Cómo no romper tu primer proyecto NestJS

(Guía práctica para trainees)

Gustavo Garozzo

2025

Una guía técnica, profesional y didáctica para sentar bases arquitectónicas sólidas.

Introducción: Los cimientos de la escalabilidad `NestJS`

NestJS es un *framework* progresivo de Node.js diseñado para construir aplicaciones del lado del servidor (APIs RESTful, Microservicios, aplicaciones compatibles con GraphQL) que sean eficientes, confiables y, sobre todo, **escalables**. Está construido y totalmente tipado con `TypeScript`, aprovechando al máximo los principios de la Programación Orientada a Objetos (POO).

Ventajas de NestJS sobre Express. Mientras que `Express` es minimalista, dejando la estructura y la arquitectura a completa discreción del desarrollador (lo que puede llevar al "monolito de espagueti" en proyectos grandes), NestJS impone un conjunto de patrones probados, inspirados en Angular, que garantizan el orden.

Cuadro 1: Comparativa NestJS vs. Express

Característica	Express (Solo)	NestJS
Arquitectura	No impone ninguna.	Impone una arquitectura modular clara (Módulos, Controladores, Servicios).
Lenguaje	JavaScript (opcionalmente TypeScript).	TypeScript por defecto y como principio.
Patrones	Implementación manual y variable.	Adopta la Inyección de Dependencias (IoC) y el Principio de Responsabilidad Única (SRP).
Productividad	Base mínima.	CLI potente, integración nativa de *testing* y herramientas de configuración.

El objetivo de esta guía es dotar al *trainee* no solo de sintaxis, sino de **criterio arquitectónico**. Un desarrollador que sabe dónde va cada pieza de código evita errores que penalizan la mantenibilidad y la escalabilidad del proyecto.

1 Estructura del proyecto: El orden es la mitad de la batalla

Un proyecto NestJS bien estructurado es un proyecto que un nuevo desarrollador puede navegar y entender rápidamente. La clave es la **modularidad**.

1.1 Organización de Módulos, Controladores y Servicios. El núcleo de NestJS son tres componentes que deben operar según el Principio de Responsabilidad Única (SRP). Un **Módulo** encapsula un **Recurso** (ej. `Usuario`) y agrupa su lógica.

1. **Módulo (`@Module`):** Es el contenedor de contexto. Agrupa controladores y servicios que están relacionados. Define las fronteras del recurso y dónde se inyectan las dependencias.

2. **Controlador (`@Controller`):** Es la capa de entrada. Su única responsabilidad es **manejar la petición HTTP**. Debe ser ligero, solo orquestar la llamada al servicio y devolver la respuesta.

3. **Servicio (`@Injectable` / Provider):** Es la capa de negocio. Contiene la **lógica real** de la aplicación (acceso a DB, cálculos, validaciones complejas, integración con terceros).

1.2 Ejemplo de estructura real recomendada. La convención de nombres debe ser consistente: carpetas en *kebab-case* (plural) y archivos en *dot-notation* (singular).

```
src/
 app.module.ts        # Módulo Raíz (solo importa otros módulos)
 main.ts              # Punto de entrada

 common/              # Elementos transversales: Interceptores, Filtros,
    Pipes
|      pipes/
|      filters/

 users/               # Módulo de Recurso (Users)
|      dto/             # Data Transfer Objects (para la forma de la data)
|   |      create-user.dto.ts
|   |      update-user.dto.ts
|   |
|      users.controller.ts # Capa HTTP: @Post('ruta'), @Get('ruta')
|      users.service.ts    # Capa de Negocio: Lógica CRUD
|      users.module.ts     # Agrupa Controller y Service

 products/            # Otro Módulo de Recurso (Products)
     ... (dto, controller, service, module)
```

Listing 1: Estructura de un proyecto NestJS escalable

Recomendación: Evita anidar más de 3 niveles de carpetas. Si lo necesitas, es una señal de que el módulo es demasiado grande y debe ser descompuesto.

2 Principios clave: Patrones que salvan el código

NestJS aplica patrones de diseño que debes entender para evitar trampas.

2.1 Inversión de Dependencias (IoC). El concepto de IoC significa que la clase que requiere una dependencia (*el consumidor*) no es responsable de crearla. El *framework* (el Inyector de Dependencias) lo hace por ti. Esto se logra a través de la inyección por constructor.

```
// users.controller.ts
import { Controller, Get } from '@nestjs/common';
import { UsersService } from './users.service';

@Controller('users')
export class UsersController {
  // CORRECTO: NestJS inyecta la instancia de UsersService
  constructor(private readonly userService: UsersService) {}

  @Get()
  findAll() {
    return this.userService.findAll();
  }
}
```

Listing 2: Uso de la Inyección de Dependencias (IoC)

Error del Trainee: Intentar crear la instancia manualmente con `new UsersService()`. Esto anula el IoC, impide el *testing* y rompe la arquitectura.

