NESTJS WEB PROGRAMMING

Professional

(Integrative-Generative AI Edition)

Author: Student Price Book Center

คำนำ

การพัฒนาแอปพลิเคชันเว็บสมัยใหม่ต้องอาศัยเครื่องมือและเทคนิคที่ทรงพลังเพื่อสร้างระบบที่มีประสิทธิภาพ ปลอดภัย และสามารถปรับขยายได้ตามความต้องการของผู้ใช้งาน NestJS เป็นหนึ่งในเฟรมเวิร์กที่ตอบโจทย์การพัฒนาแอปพลิเคชันแบบ scalable และ maintainable ด้วยสถาปัตยกรรมที่ชัดเจน สนับสนุน TypeScript และรวมแนวทางปฏิบัติที่ดีที่สุดสำหรับการพัฒนาเว็บในระดับองค์กร หนังสือเล่มนี้ถูกออกแบบมาเพื่อผู้พัฒนาที่ต้องการยกระดับทักษะไปสู่ระดับ **Professional** โดยมุ่งเน้นทั้งความเข้าใจเชิงลึกและการปฏิบัติจริง

ในบทที่ 16 เราจะเจาะลึกการจัดการ **File Uploading และ Multer** ซึ่งเป็นฟีเจอร์สำคัญของแอปพลิเคชันที่ต้องรองรับการส่งไฟล์จากผู้ใช้งาน หนังสือเล่มนี้สอนการใช้งาน **@nestjs/platform-express** ร่วมกับ Multer เพื่ออัปโหลดไฟล์หลายประเภท การกำหนด **storage destination และ filter** และการจัดการ **validation และ limit size** ผ่านตัวอย่างบูรณาการและ Ultimate Project เพื่อให้ผู้อ่านสามารถสร้างระบบอัปโหลดไฟล์ที่ปลอดภัยและมีประสิทธิภาพ

บทที่ 17 นำเสนอแนวทางการสร้าง **Task Scheduling และ Background Jobs** ซึ่งช่วยให้แอปสามารถทำงานเบื้องหลังแบบอัตโนมัติได้อย่างมีประสิทธิภาพ ผู้พัฒนาจะได้เรียนรู้การใช้ **@nestjs/schedule** ตั้งเวลา **cron, interval, และ timeout** และการผสาน **Bull / Queue ร่วมกับ Redis** สำหรับงานที่ซับซ้อนและต้องรองรับผู้ใช้งานจำนวนมาก ตัวอย่างและ Ultimate Integration Project จะช่วยให้ผู้อ่านเห็นภาพรวมและนำไปประยุกต์ใช้งานจริง

ต่อมา บทที่ 18 จะสอนเรื่อง **Unit Testing และ Integration Testing** เพื่อสร้างความมั่นใจว่าระบบทำงานได้ถูกต้องตามที่ออกแบบ หนังสือแนะนำการใช้ **Jest** สำหรับ unit test, การ **mock dependencies ใน Service**, การเขียน **E2E Test ด้วย supertest** รวมถึงการทดสอบ **Guards และ Pipes** ผ่านตัวอย่างบูรณาการและ Ultimate Project ทำให้ผู้พัฒนาสามารถสร้างระบบที่มีคุณภาพสูงและลดข้อผิดพลาดเชิง logic

บทที่ 19 เจาะลึกการ **Deploy และ Production-ready NestJS** โดยเริ่มจากการสร้าง **build ด้วย nest build** และแนวทางการ deploy บนแพลตฟอร์มยอดนิยม เช่น Docker, Heroku, Vercel และ Render นอกจากนี้ยังครอบคลุมการใช้ **PM2** สำหรับ process management และการตั้งค่า production-ready เช่น **CORS, HTTPS, และ Logging** เพื่อให้ระบบปลอดภัย มีประสิทธิภาพ และรองรับผู้ใช้งานจำนวนมาก

สุดท้าย บทที่ 20 มุ่งเน้น **Performance Optimization และ Monitoring** เพื่อยกระดับประสิทธิภาพของแอป NestJS ทั้งการทำ **caching ด้วย @nestjs/cache-manager** และ **Redis cache สำหรับ API** การติดตั้งและใช้งานเครื่องมือ **Monitoring** เช่น Prometheus, Sentry และ Datadog รวมถึงการปรับแต่ง **memory และ response time** ตัวอย่างบูรณาการและ Ultimate

Integrated Project จะช่วยให้ผู้พัฒนาสามารถสร้างระบบที่ตอบสนองรวดเร็ว เสถียร และสามารถตรวจสอบปัญหาเชิงโปรแกรมได้อย่างมืออาชีพ

หนังสือเล่มนี้ถูกออกแบบสำหรับนักพัฒนาที่มีพื้นฐาน NestJS อยู่แล้วและต้องการก้าวไปสู่ระดับมืออาชีพ ผ่านตัวอย่างโค้ดจริง ตัวอย่างบูรณาการ และ Ultimate Project ที่ครอบคลุมทุกหัวข้อสำคัญ ผู้อ่านจะได้เรียนรู้แนวทางปฏิบัติที่ดีที่สุด การตั้งค่าที่ปลอดภัย และเทคนิคการปรับแต่งประสิทธิภาพที่จำเป็นสำหรับแอปพลิเคชันเว็บที่ใช้งานจริง

ด้วยรักและปรารถนาดี

ศูนย์หนังสือราคานักเรียน

สารบัญ

หน้า

บทที่ 16

File Uploading และ Multer

(File Uploading and Multer)

เนื้อหา

- File Uploading และ Multer
- การใช้งาน @nestjs/platform-express
- การ Upload ไฟล์หลายประเภท (Multiple File Types)
- การกำหนด storage destination และ filter
- การจัดการ Validation และ Limit Size
- ตัวอย่างบูรณาการ
- ตัวอย่าง Ultimate

การพัฒนาแอปพลิเคชันเว็บสมัยใหม่มักต้องมีความสามารถในการรับส่งไฟล์จากผู้ใช้งาน ซึ่งอาจรวมถึงรูปภาพ เอกสาร หรือไฟล์ประเภทอื่น ๆ การรองรับฟีเจอร์เหล่านี้อย่างปลอดภัยและมีประสิทธิภาพเป็นสิ่งจำเป็นสำหรับนักพัฒนาที่ใช้ NestJS เพราะการจัดการไฟล์ที่ไม่ดีอาจนำไปสู่ปัญหาด้านความปลอดภัยและประสิทธิภาพของระบบ

ในบทนี้ เราจะเริ่มต้นด้วยการแนะนำ **@nestjs/platform-express** ซึ่งเป็นโมดูลหลักที่ช่วยให้นักพัฒนาสามารถรวม Express.js เข้ากับ NestJS เพื่อจัดการการรับส่งไฟล์ได้อย่างราบรื่น โมดูลนี้ทำหน้าที่เป็นสะพานเชื่อมระหว่างฟังก์ชันการทำงานของ NestJS และ middleware ของ Express โดยเฉพาะ **Multer** ซึ่งเป็นเครื่องมือยอดนิยมสำหรับการอัปโหลดไฟล์

ต่อมา เราจะสำรวจวิธีการ **อัปโหลดไฟล์หลายประเภท** ภายในระบบเดียว ไม่ว่าจะเป็นรูปภาพ ไฟล์เอกสาร หรือไฟล์สื่ออื่น ๆ การจัดการไฟล์หลายประเภทพร้อมกันต้องใช้เทคนิคการกำหนด **MIME type** และการกรองไฟล์ เพื่อให้มั่นใจว่าผู้ใช้งานส่งไฟล์ที่ถูกต้องตามที่ระบบรองรับ

นอกจากนั้น การกำหนด **storage destination และ filter** เป็นหัวใจสำคัญของการจัดการไฟล์ ระบบต้องรู้ว่าไฟล์จะถูกเก็บไว้ที่ไหน และจะทำการตรวจสอบชนิดไฟล์หรือชื่อไฟล์อย่างไร ตัวอย่างเช่น การแยกโฟลเดอร์เก็บรูปภาพและเอกสารออกจากกัน หรือการใช้ฟิลเตอร์เพื่อตรวจสอบนามสกุลไฟล์

