

O Gato Sagrado da Torre do Guardião



NestJS

Autenticação e Autorização



Exemplo aplicado:

- Introdução à Autenticação e Autorização
- Fundamentos do NestJS e Arquitetura em Módulos
- Configuração do PostgreSQL
- Configuração do Prisma (Desacoplado)
- Criação das Entidades: Usuário, Perfil e Permissões
- Implementando Autenticação com JWT
- Implementando Autorização com Guards e Decorators
- Testando com Insomnia
- Conclusão e Boas Práticas

Adendo: Apesar de o e-book apresentar a implementação de uma API básica, é importante ressaltar que ela não atende a todas as especificações de uma *API RESTful* completa. O foco principal deste material está em demonstrar os conceitos e a implementação prática de autenticação e autorização. Você pode encontrar a implementação da API em:



[https://github.com/PjBierhals/ebook-DIO-NESTJS.](https://github.com/PjBierhals/ebook-DIO-NESTJS)

■ Capítulo 1:

Introdução

Neste eBook, vamos abordar os conceitos de autenticação e autorização utilizando o framework NestJS, integrando com o Prisma ORM de forma desacoplada, utilizando banco de dados PostgreSQL, JWT e Passport para segurança, e o Insomnia como ferramenta de testes.



Capítulo 2:

Conceitos de Autenticação e Autorização

Autenticação: é o processo de identificar e verificar se um usuário é quem ele diz ser. Normalmente, isso é feito através de credenciais como e-mail e senha, mas também pode envolver autenticação multifator (2FA), biometria ou tokens de acesso temporários.

Exemplo: Imagine que você está acessando um sistema de gestão de tarefas. Você precisa inserir seu e-mail e senha na tela de login. O sistema então valida essas informações. Se estiverem corretas, você está autenticado e recebe um token para interagir com o sistema.

Autorização: ocorre após a autenticação e determina o que o usuário autenticado pode ou não fazer dentro do sistema. Isso normalmente é gerenciado por meio de papéis (roles) e permissões.

Exemplo: Um usuário com papel de admin pode acessar todas as rotas, como /users ou /settings. Já um usuário com papel de editor pode apenas acessar rotas como /posts ou /comments, sem poder ver ou modificar configurações administrativas.

A separação entre autenticação e autorização permite que o sistema identifique o usuário corretamente e aplique políticas de controle de acesso conforme suas permissões.

Capítulo 3:

Padrões do NestJS

NestJS é um framework Node.js para construir aplicações escaláveis do lado do servidor. Ele segue uma arquitetura fortemente inspirada no Angular, usando decorators, injeção de dependências, módulos, serviços e controladores. Site oficial: <https://docs.nestjs.com>

A estrutura modular ajuda a manter o projeto organizado, escalável e testável. Veja uma estrutura básica típica de uma aplicação com autenticação:

```
src/
  └── auth/
    ├── auth.controller.ts          # Módulo responsável por autenticação
    ├── auth.service.ts            # Controlador com rotas de login e registro
    ├── auth.module.ts             # Serviço com lógica de autenticação e geração de tokens
    └── dto/
      └── login.dto.ts            # Módulo que agrupa os componentes do domínio de autenticação
        # Objetos de transferência de dados
        # DTO usado para validar o login
      └── strategies/
        └── jwt.strategy.ts        # Estratégias de autenticação (JWT, Local, etc)
          # Implementação da estratégia JWT

  └── users/
    ├── users.controller.ts        # Módulo responsável pelo gerenciamento de usuários
    ├── users.service.ts          # Controlador com rotas relacionadas a usuários
    ├── users.module.ts           # Serviço com regras de negócios de usuários
    └── entities/
      └── user.entity.ts          # Módulo que agrupa os componentes do domínio de usuários
        # Entidades do domínio do usuário
        # Entidade que representa o usuário no banco de dados

  └── common/
    ├── guards/
    └── decorators/               # Recursos reutilizáveis
      # Guards como JwtAuthGuard, RoleGuard, etc.
      # Decorators personalizados como @Roles()

  └── prisma/
    └── prisma.service.ts         # Serviço de acesso ao banco encapsulado
      # Serviço que inicializa e fornece o PrismaClient

  └── app.module.ts              # Módulo raiz da aplicação que importa todos os módulos
  └── main.ts                    # Ponto de entrada da aplicação (bootstrap)
```

Capítulo 4:

Arquiteturas do NestJS

A **arquitetura modular** do NestJS organiza o código em módulos independentes. Cada módulo encapsula uma funcionalidade específica do sistema (como AuthModule, UsersModule, etc). Isso promove a reutilização, testabilidade e facilita a manutenção do código.

Controllers são responsáveis por receber as requisições HTTP e encaminhá-las para os serviços. Eles atuam como intermediários entre o mundo externo e a lógica de negócios.

Services encapsulam a lógica de negócios. São injetados nos controllers e também podem ser reutilizados em outros services. Eles interagem com repositórios, bancos de dados ou qualquer lógica necessária para a aplicação.

Vantagens:

- Separação clara de responsabilidades.
- Facilita testes unitários e integração.
- Escalabilidade da aplicação.

```
1 @Module({
2   imports: [UsersModule, AuthModule],
3   controllers: [AppController],
4   providers: [AppService],
5 })
6 export class AppModule {}
```



Capítulo 5:

Criando o App com NestJS

Para iniciar uma aplicação *NestJS*, siga os seguintes passos:

1. Abra o Prompt de comando
2. Instale o Nest CLI globalmente (caso não tenha):

```
● ● ●  
npm install -g @nestjs/cli
```

3. Crie a pasta no local desejado:

```
● ● ●  
mkdir E-Book
```

4. Crie a aplicação:

```
● ● ●  
nest new nest-auth-app
```

5. Escolha o gerenciador de pacotes:

```
● ● ●  
? Which package manager would you ❤️ to use? (Use arrow keys)  
> npm  
yarn  
pnpm
```

6. Após a instalação, entre na pasta do projeto ou abra direto no vscode:

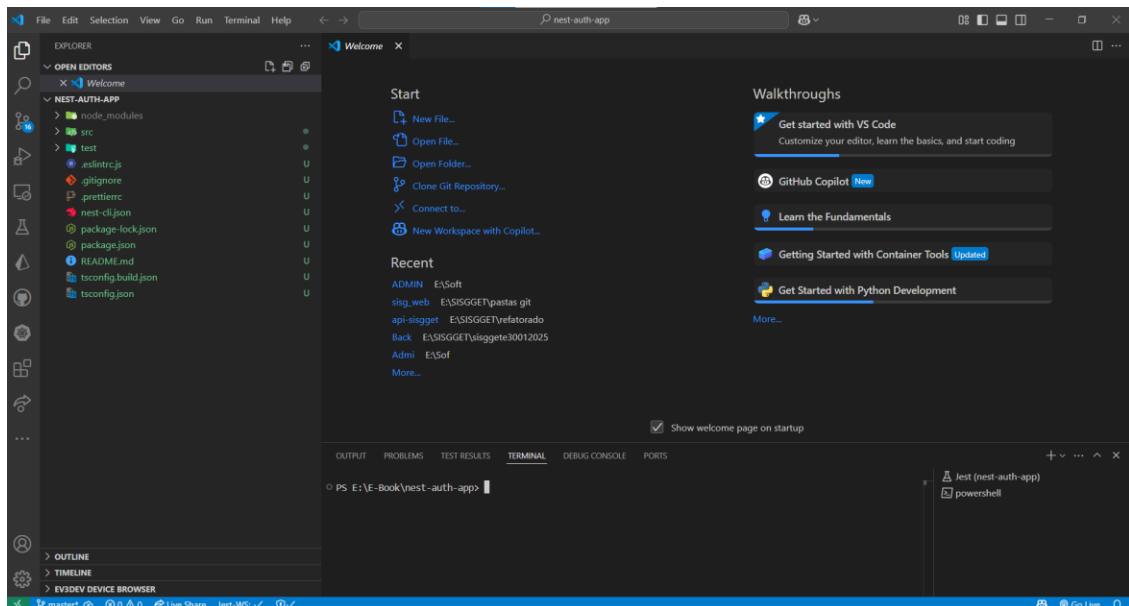
```
● ● ●  
E:\E-Book>code nest-auth-app
```

Capítulo 6:

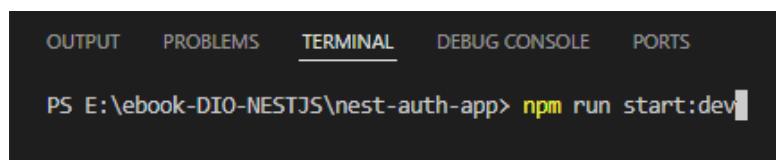
Criando o App com NestJS

O projeto abrira no framework **vscode**:

Site oficial: <https://code.visualstudio.com/>



No terminal do vscode :



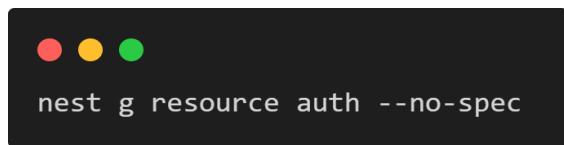
```
PS E:\ebook-DI0-NESTJS\nest-auth-app> npm run start:dev
```

Abra o navegador: <http://localhost:3000/>

Capítulo 7:

Criando Estrutura de Recursos (CRUDs)

No terminal do vscode:



```
● ● ●
nest g resource auth --no-spec
```

- **nest**->Esse é o comando principal que chama a CLI (Interface de Linha de Comando) do NestJS.
- **g**->Abreviação de generate, que significa "gerar". A CLI entende que você está pedindo para gerar algum tipo de arquivo ou estrutura.
- **resource**->Diz à CLI que você quer gerar um "recurso". No contexto do NestJS, um recurso é um conjunto completo que inclui um controller, service, módulo e, opcionalmente, DTOs e arquivos de teste. Ou seja, é uma funcionalidade completa já estruturada.
- **auth**-> É o nome do recurso que você quer criar. Neste caso, você está dizendo que quer um recurso chamado "auth".
- **--no-spec**->Esse é um modificador (flag) que indica que você **não** quer que a CLI gere arquivos de teste (os arquivos .spec.ts). Sem essa flag, o Nest geraria esses testes automaticamente para o controller e o service.



