NestJS

Iniciando um projeto NestJS

1. Instale o CLI do NestJS

npm i -g @nestjs/cli

serve para **instalar a CLI (Command Line Interface) do NestJS globalmente** na sua máquina.

Com a CLI instalada, você pode rodar o comando:

nest new nome-do-projeto

Isso gera automaticamente toda a estrutura de pastas, arquivos de configuração e dependências do NestJS.

A CLI te oferece comandos prontos como:

* nest g module nome → Gera um módulo
* nest g controller nome → Gera um controller
* nest g service nome → Gera um service

A CLI segue as melhores práticas e gera o código seguindo o padrão oficial do NestJS, evitando erros comuns.

1. Crie um novo projeto

nest new nome-do-projeto

**porque que um projeto nest já vem versionado?**

Um projeto NestJS **já vem versionado** porque o comando:

nest new nome-do-projeto

executa internamente o seguinte processo:

**1. Cria um repositório Git automaticamente**

Durante a criação do projeto, a CLI do NestJS executa:

git init

Ou seja, ela **inicializa um repositório Git** automaticamente dentro da pasta do projeto.

**Por que o Nest faz isso por padrão?**

1. **Boas práticas de desenvolvimento:**
   * O uso de versionamento com Git é considerado essencial para qualquer projeto profissional.
   * Permite que você controle alterações, volte versões, e trabalhe com branches.
2. **Preparado para subir no GitHub ou GitLab:**
   * Após a criação, você só precisa rodar:

git remote add origin https://github.com/seu-usuario/seu-repo.git

git push -u origin main

1. **Facilita colaboração em equipe:**
   * O NestJS é voltado para projetos escaláveis e profissionais. Ter um repositório Git já pronto facilita integrar com times, CI/CD, etc.

**Como saber se foi versionado?**

* A pasta .git está criada no projeto.
* O terminal exibe isso ao final da criação:

Successfully initialized git repository.

**E se você não quiser que venha versionado?**

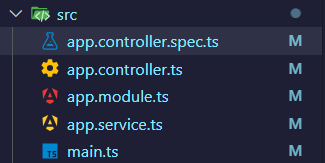
Você pode usar:

nest new nome-do-projeto --skip-git

Assim o Nest **não vai executar git init** automaticamente.

1. Estrutura inicial criada





**app.controller.spec.ts**

* **O que é?**  
  Um **arquivo de teste unitário** do controller (app.controller.ts).
* **Função:**  
  Testa se o AppController está funcionando corretamente (por exemplo, se retorna "Hello World!").
* **Importância:**  
  Garante que o comportamento da aplicação está correto sem precisar rodar o servidor.

**app.controller.ts**

* **O que é?**  
  O **controller principal** da aplicação NestJS.
* **Função:**  
  Controladores recebem as **requisições HTTP**, processam ou delegam para serviços e retornam respostas.
* **Exemplo:**  
  Um método @Get() que retorna "Hello World!".

O AppController está usando **injeção de dependência** para acessar o AppService, que é onde está a **lógica de negócio**.

Esse constructor(...) é chamado **automaticamente pelo NestJS** quando o controller é criado.

O NestJS vê que o controller precisa de um AppService e entrega automaticamente uma instância dele (isso é chamado de **injeção de dependência**).

O private readonly appService faz com que essa instância fique **disponível como uma propriedade da classe**.

**O que é Injeção de Dependência (Dependency Injection)?**

**É um padrão de design** em que um objeto (como um controller ou serviço) **recebe as suas dependências de fora**, ao invés de criá-las diretamente.

**Exemplo sem injeção de dependência:**

export class AppController {

private appService = new AppService(); // 👎 ruim

getHello() {

return this.appService.getHello();

}

}

**app.module.ts**

* **O que é?**  
  O **módulo raiz** da aplicação.
* **Função:**  
  Define os **módulos, controllers e providers** que fazem parte da aplicação.
* **Importância:**  
  É o ponto central de organização da aplicação NestJS — tudo começa aqui.
* **Exemplo:**

@Module({

imports: [],

controllers: [AppController],

providers: [AppService],

})

**E por que a classe AppModule está vazia?**

Porque **todo o conteúdo da classe é fornecido pelo decorator @Module()**, então a classe em si **não precisa de código interno**.

* A classe **não está vazia por obrigação**.
* Ela está vazia **porque normalmente não se coloca lógica dentro dela** — tudo é configurado com o decorator.
* Mesmo vazia, o NestJS precisa dela para carregar a estrutura.

**app.service.ts**

* **O que é?**  
  Um **serviço** que contém a lógica de negócio.
* **Função:**  
  É usado para separar a lógica dos controllers.  
  Ex: AppService tem um método getHello() que retorna "Hello World!".
* **Importância:**  
  Mantém o controller limpo e desacoplado da lógica interna.

**main.ts**

* **O que é?**  
  O **arquivo de bootstrap** da aplicação.
* **Função:**  
  É o ponto de entrada. Ele executa:

await NestFactory.create(AppModule);

**Por que a função em main.ts se chama bootstrap?**

A palavra **bootstrap** vem do termo **"bootstrapping"**, que é um termo comum em computação para descrever o **processo de inicializar ou dar o "start" em um sistema**.

bootstrap().catch((err) => {

  console.error('Falha ao iniciar a aplicação:', err);

  process.exit(1); // Encerra com código de erro

});

**Significado do termo:**

**Bootstrapping** é o processo de **levantar um sistema do zero**, carregando tudo que ele precisa para funcionar.

**O que são Decorators?**

**Definição simples:**

**Decorators** são **funções especiais** que você coloca com @ antes de classes, métodos, propriedades ou parâmetros, para **adicionar comportamentos extras** a eles.

**No NestJS, os decorators são usados para:**

|  |  |
| --- | --- |
| **Onde** | **Para quê** |
| Classes | Definir controladores, serviços, módulos, etc. |
| Métodos | Criar rotas HTTP (GET, POST...) ou interceptadores/middlewares. |
| Parâmetros | Extrair dados da requisição (query, body, params, headers...). |
| Propriedades | Injetar dependências automaticamente. |

Principais decorators do NestJS e o que cada um faz

**Decorators de Classe**

**@Module() - Declara um módulo (ex: AppModule) e agrupa controllers, providers etc.**

**Estrutura do @Module()**

@Module({

imports: [],

controllers: [],

providers: [],

exports: [],

})

export class AlgumModulo {}

Explicação de cada propriedade

Imports - Lista de **outros módulos** que este módulo depende.

Controllers - Lista dos **controllers** que pertencem a este módulo.

Providers - Lista dos **serviços, classes, guards, pipes** etc. que serão injetáveis.

Exports - Define o que este módulo **expõe para outros módulos** que o importarem.

**Exemplo prático:**

@Module({

imports: [DatabaseModule], // ← usa funcionalidades do módulo de banco

controllers: [UserController],

providers: [UserService, LoggerService],

exports: [UserService], // ← outros módulos podem injetar UserService

})

export class UserModule {}

**Explicando com um projeto real:**

Imagine que você tem 3 arquivos:

* user.controller.ts
* user.service.ts
* database.module.ts

O UserModule ficaria assim:

@Module({

imports: [DatabaseModule],

controllers: [UserController],

providers: [UserService],

exports: [UserService], // para que outros módulos possam usar

})

export class UserModule {}

**Usando exports para compartilhar serviços**

Se o UserService for usado em outro módulo, como AuthModule, você precisa:

1. **Exportar ele no UserModule**
2. **Importar o UserModule no AuthModule**

// auth.module.ts

@Module({

imports: [UserModule], // agora pode usar UserService aqui

})

export class AuthModule {}

**Quando você importa ou exporta um módulo no NestJS, é para reutilizar seus services (providers) e controllers em outros módulos.**

Mas tem uma diferença importante:

**Importar um módulo**

Serve para **usar os services exportados** por esse módulo.

**Exemplo:**

@Module({

imports: [UserModule], // <- importa o módulo do usuário

})

export class AuthModule {}

Isso **permite que o AuthModule use qualquer service ou provider que o UserModule tenha exportado**.

**Exportar em um módulo**

Serve para **tornar seus services disponíveis para outros módulos** que importarem ele.

**Exemplo:**

@Module({

controllers: [UserController],

providers: [UserService],

exports: [UserService], // <- tornando o serviço disponível para outros módulos

})

export class UserModule {}

**Exemplo completo:**

**user.module.ts:**

@Module({

controllers: [UserController],

providers: [UserService],

exports: [UserService], // <- outros módulos poderão usar esse service

})

export class UserModule {}

**auth.module.ts:**

@Module({

imports: [UserModule], // <- agora AuthModule pode usar UserService

providers: [AuthService],

})

export class AuthModule {}

Agora dentro de AuthService:

@Injectable()

export class AuthService {

constructor(private readonly userService: UserService) {} // ✅ funciona

}

**@Injectable() - Marca uma classe como injetável via dependência (ex: serviços).**

@Injectable() é um decorator de classe que informa ao NestJS que a classe pode ser gerenciada pelo sistema de injeção de dependência — ou seja, pode ser injetada em outros lugares.

**Sintaxe completa (com configuração):**

@Injectable({ scope: Scope.DEFAULT })

A única "opção configurável" do @Injectable() é o scope.

**O que significa ter um "escopo" (scope) em programação — e no NestJS?**

**Definição simples:**

**Escopo (scope)** define **onde, como e por quanto tempo algo existe e é reutilizado** na aplicação.

No contexto do NestJS, o **scope controla o ciclo de vida de uma instância de classe injetada com @Injectable()**.

**Analogia simples:**

Pense em uma **cafeteira**:

* Scope.DEFAULT → Uma cafeteira para toda a casa: **todo mundo compartilha**.
* Scope.REQUEST → Cada visita recebe **uma nova cafeteira só para ela**.
* Scope.TRANSIENT → Sempre que alguém pede café, **uma nova cafeteira é criada**.

**Escopos no NestJS**

**1. Scope.DEFAULT (padrão)**

Cria **uma única instância** da classe e **reutiliza** em toda a aplicação (singleton).

Ideal para: serviços reutilizáveis como UserService, DatabaseService, LoggerService, etc.

