

## **Atividade 03 | Explorando Conceitos do NestJS com IA e Ferramentas Visuais**

Express Endpoint To NestJS Controller >

Código-base:  
app.post('/usuarios', (req, res) => {  
 const novoUsuario = {  
 id: usuarios.length + 1,  
 nome: req.body.nome  
 };  
 usuarios.push(novoUsuario);  
 res.status(201).json(novoUsuario);  
});  
  
Pergunte para uma IA (ChatGPT):  
Pergunta: "Como esse código em Express seria implementado com NestJS?"

To show you exactly how this Express endpoint maps to your existing NestJS project, I'll open your current `users.controller.ts` and `users.module.ts` to align with the structure already in your codebase.

Read `users.controller.ts`  
Read `users.module.ts`  
Read `users.service.spec.ts`

Aqui está como implementar o seu endpoint Express no NestJS, de forma equivalente.

Express original:

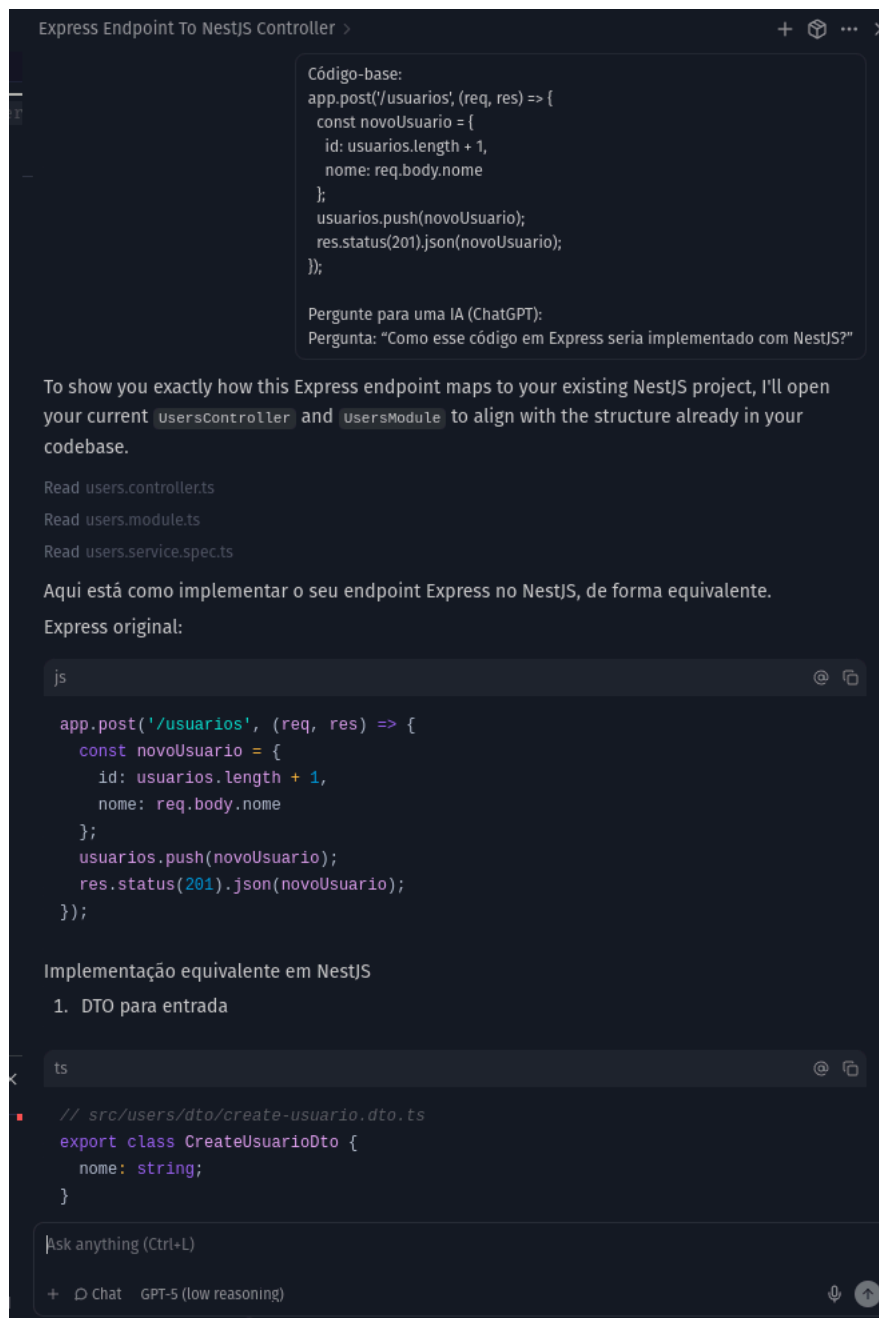
```
js  
  
app.post('/usuarios', (req, res) => {  
  const novoUsuario = {  
    id: usuarios.length + 1,  
    nome: req.body.nome  
  };  
  usuarios.push(novoUsuario);  
  res.status(201).json(novoUsuario);  
});
```

