NestJS Web Programming: Intermediate

Integrative-Generative AI Edition

Contents:

Student Price Book Center

คำนำ

ในยุคของการพัฒนาเว็บแอปพลิเคชันสมัยใหม่ การสร้างระบบที่มีความปลอดภัย มีประสิทธิภาพ และสามารถรองรับการขยายตัวได้กลายเป็นความท้าทายสำหรับนักพัฒนา NestJS ซึ่งเป็น Framework สำหรับ Node.js ที่ใช้ TypeScript ถูกออกแบบมาเพื่อช่วยแก้ปัญหาเหล่านี้ ด้วยแนวคิด Modular และสถาปัตยกรรมแบบ MVC (Model-View-Controller) รวมถึงการสนับสนุน Dependency Injection และ Decorator ทำให้ผู้พัฒนาสามารถสร้างแอปพลิเคชันที่ซับซ้อนอย่างเป็นระบบและง่ายต่อการบำรุงรักษา

หนังสือเล่มนี้เป็น **เวอร์ชัน Intermediate** ต่อยอดจากความรู้พื้นฐานของ NestJS เหมาะสำหรับผู้ที่มีพื้นฐานการสร้างโปรเจกต์ NestJS แล้ว และต้องการเข้าใจเชิงลึกเกี่ยวกับ Middleware, Pipes, Guards, การเชื่อมต่อฐานข้อมูล และการสร้าง RESTful API แบบเต็มรูปแบบ เนื้อหาจะเน้นไปที่การ **ประยุกต์ใช้งานจริง** พร้อมตัวอย่างโค้ดเต็มรูปแบบ ตั้งแต่ตัวอย่างบูรณาการ (Integrative Examples) ไปจนถึงตัวอย่าง Ultimate ที่รวมฟีเจอร์หลายอย่างเข้าด้วยกัน

บทที่ 6 **Middleware และ Exception Filters** จะพาผู้อ่านทำความเข้าใจการจัดการคำขอและข้อผิดพลาดใน NestJS เริ่มจากการเขียน Middleware สำหรับ **request logging** การใช้งาน **Global Middleware** และการจัดการข้อผิดพลาดด้วย **Exception Filters** รวมถึงการสร้าง **Custom Exception** เช่น UnauthorizedException บทนี้ยังมีตัวอย่างบูรณาการและ Ultimate Examples ที่ช่วยให้เห็นภาพการทำงานแบบครบวงจร

บทที่ 7 **Pipes และ Custom Transformations** เจาะลึกเรื่องการประมวลผลและแปลงข้อมูล โดยครอบคลุม **Built-in Pipes** เช่น ParseIntPipe และ ValidationPipe การสร้าง **Custom Pipe** การใช้ Pipe ในระดับ **Parameter, Controller และ Global** ตลอดจนการจัดการข้อผิดพลาดใน Pipe อีกทั้งยังมีตัวอย่าง Ultimate ที่รวม Middleware Logging และ Auth ทำให้ผู้เรียนสามารถเชื่อมโยงความรู้ระหว่างบทได้อย่างเป็นระบบ

บทที่ 8 **Guards และ Authorization เบื้องต้น** เน้นการจัดการ **Authentication และ Authorization** ผู้เรียนจะได้ทำความเข้าใจกับ **AuthGuard และ RoleGuard**, การสร้าง **Custom Guard**, การใช้ **@UseGuards** กับ Controller และการตรวจสอบ **JWT (JSON Web Token)** บทนี้ประกอบด้วยตัวอย่างบูรณาการ Ultimate และต่อยอด Full Stack Integration Version เพื่อให้ผู้พัฒนาสามารถสร้างระบบที่ปลอดภัยและยืดหยุ่น

บทที่ 9 **การเชื่อมต่อฐานข้อมูลด้วย TypeORM** สอนการจัดการข้อมูลเชิงสัมพันธ์ใน NestJS ครอบคลุมการติดตั้งและเชื่อมต่อ TypeORM กับ **PostgreSQL** หรือ **MySQL**, การสร้าง **Entity และ Repository**, การใช้ **@InjectRepository()** ใน Service และการใช้งาน **QueryBuilder** เพื่อสร้าง query ที่ซับซ้อน ทั้งนี้ยังมีตัวอย่าง Ultimate Integration ที่ช่วยให้ผู้อ่านเห็นภาพการทำงานร่วมกันของฐานข้อมูลและแอปพลิเคชัน

บทที่ 10 **การใช้งาน RESTful API แบบเต็มรูปแบบ** จะรวบรวมความรู้ที่เรียนมาเพื่อสร้าง API ที่สมบูรณ์แบบ ครอบคลุมการสร้าง **CRUD** ผ่าน Controller และ Service, การใช้ **HTTP Status Codes**, การทำ **API Versioning**, การจัดการข้อผิดพลาดที่ส่งคืน API และตัวอย่าง Ultimate ที่รวมฟีเจอร์ทั้งหมด การเรียนรู้บทนี้จะช่วยให้ผู้พัฒนาสามารถออกแบบ API ที่ปลอดภัย, สะอาด และพร้อมใช้งานจริงในโปรเจกต์ระดับมืออาชีพ

หนังสือเล่มนี้ถูกออกแบบให้ผู้เรียนสามารถ **ลงมือทำได้ทันที** ผ่านตัวอย่างโค้ดเต็มรูปแบบ และมีคำอธิบายเชิงลึกสำหรับแต่ละฟีเจอร์ ไม่เพียงแต่ช่วยให้เข้าใจการทำงานของ NestJS แต่ยังเตรียมผู้พัฒนาสำหรับการสร้างระบบขนาดใหญ่ที่ซับซ้อนในอนาคต บทเรียนถูกจัดเรียงอย่างเป็นลำดับขั้นตอน ทำให้สามารถเรียนรู้จากพื้นฐานจนถึงการประยุกต์ใช้งานขั้นสูง

ท้ายที่สุด หนังสือเล่มนี้หวังว่าจะเป็น **คู่มือเชิงปฏิบัติและเชิงลึก** สำหรับผู้ที่ต้องการพัฒนาทักษะ NestJS อย่างมั่นคงและยั่งยืน ไม่ว่าจะเป็นนักพัฒนาที่ทำงานในองค์กร ทีมสตาร์ทอัพ หรือผู้ที่สนใจสร้างโปรเจกต์ส่วนตัว ความเข้าใจเชิงลึกใน Middleware, Pipes, Guards, TypeORM และ RESTful API จะช่วยให้สามารถสร้างระบบที่เสถียร, ปลอดภัย, และมีคุณภาพสูง

ด้วยรักและปรารถนาดี

ศูนย์หนังสือราคานักเรียน

สารบัญ

หน้า

บทที่ 6

Middleware และ Exception Filters

(Middleware and Exception Filters)

เนื้อหา

บทที่ 6: Middleware และ Exception Filters

ในโลกของการพัฒนาเว็บแอปพลิเคชัน การจัดการคำขอและการตอบสนองอย่างมีประสิทธิภาพถือเป็นสิ่งสำคัญ Middleware จึงเข้ามามีบทบาทสำคัญในการตรวจสอบและประมวลผลคำขอก่อนที่จะถูกส่งต่อไปยัง Controller ของระบบ การเรียนรู้วิธีการเขียน Middleware สำหรับ request logging จะช่วยให้ผู้พัฒนาสามารถติดตามการทำงานของระบบ ตรวจสอบปัญหา และเก็บข้อมูลสถิติของคำขอได้อย่างง่ายดาย

นอกจากการสร้าง Middleware เฉพาะจุดแล้ว NestJS ยังสนับสนุนการใช้งาน **Global Middleware** ซึ่งสามารถนำไปประยุกต์ใช้กับทุกคำขอของแอปพลิเคชันได้ วิธีนี้ช่วยลดความซ้ำซ้อนของโค้ดและทำให้การจัดการระบบมีความเป็นระเบียบมากขึ้น ผู้พัฒนาจึงสามารถกำหนดพฤติกรรมที่ต้องการให้เกิดขึ้นในทุกคำขอได้ เช่น การตรวจสอบ authentication หรือ logging ข้อมูลพื้นฐาน

อีกหนึ่งหัวข้อสำคัญของบทนี้คือ **Exception Filters** ซึ่งเป็นเครื่องมือสำหรับการจัดการ error และ exception ใน NestJS อย่างเป็นระบบ Exception Filters ช่วยให้เราสามารถกำหนดวิธีตอบสนองต่อข้อผิดพลาดต่าง ๆ ได้อย่างชัดเจนและเหมาะสม ไม่ว่าจะเป็นการตอบกลับข้อความ error เฉพาะเจาะจงหรือการทำ logging ของ exception สำหรับการวิเคราะห์ในอนาคต

ผู้พัฒนายังสามารถสร้าง **Custom Exception** ของตนเอง เช่น UnauthorizedException เพื่อจัดการกรณีที่ผู้ใช้งานไม่ผ่านการตรวจสอบสิทธิ์ การสร้าง exception แบบกำหนดเองทำให้โค้ดมีความยืดหยุ่นและสามารถควบคุม flow ของ application ได้ดีขึ้น นอกจากนี้ยังช่วยให้ response ของ API มีความสอดคล้องและอ่านง่ายสำหรับผู้ใช้งาน