@Injectable()

export class MeuService {}

**2. Scope.REQUEST**

Cria **uma nova instância da classe por requisição HTTP**.

Ideal para: quando você precisa acessar **dados específicos da requisição**, como o usuário logado.

@Injectable({ scope: Scope.REQUEST })

export class MeuService {

constructor(@Inject(REQUEST) private readonly req: Request) {

console.log('Usuário:', req.user);

}

}

**3. Scope.TRANSIENT**

Cria **uma nova instância toda vez que a dependência for injetada**, mesmo dentro da mesma requisição.

Ideal para: **serviços que precisam de comportamento totalmente isolado**, como geração de relatórios, builders, serviços com estado temporário.

@Injectable({ scope: Scope.TRANSIENT })

export class MeuService {}

**@Controller() - Define uma classe como controller que responde a rotas HTTP.**

O decorator @Controller() marca uma **classe como responsável por lidar com requisições HTTP**.

**Estrutura geral**

@Controller('rota-base')

export class AlgumController {

@Get() findAll() {}

@Post() create() {}

@Get(':id') findOne(@Param('id') id: string) {}

}

O caminho 'rota-base' vira /rota-base na URL.  
Os métodos dentro do controller lidam com rotas específicas usando decorators como @Get() e @Post().

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

@Controller({

path: 'users',

version: '1',

scope: Scope.DEFAULT,

host: 'api.meusite.com',

})

export class UserController {

@Get()

getAll() {

return 'Todos os usuários';

}

}

**path**

Define a **rota base** para o controller.  
Pode ser string única ou array de rotas.

@Controller({ path: 'products' })

Mesmo que:

@Controller('products')

**scope**

Controla o ciclo de vida do controller (igual ao @Injectable()):

import { Scope } from '@nestjs/common';

@Controller({ scope: Scope.REQUEST }) // instância nova por requisição

**version**

Permite definir uma versão da API diretamente no controller.

@Controller({ path: 'users', version: '1' })

Com isso, se você ativar o versionamento na main.ts, você pode acessar /v1/users.

**O que é versionamento de API?**

Versionar uma API significa permitir que existam **diferentes versões da mesma API**, como:

* GET /v1/users → versão 1
* GET /v2/users → versão 2 com comportamento diferente

Isso é útil para **evoluir a aplicação sem quebrar quem ainda usa a versão antiga**.

**Como ativar versionamento no NestJS**

No arquivo main.ts, adicione o seguinte:

import { VersioningType } from '@nestjs/common';

async function bootstrap() {

const app = await NestFactory.create(AppModule);

// Ativa o versionamento por URI

app.enableVersioning({

type: VersioningType.URI, // Usa /v1, /v2, etc.

});

await app.listen(3000);

}

bootstrap();

**host**

Define um **host específico** para o controller.  
Exemplo: um controller que só responde se a requisição for para admin.site.com.

@Controller({ host: 'admin.site.com' })

Isso é útil para APIs multitenancy (vários subdomínios).

É onde você define as **rotas da sua API**, como GET, POST, PUT, DELETE, etc.

**@Global() - Torna um módulo global (não precisa ser importado em outros módulos).**

@Global() é um **decorator de classe** que transforma um módulo NestJS em um **módulo global**, ou seja, um módulo **visível para toda a aplicação sem precisar importar ele em outros módulos.**

**Quando usar @Global()?**

Quando você tem um módulo que:

Deve ser **acessível em toda a aplicação**  
Não quer **repetir imports: [...] em todo módulo**

Exemplos comuns de módulos globais:

LoggerModule - Serviço de logs em toda a aplicação

ConfigModule - Variáveis de ambiente (ex: .env)

PrismaModule - Cliente do banco com Prisma

AuthModule (às vezes) - Estratégia JWT compartilhada

Decorators de Método (dentro do controller)

**@Get() - Cria uma rota HTTP GET**

O @Get() é um **decorator de método** usado dentro de um controller para definir uma rota HTTP do tipo **GET**.

**Parâmetros que o @Get() aceita**

Ele pode receber **0 ou 1 parâmetro**, que é o **caminho (path)** da rota.

**Exemplos:**

**1. Sem parâmetros (rota base do controller)**

@Controller('users')

export class UserController {

@Get()

findAll() {

return 'Todos os usuários';

}

}

Rota: GET /users

**2. Com parâmetro de rota:**

@Get(':id')

findOne(@Param('id') id: string) {

return `Usuário com ID ${id}`;

}

Rota: GET /users/123

**3. Com múltiplas rotas (array)**

@Get(['', 'list', 'all'])

findAll() {

return 'Várias rotas para a mesma resposta';

}

Responde em:

* GET /users
* GET /users/list
* GET /users/all

**E se eu quiser usar query params?**

Aí você **não passa nada no @Get()**, e usa @Query() no método:

@Get()

findFiltered(@Query('active') active: string) {

return `Usuários ativos: ${active}`;

}

Exemplo de chamada: GET /users?active=true

**@Post() - Cria uma rota HTTP POST**

**O que é o @Post()?**

@Post()

create(@Body() body: any) {

return 'Dados recebidos';

}

@Post() é um **decorator de método** usado dentro de um @Controller() para lidar com **requisições HTTP do tipo POST**.

**Para que serve?**

É usado para **criar dados**, **enviar informações**, **realizar cadastros**, etc.

Exemplo de requisição:

POST /users

Content-Type: application/json

{

"name": "Johann",

"email": "johann@example.com"

}

**Exemplos de uso**

**1. Rota base**

@Controller('users')

export class UserController {

@Post()

createUser(@Body() body: any) {

return body;

}

}

Rota: POST /users  
O Nest vai injetar automaticamente o corpo da requisição no body.

**2. Rota específica**

@Post('register')

register(@Body() data: RegisterDto) {

return `Usuário ${data.name} cadastrado`;

}

Rota: POST /users/register

**3. Com múltiplos caminhos (pouco comum)**

@Post(['', 'create'])

createUser(@Body() data: any) {

return data;

}

Rota: POST /users e POST /users/create

Combinando com outros decorators

@Body() - Extrai o corpo da requisição (JSON)

@Headers() - Extrai cabeçalhos

@Query() - Extrai query params

@Param() - Extrai parâmetros da rota

@HttpCode(201) - Define o código de resposta

**Exemplo com tudo junto:**

@Post()

@HttpCode(201)

create(@Body() data: CreateUserDto, @Headers('authorization') token: string) {

return { data, token };

}

**@Put() - Cria uma rota HTTP PUT**

**O que é o @Put()?**

@Put()

update() {

return 'Atualização completa!';

}

@Put() é um **decorator de método** usado dentro de um controller para lidar com **requisições HTTP do tipo PUT**.

**Para que serve o método PUT?**

* O método **PUT** é utilizado para **substituir ou atualizar completamente um recurso existente**.
* Muito comum em operações de **edição de dados completos** (ex: atualizar todos os dados de um usuário).

**Exemplos práticos**

**1. Atualização por ID**

@Put(':id')

updateUser(@Param('id') id: string, @Body() body: any) {

return `Usuário ${id} atualizado com: ${JSON.stringify(body)}`;

}

Rota: PUT /users/123  
Corpo:

{

"name": "Johann",

"email": "johann@example.com"

}

**2. Definindo caminho manual**

@Put('atualizar/:id')

atualizar(@Param('id') id: string, @Body() dto: UpdateDto) {

return `Atualizado ${id} com sucesso`;

}

Rota: PUT /users/atualizar/123

Usando com outros decorators

@Param() - Captura o parâmetro da rota (:id)

@Body() - Captura o corpo da requisição (JSON)

@Query() - Captura query strings (?status=ativo)

@HttpCode() - Altera o código HTTP retornado (200, 204, etc)

**Exemplo completo**

@Put(':id')

@HttpCode(200)

updateUser(

@Param('id') id: string,

@Body() data: UpdateUserDto

) {

return {

message: `Usuário ${id} atualizado`,

data,

};

}

**@Delete() - Cria uma rota HTTP DELETE**

**O que é @Delete()?**

@Delete(':id')

remove(@Param('id') id: string) {

return `Recurso com id ${id} foi removido`;

}

@Delete() é um **decorator de método** usado em um controller para lidar com **requisições HTTP do tipo DELETE**.

**Exemplos práticos:**

**1. Excluir por ID**

@Delete(':id')

remove(@Param('id') id: string) {

return `Removido o recurso com ID ${id}`;

}

Rota: DELETE /users/1

**2. Caminho customizado**

@Delete('deletar/:id')

deletar(@Param('id') id: string) {

return `Item ${id} foi deletado`;

}

Rota: DELETE /users/deletar/1

Combinando com outros decorators

@Param() - Captura o parâmetro de rota (ex: :id)

@Body() - Em casos raros, você pode querer passar corpo junto (não comum em DELETE)

@HttpCode() - Altera o status HTTP manualmente (204, 200, etc)

**Exemplo:**

@Delete(':id')

@HttpCode(204)

remove(@Param('id') id: string) {

// Deleta e não retorna conteúdo (status 204)

}

**@Patch() - Cria uma rota HTTP PATCH**

**O que é o @Patch()?**

@Patch(':id')

updatePartial(@Param('id') id: string, @Body() data: any) {

return `Atualização parcial do item ${id}`;

}

@Patch() é um **decorator de método** usado em controllers NestJS para lidar com **requisições HTTP do tipo PATCH**.

**Para que serve?**

O método **PATCH** é utilizado para **atualizar parcialmente** um recurso.

Diferente do PUT, que espera **todos os campos** do recurso, o PATCH **aceita somente os campos que você quer alterar**.

**Exemplo de uso:**

**Suponha que você tenha um usuário:**

{

"name": "Johann",

"email": "johann@example.com",

"active": true

}

Se quiser **atualizar apenas o campo active**, você faria:

PATCH /users/123

{

"active": false

}

E isso **não apaga os outros campos**.