Implementação equivalente em NestJS

1. DTO para entrada

```
ts  
  
// src/users/dto/create-usuario.dto.ts  
export class CreateUsuarioDto {  
  nome: string;  
}
```

Ask anything (Ctrl+L)



#### ▼ Pergunta e resposta da IA

Código-base:

```
app.post('/usuarios', (req, res) => {
  const novoUsuario = {
    id: usuarios.length + 1,
    nome: req.body.nome
  };
  usuarios.push(novoUsuario);
  res.status(201).json(novoUsuario);
});
```

Pergunte para uma IA (ChatGPT):

Pergunta: "Como esse código em Express seria implementado com NestJS?"

To show you exactly how this Express endpoint maps to your existing NestJS project, I'll open your current `UsersController` and `UsersModule` to align with the structure already in your codebase.

Aqui está como implementar o seu endpoint Express no NestJS, de forma equivalente.

Express original:

```
app.post('/usuarios', (req, res) => {  
  const novoUsuario = {  
    id: usuarios.length + 1,  
    nome: req.body.nome  
  };  
  usuarios.push(novoUsuario);  
  res.status(201).json(novoUsuario);  
});
```

Implementação equivalente em NestJS

#### 1. DTO para entrada

```
// src/users/dto/create-usuario.dto.ts  
export class CreateUsuarioDto {  
  nome: string;  
}
```

#### 1. Service para armazenar e criar usuários em memória

```
// src/users/users.service.ts  
import { Injectable } from '@nestjs/common';  
import { CreateUsuarioDto } from '../dto/create-usuario.dto';  
  
type Usuario = { id: number; nome: string };  
  
@Injectable()  
export class UsersService {  
  private readonly usuarios: Usuario[] = [];  
  
  findAll() {  
    return this.usuarios;  
  }  
  
  findOne(id: number) {  
    return this.usuarios.find((u) => u.id === id);  
  }  
  
  create(dto: CreateUsuarioDto) {  
    const novoUsuario: Usuario = {  
      id: this.usuarios.length + 1,  
      nome: dto.nome,  
    };  
    this.usuarios.push(novoUsuario);  
    return novoUsuario; // Nest retorna 201 por padrão em POST  
  }  
  
  update(id: number, dto: CreateUsuarioDto) {  
    const idx = this.usuarios.findIndex((u) => u.id === id);  
    if (idx === -1) return undefined;  
    this.usuarios[idx] = { id, nome: dto.nome };  
    return this.usuarios[idx];  
  }  
  
  remove(id: number) {  
    const idx = this.usuarios.findIndex((u) => u.id === id);
```

```

    if (idx === -1) return undefined;
    const [removido] = this.usuarios.splice(idx, 1);
    return removido;
  }
}

```

#### 1. Controller com a rota POST /usuarios

```

// src/users/users.controller.ts
import { Controller, Get, Post, Put, Delete, Param, Body } from '@nestjs/common';
import { UsersService } from './users.service';
import { CreateUsuarioDto } from './dto/create-usuario.dto';

@Controller('usuarios')
export class UsersController {
  constructor(private readonly usersService: UsersService) {}

  @Get()
  findAll() {
    return this.usersService.findAll();
  }

  @Get('/:id')
  findOne(@Param('id') id: string) {
    return this.usersService.findOne(+id);
  }

  @Post()
  create(@Body() body: CreateUsuarioDto) {
    return this.usersService.create(body); // por padrão, NestJS responde 201 em POST
  }

  @Put('/:id')
  update(@Param('id') id: string, @Body() body: CreateUsuarioDto) {
    return this.usersService.update(+id, body);
  }

  @Delete('/:id')
  remove(@Param('id') id: string) {
    return this.usersService.remove(+id);
  }
}

```

#### 1. Módulo

```

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

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

```

Observações importantes

- Código de status 201: em NestJS, o método POST já retorna 201 por padrão, então você não precisa setar manualmente. Se quiser explicitar, pode usar `@HttpCode(201)` no método create.
- Validação do corpo: para validar `nome`, adicione `class-validator` e `ValidationPipe` global.
- Nome da rota: mapeei `@Controller('usuarios')` para corresponder exatamente ao seu caminho `/usuarios`. Se preferir manter em inglês (como no seu arquivo atual `@Controller('users')`), basta ajustar o caminho ou criar outro controller/rota.