บทนี้จะเริ่มจากการอธิบายหลักการทำงานของ Middleware พร้อมตัวอย่างการ implement request logging แบบง่าย จากนั้นจะแนะนำวิธีการกำหนด Global Middleware เพื่อให้เข้าใจข้อดีและข้อควรระวังในการนำไปใช้งานจริง การปูพื้นฐานในส่วนนี้จะช่วยให้ผู้เรียนมีความเข้าใจในลำดับการประมวลผลคำขอและสามารถจัดการ flow ของ application ได้อย่างมีประสิทธิภาพ

ต่อมาเราจะเจาะลึกไปที่ Exception Filters โดยเริ่มจากการใช้ Filters พื้นฐานสำหรับจัดการ HTTP exception ก่อนที่จะขยายไปสู่การสร้าง Custom Exception ของเราเอง การใช้ Filters อย่างถูกวิธีจะช่วยให้ระบบมีความเสถียร สามารถตอบสนองต่อความผิดพลาดได้อย่างเหมาะสม และช่วยให้ debugging ง่ายขึ้น

ท้ายที่สุด บทนี้จะเน้นให้ผู้เรียนเห็นความสำคัญของการผสมผสาน Middleware และ Exception Filters เข้าด้วยกัน การทำงานร่วมกันของทั้งสองส่วนนี้ช่วยให้แอปพลิเคชันมีความปลอดภัย มีความเสถียร และสามารถตรวจสอบการทำงานได้ครบถ้วน การเข้าใจหลักการเหล่านี้ไม่เพียงแต่ช่วยให้เขียนโค้ดได้ดีขึ้น แต่ยังเตรียมผู้พัฒนาให้พร้อมสำหรับการสร้างระบบขนาดใหญ่ที่ซับซ้อนในอนาคต

Middleware และ Exception Filters

- การเขียน Middleware สำหรับ request logging
- การใช้งาน Global Middleware
- การจัดการ Error ด้วย Exception Filter
- Custom Exception (เช่น UnauthorizedException)

บทที่ 6 — Middleware และ Exception Filters (อธิบายละเอียด)

ต่อไปนี้เป็นคำอธิบายเชิงลึก ทั้งแนวคิด ลำดับการทำงาน ตัวอย่างโค้ด (TypeScript/NestJS) วิธีลงทะเบียน และแนวปฏิบัติที่ดี — ครอบคลุมหัวข้อที่คุณขอ: *การเขียน Middleware สำหรับ request logging, การใช้งาน Global Middleware, การจัดการ Error ด้วย Exception Filter, และการสร้าง/ใช้ Custom Exception (เช่น UnauthorizedException)*

---

1) ภาพรวม (สั้น ๆ แต่สำคัญ)

- **Middleware**: ฟังก์ชัน (หรือคลาส) ที่รัน *ก่อน* เข้าไปถึง pipeline ของ Nest (ก่อน Guards / Interceptors / Pipes / Controller) — เหมาะกับงานระดับ low-level เช่น logging, parsing, CORS, static files, rate limiting ฯลฯ

- **Exception Filter**: เมคานิซึมของ Nest สำหรับจัดการข้อผิดพลาดที่เกิดขึ้นภายใน *context ของ Nest* (HTTP / WebSocket / RPC) — ใช้จับและแปลง exception ให้ออกเป็น response ที่ต้องการ (เช่น JSON error format)
- **ลำดับการทำงาน (request lifecycle — ย่อ)**:

  Middleware ⟶ Guards ⟶ Interceptors (before) ⟶ Pipes ⟶ Controller ⟶ Service ⟶ Interceptors (after) ⟶ Response

  ถาัมี exception เกิดขึ้นในระหว่าง pipeline ภายใน Nest (เช่นใน Controller/Service) ⟶ Exception Filters จะถูกเรียกเพื่อจัดการ

ข้อสำคัญ: ถ้า exception ถูกโยนจาก *middleware* (เพราะ middleware ทำงานก่อน Nest lifecycle) Nest's Exception Filters อาจจะไม่จับได้ — ต้องจัดการ error ภายใน middleware เองหรือส่งต่อด้วย next(err) (express) / ใช้ error handler ของ underlying adapter

---

2) Middleware — แนวคิด + ตัวอย่าง (request logging)

รูปแบบสองแบบ

1. **Class-based middleware** (implements NestMiddleware) — สามารถลงทะเบียนโดย consumer.apply(...).forRoutes(...)
2. **Functional middleware** (ฟังก์ชันแบบ Express) — ลงทะเบียนด้วย app.use(...) (เป็น global บน Express adapter)

ตัวอย่าง — Class-based LoggerMiddleware

```
// src/common/middleware/logger.middleware.ts
import { Injectable, NestMiddleware, Logger } from '@nestjs/common';
import { Request, Response, NextFunction } from 'express';

@Injectable()
export class LoggerMiddleware implements NestMiddleware {
  private logger = new Logger('HTTP');

  use(req: Request, res: Response, next: NextFunction) {
    const { method, originalUrl } = req;
    const start = Date.now();

    // เมื่อ response จบ เราจะ log เวลา และ status
    res.on('finish', () => {
      const ms = Date.now() - start;
```

```
      this.logger.log(`${method} ${originalUrl} ${res.statusCode} - ${ms}ms`);
    });

    next();
  }
}
```

ลงทะเบียนแบบ Module-scoped (เฉพาะ AppModule)

```
// src/app.module.ts
import { Module, NestModule, MiddlewareConsumer, RequestMethod } from '@nestjs/common';
import { LoggerMiddleware } from './common/middleware/logger.middleware';
import { AppController } from './app.controller';
import { AppService } from './app.service';

@Module({
  controllers: [AppController],
  providers: [AppService],
})
export class AppModule implements NestModule {
  configure(consumer: MiddlewareConsumer) {
    consumer
      .apply(LoggerMiddleware)
      .forRoutes({ path: '*', method: RequestMethod.ALL }); // ใช้กับทุก route
  }
}
```

Functional middleware (global, ใช้ app.use)

```
// src/main.ts
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';
import { Request, Response, NextFunction } from 'express';

function logger(req: Request, res: Response, next: NextFunction) {
  const start = Date.now();
  res.on('finish', () => {
    console.log(`${req.method} ${req.originalUrl} ${res.statusCode} - ${Date.now() - start}ms`);
```

```
  });
  next();
}

async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  app.use(logger); // global for Express adapter
  await app.listen(3000);
}
bootstrap();
```

ข้อควรระวังเกี่ยวกับ Middleware

- **อย่าโยน (throw) uncaught errors ใน middleware** — ถ้า middleware โยน error, Nest Exception Filters อาจจับไม่ได้ (เพราะมันอยู่นอก pipeline)
- ถ้าต้องทำ async ใน middleware ให้ **จับ error แล้วเรียก next(err)** (สำหรับ Express) เพื่อให้ underlying error handler ทำงาน
- หลีกเลี่ยงงานหนัก (blocking) ใน middleware — ถ้าต้องใช้งานหนัก ให้ย้ายไป background job / queue / worker
- บันทึก requestId หรือ correlation id เพื่อเชื่อม logs ข้าม services

---

3) Global Middleware — วิธีใช้งานและข้อแตกต่าง

มีสองวิธีหลัก:

1. **Platform-level global (app.use)** — app.use(...) ทำงานบน underlying adapter (Express/Fastify). เหมาะเมื่อต้องการ middleware ที่แนบกับ adapter โดยตรง (เช่น helmet, cors, bodyParser)
    - แต่ข้อจำกัด: สำหรับ Fastify ต้องเป็น fastify plugin — app.use อาจไม่ทำงานเหมือน Express
2. **Module-scoped "global-like" via MiddlewareConsumer** — ใช้ consumer.apply(...).forRoutes({ path: '*', method: RequestMethod.ALL }) ซึ่งเทียบเท่ากับ global ในระดับ Nest routing แต่ยังเป็นส่วนหนึ่งของ lifecycle ของ Nest (ใช้กับ NestModule)

ตัวอย่างการทำให้เป็น global สำหรับทุก route (ใน AppModule):

```
consumer.apply(LoggerMiddleware).forRoutes({ path: '*', method: RequestMethod.ALL });
```

---

4) Exception Filters — แนวคิดและการใช้งาน

ทำไมต้องใช้ Exception Filters

- ควบคุมรูปแบบ response เมื่อเกิด error (เช่น JSON format ที่เป็นมาตรฐาน)
- แยกแยะ exception แต่ละชนิด (เช่น HTTP exceptions, validation errors, custom errors)
- ศูนย์กลางการ logging ของข้อผิดพลาด และไม่รั่วไหลข้อมูลที่ sensitive ใน production