การจัดการ **Validation และ Limit size** ก็เป็นอีกหนึ่งแนวทางสำคัญในการป้องกันปัญหา เช่น ไฟล์ขนาดใหญ่เกินไปหรือไฟล์ชนิดไม่ปลอดภัย การตั้งค่าเหล่านี้ช่วยให้ระบบมีความเสถียรและป้องกันการโจมตีแบบ Denial of Service ที่อาจเกิดขึ้นจากการอัปโหลดไฟล์ขนาดใหญ่

ในบทนี้ เราจะนำเสนอแนวทางการใช้งาน Multer ร่วมกับ NestJS ตั้งแต่การตั้งค่าเบื้องต้น จนถึงการปรับแต่งขั้นสูง รวมถึงตัวอย่างการใช้งานจริงที่ครอบคลุมทั้ง **storage, filter, validation และ limit size** เพื่อให้นักพัฒนาสามารถนำไปประยุกต์ใช้ได้อย่างตรงจุด

สุดท้าย บทนี้ไม่ได้เพียงแค่สอนการอัปโหลดไฟล์เท่านั้น แต่ยังเน้นไปที่ **หลักการปฏิบัติที่ปลอดภัยและมีประสิทธิภาพ** ของการจัดการไฟล์ เพื่อให้ระบบของคุณสามารถรองรับผู้ใช้งานจำนวนมากได้โดยไม่เกิดปัญหา และสามารถจัดการไฟล์ได้อย่างเป็นระเบียบและเชื่อถือได้

ด้วยแนวทางและตัวอย่างในบทนี้ นักพัฒนาจะมีความเข้าใจอย่างลึกซึ้งเกี่ยวกับการอัปโหลดไฟล์ใน NestJS และสามารถสร้างแอปพลิเคชันที่รองรับไฟล์หลากหลายประเภทได้อย่างมั่นใจและปลอดภัย

File Uploading และ Multer

- การใช้งาน @nestjs/platform-express
- การ upload ไฟล์หลายประเภท
- การกำหนด storage destination และ filter
- การจัดการ Validation และ Limit size

ผมได้จัดทำ **บทที่ 16: File Uploading และ Multer** ไว้ในเอกสารแล้วครับ □
เนื้อหาครอบคลุมครบตามหัวข้อที่คุณระบุ ได้แก่

- การใช้งาน **@nestjs/platform-express** ร่วมกับ Multer
- การอัปโหลดไฟล์หลายประเภท (single, multiple, fields)
- การกำหนด **Storage Destination** และ **File Filter**
- การจัดการ **Validation** และการจำกัด **Limit Size**

□ **รายละเอียดเชิงลึก: File Uploading และ Multer**

1. ภาพรวมการทำงาน

Multer เป็น **middleware ของ Express** สำหรับจัดการ multipart/form-data ซึ่งเป็น encoding ที่ใช้เมื่อเราส่งไฟล์ขึ้นมาจาก client (เช่น ฟอร์ม HTML, Postman, หรือ Mobile App)

- Multer จะ **อ่าน stream ของ request body** ⟶ แยกไฟล์ออกจาก field อื่น ๆ
- ไฟล์ที่อัปโหลดจะถูกเก็บไว้ **ชั่วคราว (memory buffer)** หรือ **ถาวร (disk storage)** ตามที่เรากำหนด
- หลังจากประมวลผลแล้ว NestJS จะ inject ไฟล์เข้ามาใน controller ผ่าน decorator เช่น @UploadedFile() หรือ @UploadedFiles()

---

2. Integration ของ NestJS + Multer

NestJS ใช้ **Interceptor** ห่อ Multer เพื่อความเป็นโมดูลาร์

- FileInterceptor ⟶ รองรับอัปโหลดไฟล์เดี่ยว
- FilesInterceptor ⟶ รองรับอัปโหลดหลายไฟล์ในฟิลด์เดียว
- FileFieldsInterceptor ⟶ รองรับหลายฟิลด์ที่เป็นไฟล์
- AnyFilesInterceptor ⟶ รับไฟล์จากทุกฟิลด์

เบื้องหลัง Interceptor จะเรียกใช้ multer(options).single(), multer(options).array(), หรือ multer(options).fields() ตามประเภทที่เลือก

---

3. การ Upload หลายประเภท

3.1 Single Upload

```
@Post('avatar')
@UseInterceptors(FileInterceptor('avatar'))
uploadAvatar(@UploadedFile() file: Express.Multer.File) {
  return { name: file.originalname, size: file.size };
}
```

- ใช้กับ input type file name="avatar"
- รองรับไฟล์เดียวต่อ request

3.2 Multiple Upload

```
@Post('gallery')
@UseInterceptors(FilesInterceptor('images', 10)) // สูงสุด 10 รูป
uploadGallery(@UploadedFiles() files: Express.Multer.File[]) {
  return files.map(f => f.originalname);
}
```

- ใช้กับ name="images" + multiple attribute
- จำกัดจำนวนไฟล์สูงสุดได้

3.3 Fields Upload

```
@Post('profile')
@UseInterceptors(
  FileFieldsInterceptor([
    { name: 'avatar', maxCount: 1 },
    { name: 'documents', maxCount: 5 },
  ]),
)
uploadProfile(@UploadedFiles() files) {
```

```
  return {
    avatar: files.avatar?.map(f => f.filename),
    documents: files.documents?.map(f => f.filename),
  };
}
```

- ใช้สำหรับหลายฟิลด์ เช่น avatar และ documents
- เหมาะกับฟอร์มซับซ้อน

---

4. Storage Destination และ File Filter

4.1 Storage (Disk vs Memory)

- **DiskStorage**: เก็บไฟล์ไว้ในโฟลเดอร์ เช่น uploads/
  ⟶ เหมาะสำหรับ Production ที่ต้องโยกไฟล์ไป Cloud Storage (S3, GCS)
- **MemoryStorage**: เก็บไฟล์ใน buffer (RAM)
  ⟶ เหมาะเมื่อเราจะอัปโหลดต่อไปยัง external API โดยไม่เก็บลงดิสก์

```
import { diskStorage } from 'multer';

const storage = diskStorage({
  destination: './uploads',
  filename: (req, file, cb) => {
    const ext = file.originalname.split('.').pop();
    cb(null, `${Date.now()}.${ext}`);
  },
});
```

4.2 File Filter

```
const fileFilter = (req, file, cb) => {
  if (!file.mimetype.match(/\/(jpg|jpeg|png)$/)) {
    return cb(new Error('Only image files allowed'), false);
  }
  cb(null, true);
};
```

---

5. Validation และ Limit Size

Multer รองรับ **limits** เช่น fileSize, files, fields

```
@UseInterceptors(
```

```
  FileInterceptor('file', {
    storage,
    fileFilter,
    limits: { fileSize: 2 * 1024 * 1024 }, // 2MB
  }),
)
```

- หากไฟล์ใหญ่เกิน limit ⟶ Multer จะ throw error LIMIT_FILE_SIZE
- สามารถเขียน **Exception Filter** จับ error และส่ง response ที่เหมาะสมได้

---

6. Error Handling

NestJS + Multer Error ⟶ ต้องใช้ **Exception Filter** เพื่อ handle

```
@Catch()
export class UploadExceptionFilter implements ExceptionFilter {
  catch(exception: any, host: ArgumentsHost) {
    const res = host.switchToHttp().getResponse();
    if (exception.code === 'LIMIT_FILE_SIZE') {
      return res.status(400).json({ message: 'File too large!' });
    }
    res.status(400).json({ message: exception.message });
  }
}
```