**Exemplos**

**1. Atualização parcial por ID**

@Patch(':id')

updateUser(@Param('id') id: string, @Body() body: any) {

return `Usuário ${id} parcialmente atualizado com: ${JSON.stringify(body)}`;

}

Rota: PATCH /users/123

**2. Caminho alternativo**

@Patch('atualizar-status/:id')

atualizarStatus(@Param('id') id: string, @Body() data: { active: boolean }) {

return `Status do usuário ${id} atualizado para ${data.active}`;

}

Rota: PATCH /users/atualizar-status/123

**@All() - Aceita qualquer método HTTP**

**O que é o @All()?**

@All()

handleAll() {

return 'Essa rota aceita qualquer método HTTP';

}

@All() é um **decorator de método** que diz ao NestJS para **responder a qualquer método HTTP** (GET, POST, PUT, DELETE, PATCH, etc.) para aquela rota.

**Quando usar o @All()?**

* Quando você quer **capturar qualquer tipo de requisição**, independentemente do método HTTP usado.
* Útil para:
  + **Rotas dinâmicas**
  + **Tratamento genérico**
  + **Debug**
  + **Serviços de proxy ou fallback**

**Exemplos**

**1. Rota que aceita tudo na raiz**

@Controller('hello')

export class HelloController {

@All()

handleAnything() {

return 'Método HTTP qualquer aceito aqui!';

}

}

Funciona para:

* GET /hello
* POST /hello
* DELETE /hello
* PATCH /hello
* ...

**2. Com caminho específico**

@All('fallback')

handleAllFallback() {

return 'Você acessou uma rota de fallback';

}

Funciona para:

* POST /controller/fallback
* DELETE /controller/fallback
* PUT /controller/fallback
* etc.

**Dica: como descobrir qual método foi chamado?**

Use o decorator @Req() para acessar o método:

@All()

handleAll(@Req() req: Request) {

return `Método HTTP usado: ${req.method}`;

}

Se você está usando o @All() e quer saber **qual método HTTP (GET, POST, PUT, DELETE etc.) foi chamado**, você pode usar o decorator @Req() do NestJS para acessar a requisição e verificar o método diretamente.

**Como fazer isso:**

import { All, Controller, Req } from '@nestjs/common';

import { Request } from 'express';

@Controller('teste')

export class TesteController {

@All()

handleAll(@Req() req: Request) {

return `Método HTTP usado: ${req.method}`;

}

}

**Explicando:**

* @Req() injeta o **objeto da requisição HTTP**
* Esse objeto vem do **Express.js**, então req.method contém exatamente qual verbo HTTP foi usado

**Se quiser mais dados úteis da requisição, pode acessar:**

@All()

handleAll(@Req() req: Request) {

console.log(req.method); // 'GET', 'POST', etc.

console.log(req.url); // URL acessada

console.log(req.headers); // Cabeçalhos

console.log(req.body); // Corpo da requisição (se houver)

}

O @All() **recebe qualquer requisição HTTP**, mas **você precisa verificar manualmente dentro dele qual foi o método chamado**, e **executar a lógica correspondente**.

**Exemplo prático:**

import { All, Controller, Req, Res } from '@nestjs/common';

import { Request, Response } from 'express';

@Controller('teste')

export class TesteController {

@All()

handleAll(@Req() req: Request, @Res() res: Response) {

const method = req.method;

switch (method) {

case 'GET':

return res.send('Recebi um GET');

case 'POST':

return res.send('Recebi um POST');

case 'PUT':

return res.send('Recebi um PUT');

case 'DELETE':

return res.send('Recebi um DELETE');

default:

return res.status(405).send('Método não permitido');

}

}

}

**Se já temos @Get(), @Post(), @Put(), etc., qual o sentido de usar @All()?**

O decorator @All() **não é feito para uso comum em controladores REST.**  
Ele existe para **casos especiais** em que você **realmente precisa capturar todos os métodos HTTP em uma mesma rota** — normalmente por necessidade técnica ou comportamento genérico.

**Casos onde @All() faz sentido**

**1. Fallback genérico**

Você quer capturar **qualquer método em qualquer rota**, por exemplo, para:

* Responder “Rota não encontrada” com log
* Criar uma rota **curinga** para debug
* Interceptar antes de decidir para onde ir

@All('\*')

handleAll(@Req() req: Request) {

return `Método ${req.method} na rota ${req.url} não reconhecido`;

}

**2. Gateway / Proxy**

Você está criando um **gateway de entrada** para sua aplicação (como um API Gateway ou servidor de forwarding), que precisa **passar qualquer requisição para outro serviço**.

@All()

redirectAll(@Req() req: Request, @Res() res: Response) {

// pega o método e redireciona para outro servidor

}

**3. Mocks e Testes**

Quando você quer **simular um servidor** que aceita qualquer coisa sem precisar declarar todos os verbos.

@All()

mockServer(@Req() req: Request) {

return {

method: req.method,

body: req.body,

};

}

**@Head() - Responde apenas o cabeçalho**

**O que é o @Head()?**

@Head()

checkStatus() {

return;

}

O decorator @Head() define um **método que responde a requisições HTTP do tipo HEAD**.

**O que é uma requisição HEAD?**

* Uma requisição HEAD é **igual a uma GET**, **mas sem o corpo na resposta**.
* Ela retorna **somente os headers** (como Content-Type, Content-Length, Cache-Control, etc).
* Muito usada para:
  + Verificar se um recurso existe
  + Checar metadados sem baixar todo o conteúdo
  + Validar cache (ETag, Last-Modified, etc.)

**Exemplo básico:**

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

@Controller('health')

export class HealthController {

@Head()

ping() {

// Não precisa retornar nada; o NestJS responde com status 200

}

}

Rota: HEAD /health  
Resposta: status 200, sem corpo.

**Exemplo com headers:**

@Head()

check(@Res() res: Response) {

res.set('X-Custom-Header', 'true');

res.status(204).send();

}

Retorna um status 204 (sem conteúdo) e um header X-Custom-Header.

**Quando usar @Head()?**

* Monitoramento ou **verificação de disponibilidade** (ex: /health)
* Checagem de cache (Last-Modified, ETag)
* Verificar se um recurso existe antes de baixar (ex: em CDNs, arquivos grandes)

**@Options() - Responde a requisições OPTIONS**

**O que é o @Options()?**

@Options()

handleOptions() {

return;

}

O decorator @Options() define um método que lida com **requisições HTTP do tipo OPTIONS**.

**Para que serve o método HTTP OPTIONS?**

* Ele é usado para **descobrir quais métodos HTTP estão disponíveis** para uma determinada rota.
* Muito comum em **CORS** (Cross-Origin Resource Sharing).
* Navegadores usam OPTIONS automaticamente em requisições **pré-flight** (antes de um POST, PUT etc.).

**Exemplo de requisição OPTIONS**

OPTIONS /users HTTP/1.1

Access-Control-Request-Method: POST

Origin: https://meusite.com

O servidor responde dizendo se POST, GET, DELETE são permitidos nessa rota, junto com os headers permitidos.

**Exemplo prático com NestJS**

import { Controller, Options, Res } from '@nestjs/common';

import { Response } from 'express';

@Controller('users')

export class UserController {

@Options()

handleOptions(@Res() res: Response) {

res.set({

'Allow': 'GET, POST, PUT, DELETE, OPTIONS',

'Access-Control-Allow-Origin': '\*',

'Access-Control-Allow-Methods': 'GET, POST, PUT, DELETE, OPTIONS',

'Access-Control-Allow-Headers': 'Content-Type, Authorization',

});

res.sendStatus(204); // Sem corpo, apenas cabeçalhos

}

}

Em apps comuns, o Nest já cuida disso automaticamente se você ativar o CORS com:

app.enableCors();

**@Redirect() - Redireciona a rota**

**O que é o @Redirect()?**

**@Redirect('https://example.com', 302)**

O decorator @Redirect() é usado para dizer ao NestJS que a resposta dessa rota deve redirecionar para outra URL.

**Para que serve?**

* Redirecionar o usuário para outro site, rota ou recurso.
* Muito útil em:
  + Encaminhar usuários para páginas externas
  + Criar atalhos dentro da API
  + Lidar com URLs legadas

**Exemplo básico:**

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

@Controller()

export class AppController {

@Get('docs')

@Redirect('https://docs.nestjs.com', 302)

redirectToDocs() {

// Nenhum retorno necessário — redireciona direto

}

}

GET /docs redireciona para https://docs.nestjs.com com status 302 Found

**Exemplo com URL dinâmica:**

Você também pode retornar a URL dinamicamente com base em uma lógica.

@Get('google')

@Redirect() // ← sem parâmetros

redirectToGoogle() {

return {

url: 'https://google.com',

statusCode: 301, // ou 302

};

}

**Isso permite decidir em tempo de execução para onde redirecionar.**

**Redirecionamento condicional:**

@Get('redirect')

@Redirect()

redirectExample() {

const isLoggedIn = false;

return isLoggedIn

? { url: '/dashboard', statusCode: 302 }

: { url: '/login', statusCode: 302 };

}

O @Redirect() pode te levar para uma página com informações de um usuário específico?

Sim, pode, mas com uma limitação importante:

Ele só redireciona para uma rota ou URL — ele não renderiza ou devolve dados diretamente.

Ou seja:

* Ele encaminha o navegador para outra rota (interna ou externa).
* Se essa rota estiver configurada para exibir os dados do usuário (como /users/:id), **então sim — ele pode levar até lá.**

**Exemplo prático:**

Imagine isso em seu backend NestJS:

@Get('go-to-user/:id')

@Redirect()

redirectToUserPage(@Param('id') id: string) {

return {

url: `/users/${id}`,

statusCode: 302,

};

}

Ao acessar GET /go-to-user/123, o Nest redireciona para /users/123

Agora, se a rota /users/:id estiver configurada para mostrar as informações do usuário com aquele ID, o resultado final vai ser exatamente o que você perguntou:

"Ir para a página com todas as informações de um usuário específico."

Por que usar @Redirect() no backend se posso redirecionar no frontend?

Na maioria das aplicações modernas (com frontend em React, Angular, etc.) você deve tratar redirecionamentos no frontend.

**Ou seja:**

Sim, usar @Redirect() no backend geralmente não é necessário em aplicações SPA (Single Page Application).