2.2 Separación de Responsabilidades (SRP). El SRP en NestJS se traduce en la estricta distinción de roles entre Controladores y Servicios.

- **Controladores:** No deben tener lógica condicional (`if/else`), acceso directo a la base de datos o bucles. Solo deben llamar a un método del servicio y formatear la respuesta HTTP.
- **Servicios:** Deben ser **agnósticos a HTTP**. Solo manejan datos y lógica de negocio. Nunca deben devolver códigos de estado HTTP (ej. `200` o `404`), solo los datos o una excepción.

2.3 Manejo correcto de DTOs y Pipes. Un **DTO (Data Transfer Object)** es una clase que define la **forma** de los datos de entrada (`@Body()`) o salida. Para la validación, utiliza los paquetes `class-validator` y `class-transformer`.

```
// users/dto/create-user.dto.ts
import { IsString, IsEmail, IsNotEmpty, MinLength } from 'class-validator';

export class CreateUserDto {
  @IsString()
  @IsNotEmpty()
  @MinLength(5, { message: 'El nombre debe tener al menos 5 caracteres.' })
  name: string;

  @IsEmail({}, { message: 'El formato del email es inválido.' })
  email: string;
}
```

Listing 3: DTO con validación estricta

El Pipe de Validación (`ValidationPipe`) se usa globalmente en `main.ts` para interceptar la petición **antes** de que llegue al controlador. Si la validación falla, automáticamente lanza un `HTTP 400 Bad Request`.

```
// main.ts
import { ValidationPipe } from '@nestjs/common';
// ...
async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  app.useGlobalPipes(new ValidationPipe({
    whitelist: true,        // Ignora propiedades no definidas en el DTO
    forbidNonWhitelisted: true, // Lanza error si recibe propiedades extra
    transform: true,        // Convierte tipos (ej. string '1' a number 1)
  }));
  await app.listen(3000);
}
```

Listing 4: Configuración global de ValidationPipe

Práctica fundamental: ¡Nunca valides manualmente! Si no usas el `ValidationPipe`, estás rehaciendo el trabajo del *framework* e introduciendo inconsistencias.

3 Errores comunes y cómo evitarlos

Estos son los errores que transforman rápidamente un proyecto en un dolor de cabeza.

3.1 Abuso de controladores ("Controlador-Monolito"). El controlador comienza a crecer sin control, mezclando recursos o responsabilidades.

```
//   INCORRECTO: Lógica de negocio y validación en el Controller
@Post()
async create(@Body() data) {
  // Lógica de negocio compleja:
  const activeUsers = await this.userService.countActiveUsers();
  if (activeUsers >= 10) {
    throw new ForbiddenException('Límite de usuarios activos alcanzado.');
  }

  // Acceso a la base de datos:
  const newUser = await this.userRepository.save(data);
  return newUser;
}
```

Listing 5: Ejemplo de Abuso de Controlador (Lógica de negocio)

El refactor: Mover la lógica de negocio al Servicio.
El servicio debe ser el único lugar que contenga la lógica y las reglas de negocio.

```
//   CORRECTO: La lógica vive aquí, agnóstica a HTTP
async create(createUserDto: CreateUserDto) {
  const activeUsers = await this.countActiveUsers(); // Llamada a su propio
      método
  if (activeUsers >= 10) {
    // El servicio lanza una excepción HTTP, NestJS la captura
    throw new ForbiddenException('Límite de usuarios activos alcanzado.');
  }
  return this.userRepository.save(createUserDto);
}
```

Listing 6: users.service.ts (Lugar Correcto para la lógica)

```
//   CORRECTO: El Controller solo llama al servicio
@Post()
async create(@Body() createUserDto: CreateUserDto) {
  return this.userService.create(createUserDto);
}
```

Listing 7: users.controller.ts (Solo Orquestación)

3.2 No validar datos correctamente. **El Error:** Confiar en validaciones manuales (`if`) o en la base de datos. **La Solución:** Siempre delega la validación de la estructura y el formato a los **DTOs y Pipes**. Esto es una defensa temprana y consistente.

4 Buenas prácticas generales para un código profesional

El profesionalismo se mide en cómo manejamos los aspectos que no son de negocio.

4.1 Configuración de variables de entorno. **El Error:** Incluir secretos y configuración de entorno (claves de API, URLs de bases de datos, `JWT_SECRET`) directamente en el código fuente.

La Solución: Utilizar el módulo `@nestjs/config`. Esto inyecta variables de archivos `.env` de forma segura y tipada en tu aplicación.