สร้าง AllExceptionsFilter (จับทุก exception)

```
// src/common/filters/all-exceptions.filter.ts
import {
  ExceptionFilter,
  Catch,
  ArgumentsHost,
  HttpException,
  HttpStatus,
  Logger,
} from '@nestjs/common';
import { Request, Response } from 'express';

@Catch()
export class AllExceptionsFilter implements ExceptionFilter {
  private logger = new Logger('Exceptions');

  catch(exception: unknown, host: ArgumentsHost) {
    const ctx = host.switchToHttp();
    const response = ctx.getResponse<Response>();
    const request = ctx.getRequest<Request>();

    const status =
      exception instanceof HttpException
        ? exception.getStatus()
        : HttpStatus.INTERNAL_SERVER_ERROR;

    const responseBody =
      exception instanceof HttpException
        ? exception.getResponse()
        : { message: 'Internal server error' };
```

```
    // log ข้อมูล (อย่า log stack-sensitive ใน production แบบเปิดเผย)
    this.logger.error(
      `HTTP ${status} Error on ${request.method} ${request.url} ->
${JSON.stringify(responseBody)}`,
    );

    response.status(status).json({
      statusCode: status,
      timestamp: new Date().toISOString(),
      path: request.url,
      error: responseBody,
    });
  }
}
```

ลงทะเบียน Filter

- **Global** (ทุก controller / route)

```
// src/main.ts
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';
import { AllExceptionsFilter } from './common/filters/all-exceptions.filter';

async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  app.useGlobalFilters(new AllExceptionsFilter());
  await app.listen(3000);
}
bootstrap();
```

- **Controller หรือ Method level**
- @Controller('cats')
- @UseFilters(new AllExceptionsFilter())
- export class CatsController { ... }
-
- // หรือสำหรับ method เดียว
- @Get()

- @UseFilters(new AllExceptionsFilter())
- findAll() { ... }

จับเฉพาะชนิดของ Exception

ถ้าต้องการจับเฉพาะ UnauthorizedException:

```
@Catch(UnauthorizedException)
export class UnauthorizedFilter implements ExceptionFilter {
  catch(exception: UnauthorizedException, host: ArgumentsHost) {
    // ... handle 401 specially
  }
}
```

---

5) Custom Exception (เช่น UnauthorizedException แบบกำหนดเอง)

คุณสามารถสร้าง exception ของคุณเองโดย extend HttpException — ประโยชน์คือใส่ payload/shape ของ error ที่ต้องการได้

```
// src/common/exceptions/my-unauthorized.exception.ts
import { HttpException, HttpStatus } from '@nestjs/common';

export class MyUnauthorizedException extends HttpException {
  constructor(message = 'คุณไม่มีสิทธิ์เข้าถึง') {
    super(
      {
        statusCode: HttpStatus.UNAUTHORIZED,
        error: 'Unauthorized',
        message,
      },
      HttpStatus.UNAUTHORIZED,
    );
  }
}
```

ใช้งาน (จาก Service / Controller):

```
// ที่ไหนสักแห่งใน service
if (!user) {
  throw new MyUnauthorizedException('Token หมดอายุ หรือไม่ถูกต้อง');
}
```

หรือใช้ built-in:

throw new UnauthorizedException('Token invalid');

และถ้าต้องการจัดการ 401 แบบพิเศษ ให้เขียน @Catch(UnauthorizedException) filter

---

6) ตัวอย่างผสาน (Middleware + Filter + Custom Exception)

1. Middleware log ทุก request
2. Service พบ user ไม่ถูกต้อง ⟶ throw new MyUnauthorizedException()
3. Exception Filter (global) รับ exception แล้ว return JSON ที่เป็นมาตรฐานและ log

ตัวอย่าง Controller สั้น ๆ

```
@Controller('api')
export class ApiController {
  @Get('private')
  getPrivate() {
    // สมมติเช็ค token ล้มเหลว
    throw new MyUnauthorizedException('Token ไม่ถูกต้อง');
  }
}
```

ตัวอย่าง response (จาก AllExceptionsFilter)

```
HTTP/1.1 401 Unauthorized
Content-Type: application/json

{
  "statusCode": 401,
  "timestamp": "2025-10-12T01:23:45.678Z",
  "path": "/api/private",
  "error": {
    "statusCode": 401,
    "error": "Unauthorized",
    "message": "Token ไม่ถูกต้อง"
  }
}
```

---

7) Best practices & ข้อแนะนำเชิงปฏิบัติ

- **แยกความรับผิดชอบ**: Middleware สำหรับ concerns ระดับ infrastructure (logging, parsing, auth token parse), Exception Filters สำหรับความผิดพลาดและรูปแบบ error response
- **อย่าโยน exception แบบเปิดเผยใน middleware** — จัดการด้วย try/catch แล้วส่งต่อด้วย next(err) หรือแปลงเป็น response ที่เหมาะสม
- **ใช้ HttpException/built-in exceptions` (BadRequestException, UnauthorizedException, NotFoundException, ฯลฯ)** เพื่อให้มีสถานะ HTTP ที่ชัดเจน
- **Format response ให้เป็นมาตรฐาน** (statusCode, message, timestamp, path, details) — ง่ายต่อ frontend และ logging/observability
- **Log ระดับต่าง ๆ**: info for normal requests, warn for suspicious, error for exception — บันทึก requestId/correlationId เสมอ
- **อย่าเปิดเผย stack traces ใน production** — log stack trace ให้ internal logging (Sentry, Datadog) แต่ตอบ client ด้วยข้อความที่ปลอดภัย
- **ใช้ ValidationPipe สำหรับ DTO validation** — validation errors จะกลายเป็น BadRequestException โดยอัตโนมัติ (สามารถจับด้วย filter)
- **ทดสอบ**: เขียน unit tests สำหรับ middleware (mock req/res/next) และ integration/e2e tests เพื่อยืนยัน format ของ error responses

---

8) ข้อควรระวังเชิงเทคนิค (ที่มักพลาดกัน)

- **Exception filters ไม่จับ exception ที่เกิดนอก Nest context** (เช่น error ใน global express middleware ที่โยนโดยตรง)
- **สำหรับ Fastify** — วิธีการลงทะเบียน middleware ต่างออกไป (ควรใช้ fastify plugins)
- **ถ้า middleware ต้องทำ async และอาจโยน error** — ใช้ try { await ...; next(); } catch(e) { next(e); } เพื่อส่งต่อ error ให้ underlying adapter

---

รายละเอียดเชิงลึก — บทที่ 6: Middleware และ Exception Filters (ครบถ้วน / ลงมือทำได้เลย)

ต่อไปนี้เป็น **คู่มือเชิงลึก** — มีทั้งแนวคิดเชิงสถาปัตยกรรม, โค้ดตัวอย่างแบบไฟล์จริง, การตั้งค่า DI, การทดสอบ, ข้อควรระวัง, และ pattern ที่ใช้จริงใน Production (correlation id, structured logging, integration กับ error monitoring) — ถ้าคุณอยากให้ผมแพ็กเป็นโปรเจกต์ตัวอย่างที่สามารถรันได้ทันที บอกมาได้เลย ผมจะส่งเป็นชุดไฟล์ตัวอย่าง

---

1 — ลำดับการทำงาน (Request lifecycle — แบบละเอียด)

1. **Incoming HTTP request (Adapter)** — Express/Fastify รับ request ก่อน
2. **Platform-level middleware** (app.use(...)) — ทำงานก่อน Nest lifecycle (ผูกกับ underlying adapter)
3. **Nest Module-scoped middleware** (class-based via consumer.apply(...).forRoutes(...)) — ทำก่อน Guards / Interceptors / Pipes / Controllers
4. **Guards** (CanActivate) — ตรวจสิทธิ์ / auth
5. **Interceptors (before)** — transform/monitor input (เช่น timing)
6. **Pipes** — validation/transform (เช่น ValidationPipe)
7. **Controller ⟶ Service (business logic)**
8. **Interceptors (after)** — transform response, caching ฯลฯ
9. **Response returned to client**
10. **Exception handling path** — ถ้าเกิด exception ภายใน Nest pipeline (Controller/Service/Pipes/Interceptors/Guards) ⟶ **Exception Filters** จะจับได้.
    - **ข้อสำคัญ:** ถา้ exception เกิด *ก่อน* Nest pipeline (เช่นใน platform-level middleware ที่โยน error โดยตรง) Nest filter อาจไม่จับได้ — ต้อง next(err) หรือจัดการภายใน middleware เอง

---

2 — Middleware (เชิงลึก + ตัวอย่างไฟล์จริง)

2.1 แบบที่ใช้บ่อย

- **Functional middleware**: ใช้ app.use(fn) (Express style) — ดีสำหรับ third-party libraries (helmet, cors)
- **Class-based NestMiddleware**: สะดวกเมื่อต้องการ DI เข้ามาใช้ (LoggerService, Config) — ลงทะเบียนผ่าน consumer.apply(...).forRoutes(...)