---

7. ปัญหาใน Production และ Best Practices

1. **ไฟล์ขนาดใหญ่** ⟶ ควรใช้ Reverse Proxy (Nginx) ตั้งค่า client_max_body_size และใช้ streaming แทนการ buffer ทั้งไฟล์
2. **Security**
   - ตรวจสอบ MIME type + Extension
   - สแกนไฟล์ (เช่น ClamAV) ก่อนใช้งาน
   - หลีกเลี่ยงการเก็บไฟล์บน server ตรง ๆ ⟶ ควรอัปโหลดต่อไปยัง S3/GCS แล้วเก็บแต่ URL
3. **Path Traversal**
   - หลีกเลี่ยงการใช้ file.originalname ตรง ๆ เป็น filename
   - ใช้ UUID หรือ hash แทน
4. **Scalability**

- หากระบบต้องรองรับไฟล์จำนวนมาก ให้ใช้ **Cloud Storage**
- ใช้ CDN สำหรับการเสิร์ฟไฟล์กลับ

5. **Validation Layer**
   - ใช้ Pipe ตรวจสอบ metadata เช่น type, size
   - ใช้ DTO สำหรับข้อมูลประกอบไฟล์

---

8. Workflow ที่นิยมใน Production

1. Client ⟶ Upload ไปที่ API NestJS
2. NestJS ⟶ ใช้ Multer รับไฟล์ชั่วคราว (memory/disk)
3. Service ⟶ Upload ต่อไปยัง S3 / GCP Storage / Azure Blob
4. Database ⟶ เก็บแต่ metadata (URL, ขนาด, ชนิดไฟล์)
5. Response ⟶ ส่ง URL/ID กลับไปให้ client

สรุปเชิงลึก

- NestJS ห่อ Multer ผ่าน Interceptor ⟶ ง่ายต่อการใช้งาน
- ควรกำหนด Storage, Filter, Limit, และ Validation อย่างรัดกุม
- ใน Production ไม่ควรเก็บไฟล์บน server ตรง ๆ ⟶ ควร push ไป Cloud Storage
- Error Handling และ Security ต้องมาก่อนเสมอ
- Workflow ที่ดีคือ NestJS เป็น **gateway/validator** ของไฟล์ ไม่ใช่ storage หลัก

การใช้งาน @nestjs/platform-express

การใช้งาน @nestjs/platform-express ซึ่งเป็นหัวใจสำคัญของระบบ File Upload ใน NestJS กันแบบ **ระดับ Professional**

(เพราะ NestJS ใช้ Express เป็นพื้นฐาน และ @nestjs/platform-express คือ Bridge ที่เชื่อม Multer เข้ากับระบบ Interceptor/Decorator ของ NestJS)

---

1. ทำไมถึงต้องใช้ @nestjs/platform-express

ใน NestJS เราสามารถเลือก “platform” ได้ เช่น

- @nestjs/platform-express ⟶ ใช้ Express (Default)
- @nestjs/platform-fastify ⟶ ใช้ Fastify (ประสิทธิภาพสูงกว่า)

เมื่อใช้ Express ⟶ NestJS สามารถดึงความสามารถของ middleware ได้เต็มที่ เช่น body-parser, cookie-parser, Multer ฯลฯ

□ โดยเฉพาะ **การอัปโหลดไฟล์** ⟶ NestJS ใช้ Multer (middleware ของ Express) ⟶ แต่แทนที่จะให้เราเขียน Multer เองแบบดิบ ๆ NestJS ได้สร้าง abstraction ขึ้นมาให้ใช้งานง่ายผ่าน **Interceptor** และ decorator ที่อยู่ใน @nestjs/platform-express

---

⚡ 2. การติดตั้งแพ็กเกจ

เมื่อสร้างโปรเจกต์ NestJS ใหม่ (ผ่าน nest new) ระบบจะติดตั้ง @nestjs/platform-express ให้อยู่แล้วโดยอัตโนมัติ

แต่ถ้าต้องการติดตั้งเอง:

npm install @nestjs/platform-express multer

□ Multer ต้องติดตั้งด้วย เพราะเป็น dependency ที่ใช้สำหรับประมวลผล multipart/form-data

---

□ 3. การทำงานของ @nestjs/platform-express ภายใต้ Hood

เบื้องหลังการทำงานของ Interceptor เช่น FileInterceptor() คือ:

1. NestJS ⟶ โหลด multer() พร้อม options ที่เรากำหนด
2. แทรก Multer เข้าเป็น Express middleware เฉพาะ route นั้น
3. Multer ประมวลผล request ⟶ แยกไฟล์ออกมา ⟶ เก็บลง disk หรือ memory ⟶ เพิ่ม property req.file หรือ req.files
4. Interceptor ⟶ ดึงข้อมูลไฟล์มา ⟶ Inject เข้า Controller method parameter ผ่าน @UploadedFile() หรือ @UploadedFiles()

□ 4. ตัวอย่างการใช้งานพื้นฐาน

```
import { Controller, Post, UploadedFile, UseInterceptors } from '@nestjs/common';
import { FileInterceptor } from '@nestjs/platform-express';

@Controller('upload')
export class UploadController {
  @Post('single')
  @UseInterceptors(FileInterceptor('file'))  // <-- มาจาก @nestjs/platform-express
  uploadSingle(@UploadedFile() file: Express.Multer.File) {
    return {
      originalName: file.originalname,
      fileSize: file.size,
      mimeType: file.mimetype,
    };
```

```
  }
}
```

☐ **อธิบาย**

- FileInterceptor('file') ⟶ กำหนดให้ Multer อ่านไฟล์จากฟิลด์ชื่อ file
- เมื่อ client ส่ง multipart/form-data ⟶ Multer จะจัดการ parsing ⟶ แล้วไฟล์จะถูก inject เข้า parameter file ของ method
- @UploadedFile() เป็น decorator ที่มาพร้อม platform-express

☐ **5. Multiple Uploads และ Field Upload**

☐ **Multiple Upload (หลายไฟล์ใน field เดียว)**

```
import { Controller, Post, UploadedFiles, UseInterceptors } from '@nestjs/common';
import { FilesInterceptor } from '@nestjs/platform-express';

@Controller('upload')
export class UploadController {
  @Post('multiple')
  @UseInterceptors(FilesInterceptor('files', 5)) // สูงสุด 5 ไฟล์
  uploadMultiple(@UploadedFiles() files: Express.Multer.File[]) {
    return files.map(f => ({ name: f.originalname, size: f.size }));
  }
}
```

☐ **Fields Upload (หลาย field)**

```
import { Controller, Post, UploadedFiles, UseInterceptors } from '@nestjs/common';
import { FileFieldsInterceptor } from '@nestjs/platform-express';

@Controller('upload')
export class UploadController {
  @Post('fields')
  @UseInterceptors(
    FileFieldsInterceptor([
      { name: 'avatar', maxCount: 1 },
      { name: 'documents', maxCount: 3 },
    ]),
  )
```

```
  uploadFields(@UploadedFiles() files: Record<string, Express.Multer.File[]>) {
    return {
      avatar: files.avatar?.[0].originalname,
      documents: files.documents?.map(d => d.originalname),
    };
  }
}
```

 ทั้งหมดนี้มาจาก @nestjs/platform-express — เป็น abstraction ที่ห่อ Multer ให้ใช้สะดวกโดยไม่ต้องเขียน middleware เอง

---

 6. การกำหนด Multer Options

FileInterceptor, FilesInterceptor, FileFieldsInterceptor รับพารามิเตอร์ที่ 2 เป็น Multer Options ได้เลย เช่น

```
import { diskStorage } from 'multer';
import { extname } from 'path';

const storage = diskStorage({
  destination: './uploads',
  filename: (req, file, callback) => {
    const unique = Date.now() + '-' + Math.round(Math.random() * 1e9);
    callback(null, `${file.fieldname}-${unique}${extname(file.originalname)}`);
  },
});

@Post('custom')
@UseInterceptors(FileInterceptor('file', { storage }))
uploadWithCustomStorage(@UploadedFile() file: Express.Multer.File) {
  return { filename: file.filename };
}
```

 storage และ fileFilter เหล่านี้ เป็นของ Multer โดยตรง แต่ NestJS จะ pass ไปยัง Multer ผ่าน Interceptor