Capítulo 7:

Criando Estrutura de Recursos (CRUDs)

O NestJS CLI vai perguntar qual camada de transporte você quer usar para o recurso que está criando. Escolha **REST API**.

```
● ● ●
What transport layer do you use? (Use arrow keys)
> REST API
  GraphQL (code first)
  GraphQL (schema first)
  Microservice (non-HTTP)
  WebSockets

● ● ●
? What transport layer do you use? REST API
? Would you like to generate CRUD entry points? (Y/n) y
```

Você gostaria de gerar pontos de entrada CRUD? Responda: **Y "sim"**.

O NestJS irá gerar automaticamente os métodos padrão do **CRUD** no controller e no service do recurso. Acelerando o desenvolvimento inicial, principalmente se você estiver criando uma API padrão com **Create, Read, Update e Delete**.

Criando o restante. No terminal do vscode:

```
● ● ●
nest g resource users --no-spec
nest g resource roles --no-spec
nest g resource permissions --no-spec
nest g resource posts --no-spec
nest g resource comments --no-spec
```

Capítulo 8:

Prisma

O **Prisma** é um ORM (Object Relational Mapping) moderno que facilita o acesso a bancos de dados usando TypeScript. No NestJS, é recomendável manter o Prisma desacoplado usando um módulo próprio. Site oficial: <https://www.prisma.io>

- Instalando o Prisma:

```
● ● ●  
npm install prisma --save-dev
```

- Iniciando o prisma:

```
● ● ●  
npx prisma init
```

- Edite o arquivo .env:

```
● ● ●  
DATABASE_URL="postgresql://usuario:senha@localhost:5432/nest_auth"
```

- **postgresql://**->Indica o tipo do banco de dados. Aqui é PostgreSQL. Outros exemplos seriam mysql:// ou mongodb://.
- **usuario**->Nome de usuário usado para acessar o banco de dados. Esse é o login configurado no PostgreSQL.
- **senha**->A senha correspondente ao usuário. É usada para autenticar a conexão.
- **localhost**->É o endereço do servidor do banco de dados apontando para o próprio computador.
- **5432**-> É a porta padrão usada pelo PostgreSQL. Se você não alterou isso na configuração do banco.
- **nest_auth**->É o nome do banco de dados ao qual você está se conectando.

Capítulo 9:

Configuração do Prisma ORM.

O arquivo ***schema.prisma*** é o arquivo principal de configuração do Prisma ORM. Ele define:

- Define o banco de dados e o client Prisma.
- Cria modelos que viram tabelas no banco.
- Permite gerar tipos e consultas seguras em TypeScript.
- Tabela ***user***:

```
● ● ●
model User {
    id      String    @id @default(uuid())
    name    String
    email   String    @unique
    password String
    role    Role? @relation(fields: [roleId], references: [id])
    roleId String?
    posts   Post[]
    comments Comment[]
}
```

- Tabela ***role***:

```
● ● ●
model Role {
    id      String    @id @default(uuid())
    name    String    @unique
    permissions Permission[] @relation("RolePermissions")
    users   User[]    @relation("UserRoles")
}
```

Capítulo 9:

Configuração do Prisma ORM.

- Tabela **permission**:

```
● ● ●  
model Permission {  
    id      String @id @default(uuid())  
    name    String @unique  
    roles   Role[] @relation("RolePermissions")  
}
```

- Tabela **post**:

```
● ● ●  
model Post {  
    id      String    @id @default(uuid())  
    title   String  
    content  String  
    authorId String  
    author   User      @relation(fields: [authorId], references: [id])  
    comments Comment[]  
    createdAt DateTime @default(now())  
}
```

- Tabela **comment**:

```
● ● ●  
model Comment {  
    id      String    @id @default(uuid())  
    content  String  
    postId   String  
    authorId String  
    post     Post      @relation(fields: [postId], references: [id])  
    author   User      @relation(fields: [authorId], references: [id])  
    createdAt DateTime @default(now())  
}
```

- Por último, execute o comando que criar e aplicar uma migração no banco de dados. No terminal do vsode:

```
npx prisma migrate dev --name init
```



Capítulo 10:

Configuração o serviço do Prisma ORM.

O **PrismaService** é uma classe personalizada que estende o **PrismaClient** (do Prisma ORM) para que ele possa ser usado com injeção de dependência no NestJS. No terminal do vscode:

```
● ● ●  
nest g service prisma --no-spec
```

Classe:

```
● ● ●  
import { Injectable, OnModuleInit } from '@nestjs/common';  
import { PrismaClient } from 'generated/prisma';  
  
@Injectable()  
export class PrismaService extends PrismaClient implements OnModuleInit {  
    async onModuleInit() {  
        await this.$connect();  
        console.log('Prisma conectado!');  
    }  
  
    async onModuleDestroy() {  
        await this.$disconnect();  
        console.log('Prisma desconectado!');  
    }  
}
```



Capítulo 10:

Configuração o serviço do Prisma ORM.

No NestJS, um **Module** é uma classe que organiza e agrupa partes da aplicação, como controllers, services e outros modulos. Ele funciona como um “container” que declara o que aquele módulo oferece e o que ele precisa. O **Decorator @Global()** deixa o modulo disponível para toda a aplicação.

Criando Module. No terminal do vscode:

```
● ● ●  
nest g module prisma --no-spec
```

Classe:

```
● ● ●  
import { Global, Module } from '@nestjs/common';  
import { PrismaService } from './prisma.service';  
@Global()  
@Module({  
  providers: [PrismaService],  
  exports: [PrismaService],  
})  
export class PrismaModule {}
```



Capítulo 11:

Implementando HASH na senhas da API.

Senhas nunca devem ser armazenadas em texto puro no banco de dados. Em vez disso, usa-se **hashing**, que é um processo irreversível para proteger as senhas. Ferramentas como **bcrypt** aplicam um **salt** (valor aleatório) à senha antes de gerar o hash, tornando cada resultado único. Isso impede que senhas iguais tenham o mesmo hash e dificulta ataques. Ao autenticar, compara-se o hash da senha fornecida com o armazenado. Essa prática garante mais segurança mesmo em caso de vazamento do banco de dados.

No terminal do vscode:



```
npm install bcrypt -D @types/bcrypt
```



```
nest g service auth/hashing --no-spec
```



```
nest g service auth/hashing/bcrypt --no-spec --flat
```

Esses comandos geralmente fazem parte de uma estrutura de **injeção de dependência baseada em estratégia**:

- **hashing.service** é uma **abstração genérica**, uma classe abstrata.
- **bcrypto.Service** é uma implementação concreta que usa o pacote bcrypt para hashing de senhas.

Capítulo 11:

Implementando HASH na senhas da API.

Abstract class HashingService é uma **classe abstrata**: ela **não pode ser instanciada diretamente**, Serve como um **contrato**: qualquer classe que estenda HashingService deve implementar os métodos definidos.

Essa abordagem permite que você tenha múltiplas implementações (ex: Bcrypt, Argon2)

Classe:

```
● ● ●  
import { Injectable } from '@nestjs/common';  
  
@Injectable()  
export abstract class HashingService {  
    abstract hash(data: string | Buffer): Promise<string>;  
    abstract compare(data: string | Buffer, encrypted: string): Promise<boolean>;  
}
```

- **hash**: recebe um texto ou buffer e retorna uma `Promise<string>` com o **hash gerado**.
- **compare**: recebe o texto original e o hash, e retorna uma `Promise<boolean>` indicando se a **senha corresponde ao hash**.

Capítulo 11:

Implementando HASH na senhas da API.

A classe BcryptService que estende a classe abstrata HashingService para fazer **hash e verificação de senhas** usando a biblioteca bcrypt.

Classe:

```
● ● ●  
import { Injectable } from '@nestjs/common';
import { HashingService } from './hashing.service';
import { compare, genSalt, hash } from 'bcrypt';

@Injectable()
export class BcryptService extends HashingService {
    private readonly saltRounds = 12;
    async hash(data: string | Buffer): Promise<string> {
        const salt = await genSalt(this.saltRounds);
        return hash(data, salt);
    }

    async compare(data: string | Buffer, encrypted: string): Promise<boolean> {
        return compare(data, encrypted);
    }
}
```

- O **saltRounds** é que define o custo do hash (tempo de processamento). Quanto maior, mais seguro (mas mais lento). Valor comum: entre 10 e 14.
- Método **hash**, gera um hash seguro da senha usando bcrypt com salt de 12 rounds.
- Método **compare**, Compara a senha fornecida com o hash armazenado para verificar se são iguais.



Capítulo 12:

Instalando bibliotecas.