**Então... por que o NestJS tem @Redirect()?**

Porque existem cenários específicos onde o redirecionamento pelo backend faz sentido.

**Casos em que o @Redirect() faz sentido no backend:**

Site público sem SPA - Você precisa redirecionar para outra rota HTML

Após login via OAuth - Você redireciona do backend para o frontend com um token

Encaminhar para um recurso externo - Ex: /docs redireciona para <https://docs.seusite.com>

URL curta / redirect de marketing - Ex: /promo redireciona para /products?id=123

Fallback de rota legada - Ex: /old-route → /new-route

API Gateway ou proxy - O backend distribui chamadas para outros serviços

**@Render() - Retorna um template (usado com @nestjs/platform-express)**

**O que é o @Render()?**

@Render('home')

getHome() {

return { title: 'Bem-vindo!' };

}

@Render() é um decorator de método que diz ao NestJS para renderizar um template HTML com os dados retornados pela função.

**Quando usar @Render()?**

* Quando você está construindo páginas renderizadas no servidor (SSR) com NestJS.
* Quando você usa um engine de templates, como:
  + Handlebars (hbs)
  + EJS
  + Pug
  + Mustache
* Quando você quer retornar HTML no lugar de JSON (como em apps clássicos).

**Como usar @Render() no NestJS**

**1. Instale um engine de template (ex: Handlebars)**

**npm install hbs**

**2. Configure no main.ts**

import { NestFactory } from '@nestjs/core';

import { AppModule } from './app.module';

import { join } from 'path';

async function bootstrap() {

const app = await NestFactory.create(AppModule);

app.setBaseViewsDir(join(\_\_dirname, '..', 'views')); // pasta das views

app.setViewEngine('hbs'); // ou 'ejs', 'pug', etc.

await app.listen(3000);

}

bootstrap();

**3. Crie a view**

Crie um arquivo views/home.hbs:

<h1>{{title}}</h1>

<p>Bem-vindo ao NestJS com Handlebars!</p>

**4. Use o @Render() no controller**

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

@Controller()

export class AppController {

@Get()

@Render('home') // nome do arquivo de template (sem extensão)

getHomePage() {

return { title: 'Olá, Johann!' }; // será usado como contexto da view

}

}

A rota / agora renderiza a view home.hbs com o valor title = 'Olá, Johann!'.

Qual o sentido de renderizar HTML no servidor de uma API, se o frontend pode fazer isso?

Na maioria dos casos atuais, realmente não faz sentido.

Se você está construindo uma SPA (React, Angular, Vue, etc.), o frontend deve cuidar totalmente da interface, e o backend deve apenas fornecer dados em JSON.

Mas...

Existem cenários onde faz sim sentido renderizar HTML no backend.

**Quando faz sentido usar @Render() e renderizar HTML no backend?**

Aplicações tradicionais (sem frontend separado) - Tudo fica no Nest: lógica, templates e rotas

Aplicações que precisam de SEO (Google indexa HTML) - HTML pré-renderizado = melhor ranqueamento

Primeira página carregada mais rápido - Server-Side Rendering (SSR) entrega o HTML direto

E-mails em HTML - Templates no backend para montar corpo de e-mail

Portais administrativos rápidos - NestJS + @Render() + Handlebars para CRUD interno

Micro-sites estáticos (landing pages, formulários) - Não precisa de SPA para algo simples

Decorators de Parâmetro (usados nos métodos)

**@Param() - Parâmetros de rota (/users/:id)**

**O que é o @Param()?**

@Param('id') id: string

O @Param() é um decorator de parâmetro de método usado para extrair valores da URL que estão definidos como parâmetros dinâmicos (/users/:id, /posts/:slug, etc.).

**Para que serve?**

**Quando você define uma rota como:**

**@Get(':id')**

Você precisa capturar o valor de :id — e é exatamente isso que o @Param() faz.

**Sintaxe**

@Param() // retorna todos os parâmetros como objeto

@Param('nome') // retorna o valor de um parâmetro específico

**Exemplos práticos**

**1. Capturar um único parâmetro**

@Get(':id')

getUser(@Param('id') id: string) {

return `Usuário com ID: ${id}`;

}

Rota: GET /users/123  
Resultado: "Usuário com ID: 123"

**2. Capturar múltiplos parâmetros**

@Get(':userId/posts/:postId')

getPost(

@Param('userId') userId: string,

@Param('postId') postId: string,

) {

return `Post ${postId} do usuário ${userId}`;

}

Rota: GET /users/42/posts/99

**3. Capturar todos os parâmetros de uma vez**

@Get(':userId/posts/:postId')

getAllParams(@Param() params: { userId: string; postId: string }) {

return params; // { userId: '42', postId: '99' }

}

Isso é útil quando há muitos parâmetros e você quer trabalhar com eles como objeto.

**Dica: Combine com @ParseIntPipe (ou outros pipes)**

@Get(':id')

getById(@Param('id', ParseIntPipe) id: number) {

return `ID como número: ${id}`;

}

Isso já garante que o id seja convertido e validado como number.

**Exemplo completo com DTO**

@Get(':id')

findOne(@Param() params: { id: string }) {

return `Buscando item ${params.id}`;

}

**@Query() - Query strings (/users?limit=10)**

**O que é o @Query()?**

@Query('nome') nome: string

O @Query() é um **decorator de parâmetro de método** usado para capturar **query parameters** da URL.

**O que são query parameters?**

São valores passados na URL **depois do ponto de interrogação ?**:

GET /usuarios?ativo=true&idade=25

**Sintaxe do @Query()**

@Query() // captura todos os parâmetros como objeto

@Query('nome') // captura um parâmetro específico

@Query('idade', ParseIntPipe) idade: number // aplica pipe de transformação

**Exemplos práticos**

**1. Capturar um parâmetro específico**

@Get()

buscar(@Query('nome') nome: string) {

return `Buscando por nome: ${nome}`;

}

Rota: GET /?nome=Johann  
Retorno: "Buscando por nome: Johann"

**2. Capturar múltiplos parâmetros**

@Get()

filtrar(

@Query('nome') nome: string,

@Query('idade', ParseIntPipe) idade: number,

) {

return `Nome: ${nome}, Idade: ${idade}`;

}

Rota: GET /?nome=Ana&idade=20

**3. Capturar todos os parâmetros de uma vez**

@Get()

filtroCompleto(@Query() query: { nome?: string; ativo?: string }) {

return query;

}

Rota: GET /?nome=Joao&ativo=true  
Retorno: { nome: 'Joao', ativo: 'true' }

**Pode usar com validação e transformação?**

Sim! Pode usar pipes como ParseIntPipe, ValidationPipe, DTOs, ou bibliotecas externas (como Zod ou Joi).

**Exemplo com DTO:**

@Get()

filtro(@Query() dto: FiltroUsuarioDto) {

return dto;

}

**@Body() - Corpo da requisição (JSON)**

**O que é o @Body()?**

@Body() body: any

O @Body() é um **decorator** do NestJS que extrai e fornece o **corpo (body)** da requisição HTTP para o seu método do controller.

**Exemplo básico**

@Post()

criar(@Body() dados: any) {

return dados;

}

Se você enviar um JSON:

{ "nome": "Johann", "idade": 30 }

A variável dados receberá esse objeto.

**Formas de uso**

**1. Capturar todo o corpo**

@Post()

create(@Body() body: any) {

return body;

}

**2. Capturar campos específicos do body**

@Post()

create(@Body('nome') nome: string) {

return `Nome: ${nome}`;

}

Envia { "nome": "Johann" } → recebe "Nome: Johann"

**3. Usar com DTO (melhor prática)**

export class CriarUsuarioDto {

nome: string;

idade: number;

}

@Post()

create(@Body() dto: CriarUsuarioDto) {

return dto;

}

Isso permite que você:

* Tenha **autocompletar no VSCode**
* Faça **validação automática** com class-validator
* Escreva código mais limpo e seguro

**@Headers() - Cabeçalhos HTTP**

**O que é o @Headers()?**

@Headers('authorization') token: string

O @Headers() é um **decorator de parâmetro** que permite acessar os **valores dos headers HTTP** enviados no request.

**Exemplos práticos**

**1. Capturar todos os headers de uma requisição**

@Get()

getHeaders(@Headers() headers: Record<string, string>) {

return headers;

}

Você recebe um objeto com todos os headers da requisição.

**2. Capturar um header específico**

@Get()

getAuth(@Headers('authorization') token: string) {

return `Token recebido: ${token}`;

}

Se o cliente enviar:

Authorization: Bearer abc123

Você receberá Bearer abc123.

**3. Sensível ou não ao case?**

Não! Os nomes dos headers **não são case sensitive**, então tanto 'Authorization' quanto 'authorization' funcionam.

**Exemplo com autenticação (JWT)**

@Get('me')

getProfile(@Headers('authorization') token: string) {

const jwt = token.replace('Bearer ', '');

return this.authService.decode(jwt);

}

**@Req() - Objeto completo da requisição Express**

**O que é o @Req()?**

@Req() req: Request

@Req() é um decorator que injeta o **objeto da requisição HTTP** diretamente no método do controller — como ocorre no Express (req do req, res).

**Quando usar?**

Use @Req() quando você precisa acessar:

Cabeçalhos - req.headers

Corpo da requisição - req.body

Parâmetros da URL - req.params

Query parameters - req.query

IP do cliente - req.ip

Método (GET, POST, etc.) - req.method

Caminho da rota - req.path

**Exemplo básico**

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

import { Request } from 'express';

@Controller('users')

export class UsersController {

@Get()

getAll(@Req() req: Request) {

console.log(req.method); // GET

console.log(req.url); // /users

return 'Todos os usuários';

}

}

O req aqui é do tipo Request do Express.