---

 7. ความสัมพันธ์กับ Fastify

ถ้าเราเปลี่ยน platform ไปใช้ @nestjs/platform-fastify ⟶ Multer จะไม่สามารถใช้ได้โดยตรง ต้องใช้ plugin fastify-multipart แทน และ NestJS มี abstraction สำหรับ Fastify แยกต่างหาก ดังนั้นทุกอย่างที่กล่าวมาข้างต้น ⟶ ใช้ได้เฉพาะกับ **Express Platform** เท่านั้น

□ **สรุปเชิงลึก**

รายการ	รายละเอียด
□ @nestjs/platform-express	แพ็กเกจที่เชื่อม NestJS กับ Express
□ บทบาท	ทำให้ NestJS ใช้ Multer ผ่าน Interceptor & Decorator ได้ง่าย
□ ฟีเจอร์หลัก	FileInterceptor, FilesInterceptor, FileFieldsInterceptor, AnyFilesInterceptor
⚡ ข้อดี	ไม่ต้องเขียน Multer middleware เอง, ใช้ decorator injection, ทำงานแบบ NestJS ecosystem
⚠□ ใช้ได้เฉพาะ	เมื่อ platform เป็น Express เท่านั้น

□ **ตัวอย่างโปรแกรม NestJS หัวข้อ “File Uploading และ Multer”**

แยกเป็น

- □ **พื้นฐาน 3 โปรแกรม** ⟶ เริ่มจาก Single / Multiple / Fields Upload
- □ **แนวประยุกต์ 3 โปรแกรม** ⟶ เพิ่มการกำหนด Storage, Filter, Validation และการใช้งานร่วมกับ DTO & Pipe

---

□ **โครงสร้างโปรเจกต์หลัก (ใช้ร่วมกันทั้ง 6 ตัวอย่าง)**

```
nestjs-file-upload/
├── src/
│   ├── app.module.ts
│   ├── main.ts
│   └── upload/
│       ├── upload.module.ts
│       ├── upload.controller.ts
│       └── upload.service.ts
├── uploads/
│   └── (ไฟล์ที่ถูกอัปโหลดจะอยู่ในนี้)
```

```
├── package.json
├── tsconfig.json
└── nest-cli.json
```

☐ ตัวอย่างพื้นฐาน (Basic Examples)

☐ ตัวอย่างที่ 1: Single File Upload

☐ src/upload/upload.controller.ts

```
import {
  Controller,
  Post,
  UploadedFile,
  UseInterceptors,
} from '@nestjs/common';
import { FileInterceptor } from '@nestjs/platform-express';

@Controller('upload')
export class UploadController {
  @Post('single')
  @UseInterceptors(FileInterceptor('file'))
  uploadSingle(@UploadedFile() file: Express.Multer.File) {
    return {
      message: 'Upload success',
      filename: file.originalname,
      size: file.size,
    };
  }
}
```

☐ src/upload/upload.module.ts

```
import { Module } from '@nestjs/common';
import { UploadController } from './upload.controller';
import { UploadService } from './upload.service';
```

```
@Module({
  controllers: [UploadController],
  providers: [UploadService],
})
export class UploadModule {}
```

---

src/upload/upload.service.ts

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

@Injectable()
export class UploadService {}
```

---

src/app.module.ts

```
import { Module } from '@nestjs/common';
import { UploadModule } from './upload/upload.module';

@Module({
  imports: [UploadModule],
})
export class AppModule {}
```

---

src/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

รันเซิร์ฟเวอร์

```
npm run start
```

อัปโหลดไฟล์ด้วย Postman

- URL: http://localhost:3000/upload/single
- Method: POST
- Body: form-data ⟶ Key: file ⟶ Type: File ⟶ เลือกไฟล์

☐ **ผลลัพธ์**

```
{
  "message": "Upload success",
  "filename": "test.png",
  "size": 31245
}
```

☐ **ตัวอย่างที่ 2: Multiple Files Upload**

☐ **src/upload/upload.controller.ts (เพิ่ม method)**

```
import {
  Controller,
  Post,
  UploadedFiles,
  UseInterceptors,
} from '@nestjs/common';
import { FilesInterceptor } from '@nestjs/platform-express';

@Controller('upload')
export class UploadController {
  // ... Single upload ด้านบน

  @Post('multiple')
  @UseInterceptors(FilesInterceptor('files', 5)) // จำกัดสูงสุด 5 ไฟล์
  uploadMultiple(@UploadedFiles() files: Express.Multer.File[]) {
    return {
      message: 'Multiple upload success',
      count: files.length,
      files: files.map((f) => f.originalname),
    };
  }
```

```
}
```

▶ **ผลการรัน**

- URL: http://localhost:3000/upload/multiple
- POST ⟶ form-data ⟶ Key: files ⟶ Type: File ⟶ เลือกหลายไฟล์

ผลลัพธ์

```
{
  "message": "Multiple upload success",
  "count": 3,
  "files": ["img1.jpg", "img2.jpg", "doc.pdf"]
}
```

ตัวอย่างที่ 3: Fields Upload (หลาย key)

src/upload/upload.controller.ts (เพิ่ม method)

```
import {
  Controller,
  Post,
  UseInterceptors,
  UploadedFiles,
} from '@nestjs/common';
import { FileFieldsInterceptor } from '@nestjs/platform-express';

@Controller('upload')
export class UploadController {
  // ... Single & Multiple

  @Post('fields')
  @UseInterceptors(
    FileFieldsInterceptor([
      { name: 'avatar', maxCount: 1 },
      { name: 'documents', maxCount: 3 },
    ]),
  )
```

```
  uploadFields(@UploadedFiles() files: { avatar?: Express.Multer.File[]; documents?:
Express.Multer.File[] }) {
    return {
      avatar: files.avatar?.[0]?.originalname,
      documents: files.documents?.map((d) => d.originalname),
    };
  }
}
```

▶ **ผลการรัน**

- URL: http://localhost:3000/upload/fields
- POST ⟶ form-data ⟶
  - avatar ⟶ File (รูปเดียว)
  - documents ⟶ File (หลายไฟล์)

ผลลัพธ์

```
{
  "avatar": "profile.jpg",
  "documents": ["cv.pdf", "certificate.png"]
}
```

ตัวอย่างแนวประยุกต์ (Advanced Examples)

ตัวอย่างที่ 4: กำหนด Storage Destination & Filename

ใช้ diskStorage จาก multer เพื่อเก็บไฟล์ในโฟลเดอร์ uploads/

src/upload/upload.controller.ts

```
import {
  Controller,
  Post,
  UploadedFile,
  UseInterceptors,
} from '@nestjs/common';
import { FileInterceptor } from '@nestjs/platform-express';
import { diskStorage } from 'multer';
import { extname } from 'path';
```

```
@Controller('upload')
export class UploadController {
  @Post('custom-storage')
  @UseInterceptors(
    FileInterceptor('file', {
      storage: diskStorage({
        destination: './uploads',
        filename: (req, file, callback) => {
          const uniqueSuffix = Date.now() + '-' + Math.round(Math.random() * 1e9);
          const ext = extname(file.originalname);
          callback(null, `${file.fieldname}-${uniqueSuffix}${ext}`);
        },
      }),
    }),
  )
  uploadWithCustomStorage(@UploadedFile() file: Express.Multer.File) {
    return {
      message: 'File saved',
      path: file.path,
    };
  }
}
```

☐ **ผลลัพธ์**

```
{
  "message": "File saved",
  "path": "uploads/file-1710000000000-123456789.png"
}
```

☐ **ตัวอย่างที่ 5: File Filter + Validation (เฉพาะ .jpg/.png)**

☐ **src/upload/upload.controller.ts**

