sk-12345
```

app.service.ts

```
@Injectable()
export class AppService {
```

```
  constructor(private config: ConfigService) {}

  getApiKey() {
    return `Using API key: ${this.config.get('OPENAI_API_KEY').slice(0, 5)}*****`;
  }
}
```

⚡ ผลการรัน

```
{ "apiKey": "Using API key: sk-12*****" }
```

☐ โปรแกรมที่ 9: Multi-Config Files (Auth + Database)

☐ src/config/database.config.ts

```
export default () => ({
  db: {
    host: process.env.DB_HOST,
    port: parseInt(process.env.DB_PORT, 10),
  },
});
```

☐ src/config/auth.config.ts

```
export default () => ({
  jwt: {
    secret: process.env.JWT_SECRET,
    expiresIn: process.env.JWT_EXPIRE,
  },
});
```

☐ app.module.ts

```
ConfigModule.forRoot({
  isGlobal: true,
  load: [databaseConfig, authConfig],
});
```

☐ โปรแกรมที่ 10: ใช้ Secrets จาก Environment จริง (ไม่เก็บในไฟล์)

ใน production:

```
export JWT_SECRET=$(aws secretsmanager get-secret-value --secret-id myAppSecret)
npm run start:prod
```

ในโค้ด NestJS:

```
const secret = this.config.get('JWT_SECRET');
```

⚡ ผลลัพธ์

NestJS ใช้ secret จาก AWS Secrets Manager โดยไม่ต้องมี .env จริงในเครื่อง

☐ สรุปภาพรวม

หมวด	โปรแกรม	แนวคิดหลัก	จุดเด่น
พื้นฐาน	1	อ่านค่า .env	เข้าใจการ Inject
พื้นฐาน	2	Default Values	ปลอดภัยเมื่อค่าหาย
พื้นฐาน	3	Configuration File	แยก logic config
พื้นฐาน	4	Joi Validation	ป้องกัน config ผิด
พื้นฐาน	5	Multi-Environment	รองรับหลาย environment
ประยุกต์	6	Database Config	ใช้กับ TypeORM
ประยุกต์	7	JWT Secret	ใช้กับ Auth
ประยุกต์	8	API Key	ใช้ External API
ประยุกต์	9	Multi Config Modules	Modular Design
ประยุกต์	10	Secrets Manager	Production Security

บทที่ 11 (ตอนที่ 2): การสร้างไฟล์ .env และ .env.production ใน NestJS

☐ 1. ทำไมต้องใช้ไฟล์ .env

ใน NestJS (และ Node.js โดยทั่วไป) การตั้งค่าต่าง ๆ เช่น

- URL ของฐานข้อมูล
- API keys
- JWT secret
- พอร์ตที่แอปทำงาน

ควรเก็บไว้ในไฟล์ .env แทนการเขียน hardcode ในโค้ด เพื่อให้

☐ ปลอดภัย (ไม่ push ความลับขึ้น GitHub)

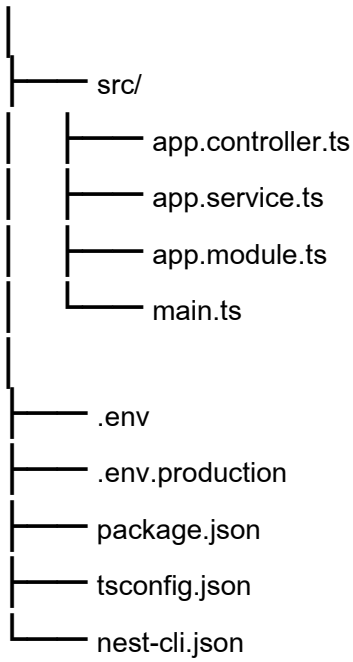
☐ ยืดหยุ่น (เปลี่ยนค่าตามสภาพแวดล้อมได้ง่าย)

☐ แยก configuration ออกจาก logic

---

2. โครงสร้างโปรเจกต์ตัวอย่าง

nestjs-env-demo/



---

3. ติดตั้งแพ็กเกจที่เกี่ยวข้อง

```
npm install @nestjs/config
```

---

4. ตัวอย่างไฟล์ .env (สำหรับ Development)

ไฟล์: .env

```
# แอปทำงานในโหมด development
NODE_ENV=development

# Port ของแอป
APP_PORT=3000

# Database สำหรับ dev
DB_HOST=localhost
DB_PORT=5432
DB_USER=dev_user
DB_PASS=dev_password
DB_NAME=dev_db

# Secret key สำหรับ JWT
```

```
JWT_SECRET=dev-secret-key
```

---

5. ตัวอย่างไฟล์ .env.production (สำหรับ Production)

ไฟล์: .env.production

```
NODE_ENV=production
APP_PORT=8080

# Database ของ production (เช่นบน Cloud)
DB_HOST=prod-db.example.com
DB_PORT=5432
DB_USER=prod_user
DB_PASS=${PROD_DB_PASS}
DB_NAME=prod_db

JWT_SECRET=${PROD_JWT_SECRET}
```

เคล็ดลับ:

ใน production environment ควร **ไม่เก็บค่าจริงในไฟล์ .env.production**
แต่ใช้การแทนที่ด้วย environment variable ของระบบ (เช่นใน Docker หรือระบบ CI/CD)

---

6. โหลดค่าจาก .env ใน NestJS

ไฟล์: app.module.ts

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

@Module({
  imports: [
    ConfigModule.forRoot({
      envFilePath: process.env.NODE_ENV === 'production' ? '.env.production' : '.env',
      isGlobal: true, // ให้ ConfigModule ใช้ได้ทุกที่ในแอป
    }),
  ],
  controllers: [AppController],
```

```
  providers: [AppService],
})
export class AppModule {}
```