**Tipagem do @Req()**

* Por padrão, o NestJS usa o Express, então você pode importar a tipagem de lá:

import { Request } from 'express';

* Mas se estiver usando **Fastify** em vez de Express, use:

import { FastifyRequest as Request } from 'fastify';

**Exemplos úteis**

**1. Obter o IP do usuário**

@Get()

getIp(@Req() req: Request) {

return req.ip;

}

**2. Obter o header manualmente**

@Get()

getToken(@Req() req: Request) {

return req.headers['authorization'];

}

**3. Acessar query e params ao mesmo tempo**

@Get(':id')

getData(@Req() req: Request) {

return {

id: req.params.id,

page: req.query.page,

};

}

**Quando evitar @Req()?**

Evite usar @Req() se você **só precisa de uma parte específica**, como:

@Body() - Para ler o body

@Param() - Para parâmetros

@Query() - Para query strings

@Headers() - Para headers

**@Res() - Objeto completo da resposta Express**

**O que é o @Res()?**

@Res() res: Response

O @Res() é um decorator que injeta diretamente o **objeto de resposta (res) do Express** (ou Fastify) no seu método de controller.

**Exemplo básico**

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

import { Response } from 'express';

@Controller('testes')

export class TestesController {

@Get()

enviarMensagem(@Res() res: Response) {

res.status(200).json({ mensagem: 'Resposta direta usando @Res()' });

}

}

Isso **bypassa o sistema padrão de resposta do NestJS**, que normalmente envia o retorno do método diretamente.

**ATENÇÃO: Quando você usa @Res(), o NestJS não envia mais a resposta automaticamente. Você é obrigado a usar res.send(), res.json(), etc.**

**Exemplo com redirecionamento**

@Get('google')

redirecionar(@Res() res: Response) {

res.redirect('https://www.google.com');

}

Quando usar @Res()?

Precisa de controle total da resposta – SIM

Quer enviar headers customizados – SIM

Redirecionamentos manuais – SIM

Enviar arquivos ou HTML puro – SIM

APIs REST simples – NÃO

**Quando não usar @Res()?**

Evite @Res() em APIs tradicionais onde você só quer devolver um JSON. Prefira deixar o Nest cuidar da resposta:

@Get()

getAll() {

return { users: [] }; // O Nest já responde com status 200 e o objeto

}

**@Next() - Próxima função do middleware (next())**

**O que é o @Next()?**

@Next() next: NextFunction

O decorator @Next() injeta a função next() do Express/Fastify no seu método ou middleware, permitindo **passar a requisição adiante** na cadeia de execução.

**Onde se usa?**

Geralmente, o @Next() é usado em:

1. **Middlewares**
2. **Controllers avançados (casos raros)**
3. **Intervenções específicas em rota**

**Exemplo básico com middleware**

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(`Requisição para: ${req.method} ${req.url}`);

next(); // segue para o próximo middleware ou rota

}

}

Aqui não usamos @Next() como decorator, pois ele só é necessário em **controllers ou handlers**. Em middlewares, ele vem como argumento.

**Exemplo com Controller (menos comum)**

import { Controller, Get, Req, Res, Next } from '@nestjs/common';

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

@Controller('test')

export class TestController {

@Get()

exemplo(@Req() req: Request, @Res() res: Response, @Next() next: NextFunction) {

if (req.query.skip === 'true') {

return next(); // pula essa rota e vai para a próxima

}

return res.send('Interceptado e respondido diretamente');

}

}

Se ?skip=true estiver na URL, ele vai para a **próxima rota compatível** com o path.

**@Session() - Sessão da requisição (quando ativado)**

**O que é o @Session()?**

@Session() session: Record<string, any>

O @Session() é um **decorator de parâmetro** que injeta o objeto de **sessão atual** no método do controller.

**Exemplo básico**

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

@Controller('contador')

export class ContadorController {

@Get()

contar(@Session() session: Record<string, any>) {

session.visitas = (session.visitas || 0) + 1;

return `Você visitou ${session.visitas} vezes`;

}

}

Toda vez que o usuário acessa a rota, a contagem aumenta e é mantida entre requisições (enquanto a sessão durar).

**Como funciona por trás?**

* O NestJS depende do [**express-session**](https://www.npmjs.com/package/express-session) para fornecer suporte a sessões.
* Isso significa que você **precisa configurar manualmente a sessão no main.ts**, assim:

// main.ts

import \* as session from 'express-session';

async function bootstrap() {

const app = await NestFactory.create(AppModule);

app.use(

session({

secret: 'chave-secreta',

resave: false,

saveUninitialized: false,

cookie: { maxAge: 60000 }, // 1 minuto

}),

);

await app.listen(3000);

}

Quando usar @Session()?

Autenticação sem JWT (via cookie) – SIM

Contador de acessos – SIM

Armazenar preferências temporárias – SIM

APIs REST modernas (stateless) – NÃO

**Atenção**

1. **Sessões são armazenadas em memória por padrão**, o que **não é ideal em produção**.
2. Você pode trocar por um armazenamento persistente (Redis, banco etc.).
3. Em aplicações **frontend modernas com JWT**, sessões são pouco usadas.

**@Ip() - IP do cliente**

**O que é o @Ip()?**

@Ip() ip: string

O @Ip() é um **decorator de parâmetro** que injeta automaticamente o **IP do cliente** na sua rota/controller.

**Exemplo de uso**

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

@Controller('acesso')

export class AcessoController {

@Get()

obterIp(@Ip() ip: string) {

return `Seu IP é: ${ip}`;

}

}

**Quando usar?**

Você pode usar o @Ip() para:

Auditoria de acessos - Registrar IP de login

Controle de região ou geolocalização - Bloquear IPs estrangeiros

Rate limiting por IP - Impedir muitos acessos do mesmo IP

Logs de segurança - Monitoramento de atividades

**Atenção com proxies!**

Se sua aplicação estiver **atrás de um proxy ou load balancer (como nginx)**, você deve ativar o trust proxy, ou o IP será sempre 127.0.0.1.

**Exemplo com proxy confiável:**

const app = await NestFactory.create(AppModule);

app.set('trust proxy', true); // Necessário para pegar o IP real

**@HostParam() - Parâmetros do host (subdomínios, etc)**

**O que é @HostParam()?**

O @HostParam() é um decorator que permite **acessar partes do hostname (domínio)** da requisição — como api.site.com ou user1.site.com.

Ele é usado junto com o decorator @Controller() configurado com um **host pattern**.

**Exemplo prático com subdomínio**

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

@Controller({ host: ':account.example.com' })

export class AccountController {

@Get()

getAccount(@HostParam('account') account: string) {

return `Você acessou a conta: ${account}`;

}

}

Se alguém acessa:

GET http://john.example.com/

A resposta será: Você acessou a conta: john

**Como funciona?**

* O @Controller({ host: ':param.exemplo.com' }) define uma **rota baseada em subdomínio**.
* O @HostParam('param') **extrai o valor** do subdomínio.

**@UploadedFile() - Arquivo enviado por upload (com @UseInterceptors)**

**O que é @UploadFile()?**

@UploadFile() file: Express.Multer.File

Ele injeta automaticamente o **arquivo enviado pelo cliente** no seu método, usando o multer por trás dos panos.

**Exemplo básico de upload de arquivo**

import { Controller, Post, UploadedFile, UseInterceptors } from '@nestjs/common';

import { FileInterceptor } from '@nestjs/platform-express';

@Controller('upload')

export class UploadController {

@Post()

@UseInterceptors(FileInterceptor('arquivo'))

upload(@UploadedFile() file: Express.Multer.File) {

return {

nome: file.originalname,

tipo: file.mimetype,

tamanho: file.size,

};

}

}

O campo 'arquivo' deve **bater com o nome do campo** enviado pelo formulário ou front-end.

**Requisitos**

**Instale o multer:**

npm install --save @nestjs/platform-express multer

**Estrutura do objeto file**

file = {

fieldname: 'arquivo',

originalname: 'foto.jpg',

encoding: '7bit',

mimetype: 'image/jpeg',

buffer: <Buffer>,

size: 102400

}

**Salvando no disco**

Se quiser armazenar o arquivo no sistema de arquivos:

@UseInterceptors(FileInterceptor('arquivo', {

dest: './uploads', // pasta onde será salvo

}))

**@UploadedFiles() - Vários arquivos (usando FilesInterceptor)**

**O que é @UploadedFiles()?**

@UploadedFiles() files: Express.Multer.File[]

Esse decorator injeta um **array de arquivos** enviados na requisição — funciona junto com o interceptor FilesInterceptor().

**Exemplo básico**

import { Controller, Post, UploadedFiles, UseInterceptors } from '@nestjs/common';

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

@Controller('upload-multiplos')

export class UploadMultiploController {

@Post()

@UseInterceptors(FilesInterceptor('arquivos', 5)) // até 5 arquivos

upload(@UploadedFiles() files: Express.Multer.File[]) {

return files.map((file) => ({

nome: file.originalname,

tipo: file.mimetype,

tamanho: file.size,

}));

}

}

O campo 'arquivos' deve ser o nome do campo do form que envia os múltiplos arquivos.

**Requisitos**

**1. Instalar as dependências:**

npm install --save @nestjs/platform-express multer

**2. Enviar a requisição com:**

* Content-Type: multipart/form-data
* Campo arquivos[] (com múltiplos arquivos) — pode ser feito no **Postman**, **insomnia**, ou **frontend** com um <input type="file" multiple />.

**Salvar arquivos no disco**

Se quiser que os arquivos sejam salvos fisicamente:

@UseInterceptors(FilesInterceptor('arquivos', 10, {

dest: './uploads', // pasta destino

}))

**@RequestTimeout() - Define o tempo máximo da requisição (pouco usado)**

**O que é o @RequestTimeout()?**

* Faz parte do módulo @nestjs/common.
* Permite **sobrescrever o timeout** padrão definido pelo TimeoutInterceptor.
* Se o processamento ultrapassar o tempo configurado, o NestJS retorna automaticamente um **HTTP 408 Request Timeout**.

**Sintaxe**

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

@Controller('demo')

export class DemoController {

// timeout de 5 segundos (5000 ms) para este handler

@Get()

@RequestTimeout(5000)

longRunningTask() {

return new Promise((resolve) => {

setTimeout(() => resolve({ ok: true }), 10\_000); // 10s → ultrapassa o timeout

});

}

}

* **Parâmetro obrigatório:** um número em milissegundos.
* Você pode aplicar em **métodos** ou **em toda classe** (decorando a classe).