```
import {
  BadRequestException,
  Controller,
```

```
  Post,
  UploadedFile,
  UseInterceptors,
} from '@nestjs/common';
import { FileInterceptor } from '@nestjs/platform-express';

@Controller('upload')
export class UploadController {
  @Post('image-only')
  @UseInterceptors(
    FileInterceptor('file', {
      fileFilter: (req, file, callback) => {
        if (!file.mimetype.match(/\/(jpg|jpeg|png)$/)) {
          return callback(new BadRequestException('Only image files are allowed!'), false);
        }
        callback(null, true);
      },
    }),
  )
  uploadImageOnly(@UploadedFile() file: Express.Multer.File) {
    return {
      message: 'Image upload success',
      filename: file.originalname,
    };
  }
}
```

□ หากอัปโหลด PDF ⟶ จะได้

```
{
  "statusCode": 400,
  "message": "Only image files are allowed!",
  "error": "Bad Request"
}
```

□ **ตัวอย่างที่ 6: Limit File Size + DTO Validation**

☐ **src/upload/upload.controller.ts**

```
import {
  Controller,
  Post,
  UploadedFile,
  UseInterceptors,
} from '@nestjs/common';
import { FileInterceptor } from '@nestjs/platform-express';

@Controller('upload')
export class UploadController {
  @Post('limited')
  @UseInterceptors(
    FileInterceptor('file', {
      limits: {
        fileSize: 2 * 1024 * 1024, // 2MB
      },
    }),
  )
  uploadLimited(@UploadedFile() file: Express.Multer.File) {
    return {
      message: 'File within size limit',
      size: file.size,
    };
  }
}
```

☐ ถ้าอัปโหลดไฟล์เกิน 2MB ⟶ จะถูก block โดย Multer

☐ **สรุปจุดสำคัญ**

ฟีเจอร์	ตัวอย่าง
Single Upload	ตัวอย่าง 1
Multiple Upload	ตัวอย่าง 2
Fields Upload	ตัวอย่าง 3

ฟีเจอร์	ตัวอย่าง
Custom Storage	ตัวอย่าง 4
File Filter	ตัวอย่าง 5
Limit Size	ตัวอย่าง 6

ตัวอย่าง “Ultimate” (File Upload + DTO Validation + Save ข้อมูลลง Database + Serve Static Files)

☐ ใช้ NestJS + Multer + TypeORM (SQLite)

☐ มีการ validate ประเภทไฟล์, จำกัดขนาด, ตั้งชื่อไฟล์อัตโนมัติ, และบันทึก metadata ลงฐานข้อมูล

☐ **โครงสร้างโปรเจกต์**

```
nestjs-file-upload-ultimate/
├── src/
│   ├── app.module.ts
│   ├── main.ts
│   ├── upload/
│   │   ├── upload.module.ts
│   │   ├── upload.controller.ts
│   │   ├── upload.service.ts
│   │   ├── dto/
│   │   │   └── create-file.dto.ts
│   │   └── entities/
│   │       └── file.entity.ts
├── uploads/
│   └── (ไฟล์จริง)
├── package.json
├── tsconfig.json
└── nest-cli.json
```

☐ **ขั้นตอนที่ 1: ติดตั้ง Library ที่จำเป็น**

```
npm install @nestjs/platform-express multer
npm install @nestjs/typeorm typeorm sqlite3
```

```
npm install class-validator class-transformer
```

☐ **ขั้นตอนที่ 2: สร้าง Entity สำหรับเก็บข้อมูลไฟล์**

☐ **src/upload/entities/file.entity.ts**

```
import { Entity, PrimaryGeneratedColumn, Column } from 'typeorm';

@Entity('files')
export class FileEntity {
  @PrimaryGeneratedColumn()
  id: number;

  @Column()
  originalName: string;

  @Column()
  filename: string;

  @Column()
  mimetype: string;

  @Column()
  size: number;

  @Column()
  path: string;

  @Column({ default: () => 'CURRENT_TIMESTAMP' })
  createdAt: string;
}
```

☐ **ขั้นตอนที่ 3: DTO สำหรับรับข้อมูล**

☐ **src/upload/dto/create-file.dto.ts**

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

```
export class CreateFileDto {
  @IsString()
  @IsNotEmpty()
  description: string;
}
```

☐ **ขั้นตอนที่ 4: UploadService**

☐ **src/upload/upload.service.ts**

```
import { Injectable } from '@nestjs/common';
import { InjectRepository } from '@nestjs/typeorm';
import { Repository } from 'typeorm';
import { FileEntity } from './entities/file.entity';
import { CreateFileDto } from './dto/create-file.dto';

@Injectable()
export class UploadService {
  constructor(
    @InjectRepository(FileEntity)
    private fileRepo: Repository<FileEntity>,
  ) {}

  async saveFileInfo(file: Express.Multer.File, dto: CreateFileDto) {
    const newFile = this.fileRepo.create({
      originalName: file.originalname,
      filename: file.filename,
      mimetype: file.mimetype,
      size: file.size,
      path: file.path,
    });
    return this.fileRepo.save(newFile);
  }

  async findAll() {
    return this.fileRepo.find();
```

```
  }
}
```

☐ **ขั้นตอนที่ 5: UploadController**

☐ **src/upload/upload.controller.ts**

```
import {
  BadRequestException,
  Body,
  Controller,
  Get,
  Post,
  UploadedFile,
  UseInterceptors,
  UsePipes,
  ValidationPipe,
} from '@nestjs/common';
import { FileInterceptor } from '@nestjs/platform-express';
import { diskStorage } from 'multer';
import { extname } from 'path';
import { CreateFileDto } from './dto/create-file.dto';
import { UploadService } from './upload.service';

@Controller('upload')
export class UploadController {
  constructor(private readonly uploadService: UploadService) {}

  @Post()
  @UseInterceptors(
    FileInterceptor('file', {
      storage: diskStorage({
        destination: './uploads',
        filename: (req, file, cb) => {
          const uniqueSuffix = Date.now() + '-' + Math.round(Math.random() * 1e9);
          const ext = extname(file.originalname);
```

```
          cb(null, `file-${uniqueSuffix}${ext}`);
        },
      }),
      fileFilter: (req, file, cb) => {
        if (!file.mimetype.match(/\/(jpg|jpeg|png|pdf)$/)) {
          return cb(new BadRequestException('Only JPG, PNG, or PDF allowed'), false);
        }
        cb(null, true);
      },
      limits: {
        fileSize: 5 * 1024 * 1024, // 5MB
      },
    }),
  )
  @UsePipes(new ValidationPipe({ transform: true }))
  async uploadFile(
    @UploadedFile() file: Express.Multer.File,
    @Body() dto: CreateFileDto,
  ) {
    if (!file) {
      throw new BadRequestException('No file uploaded');
    }
    const saved = await this.uploadService.saveFileInfo(file, dto);
    return { message: 'File uploaded successfully', file: saved };
  }

  @Get()
  async getAll() {
    return this.uploadService.findAll();
  }
}
```

☐ **ขั้นตอนที่ 6: UploadModule**

☐ **src/upload/upload.module.ts**

```
import { Module } from '@nestjs/common';
import { TypeOrmModule } from '@nestjs/typeorm';
import { UploadController } from './upload.controller';
import { UploadService } from './upload.service';
import { FileEntity } from './entities/file.entity';

@Module({
  imports: [TypeOrmModule.forFeature([FileEntity])],
  controllers: [UploadController],
  providers: [UploadService],
})
export class UploadModule {}
```

☐ **ขั้นตอนที่ 7: AppModule + TypeORM Config**

☐ **src/app.module.ts**

```
import { Module } from '@nestjs/common';
import { TypeOrmModule } from '@nestjs/typeorm';
import { UploadModule } from './upload/upload.module';