Exemplo de chamada HTTP

```
POST /usuarios
Content-Type: application/json

{
  "nome": "João"
}
```

Resposta (201):

```
{
  "id": 1,
  "nome": "João"
}
```

Se quiser, eu adapto o seu UsersController atual (que está usando `@Controller('users')` e campo `name`) para seguir o padrão em português com `usuarios / nome`.

#### ▼ Reflita:

### Respostas às Perguntas:

#### 1. Foi criada uma estrutura modular com controller e service?

✓ **Sim.** A implementação em NestJS seguiu a estrutura modular padrão:

- `UsersModule` (organiza dependências)
- `UsersController` (lida com rotas e HTTP)
- `UserService` (contém a lógica de negócios)

#### 2. A estrutura de DTO foi utilizada?

✗ **Não.** O código atual usa tipagem inline (`{ name: string }`) em vez de DTOs dedicados.

#### O que é um DTO e seu papel no NestJS:

Um **DTO (Data Transfer Object)** é um padrão usado para definir a estrutura dos dados transferidos entre camadas. No NestJS, os DTOs:

- Validam dados de entrada (com `class-validator`)
- Documentam a API automaticamente (com Swagger)
- Melhoram a manutenção e previnem erros
- Exemplo de DTO para criação de usuário:

```
export class CreateUsuarioDto {
  @IsString()
  @IsNotEmpty()
  nome: string;
}
```

#### ▼ A. Estrutura de Pastas

### Estrutura de Pastas Ideal para um Projeto NestJS

A estrutura de pastas em um projeto NestJS segue um padrão modular que facilita a manutenção e escalabilidade. Abaixo está uma estrutura de pastas recomendada para projetos médios a grandes:

```
src/
├── app.module.ts      # Módulo raiz da aplicação
├── main.ts            # Ponto de entrada da aplicação
├── common/            # Código compartilhado em toda a aplicação
│   ├── constants/    # Constantes globais
│   ├── decorators/    # Decorators personalizados
│   ├── dto/           # DTOs compartilhados
│   ├── enums/         # Enumerações
│   ├── exceptions/    # Exceções personalizadas
│   ├── filters/       # Filtros de exceção globais
│   ├── guards/        # Guards de autenticação/autorização
│   ├── interceptors/  # Interceptors globais
│   ├── interfaces/    # Interfaces compartilhadas
│   ├── middlewares/   # Middlewares personalizados
│   ├── pipes/         # Pipes de validação
│   └── utils/         # Funções utilitárias
├── config/            # Configurações da aplicação
│   ├── app.config.ts  # Configuração principal
│   ├── database.config.ts # Configuração do banco de dados
│   └── env.validation.ts # Validação de variáveis de ambiente
├── modules/           # Módulos de funcionalidades (feature modules)
│   ├── auth/          # Módulo de autenticação
│   │   ├── dto/       # DTOs específicos de autenticação
│   │   ├── entities/  # Entidades relacionadas à autenticação
│   │   ├── guards/    # Guards específicos de autenticação
│   │   ├── strategies/ # Estratégias de autenticação (JWT, OAuth, etc.)
│   │   ├── auth.controller.ts # Controlador de autenticação
│   │   ├── auth.module.ts # Módulo de autenticação
│   │   └── auth.service.ts # Serviço de autenticação
│   ├── users/         # Módulo de usuários
│   │   ├── dto/       # DTOs específicos de usuários
│   │   ├── entities/  # Entidades de usuário
│   │   ├── users.controller.ts # Controlador de usuários
│   │   ├── users.module.ts # Módulo de usuários
│   │   ├── users.service.ts # Serviço de usuários
│   │   └── users.repository.ts # Repositório de usuários (opcional)
│   └── [outros módulos]/ # Outros módulos específicos da aplicação
├── database/          # Configurações relacionadas ao banco de dados
│   ├── migrations/    # Migrações do banco de dados
│   ├── seeders/       # Dados iniciais (seeds)
│   └── ormconfig.ts    # Configuração ORM (TypeORM, Prisma, etc.)
```

## Boas Práticas de Organização

- **Modularidade:** Divida a aplicação em módulos funcionais, cada um com sua responsabilidade específica.
- **Separação de responsabilidades:** Siga o princípio de responsabilidade única, onde cada arquivo tem uma função específica.
- **Nomenclatura consistente:** Use nomes descritivos e siga padrões de nomenclatura consistentes.
- **Organização por domínio:** Agrupe arquivos relacionados ao mesmo domínio de negócio.
- **Abstração de infraestrutura:** Isole código de infraestrutura (banco de dados, serviços externos) do código de domínio.