2.2 ตัวอย่าง: Logger + Correlation ID (production-ready pattern)

วัตถุประสงค์:

- สร้าง x-correlation-id (รับจาก header ถ้ามี) — ผูกกับ request lifecycle โดยใช้ AsyncLocalStorage เพื่อให้ทุกที่ใน code (service/logger/filter) อ่าน traceId ได้
- สร้าง structured log (Winston) — log มี traceId เพื่อ trace across services

โครงสร้างตัวอย่าง (ย่อ)

```
src/
  main.ts
  app.module.ts
  app.controller.ts
```

```
app.service.ts
common/
  async-storage.ts
  logger/
    logger.service.ts
  middleware/
    correlation.middleware.ts
    logger.middleware.ts
  filters/
    all-exceptions.filter.ts
  exceptions/
    my-unauthorized.exception.ts
```

1) async-storage.ts

```
// src/common/async-storage.ts
import { AsyncLocalStorage } from 'async_hooks';
export const asyncLocalStorage = new AsyncLocalStorage<Map<string, any>>();
```

2) logger.service.ts (Winston-based)

```
// src/common/logger/logger.service.ts
import { Injectable } from '@nestjs/common';
import * as winston from 'winston';
import { format } from 'winston';

@Injectable()
export class LoggerService {
  private logger: winston.Logger;
  constructor() {
    this.logger = winston.createLogger({
      level: 'info',
      format: format.combine(
        format.timestamp(),
        format.errors({ stack: true }),
        format.json()
      ),
      transports: [
```

```
        new winston.transports.Console(),
      ],
    });
  }

  log(message: string, meta?: any) { this.logger.info(message, meta); }
  error(message: string, trace?: string, meta?: any) { this.logger.error(message, { trace, ...meta
}); }
  warn(message: string, meta?: any) { this.logger.warn(message, meta); }
}
```

3) correlation.middleware.ts

```
// src/common/middleware/correlation.middleware.ts
import { Injectable, NestMiddleware } from '@nestjs/common';
import { Request, Response, NextFunction } from 'express';
import { randomUUID } from 'crypto';
import { asyncLocalStorage } from '../async-storage';

@Injectable()
export class CorrelationMiddleware implements NestMiddleware {
  use(req: Request, res: Response, next: NextFunction) {
    const headerId = (req.headers['x-correlation-id'] as string) || randomUUID();
    const store = new Map<string, any>();
    store.set('traceId', headerId);
    // run everything for this request inside AsyncLocalStorage context
    asyncLocalStorage.run(store, () => {
      res.setHeader('x-correlation-id', headerId);
      next();
    });
  }
}
```

4) logger.middleware.ts

```
// src/common/middleware/logger.middleware.ts
import { Injectable, NestMiddleware } from '@nestjs/common';
import { Request, Response, NextFunction } from 'express';
```

```
import { LoggerService } from '../logger/logger.service';
import { asyncLocalStorage } from '../async-storage';

@Injectable()
export class LoggerMiddleware implements NestMiddleware {
  constructor(private readonly logger: LoggerService) {}
  use(req: Request, res: Response, next: NextFunction) {
    const start = Date.now();
    res.on('finish', () => {
      const store = asyncLocalStorage.getStore();
      const traceId = store?.get('traceId');
      const meta = {
        traceId,
        method: req.method,
        url: req.originalUrl,
        status: res.statusCode,
        durationMs: Date.now() - start,
        ip: req.ip,
      };
      this.logger.log(`${req.method} ${req.originalUrl} ${res.statusCode} - ${meta.durationMs}ms`,
meta);
    });
    next();
  }
}
```

5) ลงทะเบียน middleware ใน AppModule

```
// src/app.module.ts (เฉพาะส่วน configure)
import { Module, NestModule, MiddlewareConsumer, RequestMethod } from '@nestjs/common';
import { AppController } from './app.controller';
import { AppService } from './app.service';
import { LoggerService } from './common/logger/logger.service';
import { LoggerMiddleware } from './common/middleware/logger.middleware';
import { CorrelationMiddleware } from './common/middleware/correlation.middleware';
```

```
@Module({
  controllers: [AppController],
  providers: [AppService, LoggerService, LoggerMiddleware, CorrelationMiddleware],
})
export class AppModule implements NestModule {
  configure(consumer: MiddlewareConsumer) {
    consumer.apply(CorrelationMiddleware, LoggerMiddleware)
      .forRoutes({ path: '*', method: RequestMethod.ALL });
  }
}
```

คำอธิบายเชิงลึก:

- asyncLocalStorage.run(...) ทำให้ข้อมูล traceId ติดตามตลอด async call chain (ดีเมื่อใช้กับ logging และ tracing)
- ให้ middleware ทั้งสองเป็น provider (ใน providers) เพื่อให้ DI ทำงาน (LoggerMiddleware ต้องการ LoggerService)
- หลีกเลี่ยง synchronous heavy work ภายใน middleware — ถ้าต้องทำ heavy IO ให้ทำแบบ async และจัดการ error ด้วย try/catch แล้ว next(err)

---

3 — Exception Filters (เชิงลึก + ตัวอย่างไฟล์จริง)

3.1 เป้าหมายของ Exception Filter

- **แปลง exception ⟶ HTTP response** ในรูปแบบที่กำหนด (standard error shape)
- **แยกการจัดการ error แต่ละชนิด** (HttpException, ValidationError, Custom Exceptions)
- **บันทึก (log) และส่งไปยัง monitoring** (เช่น Sentry) จากศูนย์กลาง

3.2 ตัวอย่าง: AllExceptionsFilter (structured response + logger + traceId)

```
// src/common/filters/all-exceptions.filter.ts
import { ExceptionFilter, Catch, ArgumentsHost, HttpException, HttpStatus } from
'@nestjs/common';
import { Request, Response } from 'express';
import { LoggerService } from '../logger/logger.service';
import { asyncLocalStorage } from '../async-storage';

@Catch()
export class AllExceptionsFilter implements ExceptionFilter {
  constructor(private readonly logger: LoggerService) {}
```

```
  catch(exception: unknown, host: ArgumentsHost) {
    const ctx = host.switchToHttp();
    const response = ctx.getResponse<Response>();
    const request = ctx.getRequest<Request>();
    const store = asyncLocalStorage.getStore();
    const traceId = store?.get('traceId');

    const isHttp = exception instanceof HttpException;
    const status = isHttp ? exception.getStatus() : HttpStatus.INTERNAL_SERVER_ERROR;

    let message = 'Internal server error';
    let details: any = null;

    if (isHttp) {
      const res = exception.getResponse();
      if (typeof res === 'string') message = res;
      else if (typeof res === 'object' && res !== null) {
        details = res;
        // some HttpExceptions have { message: ['a','b'] } or custom object
        message = (res as any).message || (Array.isArray((res as any).message) ? (res as
any).message.join(', ') : message);
      }
    } else if (exception instanceof Error) {
      message = exception.message;
      details = { stack: exception.stack }; // careful: don't send stack in prod
    }

    const body = {
      statusCode: status,
      timestamp: new Date().toISOString(),
      path: request.url,
      traceId,
      message,
```

```
      details,
    };

    // log (stack or full exception for internal monitoring)
    this.logger.error(`HTTP ${status} ${request.method} ${request.url} - ${message}`, (exception as any)?.stack, { traceId, details });

    // optionally send to Sentry / APM here
    // Sentry.captureException(exception);

    response.status(status).json(body);
  }
}
```

3.3 ลงทะเบียน Filter (global)

```
// src/main.ts (ตัวอย่าง)
async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  const logger = app.get(LoggerService);
  app.useGlobalFilters(new AllExceptionsFilter(logger));
  await app.listen(3000);
}
```

หมายเหตุ: useGlobalFilters ต้องการ instance — ถ้าอยากให้ filter ได้รับ DI โดยตรง ให้สร้าง provider ที่ wrapper หรือเรียก app.get(...) เพื่อดึง dependency มาใส่

3.4 จับเฉพาะชนิดของ Exception

ถ้าต้องการจับ UnauthorizedException แบบเฉพาะ:

```
@Catch(UnauthorizedException)
export class UnauthorizedFilter implements ExceptionFilter {
  catch(exception: UnauthorizedException, host: ArgumentsHost) {
    // build custom 401 response
  }
}
```

หรือจับหลายชนิดด้วย @Catch(A, B, C)

3.5 การจัดการ Validation errors

- ValidationPipe (ของ Nest) จะโยน BadRequestException โดยมี response มีรายละเอียด validation error
- ใน filter ให้ตรวจว่า exception.getResponse() มี message เป็น array หรือ object แล้วทำ normalize ให้ frontend รับง่าย

---

4 — Custom Exceptions (pattern & ตัวอย่าง)

คุณสามารถ extend HttpException เพื่อสร้าง domain-specific exceptions — ทำให้โค้ดอ่านง่าย และ response shape สม่ำเสมอ

ตัวอย่าง: MyUnauthorizedException

```
// src/common/exceptions/my-unauthorized.exception.ts
import { HttpException, HttpStatus } from '@nestjs/common';

export class MyUnauthorizedException extends HttpException {
  constructor(message = 'คุณไม่มีสิทธิ์เข้าถึง') {
    super(
      {
        statusCode: HttpStatus.UNAUTHORIZED,
        error: 'Unauthorized',
        message,
      },
      HttpStatus.UNAUTHORIZED,
    );
  }
}
```