@Module({
  imports: [
    TypeOrmModule.forRoot({
      type: 'sqlite',
      database: 'db.sqlite',
      entities: [__dirname + '/**/*.entity{.ts,.js}'],
      synchronize: true,
    }),
    UploadModule,
  ],
})
export class AppModule {}
```

☐ **ขั้นตอนที่ 8: Serve Static Files**

ให้แสดงไฟล์ที่อัปโหลดผ่าน URL เช่น http://localhost:3000/uploads/xxx.png

src/main.ts

```
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';
import { join } from 'path';
import { NestExpressApplication } from '@nestjs/platform-express';

async function bootstrap() {
  const app = await NestFactory.create<NestExpressApplication>(AppModule);

  app.useStaticAssets(join(__dirname, '..', 'uploads'), {
    prefix: '/uploads/',
  });

  await app.listen(3000);
}
bootstrap();
```

▶ ผลการรัน

รันเซิร์ฟเวอร์

```
npm run start:dev
```

POST ⟶ http://localhost:3000/upload

- Body: form-data
  - Key: file ⟶ File ⟶ เลือกภาพหรือ PDF
  - Key: description ⟶ Text ⟶ "My profile picture"

Response

```
{
  "message": "File uploaded successfully",
  "file": {
    "id": 1,
    "originalName": "mypic.png",
    "filename": "file-1710000000000-123456789.png",
    "mimetype": "image/png",
    "size": 183254,
    "path": "uploads/file-1710000000000-123456789.png",
```

```
    "createdAt": "2025-10-14 07:00:00"
  }
}
```

GET ⟶ http://localhost:3000/upload

☐ จะได้รายการไฟล์ทั้งหมดที่อยู่ในฐานข้อมูล

เปิด URL ⟶

http://localhost:3000/uploads/file-1710000000000-123456789.png

☐ จะเห็นไฟล์ถูกเสิร์ฟออกมา

☐ **สิ่งที่ตัวอย่าง Ultimate ครอบคลุม**

ฟีเจอร์	สถานะ
Single file upload	☐
Custom storage + filename	☐
File filter (jpg/png/pdf)	☐
Size limit	☐
DTO + ValidationPipe	☐
TypeORM Entity & Repository	☐
Save file metadata to DB	☐
Serve static files	☐
Multiple endpoint (upload/list)	☐

การ Upload ไฟล์หลายประเภท (Multiple File Types)

การ Upload ไฟล์หลายประเภท (Multiple File Types) เป็นหัวใจสำคัญของระบบอัปโหลดที่ใช้งานจริง โดยเฉพาะในแอปที่ต้องอัปโหลดทั้งภาพ เอกสาร วิดีโอ หรือไฟล์อื่น ๆ พร้อมกัน

ใน NestJS เราสามารถจัดการได้หลายวิธี เช่น:

☐ **ภาพรวมของแนวทางการอัปโหลดหลายประเภท**

แนวทาง	รายละเอียด	เหมาะสำหรับ
1. **หลาย endpoint แยกประเภท**	เช่น /upload/image, /upload/pdf	แยกการจัดการแต่ละประเภทชัดเจน
2. **ใช้ FileFieldsInterceptor**	อัปโหลดหลายประเภทใน **request เดียว**	ฟอร์มที่ส่งภาพ+ไฟล์

แนวทาง	รายละเอียด	เหมาะสำหรับ
	โดยใช้ field name คนละชื่อ	พร้อมกัน
3. **ใช้ fileFilter + mimetype**	ตรวจสอบประเภทไฟล์ใน runtime	ตรวจสอบความถูกต้องของไฟล์
4. **Custom Storage แยกโฟลเดอร์ตามประเภท**	เช่น รูปเก็บใน uploads/images/, เอกสารเก็บใน uploads/docs/	จัดระเบียบไฟล์ในระบบ

□ ตัวอย่างที่ 1: แยก Endpoint ตามประเภทไฟล์

ในบางกรณี เราอาจอยากให้ผู้ใช้เรียก API คนละตัวเวลาอัปโหลด เช่น /upload/image สำหรับรูป และ /upload/document สำหรับเอกสาร

□ **upload.controller.ts**

```
import {
  BadRequestException,
  Controller,
  Post,
  UploadedFile,
  UseInterceptors,
} from '@nestjs/common';
import { FileInterceptor } from '@nestjs/platform-express';

@Controller('upload')
export class UploadController {
  // □ สำหรับรูปภาพเท่านั้น
  @Post('image')
  @UseInterceptors(
    FileInterceptor('file', {
      fileFilter: (req, file, cb) => {
        if (!file.mimetype.match(/\/(jpg|jpeg|png)$/)) {
          return cb(new BadRequestException('Only image files are allowed!'), false);
        }
        cb(null, true);
      },
    }),
```

```
  )
  uploadImage(@UploadedFile() file: Express.Multer.File) {
    return {
      type: 'image',
      name: file.originalname,
      mimetype: file.mimetype,
    };
  }

  // □ สำหรับเอกสารเท่านั้น
  @Post('document')
  @UseInterceptors(
    FileInterceptor('file', {
      fileFilter: (req, file, cb) => {
        if (!file.mimetype.match(/\/(pdf|msword|vnd.openxmlformats-
officedocument.wordprocessingml.document)$/)) {
          return cb(new BadRequestException('Only document files are allowed!'), false);
        }
        cb(null, true);
      },
    }),
  )
  uploadDocument(@UploadedFile() file: Express.Multer.File) {
    return {
      type: 'document',
      name: file.originalname,
      mimetype: file.mimetype,
    };
  }
}
```

□ **ผลลัพธ์**

- /upload/image ⟶ ยอมรับเฉพาะ .jpg, .jpeg, .png
- /upload/document ⟶ ยอมรับเฉพาะ .pdf, .doc, .docx

 ตัวอย่างที่ 2: อัปโหลดหลายประเภทใน Request เดียว (FileFieldsInterceptor)

ถ้าฟอร์มมีทั้ง avatar (รูปโปรไฟล์) และ documents (ไฟล์แนบ) ในการส่งข้อมูลครั้งเดียว ⟶ ใช้ FileFieldsInterceptor

 upload.controller.ts

```
import {
  Controller,
  Post,
  UploadedFiles,
  UseInterceptors,
} from '@nestjs/common';
import { FileFieldsInterceptor } from '@nestjs/platform-express';

@Controller('upload')
export class UploadController {
  @Post('multi-type')
  @UseInterceptors(
    FileFieldsInterceptor([
      { name: 'avatar', maxCount: 1 },
      { name: 'documents', maxCount: 3 },
    ]),
  )
  uploadMultipleTypes(
    @UploadedFiles()
    files: {
      avatar?: Express.Multer.File[];
      documents?: Express.Multer.File[];
    },
  ) {
    return {
      avatar: files.avatar?.map((f) => ({
        name: f.originalname,
        mimetype: f.mimetype,
      })),
      documents: files.documents?.map((f) => ({
```

```
        name: f.originalname,
        mimetype: f.mimetype,
      })),
    };
  }
}
```

วิธีทดสอบ

- POST ⟶ /upload/multi-type
- form-data:
  - avatar ⟶ File ⟶ รูปภาพ
  - documents ⟶ File ⟶ แนบ 2-3 ไฟล์

ผลลัพธ์

```
{
  "avatar": [
    { "name": "profile.jpg", "mimetype": "image/jpeg" }
  ],
  "documents": [
    { "name": "cv.pdf", "mimetype": "application/pdf" },
    { "name": "certificate.png", "mimetype": "image/png" }
  ]
}
```

ตัวอย่างที่ 3: Custom Storage แยกโฟลเดอร์

คุณสามารถกำหนดให้ภาพถูกบันทึกไว้ใน /uploads/images ส่วนเอกสารบันทึกใน /uploads/docs โดยดูจาก mimetype

upload.controller.ts

```
import {
  BadRequestException,
  Controller,
  Post,
  UploadedFile,
  UseInterceptors,
} from '@nestjs/common';
import { FileInterceptor } from '@nestjs/platform-express';
```

```
import { diskStorage } from 'multer';
import { extname } from 'path';
import * as fs from 'fs';