**Como funciona por baixo**

1. **TimeoutInterceptor**  
   Por padrão, o NestJS já inclui um interceptor de timeout (se você habilitar globalmente).
2. **Decoração**  
   O @RequestTimeout(ms) altera o valor do timeout para aquele route-handler.
3. **Disparo de exceção**  
   Se o handler não terminar antes de ms, o interceptor lança um RequestTimeoutException (HTTP 408).

**Quando usar**

* Em handlers que possam cair em loops ou chamadas externas demoradas.
* Para evitar que requisições “presas” deixem suas threads/buffers ocupados indefinidamente.
* Quando você quer **diferentes limites** de tempo para diferentes rotas (ex: uploads longos vs consultas rápidas).

**Observações**

* Se você não habilitar o TimeoutInterceptor global, o decorator **não terá efeito**.
* Habilitação global padrão (por exemplo, em main.ts):

import { NestFactory } from '@nestjs/core';

import { AppModule } from './app.module';

import { TimeoutInterceptor } from '@nestjs/common';

async function bootstrap() {

const app = await NestFactory.create(AppModule);

app.useGlobalInterceptors(new TimeoutInterceptor());

await app.listen(3000);

}

bootstrap();

* O valor padrão de timeout (sem decorator) é de **5 segundos** (5000 ms).

Decorators de Injeção de Dependência

**@Inject() - Injeta manualmente uma dependência (caso o tipo não seja suficiente).**

**O que é @Inject()?**

O @Inject() é um **decorator** que permite **informar manualmente qual token (nome/identificador)** deve ser injetado em um construtor ou método.

Ele é mais usado quando o Nest **não consegue inferir automaticamente qual provider** deve ser injetado.

**Sintaxe básica**

import { Inject } from '@nestjs/common';

constructor(@Inject('TOKEN') private readonly variavel: Tipo) {}

Quando usar @Inject()?

Injetando serviço com nome padrão – NÃO

Injetando token customizado (ex: string, símbolo) – SIM

Injetando uma instância criada manualmente – SIM

Usando useClass, useValue, useFactory, useExisting – SIM

**Exemplo com token customizado (useValue)**

**Módulo:**

@Module({

providers: [

{

provide: 'API\_URL',

useValue: 'https://api.exemplo.com',

},

],

exports: ['API\_URL'],

})

export class ConfigModule {}

**Injeção com @Inject():**

@Injectable()

export class ApiService {

constructor(@Inject('API\_URL') private readonly url: string) {}

getUrl() {

return this.url;

}

}

**Exemplos por tipo de uso**

**useClass**

providers: [{ provide: 'CustomService', useClass: MinhaClasse }]

constructor(@Inject('CustomService') private readonly service: MinhaClasse) {}

**useFactory**

providers: [{

provide: 'UUID',

useFactory: () => randomUUID(),

}]

constructor(@Inject('UUID') private readonly id: string) {}

**Importante**

* O Nest tenta **resolver automaticamente** as dependências pelo tipo (ex: private readonly userService: UserService) → isso **só funciona se os nomes são diretos**.
* Se usar **tokens personalizados**, você **tem que usar @Inject()**.

**@Optional() - Injeta a dependência apenas se existir (evita erro de instância).**

**O que é @Optional()?**

O @Optional() é um **decorator de parâmetro** usado em conjunto com a **injeção de dependência** para indicar que o **parâmetro pode ser undefined** se o NestJS **não encontrar o provider** correspondente.

**Sintaxe**

import { Optional, Inject, Injectable } from '@nestjs/common';

@Injectable()

export class MeuServico {

constructor(@Optional() @Inject('NOME\_DO\_PROVIDER') private readonly opcional?: any) {}

}

**Exemplo prático**

@Injectable()

export class LoggerService {

log(msg: string) {

console.log(`[LOG]: ${msg}`);

}

}

**AppModule com condicional**

@Module({

providers: process.env.NODE\_ENV === 'dev' ? [LoggerService] : [],

})

export class AppModule {}

**Serviço usando @Optional()**

@Injectable()

export class AppService {

constructor(@Optional() private readonly logger?: LoggerService) {}

execute() {

if (this.logger) {

this.logger.log('Executando...');

}

}

}

Se LoggerService **não estiver registrado**, a injeção não quebra a aplicação — apenas resulta em undefined.

Sem @Optional(), causa erro:

Nest can't resolve dependencies of the AppService (LoggerService). Please make sure that the argument LoggerService at index [0] is available...

**@Self() - Injeta apenas do escopo atual.**

**O que é o @Self()?**

O @Self() diz ao NestJS:

**"Resolva essa dependência somente a partir do módulo atual (local), e não procure em módulos pais."**

Se o provider **não estiver declarado no módulo local**, o Nest lançará um erro, mesmo que exista em outro lugar.

**Sintaxe**

constructor(@Self() private readonly service: MeuServico) {}

**Exemplo prático**

**Imagine dois módulos:**

* SharedModule fornece LoggerService.
* FeatureModule NÃO fornece LoggerService.

**Código:**

@Injectable()

export class MinhaClasse {

constructor(@Self() private readonly logger: LoggerService) {}

}

**Resultado:**

* Se LoggerService **não estiver no módulo atual (FeatureModule)**, um erro é lançado:

Nest can't resolve dependencies of the MinhaClasse (LoggerService).

Comparando comportamento

Sem @Self()

Busca a dependência **nos pais**

Permite herança de providers

Mais permissivo

Com @Self()

Busca somente no módulo local

Não permite fallback para módulos pais

Mais restritivo

**Por que isso é útil?**

Imagine que você tem um provider específico para cada módulo, mas acidentalmente está usando o do módulo pai. O @Self() impede esse comportamento e força que a dependência venha da fonte certa.

**Cenário: Módulo compartilhado vs módulo específico**

**Objetivo:**

Você tem um LoggerService genérico no SharedModule, mas quer que o AdminModule tenha seu **próprio logger especial**, e **nunca** use o genérico por engano.

**Estrutura de módulos:**

- SharedModule → fornece LoggerService genérico

- AdminModule → deveria usar AdminLoggerService específico

**Implementação**

**SharedModule**

@Injectable()

export class LoggerService {

log(msg: string) {

console.log('[GENERIC]:', msg);

}

}

@Module({

providers: [LoggerService],

exports: [LoggerService],

})

export class SharedModule {}

**AdminLoggerService**

@Injectable()

export class AdminLoggerService {

log(msg: string) {

console.log('[ADMIN]:', msg);

}

}

**Sem @Self() → problema silencioso**

**AdminService.ts**

@Injectable()

export class AdminService {

constructor(private readonly logger: LoggerService) {}

run() {

this.logger.log('Executando lógica de admin...');

}

}

**AdminModule.ts**

@Module({

imports: [SharedModule], // ⚠ Não registra AdminLoggerService

providers: [AdminService],

})

export class AdminModule {}

**Resultado:** o NestJS injeta o LoggerService genérico do SharedModule, **porque você não registrou o AdminLoggerService localmente.**  
→ **Você nem percebe que está usando o errado.**

**Com @Self() → proteção explícita**

Agora usamos @Self() para **garantir que o provider seja local**:

**AdminService.ts**

@Injectable()

export class AdminService {

constructor(@Self() private readonly logger: AdminLoggerService) {}

run() {

this.logger.log('Executando lógica de admin...');

}

}

**AdminModule.ts (corrigido)**

@Module({

imports: [SharedModule], // ainda tem acesso ao genérico

providers: [AdminLoggerService, AdminService], // agora tem o certo

})

export class AdminModule {}

**Se esquecer o AdminLoggerService...**

Com @Self() ativado:

Nest can't resolve dependencies of the AdminService (AdminLoggerService).

Please make sure that the argument AdminLoggerService at index [0] is available in the AdminModule context.

✅ **Erro claro → mais segurança**  
❌ Sem @Self() → **risco de comportamento errado silencioso**

**@SkipSelf() - Injeta do escopo pai, pulando o atual.**

**O que é @SkipSelf()?**

O @SkipSelf() instrui o NestJS a **não procurar a dependência no escopo local**, ou seja, **ignorar o módulo onde o injetável está sendo usado** e **buscar em módulos externos ou pais**.

**Sintaxe**

constructor(@SkipSelf() private readonly service: AlgumServico) {}

Quando usar @SkipSelf()?

Você quer **forçar** que a dependência venha de um **módulo pai – SIM**

Você quer que o serviço **compartilhado** seja injetado, e não o local – SIM

Você quer que o Nest procure primeiro **neste módulo – NÃO**

**Exemplo prático**

**Cenário:**

Você tem um LoggerService genérico no SharedModule, mas **acidentalmente registrou um vazio** no FeatureModule, e isso está escondendo o real.

**Estrutura dos módulos:**

**SharedModule com o verdadeiro LoggerService**

@Injectable()

export class LoggerService {

log(msg: string) {

console.log('[SHARED LOG]:', msg);

}

}

@Module({

providers: [LoggerService],

exports: [LoggerService],

})

export class SharedModule {}

**FeatureModule com um provider sem implementação (acidental):**

@Injectable()

export class EmptyLoggerService {

log(msg: string) {

// não faz nada

}

}

@Module({

imports: [SharedModule],

providers: [{ provide: LoggerService, useClass: EmptyLoggerService }],

})

export class FeatureModule {}

**Problema:**

Você quer que FeatureService **use o logger real do SharedModule**, mesmo que exista um override local.

**Solução com @SkipSelf()**

@Injectable()

export class FeatureService {

constructor(@SkipSelf() private readonly logger: LoggerService) {}

run() {

this.logger.log('Rodando algo...');

}

}

Isso **ignora o override local (EmptyLoggerService)** e **pega o real do módulo pai**.

**@Host() - Injeta do módulo em que o consumidor está.**

**O que é o @Host()?**

O decorator @Host() permite que você:

**Associe um @Controller() ou um @Get() (ou outro método HTTP) a um host (domínio ou subdomínio) específico**.

Isso é útil, por exemplo, quando você quer comportamentos diferentes para:

* admin.exemplo.com
* api.exemplo.com
* exemplo.com

**Onde pode ser aplicado?**

* Em cima de um @Controller() → aplica a todos os métodos daquele controller.
* Em cima de métodos (@Get(), @Post()...) → aplica apenas naquela rota.