Vamos instalar duas bibliotecas essenciais para validação e transformação de objetos em NestJS:

- **class-validator**: permite aplicar validações com decorators (ex: `@IsEmail()`, `@Length()`) diretamente nas classes DTO.
- **class-transformer**: transforma objetos JSON em instâncias de classes (e vice-versa), útil para aplicar validações corretamente.

Essas duas são amplamente usadas juntas em rotas que recebem dados do corpo da requisição (`@Body()`), garantindo que os dados estejam no formato esperado.

No terminal do vscode:

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

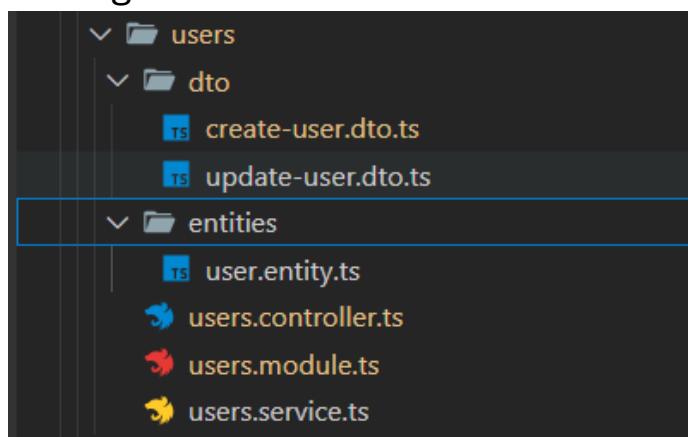
Capítulo 13:

Descrevendo módulo.

Um **módulo por resource** no NestJS organiza o código de cada funcionalidade (como users, auth, etc.) em uma estrutura padronizada com:

- **module.ts**: registra os componentes do recurso.
- **controller.ts**: define as rotas e recebe os dados da requisição.
- **service.ts**: contém a lógica de negócio.
- **dto/**: armazena os objetos de validação dos dados (ex: **create-user.dto.ts**).
- **entities/**: define a estrutura da entidade (modelo de dados).

Essa abordagem facilita a organização, escalabilidade e reutilização do código.





Capítulo 14:

Implementando módulos "Métodos".

Uma API (Interface de Programação de Aplicações) expõe rotas que são acessadas por meio de métodos HTTP. Cada método representa uma ação sobre os dados no sistema:

- ***create***: Responsável por criar um novo registro no banco de dados.
- ***findAll***: Retorna todos os registros existentes (por exemplo, todos os usuários).
- ***findOne***: Retorna um único registro com base no seu ID.
- ***findEmail***: Retorna um único registro com base no e-mail (semelhante ao findOne, mas com outro critério).
- ***update***: Atualiza os dados de um registro existente, identificando-o pelo ID.
- ***remove***: Exclui permanentemente um registro específico com base no ID.



Capítulo 15:

Implementando módulo "permissions".

Implementando arquivo ***create-permission.dto.ts***:

```
● ● ●  
import { IsNotEmpty, IsString, MinLength } from 'class-validator';  
export class CreatePermissionDto {  
    @IsNotEmpty({ message: 'É Obrigatório' })  
    @IsString({ message: 'Deve ser uma string.' })  
    @MinLength(3, { message: 'No mínimo 3' })  
    name: string;  
}
```

O arquivo ***update-permission.dto.ts*** herda todos os campos de ***create-permission***, mas torna todos eles opcionais.

PartialType(): é um helper do `@nestjs/mapped-types` que cria uma versão com todos os campos opcionais.

```
● ● ●  
import { PartialType } from '@nestjs/mapped-types';  
import { CreatePermissionDto } from './create-permission.dto';  
  
export class UpdatePermissionDto extends PartialType(CreatePermissionDto) {}
```



Capítulo 15:

Implementando módulo "permissions".

Implementando arquivo ***permissions.service.ts***:

O método construtor recebe a injeção do ***PrismaService***, que permite acessar o banco de dados utilizando o ***Prisma***, um ORM (Object-Relational Mapping) que facilita a interação com o banco de dados ao abstrair operações como ***SELECT, INSERT, UPDATE*** e ***DELETE*** de forma segura e eficiente.



```
async create(createPermissionDto: CreatePermissionDto) {
    return await this.prisma.permission.create({ data: createPermissionDto });
}

async findAll() {
    return await this.prisma.permission.findMany();
}

async findOne(id: string) {
    return await this.prisma.permission.findUnique({
        where: { id },
    });
}

async update(id: string, updatePermissionDto: UpdatePermissionDto) {
    return await this.prisma.permission.update({
        where: { id },
        data: updatePermissionDto,
    });
}

async remove(id: string) {
    return await this.prisma.permission.delete({
        where: { id },
    });
}
```



Capítulo 15:

Implementando módulo "permissions".

Implementando arquivo ***permissions.controller.ts***:

No NestJS, o controller é responsável por receber as requisições **HTTP** (como **GET**, **POST**, **PUT**, **DELETE**), chamar os serviços apropriados e retornar a resposta ao cliente. Ele atua como intermediário entre o cliente e a lógica de negócio (service).

```
● ● ●

@Controller('permissions')
export class PermissionsController {
    constructor(private readonly permissionsService: PermissionsService) {}

    @Post()
    create(@Body() createPermissionDto: CreatePermissionDto) {
        return this.permissionsService.create(createPermissionDto);
    }

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

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

    @Patch(':id')
    update(
        @Param('id') id: string,
        @Body() updatePermissionDto: UpdatePermissionDto,
    ) {
        return this.permissionsService.update(id, updatePermissionDto);
    }

    @Delete(':id')
    remove(@Param('id') id: string) {
        return this.permissionsService.remove(id);
    }
}
```



Capítulo 16:

Implementando módulo "roles".

Implementando arquivo ***create-role.dto.ts***:

```
● ● ●  
import { IsArray, IsNotEmpty, IsOptional, IsString, MaxLength } from 'class-validator';  
  
export class CreateRoleDto {  
    @IsNotEmpty({ message: 'O nome da função é obrigatório.' })  
    @IsString({ message: 'O nome da função deve ser uma string.' })  
    @MaxLength(50, {  
        message: 'O nome da função não pode ter mais de 50 caracteres.',  
    })  
    name: string;  
    @IsOptional()  
    @IsArray({ message: 'As permissões devem ser um array de IDs.' })  
    permissions?: string[];  
    permissionIds?: string[];  
}
```

O arquivo ***update-role.dto.ts*** herda todos os campos de ***create-role***, mas torna todos eles opcionais.

```
● ● ●  
import { PartialType } from '@nestjs/mapped-types';  
import { CreateRoleDto } from './create-role.dto';  
  
export class UpdateRoleDto extends PartialType(CreateRoleDto) {}
```



Capítulo 16:

Implementando módulo "roles".

Implementando arquivo **roles.service.ts**:

O método construtor recebe a injeção do **PrismaService**.



```
@Injectable()
export class RolesService {
    constructor(private readonly prisma: PrismaService) {}

    async create(createRoleDto: CreateRoleDto) {
        const { name, permissions } = createRoleDto;

        const uniquePermissionIds = new Set<string>();
        if (permissions && permissions.length > 0) {
            permissions.forEach((id) => uniquePermissionIds.add(id));
        }

        const permissionsToConnect = Array.from(uniquePermissionIds).map((id) => ({
            id,
        }));

        return await this.prisma.role.create({
            data: {
                name: name,
                permissions: {
                    connect: permissionsToConnect,
                },
            },
            include: {
                permissions: true,
            },
        });
    }
}
```



Capítulo 16:

Implementando módulo "roles".

Implementando arquivo **roles.service.ts**:

```
● ● ●

async findAll() {
    return await this.prisma.role.findMany({
        include: {
            permissions: true,
        },
    });
}

async findOne(id: string) {
    return await this.prisma.role.findUnique({
        where: { id },
        include: {
            permissions: true,
        },
    });
}

async remove(id: string) {
    return await this.prisma.role.delete({
        where: { id },
    });
}
```



Capítulo 16:

Implementando módulo "roles".

Implementando arquivo **roles.service.ts**:

```
● ● ●

async update(id: string, updateRoleDto: UpdateRoleDto) {
    const { name, permissions } = updateRoleDto;

    const existingRole = await this.prisma.role.findUnique({
        where: { id },
    });

    if (!existingRole) {
        throw new NotFoundException(`Função com ID "${id}" não encontrada.`);
    }

    const updateData: {
        name?: string;
        permissions?: {
            set?: { id: string }[];
        };
    } = {};

    if (name !== undefined) {
        updateData.name = name;
    }

    if (permissions !== undefined) {
        const uniquePermissions = new Set<string>(permissions);

        const permissionsToSet = Array.from(uniquePermissions).map((id) => ({
            id,
        }));

        updateData.permissions = {
            set: permissionsToSet,
        };
    }

    if (Object.keys(updateData).length === 0) {
        return existingRole;
    }

    return await this.prisma.role.update({
        where: { id },
        data: updateData,
        include: {
            permissions: true,
        },
    });
}
```



Capítulo 16:

Implementando módulo "rules".

Implementando arquivo **roles.controller.ts**:



```
@Controller('roles')
export class RolesController {
    constructor(private readonly rolesService: RolesService) {}

    @Post()
    create(@Body() createRoleDto: CreateRoleDto) {
        return this.rolesService.create(createRoleDto);
    }

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

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

    @Patch(':id')
    update(@Param('id') id: string, @Body() updateRoleDto: UpdateRoleDto) {
        return this.rolesService.update(id, updateRoleDto);
    }

    @Delete(':id')
    remove(@Param('id') id: string) {
        return this.rolesService.remove(id);
    }
}
```



Capítulo 17:

Implementando módulo "users".