@Controller('upload')
export class UploadController {
  @Post('auto-folder')
  @UseInterceptors(
    FileInterceptor('file', {
      storage: diskStorage({
        destination: (req, file, cb) => {
          let folder = './uploads/others';
          if (file.mimetype.match(/\/(jpg|jpeg|png)$/)) {
            folder = './uploads/images';
          } else if (file.mimetype.match(/\/(pdf|msword|vnd.openxmlformats-
officedocument.wordprocessingml.document)$/)) {
            folder = './uploads/docs';
          }
          // สร้างโฟลเดอร์ถ้ายังไม่มี
          if (!fs.existsSync(folder)) {
            fs.mkdirSync(folder, { recursive: true });
          }
          cb(null, folder);
        },
        filename: (req, file, cb) => {
          const uniqueSuffix = Date.now() + '-' + Math.round(Math.random() * 1e9);
          cb(null, `${file.fieldname}-${uniqueSuffix}${extname(file.originalname)}`);
        },
      }),
      fileFilter: (req, file, cb) => {
        if (!file.mimetype.match(/\/(jpg|jpeg|png|pdf|msword|vnd.openxmlformats-
officedocument.wordprocessingml.document)$/)) {
          return cb(new BadRequestException('Unsupported file type'), false);
        }
```

```
        cb(null, true);
      },
    }),
  )
  uploadWithAutoFolder(@UploadedFile() file: Express.Multer.File) {
    return {
      message: 'File saved in folder based on type',
      path: file.path,
      mimetype: file.mimetype,
    };
  }
}
```

☐ ตัวอย่างผลลัพธ์

```
{
  "message": "File saved in folder based on type",
  "path": "uploads/images/file-1710000000000-123456789.png",
  "mimetype": "image/png"
}
```

☐ **ข้อดีของแนวทางหลายประเภท**

☐ **ความชัดเจนของ API** ⟶ แยกตามประเภทไฟล์

☐ **ลดโอกาส error** ⟶ ตรวจสอบ mimetype และเก็บแยกโฟลเดอร์

☐ **รองรับการอัปโหลดหลายประเภทพร้อมกัน** ⟶ ด้วย FileFieldsInterceptor

☐ **พร้อมต่อยอด** ⟶ สามารถเชื่อมกับ DB บันทึกประเภทไฟล์และ path ได้ทันที

☐ **สรุปตารางฟีเจอร์**

แนวทาง	ใช้ Decorator	ประเภท	ข้อดี
แยก Endpoint	@FileInterceptor	ทีละประเภท	API ชัดเจน, ง่าย
FileFieldsInterceptor	@FileFieldsInterceptor	หลายประเภทใน 1 request	เหมาะกับฟอร์มที่ส่งหลาย field
Custom Storage	diskStorage	Dynamic path	จัดระเบียบไฟล์ในระบบ

ตัวอย่าง **NestJS – File Uploading และ Multer** แบ่งเป็น

- ☐ **ตัวอย่างพื้นฐาน 3 โปรแกรม** ⟶ Single / Multiple / Fields Upload
- ☐ **ตัวอย่างแนวประยุกต์ 3 โปรแกรม** ⟶ Custom Storage / File Filter & Validation / Limit Size + DB integration

---

☐ โครงสร้างโปรเจกต์หลัก (ใช้ร่วมกันทุกตัวอย่าง)

```
nestjs-file-upload/
├── src/
│   ├── app.module.ts
│   ├── main.ts
│   └── upload/
│       ├── upload.module.ts
│       ├── upload.controller.ts
│       └── upload.service.ts
├── uploads/
│   └── (ไฟล์อัปโหลด)
├── package.json
├── tsconfig.json
└── nest-cli.json
```

☐ ตัวอย่างพื้นฐาน (Basic Examples)

☐ ตัวอย่างที่ 1: Single File Upload

src/upload/upload.controller.ts

```
import {
  Controller,
  Post,
  UploadedFile,
  UseInterceptors,
} from '@nestjs/common';
import { FileInterceptor } from '@nestjs/platform-express';

@Controller('upload')
export class UploadController {
```

```
  @Post('single')
  @UseInterceptors(FileInterceptor('file'))
  uploadSingle(@UploadedFile() file: Express.Multer.File) {
    return {
      message: 'Upload success',
      filename: file.originalname,
      size: file.size,
    };
  }
}
```

---

src/upload/upload.module.ts

```
import { Module } from '@nestjs/common';
import { UploadController } from './upload.controller';
import { UploadService } from './upload.service';

@Module({
  controllers: [UploadController],
  providers: [UploadService],
})
export class UploadModule {}
```

---

src/upload/upload.service.ts

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

@Injectable()
export class UploadService {}
```

---

src/app.module.ts

```
import { Module } from '@nestjs/common';
import { UploadModule } from './upload/upload.module';

@Module({
  imports: [UploadModule],
```

```
})
export class AppModule {}
```

---

src/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**

- POST ⟶ http://localhost:3000/upload/single
- Body ⟶ form-data ⟶ Key: file ⟶ เลือกไฟล์

```
{
  "message": "Upload success",
  "filename": "test.png",
  "size": 31245
}
```

ตัวอย่างที่ 2: Multiple Files Upload

upload.controller.ts (เพิ่ม method)

```
import { FilesInterceptor } from '@nestjs/platform-express';
import { UploadedFiles } from '@nestjs/common';

@Post('multiple')
@UseInterceptors(FilesInterceptor('files', 5))
uploadMultiple(@UploadedFiles() files: Express.Multer.File[]) {
  return {
    message: 'Multiple upload success',
    count: files.length,
    files: files.map((f) => f.originalname),
```

```
  };
}
```

▶ **ผลการรัน**

- POST ⟶ /upload/multiple
- Key: files ⟶ เลือกหลายไฟล์

```
{
  "message": "Multiple upload success",
  "count": 3,
  "files": ["img1.jpg", "img2.jpg", "doc.pdf"]
}
```

ตัวอย่างที่ 3: Fields Upload (หลาย key)

upload.controller.ts (เพิ่ม method)

```
import { FileFieldsInterceptor } from '@nestjs/platform-express';
import { UploadedFiles } from '@nestjs/common';

@Post('fields')
@UseInterceptors(
  FileFieldsInterceptor([
    { name: 'avatar', maxCount: 1 },
    { name: 'documents', maxCount: 3 },
  ]),
)
uploadFields(@UploadedFiles() files: { avatar?: Express.Multer.File[]; documents?:
Express.Multer.File[] }) {
  return {
    avatar: files.avatar?.[0]?.originalname,
    documents: files.documents?.map((d) => d.originalname),
  };
}
```

▶ **ผลการรัน**

- POST ⟶ /upload/fields

- avatar ⟶ File (1)
- documents ⟶ File (หลายไฟล์)

```
{
  "avatar": "profile.jpg",
  "documents": ["cv.pdf", "certificate.png"]
}
```

☐ ตัวอย่างแนวประยุกต์ (Advanced Examples)

☐ ตัวอย่างที่ 4: Custom Storage + Filename

```
import { diskStorage } from 'multer';
import { extname } from 'path';

@Post('custom-storage')
@UseInterceptors(
  FileInterceptor('file', {
    storage: diskStorage({
      destination: './uploads',
      filename: (req, file, cb) => {
        const uniqueSuffix = Date.now() + '-' + Math.round(Math.random() * 1e9);
        const ext = extname(file.originalname);
        cb(null, `file-${uniqueSuffix}${ext}`);
      },
    }),
  }),
)
uploadWithCustomStorage(@UploadedFile() file: Express.Multer.File) {
  return {
    message: 'File saved',
    path: file.path,
  };
}
```

☐ ผลลัพธ์

```
{
```

```
  "message": "File saved",
  "path": "uploads/file-1710000000000-123456789.png"
}
```