☐ คำอธิบาย:

- envFilePath ตรวจสภาพแวดล้อมปัจจุบัน และโหลดไฟล์ .env ที่เหมาะสม
- isGlobal: true ช่วยให้ไม่ต้อง import ConfigModule ในทุกโมดูลย่อย

☐ 7. ใช้งานค่าในโค้ด

☐ **ไฟล์: app.service.ts**

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

@Injectable()
export class AppService {
  constructor(private readonly configService: ConfigService) {}

  getHello(): string {
    const env = this.configService.get<string>('NODE_ENV');
    const port = this.configService.get<number>('APP_PORT');
    const dbHost = this.configService.get<string>('DB_HOST');

    return `Running in ${env} mode on port ${port} | Database Host: ${dbHost}`;
  }
}
```

☐ คำอธิบาย:

- ConfigService ใช้ดึงค่าจาก .env
- TypeScript จะช่วยตรวจสอบประเภท (string, number)
- ใช้ใน runtime เพื่อปรับพฤติกรรมตามสภาพแวดล้อม

☐ 8. แสดงผลใน Controller

☐ **ไฟล์: 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();
  }
}
```

□ 9. จุดเริ่มต้นของแอป

□ **ไฟล์: main.ts**

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

async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  const configService = app.get(ConfigService);
  const port = configService.get<number>('APP_PORT', 3000);

  await app.listen(port);
  console.log(`□ Application running on: http://localhost:${port}`);
}
bootstrap();
```

□ 10. ผลการรัน

□ ***Development Mode (.env)***

```
> npm run start:dev
```

Output:

```
□ Application running on: http://localhost:3000
```

Browser Output:

```
Running in development mode on port 3000 | Database Host: localhost
```

☐ *Production Mode (.env.production)*

NODE_ENV=production npm run start

Output:

☐ Application running on: http://localhost:8080

Browser Output:

Running in production mode on port 8080 | Database Host: prod-db.example.com

---

☐ สรุปแนวคิดสำคัญ

แนวคิด	คำอธิบาย
.env	เก็บค่าคอนฟิกของสภาพแวดล้อม Development
.env.production	เก็บค่าของ Production (แต่ไม่ควรเก็บ secrets จริง)
ConfigModule	โหลดและจัดการค่า environment
ConfigService	ใช้ดึงค่าใน runtime
isGlobal: true	ทำให้ Config ใช้ได้ทุกที่โดยไม่ต้อง import ซ้ำ
process.env.NODE_ENV	ใช้สลับไฟล์ .env ตามสภาพแวดล้อม
Security Tip	ใน production ควรใช้ระบบเช่น Docker secrets, Kubernetes Secrets, หรือ Vault แทนไฟล์ .env

---

- ☐ **3 โปรแกรมพื้นฐาน** (Basic) — สาธิตแนวคิดหลัก
- ☐ **3 โปรแกรมแนวประยุกต์** (Applied) — ใช้ในบริบทจริง เช่น Database, JWT, Third-Party API

แต่ละโปรแกรมจะมีครบ:

1. โครงสร้างโปรเจกต์
2. ไฟล์เต็ม (ทุกไฟล์สำคัญ)
3. คำอธิบายโค้ดทีละส่วน
4. ผลการรันจริง

☐ ส่วนที่ 1 — โปรแกรมพื้นฐาน (Basic)

☐ โปรแกรมที่ 1: อ่านค่า .env แบบง่ายที่สุด

☐ *โครงสร้างโปรเจกต์*

basic-env-1/

```
├── src/
│   ├── app.controller.ts
│   ├── app.service.ts
│   ├── app.module.ts
│   └── main.ts
├── .env
├── package.json
└── tsconfig.json
```

☐ ***ไฟล์ .env***

```
APP_NAME=NestEnvDemo
APP_PORT=3000
```

☐ ***app.module.ts***

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

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

☐ ***app.service.ts***

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

@Injectable()
export class AppService {
  constructor(private readonly configService: ConfigService) {}

  getHello(): string {
    const appName = this.configService.get('APP_NAME');
    const appPort = this.configService.get('APP_PORT');
```

```
    return `App ${appName} is running on port ${appPort}`;
  }
}
```

 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() {
    return this.appService.getHello();
  }
}
```

 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();
```

 ผลการรัน

 Application is running on: http://localhost:3000

เปิดเบราว์เซอร์:

App NestEnvDemo is running on port 3000

 โปรแกรมที่ 2: โหลด .env.production ตาม NODE_ENV

 โครงสร้าง

basic-env-2/

```
├── src/
│   ├── app.module.ts
│   ├── app.service.ts
│   ├── app.controller.ts
│   └── main.ts
├── .env
├── .env.production
```

.env

```
NODE_ENV=development
APP_PORT=3000
```

.env.production

```
NODE_ENV=production
APP_PORT=8080
```

app.module.ts

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

@Module({
  imports: [
    ConfigModule.forRoot({
      envFilePath:
        process.env.NODE_ENV === 'production' ? '.env.production' : '.env',
      isGlobal: true,
    }),
  ],
  controllers: [AppController],
  providers: [AppService],
})
export class AppModule {}
```

app.service.ts

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