ใช้ใน Service:

```
if (!user) throw new MyUnauthorizedException('Token หมดอายุ หรือไม่ถูกต้อง');
```

ข้อดี:

- จำนวน code ใน controller/service อ่านง่าย
- Filter สามารถตรวจ type ของ exception แล้วจัดการเฉพาะได้

---

5 — Interplay: Middleware vs Guards vs Interceptors vs Filters (สรุปเชิงปฏิบัติ)

- **Middleware**: low-level, pre-routing, ไม่เหมาะกับการพูดถึง business logic มาก — เหมาะกับ logging, correlation, request body parsing (adapter)
- **Guards**: ใช้สำหรับ auth/roles — ถูกเรียกหลัง middleware แต่ก่อน controller

- **Interceptors**: สำหรับ response shaping, caching, timing, transforming payloads — เรียกก่อน/หลัง controller
- **Filters**: จัดการข้อผิดพลาด — ถูกเรียกเมื่อ exception ถูกโยนภายใน Nest pipeline

Order (สั้น ๆ): Middleware ⟶ Guards ⟶ Interceptors (before) ⟶ Pipes ⟶ Controller/Service ⟶ Interceptors (after) ⟶ Response ⟶ Filters (on exception)

---

6 — Edge cases & ข้อควรระวัง (สำคัญ)

1. **Error thrown in platform-level middleware (app.use(...))** — Nest filter อาจไม่จับได้ ⟶ ต้อง next(err) เพื่อส่งให้ underlying express error handler หรือจัดการภายใน middleware
2. **Fastify differences** — ไม่สามารถใช้ Express middleware ทุกตัวได้ — ต้อง register fastify plugins หรือใช้ adapter-specific techniques (ระวัง)
3. **อย่าส่ง stack trace กลับ client ใน production** — log ไว้ internal (Winston ⟶ file / external service) แต่ตอบ client ด้วยข้อความที่ปลอดภัย
4. **Long-running/blocking work** — หลีกเลี่ยงใน middleware, ทำผ่าน queue (Bull + Redis) หรือ background worker
5. **AsyncLocalStorage**: ดีมากสำหรับ correlation id/tracing แต่ต้องระวัง library ที่ไม่อนุรักษ์ context อาจทำให้ store หายได้ (thrift, native modules บางตัว)
6. **Testing**: ทดสอบ middleware/filters ทั้งใน unit และ e2e (supertest) เพื่อยืนยันว่า response shape ถูกต้อง

---

7 — ตัวอย่างการทดสอบ (Jest + supertest)

7.1 Unit test สำหรับ LoggerMiddleware (Jest)

```
// test/logger.middleware.spec.ts
import { LoggerMiddleware } from '../src/common/middleware/logger.middleware';
import { LoggerService } from '../src/common/logger/logger.service';

describe('LoggerMiddleware', () => {
  let middleware: LoggerMiddleware;
  let logger: LoggerService;

  beforeEach(() => {
    logger = { log: jest.fn(), error: jest.fn(), warn: jest.fn() } as unknown as LoggerService;
    middleware = new LoggerMiddleware(logger);
```

```
  });

  it('calls next and logs on finish', (done) => {
    const req: any = { method: 'GET', originalUrl: '/test', ip: '127.0.0.1' };
    const res: any = {
      statusCode: 200,
      on: (ev: string, cb: Function) => { if (ev === 'finish') setImmediate(cb); },
    };
    const next = jest.fn();

    middleware.use(req, res, next);

    setImmediate(() => {
      expect(next).toHaveBeenCalled();
      expect(logger.log).toHaveBeenCalled();
      done();
    });
  });
});
```

7.2 E2E test (supertest) — ตรวจ response shape ของ exception

```
// test/app.e2e-spec.ts
import * as request from 'supertest';
import { Test } from '@nestjs/testing';
import { AppModule } from '../src/app.module';
import { AllExceptionsFilter } from '../src/common/filters/all-exceptions.filter';
import { LoggerService } from '../src/common/logger/logger.service';

describe('App (e2e)', () => {
  let app;
  beforeAll(async () => {
    const moduleRef = await Test.createTestingModule({ imports: [AppModule] }).compile();
    app = moduleRef.createNestApplication();
    const logger = moduleRef.get(LoggerService);
    app.useGlobalFilters(new AllExceptionsFilter(logger));
```

```
    await app.init();
  });

  it('/private -> 401 structured error', async () => {
    await request(app.getHttpServer())
      .get('/private')
      .expect(401)
      .expect(res => {
        expect(res.body).toHaveProperty('statusCode', 401);
        expect(res.body).toHaveProperty('traceId');
        expect(res.body).toHaveProperty('message');
      });
  });

  afterAll(async () => app.close());
});
```

---

8 — Integration กับ Monitoring / APM / Error tracking

- **Sentry**: ใน filter catch() ให้ Sentry.captureException(exception) พร้อม tags (env, traceId)
- **APM (Datadog, Elastic)**: ส่ง timing + error metrics จาก interceptors/filters
- **Structured logging**: ส่ง logs แบบ JSON (Winston) เพื่อให้ log shipping ⟶ ELK / Splunk / Datadog ง่าย

ตัวอย่าง (ใน filter):

```
// pseudo: inside AllExceptionsFilter.catch
Sentry.withScope(scope => {
  scope.setTag('traceId', traceId);
  scope.setExtra('request', {method: request.method, url: request.url});
  Sentry.captureException(exception);
});
```

---

9 — Tips / Best Practices สั้น ๆ (เชิงปฏิบัติ)

- **Consistency**: กำหนด standard error shape ให้ frontend / mobile ใช้งานได้ทันที
- **CorrelationId**: ทำให้ทุก log & error มี traceId เพื่อ trace across logs / services

- **Log levels**: info (normal), warn (suspicious), error (exception) — อย่าล้น logs ด้วย debug ใน production
- **Avoid throwing in middleware**: ถาจำเป็น จัดการ next(err) หรือแปลงเป็น response
- **Use ValidationPipe globally**: app.useGlobalPipes(new ValidationPipe({ whitelist: true, transform: true })) — validation errors จะกลายเป็น BadRequestException ⟶ filter จะ normalize
- **Tests**: ตรวจทั้ง unit (middleware/filter) และ e2e (response shapes)
- **Document error contract**: บอก frontend/clients ว่า response error จะมี fields ใดบ้าง (statusCode, message, traceId, details)

---

10 — Common patterns & extensions (ที่ใช้กันจริง)

- **Correlation + Logger + Trace exporter** (x-correlation-id ⟶ AsyncLocalStorage ⟶ log + APM)
- **Custom HttpException base class**: ถ้าต้องการ payload เฉพาะ ให้สร้าง BaseHttpException แล้ว extend ⟶ filter ตรวจ type และแปลงเฉพาะ
- **Error Codes**: ส่ง code หรือ errorKey ใน response เพื่อให้ frontend mapping ข้อความได้ง่าย (i18n)
- **Centralized error mapping**: map DB constraint errors ⟶ ConflictException (409) ใน filter/service layer

---

11 — Quick checklist ก่อน deploy

- ☐ ใช้ global ValidationPipe
- ☐ ใช้ structured logger (Console+File) และ log rotation
- ☐ ไม่ส่ง stack trace ใน production responses
- ☐ มี global exception filter (standard response)
- ☐ Inject traceId ในทุก request (CorrelationMiddleware)
- ☐ เชื่อมกับ error tracking (Sentry) และ metrics (Prometheus/Datadog)
- ☐ เขียน e2e test เพื่อตรวจ error contract

การเขียน Middleware สำหรับ Request Logging

ใน NestJS — ครอบคลุมทั้ง **แนวคิด**, **ลำดับการทำงาน**, **รูปแบบการเขียน (Class-based / Function-based)**, **การลงทะเบียน (Scoped / Global)**, **ตัวอย่างโค้ดแบบเต็มไฟล์**, **Best Practices**, และ **ข้อควรระวังเชิงเทคนิค**

□ 1. แนวคิดของ Middleware ใน NestJS (เฉพาะด้าน Request Logging)

Middleware คือฟังก์ชันที่ถูกเรียก *ก่อน* ที่ request จะเข้าสู่ NestJS Request Lifecycle (เช่น Guards ⟶ Interceptors ⟶ Pipes ⟶ Controller ⟶ Service)

⟶ จึงเหมาะมากสำหรับงาน **logging** เช่น

- บันทึก HTTP method, URL, status code, response time
- เก็บข้อมูลสำหรับ observability / monitoring
- ใส่ Request ID / Correlation ID
- ตรวจสอบข้อมูลเบื้องต้น (Pre-processing)

ลำดับเหตุการณ์ ของ request หนึ่งครั้งใน NestJS (HTTP):

Incoming Request

↓

Middleware (เช่น LoggerMiddleware)

↓

Guards

↓

Interceptors (Before)

↓

Pipes

↓

Controller

↓

Service

↓

Interceptors (After)

↓

Response Sent

□ ดังนั้น Logger Middleware สามารถวัดเวลาที่ request ใช้ทั้งหมดได้อย่างแม่นยำ และยังไม่ขึ้นกับ logic ภายใน Controller