Implementando arquivo ***create-user.dto.ts***:

```
● ● ●  
import { IsString, IsEmail, MinLength, IsOptional } from 'class-validator';  
  
export class CreateUserDto {  
    @IsString()  
    name: string;  
  
    @IsEmail()  
    email: string;  
  
    @IsString()  
    @MinLength(6)  
    password: string;  
  
    @IsOptional()  
    role?: string;  
}
```

O arquivo ***update-user.dto.ts*** herda todos os campos de ***CreateUserDto***, mas torna todos eles opcionais.

```
● ● ●  
import { PartialType } from '@nestjs/mapped-types';  
import { CreateUserDto } from './create-user.dto';  
  
export class UpdateUserDto extends PartialType(CreateUserDto) {}
```



Capítulo 17:

Implementando módulo "users".

Implementando arquivo ***users.service.ts***:



```
async create(createUserDto: CreateUserDto) {
    const { password, role, ...userData } = createUserDto;
    const passwordHash = await this.Crypt.hash(password);

    const data: any = {
        ...userData,
        password: passwordHash,
    };

    if (role) {
        data.role = {
            connect: { id: role },
        };
    }

    return await this.prisma.user.create({
        data,
        include: {
            role: true,
        },
    });
}
```



Capítulo 17:

Implementando módulo "users".

Implementando arquivo ***users.service.ts***:

```
● ● ●

async findAll() {
    return await this.prisma.user.findMany({
        include: {
            role: true,
        },
    });
}

async findOne(id: string) {
    return await this.prisma.user.findFirst({
        where: { id },
    });
}
async findEmail(email: string) {
    return await this.prisma.user.findUnique({
        where: { email },
        include: {
            role: {
                include: {
                    permissions: true,
                },
            },
        },
    });
}
```



Capítulo 17:

Implementando módulo "users".

Implementando arquivo ***users.service.ts***:

```
● ● ●

async update(id: string, updateUserDto: UpdateUserDto) {
  const exitUser = await this.prisma.user.findUnique({ where: { id } });

  if (!exitUser) {
    return null;
  }

  const { password, role, ...userData } = updateUserDto;

  const dataToUpdate: any = {
    ...userData,
  };

  if (password) {
    const passwordHash = await this.Crypt.hash(password);
    dataToUpdate.password = passwordHash;
  }

  if (role) {
    dataToUpdate.role = {
      connect: { id: role },
    };
  }

  return await this.prisma.user.update({
    where: { id },
    data: dataToUpdate,
    include: {
      role: true,
    },
  });
}

async remove(id: string) {
  return await this.prisma.user.delete({
    where: { id },
  });
}
```



Capítulo 17:

Implementando módulo "users".

Implementando arquivo **users.controller.ts**:

```
● ● ●

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

  @Post()
  create(@Body() createUserDto: CreateUserDto) {
    return this.usersService.create(createUserDto);
  }

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

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

  @Get('by-email/:email')
  findEmail(@Param('email') email: string) {
    return this.usersService.findEmail(email);
  }

  @Patch(':id')
  update(@Param('id') id: string, @Body() updateUserDto: UpdateUserDto) {
    return this.usersService.update(id, updateUserDto);
  }

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



Capítulo 18:

Implementando módulo "auth".

Implementando arquivo ***auth.dto.ts***:

```
● ● ●

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

export class AuthDto {
  @IsNotEmpty({ message: 'O e-mail é obrigatório.' })
  @IsEmail({}, { message: 'O e-mail deve ser válido.' })
  email!: string;

  @IsNotEmpty({ message: 'A senha é obrigatória.' })
  @IsString({ message: 'A senha deve ser uma string.' })
  @Length(6, 50, { message: 'A senha deve ter entre 6 e 50 caracteres.' })
  @Matches(/^(?=.*[A-Za-z])(?=.*\d)[A-Za-z\d@$!%*?&]{6,}$/, {
    message:
      'A senha deve conter pelo menos uma letra, um número e ter no mínimo 6 caracteres.',
  })
  password!: string;
}
```



Capítulo 18:

Implementando módulo "auth".

Implementando arquivo ***auth.service.ts***:

```
● ● ●  
interface AuthenticatedUser extends User {  
    role: Role & { permissions: Permission[] };  
}
```

Essa interface define que o objeto retornado após autenticação será um **User**, mas também incluirá:

- **O role:** Papel do usuário é dentro do role, a lista de permissions (permissões)

Isso é útil para que o token JWT inclua **nível de acesso**, e possamos proteger rotas com base em permissões e papéis.

```
● ● ●  
export class AuthService {  
    constructor(  
        private readonly userService: UsersService,  
        private jwtService: JwtService,  
        private readonly Crypt: HashingService,  
    ) {}
```

- **userService:** é usado para buscar o usuário pelo e-mail.
- **jwtService:** é usado para gerar o token JWT.
- **Crypt:** serviço customizado para comparar senhas (ex: usando bcrypt).



Capítulo 18:

Implementando módulo "auth".

Implementando arquivo ***auth.service.ts***:

Esse método autentica o usuário com base no e-mail e senha fornecidos:

```
● ● ●

async validateUser(authDto: AuthDto): Promise<AuthenticatedUser | null> {
    const existUser = await this.userService.findEmail(authDto.email);

    if (!existUser) {
        throw new UnauthorizedException('Credenciais inválidas.');
    }

    const isPasswordValid = await this.Crypt.compare(
        authDto.password,
        existUser.password,
    );

    if (isPasswordValid) {
        return existUser;
    }

    throw new UnauthorizedException('Credenciais inválidas.');
}
```

- Busca o usuário no banco usando o ***e-mail***.
- Se o usuário não existir, lança **UnauthorizedException**.
- Compara a senha fornecida com a senha criptografada no banco, usando o serviço de ***hashing***.
- Se a senha for válida, retorna o objeto do usuário (incluindo role e permissions).
- Se a senha estiver incorreta, lança **UnauthorizedException**.



Capítulo 18:

Implementando módulo "auth".

Implementando arquivo **auth.service.ts**:

Esse método **login** é responsável por gerar e retornar um **token JWT** para o usuário autenticado.

Recebe um usuário já autenticado (com dados como id, email, role e permissions).

```
● ● ●

async login(user: AuthenticatedUser) {
  const payload = {
    email: user.email,
    sub: user.id,
    role: user.role ? user.role.name : null,
    permissions:
      user.role && user.role.permissions
        ? user.role.permissions.map((perm) => perm.name)
        : [],
  };
  return { access_token: this.jwtService.sign(payload) };
}
```

- Cria o **payload** que será inserido no **token JWT**:
- **sub**: ID do usuário (subject).
- **email**: e-mail do usuário.
- **role**: nome do papel (se existir).
- **permissions**: lista com os nomes das permissões do usuário.
- Gera o **token JWT** usando this.jwtService.sign(payload).
- Retorna o **token JWT encapsulado** no objeto { access_token }.



Capítulo 18:

Implementando módulo "auth".

Implementando arquivo **auth.controller.ts**:

```
● ● ●  
@Controller('auth')  
export class AuthController {  
    constructor(private readonly authService: AuthService) {}  
  
    @Post('login')  
    async login(@Request() req: any) {  
        return this.authService.login(req.user);  
    }  
}
```



Capítulo 19:

Implementando JSON Web Token(JWT)e Passaport.

JWT é um padrão seguro e compacto para transmitir informações entre partes como um objeto **JSON assinado** digitalmente, usado em autenticação e autorização. É composto por três partes: **header** (informações sobre o algoritmo e tipo), **payload** (dados como ID do usuário ou permissões), e **signature** (assinatura digital para garantir a integridade). Após o login, o servidor gera o token e o envia ao cliente, que o utiliza em requisições subsequentes, enviando-o no cabeçalho HTTP (Authorization: Bearer <token>).

Passport é uma biblioteca de autenticação para Node.js que oferece uma estrutura flexível e modular para implementar diversas estratégias de login, como **JWT, OAuth, Google, Facebook, Local** (usuário/senha), entre outras. Ele funciona com middleware no Express (ou NestJS) e trata a autenticação de forma desacoplada da lógica principal da aplicação.

No terminal do vscode:

```
● ● ●  
npm install --save @nestjs/jwt
```

```
● ● ●  
npm install --save-dev @types/passport @types/passport-local @types/passport-jwt
```



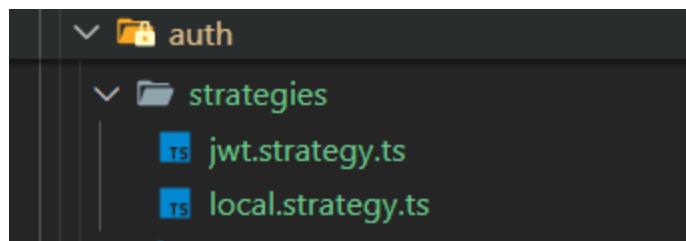
Capítulo 20:

Implementando strategy .

No **Passport**, uma *strategy* (estratégia) define como um usuário será autenticado — por exemplo, com usuário e senha (**LocalStrategy**), token JWT (**JwtStrategy**), ou provedores como Google e Facebook. Cada estratégia é um módulo independente e reutilizável que encapsula a lógica de verificação. No NestJS, você implementa uma classe que estende uma estratégia (como **PassportStrategy(Strategy)**) e define como validar os dados do usuário. Isso torna o sistema de autenticação flexível e modular, permitindo adicionar ou trocar métodos de login facilmente.

Crie uma nova pasta chamada `strategies` dentro do diretório `auth`, e dentro dessa pasta, crie dois arquivos:

- **`Jwt.strategy.ts`:** Para a estratégia de autenticação baseada em token JWT (JwtStrategy)
- **`local.strategy.ts`:** Para a estratégia de autenticação com usuário e senha (LocalStrategy).





Capítulo 21:

Implementando "local.strategy".

No contexto de autenticação com **Passport**, a "**local strategy**" é uma das formas de autenticar um usuário com base em **credenciais tradicionais**, como **e-mail** e **senha**, ao invés de tokens (como JWT, OAuth, etc.).