☐ **ตัวอย่างที่ 5: File Filter + Validation**

```
@Post('image-only')
@UseInterceptors(
  FileInterceptor('file', {
    fileFilter: (req, file, cb) => {
      if (!file.mimetype.match(/\/(jpg|jpeg|png)$/)) {
        return cb(new BadRequestException('Only image files are allowed!'), false);
      }
      cb(null, true);
    },
  }),
)
uploadImageOnly(@UploadedFile() file: Express.Multer.File) {
  return {
    message: 'Image upload success',
    filename: file.originalname,
  };
}
```

☐ หากอัปโหลดไฟล์ไม่ใช่ภาพ ⟶ Response 400

☐ **ตัวอย่างที่ 6: Limit File Size + Database Integration (Ultimate)**

Entity: file.entity.ts

```
import { Entity, PrimaryGeneratedColumn, Column } from 'typeorm';

@Entity('files')
export class FileEntity {
  @PrimaryGeneratedColumn()
  id: number;
  @Column() originalName: string;
  @Column() filename: string;
```

```
  @Column() mimetype: string;
  @Column() size: number;
  @Column() path: string;
  @Column({ default: () => 'CURRENT_TIMESTAMP' }) createdAt: string;
}
```

Service: upload.service.ts

```
@Injectable()
export class UploadService {
  constructor(@InjectRepository(FileEntity) private fileRepo: Repository<FileEntity>) {}

  async saveFile(file: Express.Multer.File) {
    const entity = this.fileRepo.create({
      originalName: file.originalname,
      filename: file.filename,
      mimetype: file.mimetype,
      size: file.size,
      path: file.path,
    });
    return this.fileRepo.save(entity);
  }

  async findAll() {
    return this.fileRepo.find();
  }
}
```

Controller: upload.controller.ts

```
@Post('limited')
@UseInterceptors(
  FileInterceptor('file', {
    limits: { fileSize: 2 * 1024 * 1024 },
    storage: diskStorage({
      destination: './uploads',
      filename: (req, file, cb) => {
        const uniqueSuffix = Date.now() + '-' + Math.round(Math.random() * 1e9);
```

```
        cb(null, `file-${uniqueSuffix}${extname(file.originalname)}`);
      },
    }),
  }),
)
async uploadLimited(@UploadedFile() file: Express.Multer.File) {
  if (!file) throw new BadRequestException('No file uploaded');
  const saved = await this.uploadService.saveFile(file);
  return { message: 'File uploaded & saved to DB', file: saved };
}
```

▶ **ผลการรัน Ultimate**

- POST ⟶ /upload/limited
- File ≤ 2MB ⟶ บันทึกไฟล์ + ข้อมูลลง DB
- GET ⟶ /upload ⟶ แสดงรายการไฟล์ทั้งหมด

```
{
  "message": "File uploaded & saved to DB",
  "file": {
    "id": 1,
    "originalName": "mypic.png",
    "filename": "file-1710000000000-123456789.png",
    "mimetype": "image/png",
    "size": 183254,
    "path": "uploads/file-1710000000000-123456789.png",
    "createdAt": "2025-10-14 07:00:00"
  }
}
```

Ultimate 2.0 – Multi-type Upload + Multiple Files + Validation + DB + Serve Static + Dynamic Folder แบบ **Production-ready** ให้ครบทุกฟีเจอร์

โครงสร้างโปรเจกต์ Ultimate 2.0

```
nestjs-file-upload-ultimate/
├── src/
```

```
│   ├── app.module.ts
│   ├── main.ts
│   └── upload/
│       ├── upload.module.ts
│       ├── upload.controller.ts
│       ├── upload.service.ts
│       ├── dto/
│       │   └── create-file.dto.ts
│       └── entities/
│           └── file.entity.ts
├── uploads/
│   ├── images/
│   ├── documents/
│   └── others/
├── package.json
├── tsconfig.json
└── nest-cli.json
```

---

1 Entity – file.entity.ts

```
import { Entity, PrimaryGeneratedColumn, Column } from 'typeorm';

@Entity('files')
export class FileEntity {
  @PrimaryGeneratedColumn()
  id: number;

  @Column()
  originalName: string;

  @Column()
  filename: string;

  @Column()
```

```
  mimetype: string;

  @Column()
  size: number;

  @Column()
  path: string;

  @Column()
  type: string; // image, document, other

  @Column({ default: () => 'CURRENT_TIMESTAMP' })
  createdAt: string;
}
```

---

2 DTO – create-file.dto.ts

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

export class CreateFileDto {
  @IsString()
  @IsOptional()
  description?: string;
}
```

---

3 Service – upload.service.ts

```
import { Injectable } from '@nestjs/common';
import { InjectRepository } from '@nestjs/typeorm';
import { Repository } from 'typeorm';
import { FileEntity } from './entities/file.entity';

@Injectable()
export class UploadService {
  constructor(
    @InjectRepository(FileEntity)
```

```
    private fileRepo: Repository<FileEntity>,
  ) {}

  async saveFile(file: Express.Multer.File, type: string) {
    const entity = this.fileRepo.create({
      originalName: file.originalname,
      filename: file.filename,
      mimetype: file.mimetype,
      size: file.size,
      path: file.path,
      type,
    });
    return this.fileRepo.save(entity);
  }

  async findAll() {
    return this.fileRepo.find();
  }

  async deleteFile(id: number) {
    const file = await this.fileRepo.findOneBy({ id });
    if (!file) return null;
    const fs = require('fs');
    if (fs.existsSync(file.path)) fs.unlinkSync(file.path);
    return this.fileRepo.remove(file);
  }
}
```

4 Controller – upload.controller.ts

```
import {
  Controller,
  Post,
  UploadedFiles,
  UseInterceptors,
```

```
  Body,
  Get,
  Delete,
  Param,
  BadRequestException,
} from '@nestjs/common';
import { FileFieldsInterceptor } from '@nestjs/platform-express';
import { diskStorage } from 'multer';
import { extname } from 'path';
import { UploadService } from './upload.service';
import { CreateFileDto } from './dto/create-file.dto';
import * as fs from 'fs';

@Controller('upload')
export class UploadController {
  constructor(private readonly uploadService: UploadService) {}

  @Post('multi')
  @UseInterceptors(
    FileFieldsInterceptor([
      { name: 'images', maxCount: 5 },
      { name: 'documents', maxCount: 5 },
      { name: 'others', maxCount: 5 },
    ], {
      storage: diskStorage({
        destination: (req, file, cb) => {
          let folder = './uploads/others';
          if (file.fieldname === 'images') folder = './uploads/images';
          else if (file.fieldname === 'documents') folder = './uploads/documents';
          if (!fs.existsSync(folder)) fs.mkdirSync(folder, { recursive: true });
          cb(null, folder);
        },
        filename: (req, file, cb) => {
          const uniqueSuffix = Date.now() + '-' + Math.round(Math.random() * 1e9);
```

```
          cb(null, `${file.fieldname}-${uniqueSuffix}${extname(file.originalname)}`);
        },
      }),
      fileFilter: (req, file, cb) => {
        const allowedTypes = {
          images: /\/(jpg|jpeg|png|gif)$/,
          documents: /\/(pdf|msword|vnd.openxmlformats-
officedocument.wordprocessingml.document)$/,
          others: /.*/,
        };
        const regex = allowedTypes[file.fieldname];
        if (!regex.test(file.mimetype)) {
          return cb(new BadRequestException(`Invalid file type for ${file.fieldname}`), false);
        }
        cb(null, true);
      },
      limits: { fileSize: 5 * 1024 * 1024 },
    }),
  )
  async uploadMulti(
    @UploadedFiles() files: {
      images?: Express.Multer.File[];
      documents?: Express.Multer.File[];
      others?: Express.Multer.File[];
    },
    @Body() dto: CreateFileDto,
  ) {
    const result = [];

    for (const type of ['images', 'documents', 'others']) {
      const fileArray = files[type] || [];
      for (const file of fileArray) {
        const saved = await this.uploadService.saveFile(file, type);
        result.push(saved);
```