**Exemplo básico com subdomínio**

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

@Controller()

@Host('admin.exemplo.com')

export class AdminController {

@Get()

getAdminData() {

return 'Dados de admin';

}

}

Neste exemplo, a rota só responderá **se a requisição vier do host admin.exemplo.com**.

**Suporte a curingas**

Você pode usar \* como coringa:

@Host('\*.exemplo.com')

Assim, tanto admin.exemplo.com, user.exemplo.com quanto api.exemplo.com seriam aceitos.

**Exemplo com parâmetro de host**

Você pode capturar partes do domínio como parâmetros com @HostParam():

@Controller()

@Host(':account.exemplo.com')

export class AccountController {

@Get()

handle(@HostParam('account') account: string) {

return `Você está acessando a conta: ${account}`;

}

}

* A URL johann.exemplo.com vai retornar Você está acessando a conta: johann

**1. Suporte a múltiplos subdomínios em um único backend**

Imagine que você tem uma aplicação **multitenant** (multiempresa), como:

* empresa1.exemplo.com
* empresa2.exemplo.com

Com @Host(), você pode criar controladores **isolados** para cada um deles, sem precisar tratar isso manualmente dentro das rotas.

**Benefício:** separação de lógica por cliente, mantendo a organização limpa e segura.

**2. Isolamento de contexto**

Você pode criar rotas específicas para diferentes **tipos de acesso**, por exemplo:

admin.exemplo.com - Painel administrativo

api.exemplo.com - API pública

user.exemplo.com - Interface do usuário

Cada rota pode ter **middlewares**, **autenticação** e **serviços distintos**, controlados facilmente por @Host().

**3. Mais segurança e controle**

* Você pode garantir que **certas rotas só estejam acessíveis via um domínio específico**.
* Evita que usuários acessem rotas administrativas de lugares onde não deveriam.

**4. Organização do código**

Sem @Host(), você precisaria de lógica manual para analisar o host da requisição e redirecionar:

@Get()

getData(@Req() req: Request) {

const host = req.headers.host;

if (host.includes('admin')) { ... }

}

Com @Host(), essa lógica fica **declarativa e limpa**, sem if espalhado no código.

**5. Pronto para escalar com multitenancy**

Se um dia seu projeto precisar crescer para oferecer serviços para vários clientes (multi-tenant), o @Host() te dá uma base forte para:

* Roteamento por tenant
* Separação de dados por subdomínio
* Deploy único para múltiplas marcas

**Exemplo prático**

**Com rotas:**

@Controller('admin')

export class AdminController {}

@Controller('api')

export class ApiController {}

Tudo responde a exemplo.com/admin, exemplo.com/api.

**Com @Host():**

@Host('admin.exemplo.com')

@Controller()

export class AdminController {}

@Host('api.exemplo.com')

@Controller()

export class ApiController {}

Cada controller responde **apenas ao subdomínio específico** — isso não depende de rotas, e sim do domínio.

**Quando você realmente precisa de @Host()**

* Quando **clientes diferentes têm seus próprios domínios ou subdomínios** (ex: empresa1.app.com, empresa2.app.com);
* Quando você precisa **servir versões diferentes da aplicação** a partir do mesmo backend (ex: mobile.app.com, web.app.com);
* Quando quer usar **cookies/sessões isoladas por subdomínio** (evita conflito);
* Quando deseja **respostas ou middlewares específicos por domínio**, sem lógica condicional espalhada.

**Por que com rotas (/admin, /users) funciona no localhost...**

...mas com @Host() (usando subdomínios) é mais complexo?

Porque **os subdomínios não são controlados pelo NestJS ou pelo frontend** —  
**eles precisam ser configurados no DNS ou no arquivo de hosts do seu sistema operacional.**

**Quando usamos rotas:**

No localhost:3000/admin, o NestJS identifica a rota /admin e responde.  
O **host** continua sendo localhost, o que é simples de simular localmente.

**Quando usamos @Host('admin.site.com'):**

Aqui o NestJS só responde se o **host HTTP for exatamente** admin.site.com.

Exemplo:

GET http://admin.site.com/

Se você fizer um request para localhost, **o Nest ignora**, mesmo que a porta seja a mesma.

**Como simular isso no seu PC?**

**1. Use o arquivo hosts do sistema**

Você pode mapear subdomínios *falsos* para 127.0.0.1.

**🗂️ Windows:**

1. Abra como administrador:  
   C:\Windows\System32\drivers\etc\hosts
2. Adicione as linhas:

127.0.0.1 admin.localhost

127.0.0.1 api.localhost

1. Salve.

**2. Agora acesse pelo navegador ou frontend:**

* http://admin.localhost:3000 → vai bater no controller com @Host('admin.localhost')
* http://api.localhost:3000 → no controller com @Host('api.localhost')

✔️ Isso simula múltiplos domínios, **mesmo em ambiente local.**

**No Frontend, como o cliente acessa o domínio correto?**

1. Se for **SPA (React, Vue, etc)**:
   * O frontend é **hospedado sob o mesmo subdomínio**:
     + admin.exemplo.com serve o painel admin
     + app.exemplo.com serve a aplicação do usuário
   * O frontend se comunica com a **mesma origem**, então o Nest já entende qual subdomínio está acessando.
2. Se o frontend for separado:
   * Você pode usar axios, fetch ou GraphQL com base na origem:

const api = axios.create({

baseURL: window.location.hostname.includes('admin')

? 'https://admin.api.exemplo.com'

: 'https://api.exemplo.com',

});

Decorators de Guards, Pipes, Interceptors, etc.

**@UseGuards() - Usa um ou mais Guards (para autenticação, roles, etc.)**

## O que é um Guard?

Um **Guard** é uma classe que implementa a interface CanActivate e determina **se a requisição pode continuar ou não**.

Ele funciona **antes do controller**, parecido com middleware, mas com **lógica de permissão**.

## Exemplo real de uso do @UseGuards()

### 1. Criando um Guard:

import { CanActivate, ExecutionContext, Injectable } from '@nestjs/common';

@Injectable()

export class AuthGuard implements CanActivate {

canActivate(context: ExecutionContext): boolean {

const request = context.switchToHttp().getRequest();

// Exemplo: só permite continuar se houver token

return !!request.headers.authorization;

}

}

### 2. Aplicando com @UseGuards() em um controller:

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

import { AuthGuard } from './auth.guard';

@Controller('profile')

export class ProfileController {

@Get()

@UseGuards(AuthGuard)

getProfile() {

return { msg: 'Acesso liberado ao perfil' };

}

}

Se não houver authorization no header, o acesso será negado automaticamente com status **403 Forbidden**.

## Para que serve na prática?

* **Autenticação (JWT, OAuth, API Key...)**
* **Autorização por roles (admin, user, etc.)**
* **Verificar se um recurso pertence ao usuário**
* **Rate limit personalizado**

## Exemplo com múltiplos guards:

@UseGuards(AuthGuard, RolesGuard)

@Get('admin')

getAdminData() {

return 'Área de admin';

}

Todos os guards listados devem retornar true. Se **um só negar**, o acesso é bloqueado.

## O que acontece se negar?

Se um guard retornar false, o NestJS automaticamente retorna:

{

"statusCode": 403,

"message": "Forbidden"

}

**@UsePipes() - Usa um ou mais Pipes (validação, transformação, etc.)**

## O que é um Pipe?

Um **pipe** no NestJS é uma classe que implementa a interface PipeTransform. Ela possui um método transform(), que é chamado **antes do controller receber os dados**.

Pode ser usado para:

* ✅ **Validar** (ex: verificar se id é número)
* 🔄 **Transformar** (ex: converter string para número)
* ❌ **Rejeitar** (se os dados forem inválidos)

## Exemplo prático com @UsePipes()

### 1. Pipe para verificar se id é um número:

import { PipeTransform, Injectable, BadRequestException } from '@nestjs/common';

@Injectable()

export class ParseIntPipe implements PipeTransform {

transform(value: string) {

const val = parseInt(value, 10);

if (isNaN(val)) {

throw new BadRequestException('ID deve ser um número');

}

return val;

}

}

### 2. Usando com @UsePipes():

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

import { ParseIntPipe } from './parse-int.pipe';

@Controller('users')

export class UserController {

@Get(':id')

@UsePipes(ParseIntPipe)

getUser(@Param('id') id: number) {

return { id };

}

}

## Também pode usar pipes **globais**:

async function bootstrap() {

const app = await NestFactory.create(AppModule);

app.useGlobalPipes(new ValidationPipe());

await app.listen(3000);

}

## Exemplo com ValidationPipe e DTO:

import { IsString, IsEmail } from 'class-validator';

export class CreateUserDto {

@IsString()

name: string;

@IsEmail()

email: string;

}

ts

CopiarEditar

@Post()

@UsePipes(new ValidationPipe())

createUser(@Body() body: CreateUserDto) {

return body;

}

Se o corpo for inválido, você recebe um erro automático com os detalhes.

**@UseFilters() - Usa filtros para tratamento de exceções**

## O que é um Filter?

Um **filter** é uma classe que implementa a interface ExceptionFilter. Ele captura **erros lançados no controller, service, etc.**, e permite:

* Customizar a resposta da API
* Logar erros de forma especial
* Lidar com exceções específicas (ex: HttpException, NotFoundException, QueryFailedError, etc.)