□ 2. รูปแบบการเขียน Middleware

NestJS รองรับ 2 รูปแบบ:

1. **Class-based Middleware**
    - เป็นรูปแบบ OOP

- Implement NestMiddleware interface
- เหมาะเมื่อ middleware ต้องมี dependency injection (เช่น LoggerService, ConfigService)
- ลงทะเบียนผ่าน MiddlewareConsumer ใน AppModule หรือ Module อื่น ๆ

2. **Function-based Middleware**
   - เขียนแบบ Express-style (req, res, next)
   - เรียก app.use() ใน main.ts ⟶ กลายเป็น *global middleware*
   - ไม่รองรับ dependency injection โดยตรง
   - เหมาะกับ logging ง่าย ๆ ที่ไม่ต้องพึ่ง Nest DI

---

3. ตัวอย่าง Class-based Logger Middleware (พร้อมคำอธิบายบรรทัดต่อบรรทัด)

โครงสร้างไฟล์

```
src/
├── common/
│   └── middleware/
│       └── logger.middleware.ts
├── app.module.ts
├── main.ts
└── app.controller.ts
```

logger.middleware.ts

```
// src/common/middleware/logger.middleware.ts
import { Injectable, NestMiddleware, Logger } from '@nestjs/common';
import { Request, Response, NextFunction } from 'express';

@Injectable()
export class LoggerMiddleware implements NestMiddleware {
  private readonly logger = new Logger('HTTP'); // สร้าง Logger tag ชื่อ "HTTP"

  use(req: Request, res: Response, next: NextFunction): void {
    const { method, originalUrl } = req;
    const start = Date.now(); // จับเวลาตั้งแต่เริ่มรับ request

    // เมื่อ response ส่งกลับเสร็จแล้ว -> trigger 'finish' event
    res.on('finish', () => {
```

```
      const statusCode = res.statusCode;
      const responseTime = Date.now() - start;

      // รูปแบบ log เช่น: GET /api/users 200 - 42ms
      this.logger.log(`${method} ${originalUrl} ${statusCode} - ${responseTime}ms`);
    });

    next(); // ส่งต่อไปยังขั้นตอนถัดไปใน lifecycle
  }
}
```

☐ จุดสำคัญ:

- res.on('finish', ...) คือจุดที่ response เสร็จสมบูรณ์ ⟶ สามารถ log response time ได้แม่นยำ
- ใช้ Logger ของ NestJS เพื่อ log ⟶ จะได้ log ที่มี timestamp + context พร้อมใน console
- next() สำคัญมาก ถ้าไม่เรียก ⟶ request จะค้างไม่ไปต่อ

☐ **4. การลงทะเบียน Middleware (Module-scoped / Global)**

☐ **Module-scoped (เช่นใช้กับทุก route ของ AppModule)**

```
// src/app.module.ts
import { Module, NestModule, MiddlewareConsumer, RequestMethod } from '@nestjs/common';
import { LoggerMiddleware } from './common/middleware/logger.middleware';
import { AppController } from './app.controller';

@Module({
  controllers: [AppController],
})
export class AppModule implements NestModule {
  configure(consumer: MiddlewareConsumer) {
    consumer
      .apply(LoggerMiddleware) // ใส่ middleware ที่ต้องการใช้
      .forRoutes({ path: '*', method: RequestMethod.ALL }); // ใช้กับทุก route ทุก method
  }
}
```

☐ .forRoutes(...) สามารถกำหนดเฉพาะ controller, path, หรือ method ได้ เช่น:

```
consumer
  .apply(LoggerMiddleware)
  .forRoutes({ path: 'users', method: RequestMethod.GET });
```

☐ 5. การเขียน Logger แบบ Global (Function-based)

ในบางกรณี เราอาจต้องการ logger แบบง่าย ๆ สำหรับทุก request โดยไม่ต้องใช้ DI ⟶ ใช้ app.use() ใน main.ts ได้

☐ **main.ts**

```
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';
import { Request, Response, NextFunction } from 'express';

function globalLogger(req: Request, res: Response, next: NextFunction) {
  const start = Date.now();
  res.on('finish', () => {
    console.log(`${req.method} ${req.originalUrl} ${res.statusCode} - ${Date.now() - start}ms`);
  });
  next();
}

async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  app.use(globalLogger); // Global middleware
  await app.listen(3000);
}
bootstrap();
```

☐ **ข้อสังเกต**

- วิธีนี้เหมาะกับ logging เบื้องต้น
- ไม่รองรับ dependency injection (ไม่สามารถใช้ LoggerService ได้ในฟังก์ชันนี้)
- เหมาะกับ Express Adapter ⟶ ถ้าใช้ Fastify ต้องเขียนเป็น Fastify plugin แทน

☐ 6. ผลการทดสอบ (Sample Output)

ลองรัน server แล้วเรียก API ใด ๆ เช่น

```
curl http://localhost:3000/
```

☐ ผลลัพธ์ใน console (Class-based Logger):

[Nest] 17200 - 10/12/2025, 7:42:15 AM [HTTP] GET / 200 - 3ms

☐ ผลลัพธ์ใน console (Function-based Logger):

GET / 200 - 2ms

☐ **7. Best Practices สำหรับ Request Logging Middleware**

แนวปฏิบัติที่ดี	รายละเอียด
☐ **Log หลัง response**	เพื่อให้ได้ status code และ response time ที่ถูกต้อง (ใช้ res.on('finish', ...))
☐ **ใช้ Nest Logger**	จะได้ log format ที่มี timestamp/context และสามารถ redirect ไป external logger ภายหลังได้
☐ **รองรับ async**	middleware ควรไม่ block, ถ้ามีงาน async เช่นเขียนลง DB ⟶ ใช้ non-blocking method
☐ **เพิ่ม requestId / correlationId**	เพื่อเชื่อมโยง log หลาย ๆ ระบบ เช่น API Gateway ⟶ Microservice
☐ **Log เฉพาะสิ่งจำเป็นใน Production**	อย่า log sensitive data เช่น password, tokens, headers ทั้งหมด
☐ **แยก middleware เป็นไฟล์แยก**	เพื่อให้ง่ายต่อ maintenance และ testing

---

⚠☐ **8. ข้อควรระวังเชิงเทคนิค**

ประเด็น	รายละเอียด
☐ ลืมเรียก next()	จะทำให้ request ไม่ถูกส่งต่อไป controller ⟶ server ค้าง
☐ โยน error ตรง ๆ ใน middleware	Exception Filters ของ NestJS จะ *ไม่จับ* error ที่โยนออกจาก middleware เพราะมันอยู่นอก Nest lifecycle
☐ Blocking code	เช่น sync file I/O หรือ loop หนัก ๆ ใน middleware ⟶ ส่งผลกระทบกับทุก request เพราะ middleware อยู่ต้น pipeline
☐ ล็อกข้อมูลส่วนตัว	เช่น Authorization header หรือ password ⟶ เสี่ยงต่อความปลอดภัย

☐ **9. ขั้นสูง (Advanced Techniques)**

1. **Structured Logging**

   ใช้ JSON format หรือ structured format เพื่อให้ Log Aggregation Tools (ELK, Datadog, etc.) parse ได้ง่าย

2. this.logger.log(JSON.stringify({
3. method,
4. url: originalUrl,
5. statusCode,
6. responseTime: `${responseTime}ms`,
7. }));
8. **Async Logging / Queue**

   หากต้องเขียน log ลงฐานข้อมูลหรือระบบภายนอก (เช่น Kafka) ⟶ อย่าทำตรง ๆ ใน middleware ⟶ ควรใช้ async worker หรือ queue เพื่อไม่ block request

9. **Multi-context Logging**

   ผูก requestId กับ AsyncLocalStorage ⟶ ทำให้ log ใน service สามารถรู้ว่าเป็น request เดียวกัน

□ **สรุปสั้น ๆ**

- □ Middleware สำหรับ logging คือจุดที่เหมาะที่สุดในการเก็บข้อมูล request ที่ “เข้ามา” และ “ออกไป”
- □ มี 2 วิธีหลัก: Class-based (Nest DI ได้) และ Function-based (global express style)
- □ ควร log หลัง response เสร็จ เพื่อจับ status และ response time
- □ ระวังเรื่องข้อมูลที่ sensitive
- □ ควรใช้ LoggerService ของ Nest ⟶ ได้ timestamp, context, level, integration-friendly

---

ตัวอย่าง **โปรแกรม NestJS สำหรับ “การเขียน Middleware สำหรับ Request Logging”**

- □ **5 ตัวอย่างพื้นฐาน (Basic)** ⟶ เน้นการเข้าใจรูปแบบ Middleware ทั้ง class-based / function-based / การลงทะเบียนแบบต่าง ๆ
- □ **5 ตัวอย่างแนวประยุกต์ (Applied)** ⟶ นำ middleware ไปใช้กับ use case จริง เช่น log ลงไฟล์, ใส่ request ID, เงื่อนไขการ log, log แบบ JSON, แยก environment