```
● ● ●  
interface AuthenticatedUser {  
    id: string;  
    email: string;  
    password?: string;  
    name: string;  
    role: {  
        id: string;  
        name: string;  
        permissions: { id: string; name: string }[];  
    };  
}
```

Essa interface descreve o formato de dados do usuário após a autenticação, omitindo a senha por segurança. Serve como um contrato de tipo para o que será retornado ao NestJS após a validação.



Capítulo 21:

Implementando "local.strategy".

Esse super configura o comportamento da estratégia:

- **usernameField: 'email'**: Passport usará o campo email em vez do padrão username.
- **passwordField: 'password'**: Campo que contém a senha.
- **passReqToCallback: false**: Indica que o objeto da requisição (req) não será passado para o método validate.



```
export class LocalStrategy extends PassportStrategy(Strategy) {  
  constructor(private authService: AuthService) {  
    super({  
      usernameField: 'email',  
      passwordField: 'password',  
      passReqToCallback: false,  
    });  
  }  
}
```



Capítulo 21:

Implementando "local.strategy".

O método validate mostrado acima faz parte de uma estratégia de autenticação (geralmente **LocalStrategy**) que autentica o usuário com e-mail e senha. Ele é usado pelo **Passport.js** no fluxo de autenticação local no NestJS. Vamos analisar passo a passo:

```
● ● ●  
async validate(email: string, password: string): Promise<AuthenticatedUser> {  
    const authDto: AuthDto = { email, password };  
    const user = await this.authService.validateUser(authDto);  
  
    if (!user) {  
        throw new UnauthorizedException('Credenciais inválidas');  
    }  
    // eslint-disable-next-line @typescript-eslint/no-unused-vars  
    const { password: _, ...result } = user;  
    return result as AuthenticatedUser;  
}
```

- Recebe e-mail e senha.
- Valida o usuário via **AuthService**.
- Remove a senha do retorno.
- Retorna o usuário autenticado.
- Lança erro se as credenciais forem inválidas.



Capítulo 22:

Implementando "jwt.strategy".

A classe **JwtStrategy** estende a estratégia de autenticação baseada em **JWT (JSON Web Token)** fornecida pelo pacote **passport-jwt**, que é uma estratégia do **Passport.js**, um middleware de autenticação muito utilizado em aplicações Node.js.

```
● ● ●  
interface JwtPayload {  
  email: string;  
  sub: string;  
  role: string | null;  
  permissions: string[];  
}
```

Essa interface define o formato esperado do **payload** dentro do **token JWT**:



Capítulo 22:

Implementando "jwt.strategy".

O método construtor recebe com injeção de dependência:

- **configService**: Utilizado para acessar variáveis de ambiente (como **JWT_SECRET**), garantindo que o segredo do JWT esteja seguro e configurável fora do código

```
● ● ●  
export class JwtStrategy extends PassportStrategy(Strategy) {  
    constructor(  
        private usersService: UsersService,  
        private configService: ConfigService,  
    ) {  
        super({  
            jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(),  
            ignoreExpiration: false,  
            secretOrKey: configService.get<string>('JWT_SECRET'),  
        });  
    }  
}
```

Esse super configura o comportamento da estratégia:

- **jwtFromRequest**: Define de onde o token será extraído. Aqui, ele é extraído do header **Authorization como Bearer token**.
- **ignoreExpiration**: false: Garante que tokens expirados não sejam aceitos.
- **SecretOrKey**: Chave secreta usada para verificar a **assinatura do JWT**.



Capítulo 22:

Implementando "jwt.strategy".

O método ***validate(payload: JwtPayload)*** é chamado automaticamente pelo ***Passport*** após o ***token JWT*** ser verificado com sucesso.

```
● ● ●  
async validate(payload: JwtPayload): Promise<any> {  
    return {  
        id: payload.sub,  
        email: payload.email,  
        name: payload.name,  
        role: payload.role,  
        permissions: payload.permissions,  
    };  
}
```

1. Recebe os dados decodificados do ***token (payload), como sub, email, role e permissions.***
2. Retorna um objeto com as informações do ***usuário autenticado***, que será injetado automaticamente no ***@Request() via req.user.***



Capítulo 23:

Adicionando decorators.

Os **decorators** no NestJS são uma parte fundamental da estrutura do framework. Eles são usados para anotar e configurar classes, métodos, propriedades e parâmetros com metadados que o NestJS utiliza para construir e gerenciar sua aplicação.

Esses decorators são baseados no padrão de decorators do TypeScript e facilitam a injeção de dependências, o roteamento, a validação, entre outros recursos.

Um decorator é uma função especial que pode ser aplicada a uma classe, método, propriedade ou parâmetro para adicionar comportamento ou metadados.

No terminal do vscode:

```
OUTPUT PROBLEMS TERMINAL DEBUG CONSOLE PORTS  
PS E:\ebook-DIO-NESTJS\nest-auth-app> nest g decorator auth/decorators/permissions --flat
```

```
OUTPUT PROBLEMS TERMINAL DEBUG CONSOLE PORTS  
PS E:\ebook-DIO-NESTJS\nest-auth-app> nest g decorator auth/decorators/roles --flat
```



Capítulo 24:

Implementando "permissions.decorator".

Esse trecho de código define um decorator personalizado no NestJS chamado `@RequiredPermissions`. Ele serve para associar metadados a rotas ou métodos, usado junto com guards para aplicar regras de autorização baseadas em permissões.

- **`SetMetadata`**, que é usada para anexar metadados personalizados a um manipulador (método de controller, por exemplo).
- Define uma constante que representa a chave do metadata.
- Define um decorator chamado **`@RequiredPermissions`**.
- Ele aceita uma lista de **`permissões (string[])`**.



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

export const PERMISSIONS_KEY = 'permissions';
export const RequiredPermissions = (...permissions: string[]) =>
  SetMetadata(PERMISSIONS_KEY, permissions);
```



Capítulo 25:

Implementando "roles.decorator".

Esse código cria um **decorador personalizado** chamado **@Roles** no NestJS. Ele é usado para marcar rotas ou métodos com os **papéis (roles)** que têm permissão para acessá-los — por exemplo: 'admin', 'user', 'moderator'.

- Cria o **decorador @Roles()**, que pode ser usado em rotas.
- Ele aceita múltiplos argumentos (string[]) representando os papéis permitidos.
- Os valores são armazenados como metadados com a chave 'roles'.



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

export const ROLES_KEY = 'roles';
export const Roles = (...roles: string[]) => SetMetadata(ROLES_KEY, roles);
```



Capítulo 26:

Adicionando guards.

Guards são mecanismos de segurança que atuam antes da execução dos manipuladores de rota (**controllers**). Eles são usados principalmente para controle de acesso, verificando se a requisição pode ou não continuar com base em certas condições — como autenticação, permissões, roles, etc.

No terminal do vscode:



```
OUTPUT PROBLEMS TERMINAL DEBUG CONSOLE PORTS
PS E:\ebook-DIO-NESTJS\nest-auth-app> nest g guard auth/guards/permissions --no-spec --flat
```



```
OUTPUT PROBLEMS TERMINAL DEBUG CONSOLE PORTS
PS E:\ebook-DIO-NESTJS\nest-auth-app> nest g guard auth/guards/roles --no-spec --flat
```

- **@RequiredPermissions(...)** é um decorator personalizado.
- Usa **SetMetadata** para associar permissões ao handler.
- Requer um **guard (como PermissionsGuard)** para verificar essas permissões.
- Facilita a criação de uma autorização baseada em permissões, clara e reutilizável.



Capítulo 26:

Adicionando guards.

O **Reflector** é uma classe utilitária do NestJS usada para acessar metadados definidos com decorators como `@SetMetadata` (ex: `@Roles`, `@Permissions`).

Ou seja, quando você cria decorators personalizados (como `@Roles('admin')`), o Reflector permite ler esses valores no momento da execução — por exemplo, dentro de um Guard, Interceptor ou Pipe.

O **canActivate** é o método principal de qualquer Guard no NestJS. Ele define a lógica de autorização — ou seja, se o usuário pode ou não acessar a rota.

Quando uma requisição é feita, o NestJS chama o método `canActivate` de todos os guards associados àquela rota. Se todos retornarem `true`, o acesso é permitido. Se qualquer um retornar `false` ou lançar uma exceção, o acesso é negado.

ExecutionContext representa o contexto de execução da requisição atual, e permite acessar:

- A requisição `HTTP(context.switchToHttp().getRequest())`.
- O método chamado (`handler`)
- A classe controller.
- O tipo de execução (HTTP, WebSocket, RPC, etc.).



Capítulo 27:

Implementando "permissions.guard".

Implementando arquivo permissions.guard.ts:

```
● ● ●

export class PermissionsGuard implements CanActivate {
  constructor(private reflector: Reflector) {}

  canActivate(context: ExecutionContext): boolean {
    const requiredPermissions = this.reflector.getAllAndOverride<string[]>(
      PERMISSIONS_KEY,
      [context.getHandler(), context.getClass()],
    );

    if (!requiredPermissions) {
      return true;
    }

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

    if (!user || !user.permissions || !Array.isArray(user.permissions)) {
      return false;
    }

    return requiredPermissions.every((reqPerm) =>
      user.permissions.some(
        (userPerm) => userPerm.toLowerCase() === reqPerm.toLowerCase(),
      ),
    );
  }
}
```



Capítulo 27:

Implementando "permissions.guard".

Descrição do arquivo permissions.guard.ts:

O **Reflector** é injetado no construtor para acessar metadados (ou seja, as permissões definidas por decorators como `@RequiredPermissions()`).

O método **canActivate** será chamado automaticamente pelo NestJS sempre que essa rota for acessada. O retorno (true ou false) indica se o usuário pode continuar.

Método do **Reflector** que procura por metadados e pega o valor mais específico encontrado. A tipagem `<string[]>` diz que estamos esperando um array de permissões.

PERMISSIONS_KEY É a chave do metadado que estamos procurando, este nome foi definido no decorator.

É um array com dois elementos:

- **context.getHandler()**: retorna o **método da rota** que está sendo acessado.
- **context.getClass()**: retorna a **classe controller** onde o método está.



Capítulo 27:

Implementando "permissions.guard".

Descrição do arquivo permissions.guard.ts:

Se nenhuma permissão foi definida com o decorador **@RequiredPermissions(...)**, então, a rota não exige permissão específica. Portanto, o acesso é liberado (return true). Convertendo o ExecutionContext para contexto HTTP, e acessamos request.user. Esse user deve ter sido adicionado anteriormente, normalmente por um JWT.

- **every()** → todos os reqPerm (permessões exigidas) devem estar no usuário.
- **some()** → procura se pelo menos uma permissão do usuário corresponde à permissão exigida.
- **toLowerCase()** → ignora diferenças entre maiúsculas e minúsculas.



Capítulo 28:

Implementando "roles.guard".

Implementando arquivo roles.guard.ts:



```
export class RolesGuard implements CanActivate {
  constructor(private reflector: Reflector) {}

  canActivate(context: ExecutionContext): boolean {
    const requiredRoles = this.reflector.getAllAndOverride<string[]>(
      ROLES_KEY,
      [context.getHandler(), context.getClass()],
    );

    if (!requiredRoles) {
      return true;
    }

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

    if (!user || !user.role) {
      return false;
    }

    return requiredRoles.some(
      (role) => user.role.toLowerCase() === role.toLowerCase(),
    );
  }
}
```



Capítulo 28:

Implementando "roles.guard".

Descrição do arquivo permissions.guard.ts:

Método do **Reflector** que procura por metadados e pega o valor mais específico encontrado. A tipagem `<string[]>` diz que estamos esperando um array de permissões.

ROLES_KEY É a chave do metadado que estamos procurando, este nome foi definido no decorator.

É um array com dois elementos:

- **context.getHandler()** → retorna o método da rota que está sendo acessado.
- **context.getClass()** → retorna a classe controller onde o método está.



Capítulo 28:

Implementando "roles.guard".

Descrição do arquivo permissions.guard.ts:

Se nenhuma permissão foi definida com o decorador **@RequiredPermissions(...)**, então, a rota não exige permissão específica. Portanto, o acesso é liberado (return true).

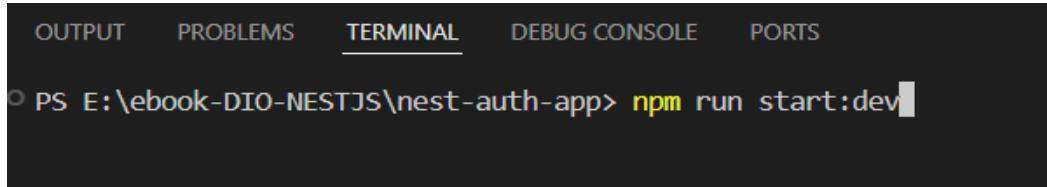
Convertendo o ExecutionContext para contexto HTTP, e acessamos request.user. Esse user deve ter sido adicionado anteriormente, normalmente por um JWT.

- **some()** → procura se pelo menos uma permissão do usuário corresponde à permissão exigida.
- **toLowerCase()** → ignora diferenças entre maiúsculas e minúsculas.

Capítulo 29:

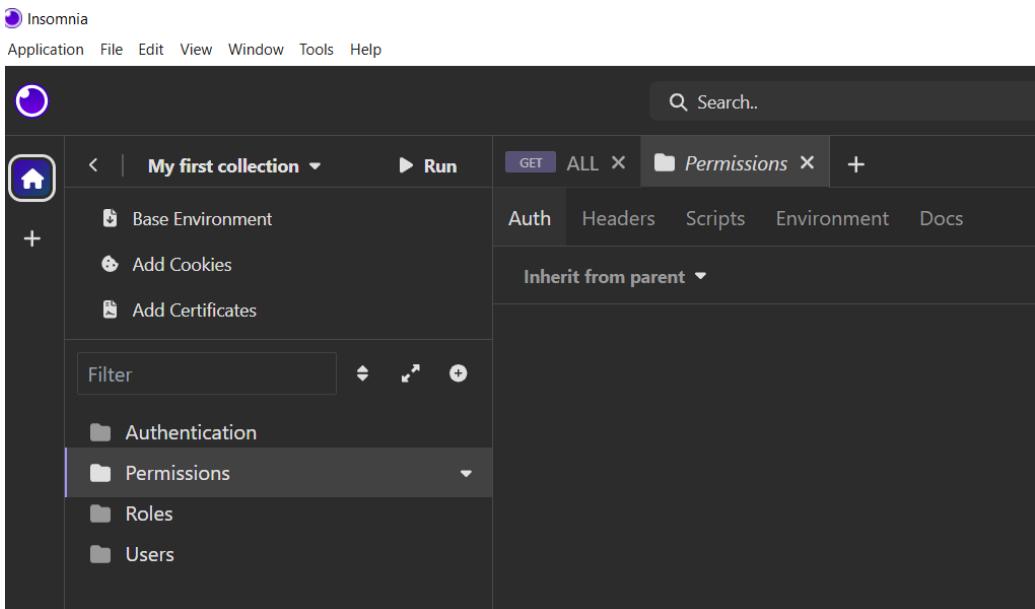
Teste no Insomnia.

No terminal do vscode:



```
OUTPUT PROBLEMS TERMINAL DEBUG CONSOLE PORTS
○ PS E:\ebook-DIO-NESTJS\nest-auth-app> npm run start:dev
```

Abra o Insomnia Crie quatro pasta:



The screenshot shows the Insomnia application interface. At the top, there's a navigation bar with icons for Home, Collection, Run, and Help. Below the navigation bar, the main area displays a collection titled "My first collection". The collection has a status bar indicating "GET ALL" and a folder icon labeled "Permissions". The "Permissions" folder is currently selected. The interface includes a search bar at the top right and a sidebar on the left containing a "Filter" input field and a list of sub-pastes: "Base Environment", "Add Cookies", "Add Certificates", "Authentication", "Permissions" (which is highlighted with a purple border), "Roles", and "Users".



Capítulo 29:

Teste no Insomnia "permissions".

Dentro da pasta Permissions crie os seguintes métodos:

The screenshot shows the Insomnia REST Client interface. At the top, there's a sidebar with a folder icon labeled 'Permissions' containing five items: 'CREATE' (POST), 'UPDATE' (PATCH), 'ALL' (GET), 'FIND' (GET), and 'DELETE' (DEL). Below this, the main window has a title bar 'Permissions' with a 'POST' button and a 'CREATE' tab selected. The URL field shows 'http://localhost:3000/permissions'. On the right, there's a 'Send' button. The bottom section is divided into tabs: 'Params', 'Body' (which is active and highlighted in green), 'Auth', 'Headers' (with a count of 4), 'Scripts', and 'Docs'. Under the 'Body' tab, the 'JSON' dropdown is selected, and the body content is displayed as:

```
1 {  
2   "name": "CRIAR"  
3 }
```

- <http://localhost:3000/>: endereço onde foi startado o serviço.
- permissions: endereço criado no arquivo de controller.