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

@Injectable()
export class AuthService {
  private readonly jwtSecret: string;

  constructor(private configService: ConfigService) {
    // Acceso seguro y tipado a la variable de entorno
    this.jwtSecret = this.configService.get<string>('JWT_SECRET');

    if (!this.jwtSecret) {
        // Validación en tiempo de ejecución (importante)
        throw new Error('JWT_SECRET no está definido en el entorno.');
    }
  }
  // ...
}
```

Listing 8: Uso de ConfigService

4.2 Manejo de excepciones (Excepciones HTTP). NestJS provee excepciones HTTP estándar. Usa siempre estas clases para errores que deben ser devueltos al cliente (ej. `404`, `403`, `409`).

```
import { NotFoundException, ConflictException } from '@nestjs/common';

async findProduct(id: number) {
  const product = await this.productRepository.findOne(id);

  if (!product) {
    // NestJS la transforma en HTTP 404 (Not Found)
    throw new NotFoundException(`Producto con ID ${id} no encontrado.`);
  }
  return product;
}
```

Listing 9: Lanzamiento de Excepciones HTTP estándar

Nunca devuelvas un error genérico (HTTP 500) por algo que el cliente hizo mal.

4.3 Logging y Validaciones. **Logging:** Utiliza el servicio `Logger` de NestJS, y **nunca** `console.log()`. El `Logger` permite controlar niveles de severidad y contexto (fundamental para la observabilidad).

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

```
@Injectable()
export class ProductsService {
  // Inyectar el contexto para saber dónde ocurrió el evento
  private readonly logger = new Logger(ProductsService.name);

  async update(id: number, updateDto: any) {
    this.logger.log(`Intentando actualizar producto ID: ${id}`);
    // ...
    this.logger.warn(`El producto ID ${id} tiene una advertencia de stock
        bajo.`);
  }
}
```

Listing 10: Uso del Logger de NestJS

4.4 Testing Básico. El *testing* es la garantía de que el código no se romperá. Gracias a la Inyección de Dependencias, NestJS hace que el *testing* unitario de servicios sea sencillo.

- **Test Unitario:** Aísla la lógica de negocio (Servicios) y **simula (*mockea*)** sus dependencias (ej. el repositorio de base de datos).

```
// users.service.spec.ts
import { Test } from '@nestjs/testing';
import { UsersService } from './users.service';

describe('UsersService', () => {
  let service: UsersService;

  // 1. Simular la dependencia (ej. un Repositorio o una DB)
  const mockUserRepository = {
    save: jest.fn(dto => ({ id: 1, ...dto })), // Mockear el resultado de
        save
    findAll: jest.fn(() => [{ id: 1, email: 'test@mail.com' }]),
  };

  beforeEach(async () => {
    const module = await Test.createTestingModule({
      providers: [
        UsersService,
        {
          // 2. Proporcionar el mock al Inyector de Dependencias
          provide: 'UserRepository',
          useValue: mockUserRepository,
        },
      ],
    }).compile();

    service = module.get<UsersService>(UsersService);
  });

  it('debe crear un usuario y llamar al repositorio.save', async () => {
    const dto = { name: 'Gustavo', email: 'g@test.com' };
    const user = await service.create(dto);
    expect(user.name).toEqual(dto.name);
    // 3. Verificar que la función mockeada fue llamada
    expect(mockUserRepository.save).toHaveBeenCalled();
```

```
  });
});
```

Listing 11: Test Unitario Básico de un Servicio (Jest)

Conclusión: El camino del desarrollador profesional

Has recorrido los cimientos arquitectónicos que distinguen un proyecto mantenible de un "monolito de espagueti". Como *trainee*, comprender **dónde** va el código es más importante que saber **cómo** escribir la sintaxis.

Resumen de los aprendizajes clave.

- **Separación Estricta:** Controladores (HTTP) $\neq$ Servicios (Negocio).
- **IoC:** Siempre inyecta dependencias por constructor. Nunca uses `new`.
- **Defensa:** Usa DTOs y el `ValidationPipe` global como tu primera línea de defensa de datos.
- **Configuración:** Externaliza secretos con `@nestjs/config`.
- **Testing:** Los servicios deben ser testeables y agnósticos a HTTP.

Recomendaciones para seguir mejorando. El siguiente paso en tu curva de aprendizaje debe enfocarse en los componentes transversales:

1. Dominar los **Guards** (para autorización y permisos).
2. Implementar **Interceptors** (para transformación de respuestas y *logging* de tiempos de ejecución).
3. Profundizar en **TypeORM o Prisma** (para acceso robusto y tipado a la base de datos).

La experiencia de un desarrollador real se basa en la **simplicidad** y la **predictibilidad**. Siempre cuestiona si el código que estás escribiendo es fácil de testear y entender para la persona que venga después de ti.

Créditos

Autor: Gustavo Garozzo
Año: 2025