แต่ละตัวอย่างจะมี:

- □ โครงสร้างโปรเจกต์
- □ ไฟล์โค้ดแบบเต็ม
- □ คำอธิบายโค้ด (ทีละส่วน)
- □ ตัวอย่างผลการรันจริง (เช่นจาก curl)

☐ SECTION 1: ตัวอย่างพื้นฐาน 5 โปรแกรม

☐ ตัวอย่างที่ 1 — LoggerMiddleware (Class-based)

☐ พื้นฐานที่สุด: สร้าง middleware แบบ class, log method, URL, status, response time

☐ **โครงสร้างโปรเจกต์**

```
basic-logger-1/
├── src/
│   ├── common/
│   │   └── middleware/
│   │       └── logger.middleware.ts
│   ├── app.module.ts
│   ├── app.controller.ts
│   ├── app.service.ts
│   └── main.ts
└── package.json
```

☐ **logger.middleware.ts**

```
import { Injectable, NestMiddleware, Logger } from '@nestjs/common';
import { Request, Response, NextFunction } from 'express';

@Injectable()
export class LoggerMiddleware implements NestMiddleware {
  private logger = new Logger('HTTP');

  use(req: Request, res: Response, next: NextFunction) {
    const { method, originalUrl } = req;
    const start = Date.now();

    res.on('finish', () => {
      const time = Date.now() - start;
      this.logger.log(`${method} ${originalUrl} ${res.statusCode} - ${time}ms`);
    });

    next();
```

```
  }
}
```

app.module.ts

```
import { Module, NestModule, MiddlewareConsumer, RequestMethod } from '@nestjs/common';
import { LoggerMiddleware } from './common/middleware/logger.middleware';
import { AppController } from './app.controller';
import { AppService } from './app.service';

@Module({
  controllers: [AppController],
  providers: [AppService],
})
export class AppModule implements NestModule {
  configure(consumer: MiddlewareConsumer) {
    consumer
      .apply(LoggerMiddleware)
      .forRoutes({ path: '*', method: RequestMethod.ALL });
  }
}
```

app.controller.ts

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

@Controller()
export class AppController {
  @Get()
  getHello() {
    return 'Hello Middleware!';
  }
}
```

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);
}
bootstrap();
```

☐ **ผลการรัน**

```
$ curl http://localhost:3000/
Hello Middleware!
Console:
[Nest] 16732   - 10/12/2025, 8:01:21 AM   [HTTP] GET / 200 - 3ms
```

☐ **ตัวอย่างที่ 2 — Functional Middleware (Global)**

☐ ใช้ app.use() ⟶ เขียน logger แบบง่ายสุด

☐ main.ts

```
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';
import { Request, Response, NextFunction } from 'express';

function simpleLogger(req: Request, res: Response, next: NextFunction) {
  const start = Date.now();
  res.on('finish', () => {
    console.log(`${req.method} ${req.originalUrl} ${res.statusCode} - ${Date.now() - start}ms`);
  });
  next();
}

async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  app.use(simpleLogger);
  await app.listen(3000);
}
bootstrap();
```

☐ ผลการรัน:

```
GET / 200 - 2ms
```

☐ **ตัวอย่างที่ 3 — ใช้ Middleware เฉพาะบาง Route**

☐ app.module.ts

```
consumer
  .apply(LoggerMiddleware)
  .forRoutes({ path: 'hello', method: RequestMethod.GET });
```

☐ app.controller.ts

```
@Controller()
export class AppController {
  @Get('hello')
  hello() { return 'hello'; }

  @Get('no-log')
  noLog() { return 'no log here'; }
}
```

☐ ทดสอบ

```
curl http://localhost:3000/hello
# -> มี log

curl http://localhost:3000/no-log
# -> ไม่มี log
```

☐ **ตัวอย่างที่ 4 — Middleware สำหรับ Logging Header เพิ่ม**

☐ logger.middleware.ts

```
this.logger.log(
  `${method} ${originalUrl} ${res.statusCode} - ${time}ms - UA:${req.headers['user-agent']}`,
);
```

⟶ Log จะเพิ่ม user-agent:

```
[HTTP] GET / 200 - 2ms - UA:curl/8.5.0
```

☐ **ตัวอย่างที่ 5 — Logging เฉพาะ Environment = development**

☐ logger.middleware.ts

```
if (process.env.NODE_ENV === 'development') {
  res.on('finish', () => {
```

```
    this.logger.log(`${method} ${originalUrl} ${res.statusCode}`);
  });
}
```

⟶ ใน production จะไม่ log

☐ SECTION 2: ตัวอย่างแนวประยุกต์ (Applied) 5 โปรแกรม

☐ ตัวอย่างที่ 6 — Logger Middleware + เขียนลงไฟล์

☐ logger.middleware.ts

```
import { Injectable, NestMiddleware } from '@nestjs/common';
import { Request, Response, NextFunction } from 'express';
import * as fs from 'fs';
import * as path from 'path';

@Injectable()
export class FileLoggerMiddleware implements NestMiddleware {
  private logPath = path.join(__dirname, '../../../logs/access.log');

  use(req: Request, res: Response, next: NextFunction) {
    const start = Date.now();
    res.on('finish', () => {
      const log = `${new Date().toISOString()} ${req.method} ${req.originalUrl} ${res.statusCode}
- ${Date.now() - start}ms\n`;
      fs.appendFileSync(this.logPath, log);
    });
    next();
  }
}
```

☐ สร้างโฟลเดอร์ logs/ ก่อนรัน ⟶ Log จะบันทึกในไฟล์

☐ ตัวอย่างที่ 7 — Logger Middleware + Request ID (Correlation ID)

☐ logger.middleware.ts

```
import { v4 as uuid } from 'uuid';
```

```
use(req: Request, res: Response, next: NextFunction) {
  const requestId = uuid();
  (req as any).requestId = requestId;

  res.on('finish', () => {
    console.log(`[${requestId}] ${req.method} ${req.originalUrl} ${res.statusCode}`);
  });

  next();
}
```

☐ app.controller.ts

```
@Get()
getHello(@Req() req: Request) {
  return `Request ID: ${(req as any).requestId}`;
}
```

☐ Output:

```
[4b2f-xxxx] GET / 200
```

☐ **ตัวอย่างที่ 8 — Logger Middleware + JSON Structured Logging**

☐ logger.middleware.ts

```
res.on('finish', () => {
  const log = {
    timestamp: new Date().toISOString(),
    method: req.method,
    path: req.originalUrl,
    status: res.statusCode,
    responseTime: Date.now() - start,
  };
  console.log(JSON.stringify(log));
});
```

☐ Output:

```
{"timestamp":"2025-10-
12T08:04:54.071Z","method":"GET","path":"/","status":200,"responseTime":3}
```

เหมาะกับส่งเข้า Elasticsearch / Loki / Datadog

□ ตัวอย่างที่ 9 — Logger Middleware + เงื่อนไขการ log เฉพาะ error

□ logger.middleware.ts

```
res.on('finish', () => {
  if (res.statusCode >= 400) {
    console.error(`□ ${req.method} ${req.originalUrl} ${res.statusCode}`);
  }
});
```

⟶ Log เฉพาะเวลามี error 4xx / 5xx ⟶ ช่วยลด noise

□ ตัวอย่างที่ 10 — Logger Middleware + Async Logging ผ่าน Queue

แนวคิด:

- Middleware ไม่เขียน log ตรง ๆ (เพื่อไม่ block)
- ส่งข้อมูลเข้า queue (เช่น BullMQ หรือ RabbitMQ)
- Worker จะรับไปเขียน log ภายหลัง

□ logger.middleware.ts

```
import { Injectable, NestMiddleware } from '@nestjs/common';
import { Request, Response, NextFunction } from 'express';
import { LogQueueService } from '../queue/log-queue.service';

@Injectable()
export class QueueLoggerMiddleware implements NestMiddleware {
  constructor(private readonly logQueue: LogQueueService) {}

  use(req: Request, res: Response, next: NextFunction) {
    const start = Date.now();
    res.on('finish', () => {
      this.logQueue.add({
        method: req.method,
        path: req.originalUrl,
        status: res.statusCode,
        time: Date.now() - start,
      });
    });
```

```
    next();
  }
}
```

□ ใน LogQueueService ใช้ Bull queue ⟶ Worker จะเขียนลง DB หรือไฟล์ภายหลัง

□ **สรุปรวม**

ประเภท	ตัวอย่าง	จุดสำคัญ
พื้นฐาน 1	Class-based logger	โครงหลักของ Nest Middleware
พื้นฐาน 2	Function global	Express-style, ง่าย
พื้นฐาน 3	Scoped middleware	กำหนดเฉพาะ path
พื้นฐาน 4	Logging headers	เพิ่มข้อมูล User-Agent
พื้นฐาน 5	Conditional env	ควบคุม log ตาม environment
ประยุกต์ 6	Log to file	เขียนลง access.log
ประยุกต์ 7	Request ID	เพิ่ม correlation ID
ประยุกต์ 8	JSON log	สำหรับ ELK
ประยุกต์ 9	Error-only log	ลด noise
ประยุกต์ 10	Queue logging	Async, scalable