```
● ● ●  
@Controller('permissions')  
export class PermissionsController {
```

- O **JSON (JavaScript Object Notation)** simples, que contém uma única chave chamada name com o valor "CRIAR".
- Send: Envia a requisição para o banco.

Crie uma permissão para "LER", "ALTERAR" e "DELETAR"



Capítulo 29:

Teste no Insomnia "permissions".

Selecione o métodos update:

The screenshot shows the Insomnia REST client interface. At the top, there are tabs for Permissions (selected), POST, CREATE, GET, ALL, PATCH (selected), UPDATE, and a plus sign. Below the tabs, it says PATCH ▾ http://localhost:3000/permissions/a9c4f2ad-b938-4b26-9654-31407add84f9. To the right is a purple Send button with a dropdown arrow. Below the URL, there are tabs for Params, Body (selected, indicated by a green dot), Auth, Headers (with a count of 4), Scripts, and Docs. Under the Body tab, it says JSON ▾. The JSON content is:
1 ▾ {
2 "name": "LER"
3 }

- http://localhost:3000/: endereço onde foi "startado" o serviço.
- permissions: endereço criado no arquivo de controller.
- /Id do dados cadastrado no banco que vai ser alterado.

O método PATCH é um dos métodos HTTP utilizados para atualizar parcialmente um recurso existente em uma API.

- O JSON (JavaScript Object Notation) simples, que contém chave a ser alterada valor.
- Send: Envia a requisição para o banco.



Capítulo 29:

Teste no Insomnia "permissions".

Selecione o métodos All, Find e Delete:

The screenshot shows the Insomnia REST client interface. At the top, there is a navigation bar with tabs: 'Permissions X', 'POST CREATE X', 'GET ALL X', 'PTCH UPDATE X', and a '+' button. Below the navigation bar, the URL 'http://localhost:3000/permissions/' is displayed, along with a 'Send' button. Underneath the URL, there are tabs for 'Params', 'Body', 'Auth', 'Headers (3)', 'Scripts', and 'Docs'. A dropdown menu labeled 'No Body' is visible.

The screenshot shows the 'Permissions' collection with the 'GET FIND X' tab selected. The URL 'http://localhost:3000/permissions/a7038682-71ba-488d-81ed-1b5e54e83e10' is shown, along with a 'Send' button. Below the URL, there are tabs for 'Params', 'Body', 'Auth', 'Headers (3)', 'Scripts', and 'Docs'.

The screenshot shows the 'Permissions' collection with the 'DEL DELETE X' tab selected. The URL 'http://localhost:3000/permissions/7a449990-834d-4c02-99ca-0d6a875e9ee0' is shown, along with a 'Send' button. Below the URL, there are tabs for 'Params', 'Body', 'Auth', 'Headers (3)', 'Scripts', and 'Docs'.

- <http://localhost:3000/>: endereço onde foi startado o serviço.
- permissions: endereço criado no arquivo de controller.

No método Find e Delete

- /Id do dados cadastrado no banco, que vai ser buscado ou deletado.

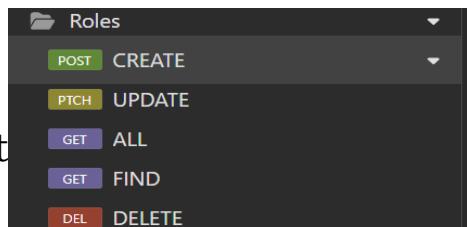


Capítulo 30:

Teste no Insomnia "roles".

Dentro da pasta Roles crie os seguintes métodos:

Selecione o mét



POST ▾ http://localhost:3000/roles	Send ▾	201 Created	1.99 s	322 B		
Params	Body	Auth	Headers	4		
JSON	Preview	Headers	7	Cookies		
1 ↴ { 2 "name": "ADMIN", 3 "permissions": [4 "5338f664-1361-4088-8c95-2a9712a5f6b4", 5 "01b711e5-dd14-4bc7-ac01-a14c724e6d47", 6 "d9150f3c-63a3-4979-8391-aab6feb931d5", 7 "a9c4f2ad-b938-4b26-9654-31407add84f9" 8] 9 }	1 ↴ { 2 "id": "b3e3deaf-da0c-4486-b1b6-0306aecacf752", 3 "name": "ADMIN", 4 "permissions": [5 { 6 "id": "5338f664-1361-4088-8c95-2a9712a5f6b4", 7 "name": "CRIAR" 8 }, 9 { 10 "id": "01b711e5-dd14-4bc7-ac01-a14c724e6d47", 11 "name": "ALTERAR" 12 }, 13 { 14 "id": "d9150f3c-63a3-4979-8391-aab6feb931d5", 15 "name": "DELETAR" 16 }, 17 { 18 "id": "a9c4f2ad-b938-4b26-9654-31407add84f9", 19 "name": "LER" 20 } 21] 22 }	Scripts	Do	Tests 0 / 0	→ Mock	Console

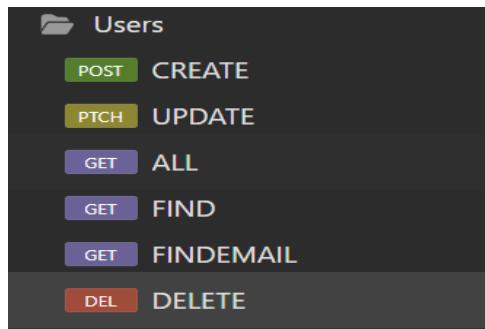
permissions. Recebe um array com las da permissões criadas anteriormente. Associando permissões ao papel usuário. Crie um ROLE para USUARIO só com permissions de "LER" e um ROLE para GESTOR com permissions de "LER" e "ALTERAR".



Capítulo 31:

Teste no Insomnia "users".

Dentro da pasta Roles crie os seguintes métodos:



Selecione o métodos create e crie os seguintes usuários com as ROLES:

- Admin
- Gestor
- Usuario

O atributo "role" é onde vais setar o id da ROLE.

Params	Body	Auth	Headers	Scripts	Docs	Preview	Headers	Cookies	Tests	→ Mock	Console	
	<pre>1 ▶ { 2 "name": "Mordock", 3 "email": "mordock.comm@example.com", 4 "password": "SenhaSegura123!", 5 "role": "b3e3deaf-da0c-4486-b1b6-0306aecaf752" 6 }</pre>					<pre>1 ▶ { 2 "id": "917a5698-3e94-4b3a-80fc-36c128f85d34", 3 "name": "Mordock", 4 "email": "mordock.comm@example.com", 5 "password": 6 "\$2b\$12\$vF6E.TRT71Y0e4.GZg.V00hWpyDjKxnbGbzjNlxKQFXoHtMP202ri", 7 "roleId": "b3e3deaf-da0c-4486-b1b6-0306aecaf752", 8 "role": { 9 "id": "b3e3deaf-da0c-4486-b1b6-0306aecaf752", 10 "name": "ADMIN" 11 }</pre>						



Capítulo 32:

Configurando na API papéis e permissões.

No arquivo auth.module.ts:

```
● ● ●

@Module({
  imports: [
    UsersModule,
    PassportModule,
    ConfigModule.forRoot({
      isGlobal: true, // Opcional, mas útil para acessar em qualquer lugar
      envFilePath: '.env', // Garante que ele lê do .env
    }),
    JwtModule.registerAsync({
      imports: [ConfigModule],
      inject: [ConfigService],
      useFactory: async (configService: ConfigService) => ({
        secret: configService.get<string>('JWT_SECRET'), // <-- Lendo a chave aqui
        signOptions: { expiresIn: '1h' },
      }),
    }),
  ],
  providers: [
    AuthService,
    UsersService,
    {
      provide: HashingService,
      useClass: BcryptService,
    },
    LocalStrategy,
    JwtStrategy,
  ],
  controllers: [AuthController],
  exports: [AuthService, JwtModule],
})
export class AuthModule {}
```



Capítulo 32:

Configurando na API papéis e permissões.

Descrição arquivo auth.module.ts:

Este módulo é responsável por gerenciar a autenticação de usuários na sua aplicação NestJS usando:

- JWT (JSON Web Token)
- Estratégias do Passport
- Senhas com hash (bcrypt)
- Variáveis de ambiente (.env)
- **UsersModule:** importa a lógica de gerenciamento de usuários (ex: buscar usuários por e-mail, salvar, etc.). Ele será usado principalmente pelo AuthService.
- **PassportModule:** integra o Passport.js ao NestJS, permitindo o uso de estratégias como LocalStrategy (login com e-mail/senha) e JwtStrategy (validação do token JWT).
- **ConfigModule.forRoot({ ... }):** permite acessar variáveis definidas no arquivo .env (como JWT_SECRET).
- **isGlobal:** true faz com que esse módulo esteja disponível em todos os outros módulos sem precisar importar manualmente.
- **JwtModule.registerAsync({ ... }):** registra o módulo JWT de forma **assíncrona**, pegando a JWT_SECRET diretamente do ConfigService.



Capítulo 32:

Configurando na API papéis e permissões.

Descrição arquivo auth.module.ts:

Providers:

- **AuthService**: responsável por toda a lógica de autenticação, como login, geração e validação de tokens JWT.
- **UserService**: utilizado para buscar os dados do usuário no banco, geralmente com base no e-mail ou ID. Importado do UsersModule.
- **HashingService**: é um token de injeção (interface) que abstrai a lógica de hashing de senhas. Permite trocar a implementação se necessário.
- **BcryptService**: é a implementação concreta do HashingService usando a biblioteca bcrypt. Usado para criar e comparar hashes de senha.
- **LocalStrategy**: define como o sistema deve autenticar um usuário usando e-mail e senha. Usa a estratégia passport-local.
- **JwtStrategy**: valida os tokens JWT recebidos nas requisições (geralmente no cabeçalho Authorization) para proteger rotas autenticadas



Capítulo 33:

Teste no Insomnia "Login".

Dentro da pasta Authentication crie os seguinte método:

The screenshot shows the Insomnia REST Client interface. On the left, there's a tree view with a folder named 'Authentication' containing a single item labeled 'POST Login'. Below this, the main area shows a POST request to 'http://localhost:3000/auth/login'. The 'Body' tab is selected, displaying a JSON payload:

```
1 {  
2   "email": "mordock.com@example.com",  
3   "password": "SenhaSegura123!"  
4 }
```

On the right, the response details are shown: '201 Created', '408 ms', '340 B', and '27 Minutes Ago'. The 'Preview' tab shows the JSON response:

```
1 {  
2   "access_token":  
3     "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJlbWFpbC16ImIvcmlvcmRvY2suY29tbUB1eGFtcGxI  
4     LmNvbSIsInIyIiGiJkx2EINjk4LNU10TQNTG1zY5b4MGZjLTMyZEoGY4NQNCiInJvGU01  
5     JBER1JTiiIisInBlcmtpc3Npb25zIjpbtKNSSUFStiwiQUxURVJBUiTsIKRFTEVUQVIIICJMRVILXSwI  
6     aWFO1joxNxLzWjAxODYyLCJleHAiOiJE3NTAyMDU0NjJ9.XC_ms0HzrB1vhQer65RaV6BLKZs0L4C  
7     5GMcz9oxyU"  
8 }
```

O serviço de autenticação (AuthService) recebe dois parâmetros:

- E-mail do usuário
- Senha informada no login

Ele utiliza esses dados para:

- Buscar o usuário no banco de dados usando o e-mail.
- Verificar se a senha está correta, comparando com o hash salvo no banco (usando BcryptService).
- Se tudo estiver certo, ele gera e retorna um token JWT, que contém as informações do usuário e tem tempo de expiração (ex: 1h).

Esse token pode então ser usado nas requisições futuras como prova de que o usuário está autenticado.



Capítulo 34:

TOKEN.

Em termos gerais, um token é um objeto digital que serve como representação de um ativo, identidade ou direito de acesso. No contexto da segurança, um token geralmente contém informações sobre a identidade do usuário e suas permissões, sendo usado para autenticação e autorização em sistemas digitais. Além disso, o termo "token" também pode se referir a dispositivos físicos usados para gerar códigos de segurança, como tokens de autenticação usados em bancos. site: <https://jwt.io/>

The screenshot shows the homepage of jwt.io. At the top, there's a navigation bar with links for 'Debugger', 'Introduction', 'Libraries', and 'Ask'. Below the navigation, the main title is 'JSON Web Token (JWT) Debugger'. There are two main sections: 'JWT Decoder' and 'JWT Encoder'. The 'JWT Decoder' section contains a text input field for pasting a JWT token. Below the input field, it says 'Encoded Value' and shows the token 'eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6Ikpvag4gRG9lIiwiYnRtaW4iOnRydWUsImhdC1GHTUxNjizOTAyMn0.KMUFsIDTnPmyG3nIiGM6H9FNFUROf3wh7SmqJp-QV30'. It also includes sections for 'Decoded Header' (JSON and Claims Table) and 'Decoded Payload' (JSON and Claims Table). The 'Decoded Header' shows the following JSON:

```
{  
  "alg": "HS256",  
  "typ": "JWT"  
}
```

The 'Decoded Payload' shows the following JSON:

```
{  
  "sub": "1234567890",  
  "name": "John Doe",  
  "admin": true,  
  "iat": 1516239022  
}
```

Capítulo 34:

TOKEN.

Aqui insira o token gerado no login do insomnia para obter se é um JWT válido.

 **JWT**
Debugger

JWT Decoder **JWT Encoder**

Paste a JWT below that you'd like to decode, validate, and verify.

ENCODED VALUE

COPY CLEAR

Valid JWT

Invalid Signature



Capítulo 34:

TOKEN.

TOKEN JWT decodificado:

DECODED HEADER

JSON	CLAIMS TABLE
{	
"alg": "HS256",	
"typ": "JWT"	
}	

DECODED PAYLOAD

JSON	CLAIMS TABLE
{	
"email": "mordock.comm@example.com",	
"sub": "917a5698-3e94-4b3a-80fc-36c128f85d34",	
"role": "ADMIN",	
"permissions": [
"CRIAR",	
"ALTERAR",	
"DELETAR",	
"LER"	
],	
"iat": 1750201862,	
"exp": 1750205462	
}	

JWT SIGNATURE VERIFICATION (OPTIONAL)

Enter the secret used to sign the JWT below:

SECRET

signature verification failed

a-string-secret-at-least-256-bits-long



Capítulo 34:

Aplicando papéis e permissões nas rotas.

Implementando arquivo roles.controller.ts:

```
● ● ●

@UseGuards(AuthGuard('jwt'), RolesGuard, PermissionsGuard)
@Controller('roles')
export class RolesController {
    constructor(private readonly rolesService: RolesService) {}

    @Post()
    @Roles('ADMIN')
    @RequiredPermissions('CRIAR')
    create(@Body() createRoleDto: CreateRoleDto) {
        return this.rolesService.create(createRoleDto);
    }

    @Get()
    @Roles('ADMIN', 'GESTOR')
    @RequiredPermissions('LER')
    findAll() {
        return this.rolesService.findAll();
    }

    @Get(':id')
    @Roles('ADMIN', 'GESTOR')
    @RequiredPermissions('LER')
    findOne(@Param('id') id: string) {
        return this.rolesService.findOne(id);
    }

    @Patch(':id')
    @Roles('ADMIN', 'GESTOR')
    @RequiredPermissions('ALTERAR')
    update(@Param('id') id: string, @Body() updateRoleDto: UpdateRoleDto) {
        return this.rolesService.update(id, updateRoleDto);
    }

    @Delete(':id')
    @Roles('ADMIN')
    @RequiredPermissions('DELETAR')
    remove(@Param('id') id: string) {
        return this.rolesService.remove(id);
    }
}
```



Capítulo 34:

Aplicando papéis e permissões nas rotas.

Implementando arquivo roles.controller.ts:

1. **@UseGuards(AuthGuard('jwt'))**, RolesGuard, PermissionsGuard).

Aplica três guards de segurança na rota:

- **AuthGuard('jwt')**: Garante que o usuário está autenticado com um token JWT válido.
- **RolesGuard**: Verifica se o usuário tem o papel (role) ADMIN, definido pelo decorador @Roles('ADMIN').
- **PermissionsGuard**: Verifica se o usuário tem a permissão específica chamada CRIAR, definida pelo decorador @RequiredPermissions('CRIAR').

2. **@Roles('ADMIN')**

Esse decorador (criado com SetMetadata) indica que somente usuários com o papel ADMIN podem acessar essa rota.

3. **@RequiredPermissions('CRIAR')**

Esse decorador define que o usuário precisa possuir a permissão chamada "CRIAR" (ex: cadastrar cargos, usuários, etc.).

O valor "CRIAR" geralmente está ligado a um sistema de permissões armazenado no banco, onde o usuário tem um array de permissões, como:

Agora podes aplicar para as outras controllers.



Capítulo 35:

Teste no Insomnia "roles".
Papeis e permissões nas rotas.

Faço login como Usuario:

```
▼ {  
  "email": "usuario@example.com",  
  "password": "SenhaSegura123!"  
}
```

Agora vá na pasta Roles método ALL, quando clicar em SEND, a resposta vai ser não autorizado.

The screenshot shows the Insomnia interface with a 401 Unauthorized status code. The response body is a JSON object with a message of "Unauthorized" and a statusCode of 401.

```
401 Unauthorized 23 ms 43 B  
Preview Headers 7 Cookies Tests 0 / 0 → Mock Console  
review ▾  
1 ▾ {  
2   "message": "Unauthorized",  
3   "statusCode": 401  
4 }
```

Copie o token do login, aba Auth, clique em Inherit from parent e selecione Bearer Token, cole o token e clique em SEND.

The screenshot shows the Auth tab in the Insomnia interface. The Bearer Token section is selected, with the Enabled checkbox checked and a token value entered. The Headers tab shows three entries: Authorization, Content-Type, and Accept.

```
GET ▾ http://localhost:3000/roles Send ▾  
Params Body Auth [●] Headers [3] Scripts  
Bearer Token ▾  
ENABLED   
TOKEN .....  
PREFIX
```



Capítulo 35:

Teste no Insomnia "roles".
Papeis e permissões nas rotas.

A resposta vai mudar:

The screenshot shows the Insomnia REST client interface. At the top, there's an orange header bar with the status code **403 Forbidden**, a time of **155 ms**, and a size of **69 B**. Below the header are tabs for **Preview**, **Headers** (with a count of 7), **Cookies**, **Tests 0 / 0**, **→ Mock**, and **Console**. The **Preview** tab is selected and displays the following JSON response:

```
1 ▾ {  
2   "message": "Forbidden resource",  
3   "error": "Forbidden",  
4   "statusCode": 403  
5 }
```

O erro 403 Forbidden, em termos simples, significa que o servidor recusou a sua solicitação de acesso a um recurso, mesmo que a solicitação tenha sido recebida e compreendida.

Faça o teste com login de ADMIN.



Capítulo 36:

Conclusão.

O uso de decorators como `@UseGuards`, `@Roles` e `@RequiredPermissions` no NestJS oferece um poderoso sistema de segurança baseado em autenticação e autorização. Ao aplicar `@UseGuards(AuthGuard('jwt'), RolesGuard, PermissionsGuard)` no nível do controller, você garante que **todas as rotas** estejam protegidas, exigindo:

- Um **token JWT válido** para autenticação;
- Um **papel específico** (como ADMIN) para controle de acesso;
- E **permissões granulares** (como CRIAR, EDITAR, etc.) para ações específicas.

Essa abordagem permite construir APIs seguras e flexíveis, com controle detalhado sobre **quem pode acessar o quê** dentro da aplicação. Ideal para sistemas corporativos, administrativos ou qualquer ambiente onde permissões detalhadas sejam essenciais.