## Exemplo simples

### 1. Criando um filtro de exceção:

import { ExceptionFilter, Catch, ArgumentsHost, HttpException } from '@nestjs/common';

import { Response } from 'express';

@Catch(HttpException)

export class HttpErrorFilter implements ExceptionFilter {

catch(exception: HttpException, host: ArgumentsHost) {

const ctx = host.switchToHttp();

const response = ctx.getResponse<Response>();

const status = exception.getStatus();

const message = exception.getResponse();

response.status(status).json({

success: false,

statusCode: status,

message,

timestamp: new Date().toISOString(),

});

}

}

### 2. Usando com @UseFilters():

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

import { HttpErrorFilter } from './http-error.filter';

@Controller('test')

export class TestController {

@Get()

@UseFilters(HttpErrorFilter)

getError() {

throw new BadRequestException('Esse erro foi capturado');

}

}

## Exemplo de uso global no main.ts:

import { HttpErrorFilter } from './common/filters/http-error.filter';

async function bootstrap() {

const app = await NestFactory.create(AppModule);

app.useGlobalFilters(new HttpErrorFilter());

await app.listen(3000);

}

## Quando usar?

* Tratamento customizado de erros (como UnauthorizedException)
* Logging personalizado em arquivos, bancos, etc.
* Padronizar todas as respostas de erro da sua API

## ❌ Sem @UseFilters()

Você recebe erros como este:

json

CopiarEditar

{

"statusCode": 400,

"message": "Erro padrão",

"error": "Bad Request"

}

## Com @UseFilters()

Você pode personalizar assim:

json

CopiarEditar

{

"success": false,

"statusCode": 400,

"message": "Esse erro foi capturado",

"timestamp": "2025-06-22T02:13:59.000Z"

}

**@UseInterceptors() - Usa interceptores para logs, cache, etc.**

## O que é um Interceptor?

Um **Interceptor** é uma classe que implementa a interface NestInterceptor, e pode ser usada para:

Manipular o request ou a response - Adicionar/remover campos do resultado

Logar tempo de execução - Calcular performance da rota

Modificar o resultado - Envolver a resposta em uma estrutura

Transformar erros - Converter exceções em mensagens padrão

Aplicar cache - Salvar e retornar dados do cache

## Exemplo real de uso

### 1. Criando um interceptor de log:

import {

Injectable,

NestInterceptor,

ExecutionContext,

CallHandler,

} from '@nestjs/common';

import { Observable, tap } from 'rxjs';

@Injectable()

export class LoggingInterceptor implements NestInterceptor {

intercept(context: ExecutionContext, next: CallHandler): Observable<any> {

console.log('Antes da execução...');

const now = Date.now();

return next.handle().pipe(

tap(() =>

console.log(`Depois... Tempo: ${Date.now() - now}ms`),

),

);

}

}

### 2. Aplicando com @UseInterceptors():

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

import { LoggingInterceptor } from './logging.interceptor';

@Controller('test')

export class TestController {

@Get()

@UseInterceptors(LoggingInterceptor)

getData() {

return { message: 'Executado com sucesso' };

}

}

## Exemplo de uso global (main.ts):

import { LoggingInterceptor } from './logging.interceptor';

async function bootstrap() {

const app = await NestFactory.create(AppModule);

app.useGlobalInterceptors(new LoggingInterceptor());

await app.listen(3000);

}

## Exemplo com transformação:

import {

Injectable,

NestInterceptor,

ExecutionContext,

CallHandler,

} from '@nestjs/common';

import { map, Observable } from 'rxjs';

@Injectable()

export class TransformInterceptor implements NestInterceptor {

intercept(context: ExecutionContext, next: CallHandler): Observable<any> {

return next.handle().pipe(

map((data) => ({

success: true,

data,

})),

);

}

}

**Resposta modificada:**

json

CopiarEditar

{

"success": true,

"data": {

"message": "Executado com sucesso"

}

}

**@SetMetadata() - Define metadados personalizados (usado com @Roles, por exemplo).**

## O que são metadados?

Metadados são **informações associadas a elementos do código** (como rotas ou classes) que **não fazem parte diretamente da lógica**, mas podem ser usadas em tempo de execução por outras ferramentas ou sistemas (ex: autenticação por função, permissões etc.).

## Sintaxe

@SetMetadata(CHAVE, VALOR)

## Exemplo real: controle de acesso por role

### 1. Criar um Guard baseado em roles:

import { CanActivate, ExecutionContext, Injectable } from '@nestjs/common';

import { Reflector } from '@nestjs/core';

@Injectable()

export class RolesGuard implements CanActivate {

constructor(private reflector: Reflector) {}

canActivate(context: ExecutionContext): boolean {

const roles = this.reflector.get<string[]>('roles', context.getHandler());

if (!roles) return true;

const request = context.switchToHttp().getRequest();

const user = request.user;

return roles.includes(user.role);

}

}

### 2. Definir os roles com @SetMetadata():

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

import { RolesGuard } from './roles.guard';

@Controller('admin')

export class AdminController {

@Get()

@SetMetadata('roles', ['admin']) // <-- Aqui está o uso

@UseGuards(RolesGuard)

getAdminData() {

return 'Acesso de administrador';

}

}

## Como funciona?

* @SetMetadata('roles', ['admin']) adiciona um metadado chamado 'roles' com o valor ['admin'].
* O RolesGuard lê esse metadado via Reflector e decide se o usuário pode acessar ou não.

## Exemplo alternativo: rota sem log

@SetMetadata('log', false)

@Get('status')

status() {

return { status: 'OK' };

}

No interceptor, você poderia ler:

const log = this.reflector.get<boolean>('log', context.getHandler());

## Dica: use constantes para evitar strings mágicas

export const ROLES\_KEY = 'roles';

@SetMetadata(ROLES\_KEY, ['admin'])

Assim evita erro de digitação e melhora a manutenção.

Outros Decorators comuns

**@Catch() - Usado em filtros de exceção personalizados**

## O que é @Catch()?

É um **decorator de classe** que define **quais tipos de erro o filtro vai capturar**.

Você o usa junto com classes que implementam a interface ExceptionFilter.

## Sintaxe:

@Catch(HttpException)

export class MeuFiltro implements ExceptionFilter {

catch(exception: HttpException, host: ArgumentsHost) {

// tratamento aqui

}

}

Você pode capturar:

* **Uma exceção específica**: @Catch(BadRequestException)
* **Múltiplas exceções**: @Catch(BadRequestException, NotFoundException)
* **Todas as exceções**: @Catch() (sem parâmetros)

## Exemplo real

### 1. Filtro personalizado:

import { ExceptionFilter, Catch, ArgumentsHost, HttpException } from '@nestjs/common';

import { Response } from 'express';

@Catch(HttpException)

export class HttpErrorFilter implements ExceptionFilter {

catch(exception: HttpException, host: ArgumentsHost) {

const ctx = host.switchToHttp();

const response = ctx.getResponse<Response>();

const status = exception.getStatus();

const message = exception.getResponse();

response.status(status).json({

success: false,

statusCode: status,

message,

timestamp: new Date().toISOString(),

});

}

}

### 2. Usando com @UseFilters():

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

import { HttpErrorFilter } from './http-error.filter';

@Controller('test')

export class TestController {

@Get()

@UseFilters(HttpErrorFilter)

getError() {

throw new BadRequestException('Erro tratado pelo filtro');

}

}

## Capturar todos os erros:

@Catch()

export class AllExceptionsFilter implements ExceptionFilter {

catch(exception: any, host: ArgumentsHost) {

console.error('Erro capturado:', exception);

const ctx = host.switchToHttp();

const response = ctx.getResponse();

response.status(500).json({

success: false,

message: 'Erro interno do servidor',

});

}

}

**@Roles() - Define roles para um método (usado com guard)**

### Para que serve @Roles()?

Ele associa um ou mais "papéis" (roles) a um endpoint, como por exemplo:

@Roles('admin') // Apenas administradores acessam

Isso **não funciona sozinho**. Ele é um **atalho para @SetMetadata('roles', [...])**, e precisa de um **guard** que use o Reflector para verificar os papéis.

## Como criar @Roles()

import { SetMetadata } from '@nestjs/common';

export const ROLES\_KEY = 'roles';

export const Roles = (...roles: string[]) => SetMetadata(ROLES\_KEY, roles);

## Exemplo completo de uso

### 1. Guard que lê os roles:

import { CanActivate, ExecutionContext, Injectable } from '@nestjs/common';

import { Reflector } from '@nestjs/core';

@Injectable()

export class RolesGuard implements CanActivate {

constructor(private reflector: Reflector) {}

canActivate(context: ExecutionContext): boolean {

const requiredRoles = this.reflector.getAllAndOverride<string[]>('roles', [

context.getHandler(),

context.getClass(),

]);

if (!requiredRoles) return true;

const { user } = context.switchToHttp().getRequest();

return requiredRoles.includes(user.role); // role precisa estar no request

}

}

### 2. Usando @Roles() em um controller:

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

import { Roles } from './roles.decorator';

import { RolesGuard } from './roles.guard';

@Controller('admin')

@UseGuards(RolesGuard)

export class AdminController {

@Get()

@Roles('admin')

getAdminContent() {

return 'Somente administradores veem isso';

}

}

## Quando usar @Roles()?

* Criar áreas restritas por tipo de usuário
* Proteger rotas específicas (ex: /admin, /financeiro)
* Integrar com sistemas de autenticação (JWT, OAuth, etc.)

**@HttpCode() - Define manualmente o status code de uma resposta (ex: 201 em POST)**

O decorator @HttpCode() no NestJS é usado para **alterar o código de status HTTP da resposta** de uma rota. Ele é útil quando você deseja retornar um **código diferente do padrão** para uma determinada requisição.

## Sintaxe

@HttpCode(statusCode: number)

## Exemplo básico

import { Controller, Post, HttpCode } from '@nestjs/common';

@Controller('auth')

export class AuthController {

@Post('login')

@HttpCode(200) // <- sem isso, o padrão seria 201 (Created)

login() {

return { message: 'Login realizado com sucesso' };

}

}

### Por padrão:

* Métodos @Post() → retornam **201 (Created)**
* Métodos @Put() e @Patch() → retornam **200**
* Métodos @Delete() → retornam **200**
* Métodos @Get() → retornam **200**

Se você quiser **modificar** o status (por exemplo, retornar 204 No Content, ou 202 Accepted), você usa @HttpCode().

## Exemplo com @Delete()

@Delete(':id')

@HttpCode(204)

removeUser(@Param('id') id: string) {

// Lógica de remoção

}

**Resposta:** status 204 (sem conteúdo)

1. Criar módulos, controllers e services

nest g module users

nest g controller users

nest g service users

Erro:

Delete '␣' eslint(prettier/prettier)

Solução:

npx prettier --write .