การใช้งาน Global Middleware

หัวข้อ **“การใช้งาน Global Middleware”** ใน NestJS เป็นส่วนสำคัญที่ใช้บ่อยในระบบจริง เพราะ Global Middleware จะถูกเรียก **ก่อนถึงทุก Route / Controller** คล้าย ๆ กับ “ด่านแรก” ที่ Request เข้ามาเจอในระบบทั้งหมด ซึ่งเหมาะกับการใช้งานในงานต่อไปนี้ เช่น

- □ Logging ทั้งระบบ
- □ ตรวจสอบ Token / Header เบื้องต้น
- □ ตั้งค่า CORS / Header สำหรับทุก Request
- □ ตรวจสอบ Request ก่อนเข้า Business Logic
- ⚡ ทำงานบางอย่างซ้ำ ๆ ที่ไม่อยากเขียนในทุก Controller

□ **โครงสร้างของ Middleware ใน NestJS (Global Middleware)**

NestJS รองรับการเขียน Global Middleware ได้ 2 วิธีหลัก:

1. □ **แบบ Functional Middleware** (ฟังก์ชันธรรมดา)

2. □ **แบบ Class-based Middleware** (implements NestMiddleware)

และการทำให้เป็น “Global” ทำได้ผ่าน

- app.use() ⟶ ใน main.ts
- หรือ ลงทะเบียนใน AppModule ผ่าน configure() ของ NestModule

□ **หลักการทำงาน**

Client Request

↓

(Global Middleware) ⟶ Logging / Auth / Transform

↓

Route-specific Middleware (optional)

↓

Guards ⟶ Interceptors ⟶ Pipes

↓

Controller ⟶ Service ⟶ Response

□ **ตัวอย่างโค้ดเชิงลึก**

ด้านล่างนี้ ผมจะให้ **5 โปรแกรมพื้นฐาน** + **5 โปรแกรมแนวประยุกต์** สำหรับ “การใช้งาน Global Middleware”

(ทุกตัวอย่างเป็นโค้ด **เต็มไฟล์** + โครงสร้างโปรเจกต์ + คำอธิบาย + ผลการรัน)

□ **A. โปรแกรมพื้นฐาน (5 ตัวอย่าง)**

□ **โครงสร้างโปรเจกต์ (ทุกตัวอย่างใช้เหมือนกัน)**

global-middleware-demo/

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

└── nest-cli.json

ตัวอย่างที่ 1: Global Logging Middleware (Functional)

src/middlewares/logger.middleware.ts

```
import { Request, Response, NextFunction } from 'express';

export function logger(req: Request, res: Response, next: NextFunction) {
  console.log(`[${new Date().toISOString()}] ${req.method} ${req.url}`);
  next(); // ไปยัง middleware ถัดไปหรือ controller
}
```

src/main.ts

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

async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  // ใช้งาน middleware ทั่วทั้งระบบ
  app.use(logger);
  await app.listen(3000);
}
bootstrap();
```

src/app.controller.ts

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

@Controller()
export class AppController {
  @Get()
  getHello() {
    return 'Hello from NestJS!';
  }
}
```

คำอธิบาย:

- logger() จะถูกเรียกก่อนทุก route

- app.use(logger) ⟶ ทำให้ logger ใช้ global

ผลการรัน:

[2025-10-12T10:00:00.123Z] GET /

Hello from NestJS!

ตัวอย่างที่ 2: Class-based Middleware แบบ Global

src/middlewares/logger.middleware.ts

```
import { Injectable, NestMiddleware } from '@nestjs/common';
import { Request, Response, NextFunction } from 'express';

@Injectable()
export class LoggerMiddleware implements NestMiddleware {
  use(req: Request, res: Response, next: NextFunction) {
    console.log(`[LoggerMiddleware] ${req.method} ${req.url}`);
    next();
  }
}
```

src/app.module.ts

```
import { MiddlewareConsumer, Module, NestModule } from '@nestjs/common';
import { AppController } from './app.controller';
import { AppService } from './app.service';
import { LoggerMiddleware } from './middlewares/logger.middleware';

@Module({
  controllers: [AppController],
  providers: [AppService],
})
export class AppModule implements NestModule {
  configure(consumer: MiddlewareConsumer) {
    // ใช้กับทุก route ⟶ global
    consumer.apply(LoggerMiddleware).forRoutes('*');
  }
}
```

คำอธิบาย:

- ใช้ class + implements NestMiddleware
- forRoutes('*') ⟶ ครอบทุก route = global middleware

ตัวอย่างที่ 3: Global Middleware + Logging Response Time

src/middlewares/timing.middleware.ts

```
import { Request, Response, NextFunction } from 'express';

export function timing(req: Request, res: Response, next: NextFunction) {
  const start = Date.now();
  res.on('finish', () => {
    const duration = Date.now() - start;
    console.log(` ${req.method} ${req.url} took ${duration}ms`);
  });
  next();
}
```

src/main.ts

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

async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  app.use(timing);
  await app.listen(3000);
}
bootstrap();
```

คำอธิบาย:

- res.on('finish') ทำงานหลังส่ง response
- ใช้วัด performance ของทุก request

ตัวอย่างที่ 4: Global Middleware + Custom Header

src/middlewares/header.middleware.ts

```
import { Request, Response, NextFunction } from 'express';
```

```
export function customHeader(req: Request, res: Response, next: NextFunction) {
  res.setHeader('X-App-Version', '1.0.0');
  next();
}
```

☐ **src/main.ts**

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

async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  app.use(customHeader);
  await app.listen(3000);
}
bootstrap();
```

☐ **ผลลัพธ์:**

เมื่อเรียก / ⟶ response จะมี header:

X-App-Version: 1.0.0

☐ **ตัวอย่างที่ 5: Global Middleware + Block HTTP Method**

☐ **src/middlewares/block.middleware.ts**

```
import { Request, Response, NextFunction } from 'express';

export function blockPutMethod(req: Request, res: Response, next: NextFunction) {
  if (req.method === 'PUT') {
    return res.status(403).json({ message: 'PUT method is not allowed' });
  }
  next();
}
```

☐ **src/main.ts**

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

```
async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  app.use(blockPutMethod);
  await app.listen(3000);
}
bootstrap();
```

□ **ผลลัพธ์:**

ส่ง PUT ⟶ 403 Forbidden

□ **B. โปรแกรมแนวประยุกต์ (5 ตัวอย่าง)**

□ **1. Logging + Auth Check Global Middleware**

⟶ Log ทุก request + ตรวจว่า header x-api-key ตรงหรือไม่

□ **2. Global Middleware สำหรับ CORS + Security Header**

⟶ ตั้งค่า CORS และ CSP/Helmet สำหรับทุก route

□ **3. Global Middleware เชื่อมต่อ External Logger (เช่น Winston)**

⟶ ส่ง log ทุก request ไป Elastic / Datadog

□ **4. Global Middleware แปลง Request Body ก่อนเข้า Controller**

⟶ เช่น sanitize input, trim string

□ **5. Global Middleware ตรวจสอบ Maintenance Mode**

⟶ หากระบบอยู่ในโหมด Maintenance ⟶ Block ทุก request ยกเว้น /status

□ **ตัวอย่างโปรแกรมพื้นฐานจำนวน 5 โปรแกรม**

□ **ตัวอย่างโปรแกรมแนวประยุกต์จำนวน 5 โปรแกรม**

สำหรับหัวข้อ

□ **“การใช้งาน Global Middleware”** ใน NestJS

ทุกตัวอย่างจะมี

- □ **โครงสร้างโปรเจกต์**
- □ **โค้ดเต็มไฟล์**
- □ **คำอธิบายโค้ดทีละส่วน**
- □ **ผลการรันตัวอย่าง**

เพื่อให้คุณเห็นภาพครบแบบมืออาชีพ □

□ **โครงสร้างโปรเจกต์พื้นฐาน (ใช้ร่วมกันทุกตัวอย่าง)**

```
global-middleware-demo/
├── src/
│   ├── app.controller.ts
│   ├── app.module.ts
│   ├── app.service.ts
│   ├── main.ts
│   └── middlewares/
│       ├── logger.middleware.ts
│       ├── timing.middleware.ts
│       ├── header.middleware.ts
│       ├── block.middleware.ts
│       └── class-logger.middleware.ts
├── package.json
├── tsconfig.json
└── nest-cli.json
```

☐ **A. ตัวอย่างโปรแกรมพื้นฐาน 5 โปรแกรม**

☐ **ตัวอย่างที่ 1: Functional Global Logging Middleware**

☐ **src/middlewares/logger.middleware.ts**

```
import { Request, Response, NextFunction } from 'express';

export function logger(req: Request, res: Response, next: NextFunction) {
  console.log(`[${new Date().toISOString()}] ${req.method} ${req.url}`);
  next(); // ไปยัง middleware หรือ controller ถัดไป
}
```

☐ **src/main.ts**

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

async function bootstrap() {
  const app = await NestFactory.create(AppModule);
```