## Estrutura de um Módulo

Cada módulo em NestJS geralmente segue esta estrutura:

```
module-name/  
├── dto/           # Data Transfer Objects  
├── entities/      # Entidades de banco de dados  
├── interfaces/    # Interfaces específicas do módulo  
├── module-name.controller.ts # Controlador - lida com requisições HTTP  
├── module-name.service.ts   # Serviço - lógica de negócios  
├── module-name.module.ts    # Definição do módulo  
├── module-name.repository.ts # Repositório (opcional)  
└── tests/               # Testes unitários e de integração
```

#### ▼ Estrutura para Aplicações Menores

Para projetos menores, você pode simplificar a estrutura:

```
src/  
├── app.module.ts  
├── main.ts  
├── auth/  
│   ├── auth.controller.ts  
│   ├── auth.module.ts  
│   ├── auth.service.ts  
│   └── dto/  
├── users/  
│   ├── users.controller.ts  
│   ├── users.module.ts  
│   ├── users.service.ts  
│   ├── dto/  
│   └── entities/  
└── common/  
    ├── filters/  
    ├── guards/  
    └── interceptors/
```

#### ▼ Recursos Adicionais

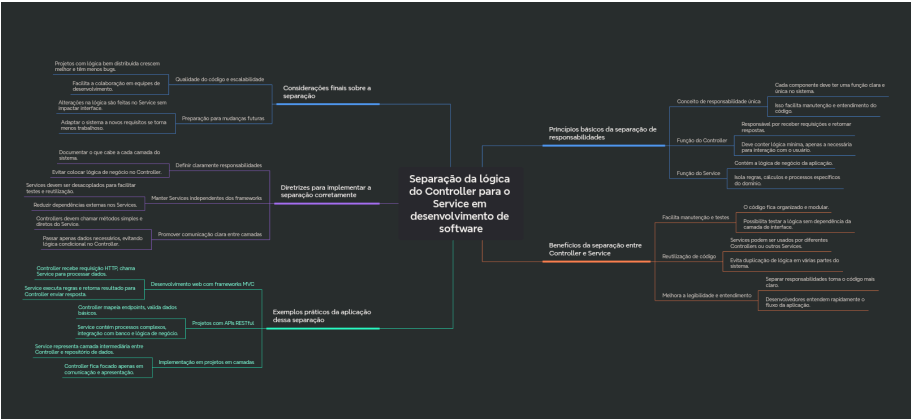
Além da estrutura básica, projetos maiores podem incluir:

- **libs/** - Bibliotecas internas compartilhadas entre projetos
- **documentation/** - Documentação da API e do projeto
- **scripts/** - Scripts de automação e CI/CD
- **test/** - Testes E2E e configurações de teste

Lembre-se que esta estrutura é flexível e pode ser adaptada às necessidades específicas do seu projeto. O importante é manter a consistência e seguir os princípios do NestJS de modularidade e injeção de dependências.


## ▼ B. Mapa Mental sobre Separação de Responsabilidades





<https://mapify.so/share-link/WgTJvXGvAm>

## ▼ C. Esquema Visual dos Decorators



# Decorators Comuns em NestJS

Este documento explora os decorators mais comuns utilizados no framework NestJS, detalhando suas funcionalidades e casos de uso. O objetivo é fornecer um guia prático para desenvolvedores que desejam aprofundar seus conhecimentos sobre como os decorators facilitam a construção de aplicações robustas e bem estruturadas em NestJS.

## O que são Decorators?

Decorators são uma funcionalidade do TypeScript (e, por extensão, do JavaScript) que permite adicionar metadados e modificar o comportamento de classes, métodos, propriedades ou parâmetros de funções. Em NestJS, os decorators são amplamente utilizados para definir rotas, injetar dependências, configurar middlewares e muito mais. Eles simplificam a configuração e a organização do código, tornando-o mais legível e fácil de manter.

## Decorators Comuns em NestJS

A seguir, apresentamos os decorators mais comuns em NestJS, agrupados por categoria:

### 1. Decorators de Rota (Controllers)

Estes decorators são usados para definir os endpoints da sua API e associá-los a métodos específicos dentro de um controller.

- @Controller(path) : Define uma classe como um controller e especifica o prefixo da rota para todas as rotas dentro dessa classe.



<https://app.napkin.ai/page/CgoiCHByb2Qtb25lEiwKBFBhZ2UaJDc3M2I3ZTlhLTUzYjYtNDdlOS05OWNmLWY1M2Nm;s=1>