Excelente escolha! Essas três tecnologias estão muito bem integradas entre si. Aqui está uma orientação prática para você aprender na ordem mais eficiente e lógica:

**🚀 Ordem recomendada para aprender:**

1. **Node.js**
   * **Por quê?** É a base das outras duas. Tudo no Nest.js e TypeORM roda sobre Node.js.
   * **O que aprender em Node.js:**
     + O que é o Node.js e como ele funciona (event loop, single thread, etc).
     + Como criar um servidor simples com http.
     + Como usar fs, path, events, process, etc.
     + Uso do npm, criação de pacotes e scripts.
     + Como lidar com módulos ES e CommonJS.
     + Uso básico de Express.js (opcional, mas ajuda a entender o que o Nest abstrai).
     + Manipulação de requisições/respostas HTTP.

**Duração estimada**: 1 a 2 semanas com projetos pequenos.

1. **Nest.js**
   * **Por quê?** É um framework para Node.js que facilita a criação de APIs robustas com estrutura organizada.
   * **O que aprender em Nest.js:**
     + Conceito de módulos, controladores e serviços.
     + Injeção de dependência.
     + Middlewares e pipes.
     + Interceptadores e guards (para autenticação/autorização).
     + Trabalhar com rotas e validação com class-validator.
     + Testes básicos com Jest.
     + Como integrar com banco de dados (aqui entra o TypeORM).

**Duração estimada**: 2 a 3 semanas com um projeto prático (ex: API de tarefas, loja virtual).

1. **TypeORM**
   * **Por quê?** Ele é o ORM (Object Relational Mapper) que você usará dentro do Nest.js para lidar com banco de dados.
   * **O que aprender em TypeORM:**
     + Criação de entidades e relações (OneToMany, ManyToOne, etc.).
     + Uso de repositórios e QueryBuilder.
     + Migrations.
     + Relacionamento entre tabelas.
     + Trabalhar com PostgreSQL ou MySQL.
     + Integração com Nest.js (como importar o módulo do TypeORM).

**Duração estimada**: 1 semana integrada ao seu projeto Nest.js.

**📚 Exemplo de plano de estudos:**

| **Semana** | **Conteúdo** |
| --- | --- |
| 1 | Node.js básico: criar API com http e express |
| 2 | Nest.js básico: módulos, controllers, services |
| 3 | Nest.js avançado: guards, pipes, validação |
| 4 | TypeORM: entidades, migrations, integração com Nest.js |
| 5 | Projeto completo: API com autenticação, banco de dados e testes |

**🧠 Dica:**

Faça **um projeto real e simples** com cada tecnologia. Por exemplo, um sistema de tarefas (todo list) com login e banco de dados.

Se quiser, posso criar um passo a passo prático com os comandos, instalação e arquivos iniciais. Deseja começar pelo Node.js com um exemplo básico?

poderia primeiramente criar um plano de estudo de node.js

Claro! Aqui está um **plano de estudo prático e progressivo de Node.js**, pensado para quem quer aprender bem a base antes de avançar para frameworks como Nest.js e ferramentas como TypeORM:

**🗂️ Plano de Estudo de Node.js (2 semanas)**

**🎯 Objetivo: Entender como o Node.js funciona, aprender a construir APIs e preparar o terreno para o Nest.js.**

**🗓️ Semana 1: Fundamentos do Node.js**

**📘 Dia 1 – Introdução e ambiente**

* O que é o Node.js, para que serve.
* Instalação do Node.js e uso do terminal.
* Testar node e npm no terminal.
* Criar seu primeiro script: hello.js.

**Exercício:** Criar um script que imprime nome, data atual e um número aleatório.

**📘 Dia 2 – Módulos nativos**

* fs: leitura e escrita de arquivos.
* path: manipular caminhos de arquivos.
* os, process.

**Exercício:** Criar um script que lê um arquivo .txt, adiciona uma linha e salva de novo.

**📘 Dia 3 – Módulos e exportações**

* Criar e importar módulos.
* require, module.exports.
* Noções de escopo e organização.

**Exercício:** Criar um módulo de calculadora com 4 funções básicas.

**📘 Dia 4 – Event Loop e eventos**

* Conceito de assíncrono e callback.
* setTimeout, setInterval, Promise, async/await.
* Módulo events.

**Exercício:** Criar um sistema de alarme que dispara eventos com atraso.

**📘 Dia 5 – Criando servidor HTTP**

* Criar servidor com http.
* Entender requisição/resposta.
* Rotas básicas.

**Exercício:** Criar um servidor com 3 rotas: /, /about, /contact.

**📘 Dia 6 e 7 – Projeto: API de tarefas sem Express**

* Criar CRUD de tarefas usando apenas http.
* Usar fs para salvar as tarefas em um .json.

**🗓️ Semana 2: Node.js com Express e ferramentas modernas**

**📘 Dia 8 – Introdução ao Express**

* O que é Express.js.
* Instalar com npm init + npm install express.
* Criar API simples com rotas GET e POST.

**Exercício:** Recriar a API de tarefas usando Express.

**📘 Dia 9 – Middlewares e JSON**

* Como Express processa requisições.
* app.use(), app.json().
* Middleware customizado.

**Exercício:** Criar middleware que registra hora da requisição.

**📘 Dia 10 – CRUD completo com Express**

* Rotas: GET, POST, PUT, DELETE.
* Como lidar com ID de tarefas (ex: com UUID).
* Armazenamento temporário em array ou arquivo .json.

**Exercício:** Finalizar CRUD com update e delete.

**📘 Dia 11 – Trabalhando com Postman**

* Testar rotas com Postman.
* Enviar corpo JSON.
* Ver respostas.

**📘 Dia 12 – Trabalhando com dados simulados**

* Adicionar biblioteca uuid.
* Separar rotas e lógica em arquivos diferentes (modularização).

**📘 Dia 13 – Projeto final com Express**

* API RESTful de tarefas com todas as rotas.
* Salvar dados em arquivo .json ou usar um array temporário.

**📘 Dia 14 – Extra: Ambiente e ferramentas**

* Debug com nodemon.
* Variáveis de ambiente com dotenv.
* Preparar projeto para futuro uso com banco de dados.

**🧩 Recursos úteis:**

* [Node.js Docs](https://nodejs.org/en/docs)
* [Express.js Docs](https://expressjs.com/)
* Curso gratuito: [Node.js no YouTube - Curso em Vídeo](https://www.youtube.com/watch?v=DiXbJL3iWVs)

**✅ Ao final do plano, você terá aprendido:**

* Como funciona o Node.js.
* Como manipular arquivos e eventos.
* Como criar e estruturar uma API com Express.
* Como fazer requisições e lidar com dados.

Se quiser, posso já criar a **estrutura inicial do projeto** e o **código do primeiro dia**. Deseja começar com isso agora?

Agora quero que crie a primeira semana do plano de estudo do Nest.js

Perfeito! Depois de dominar os fundamentos do Node.js, a transição para o **Nest.js** será bem mais tranquila. Abaixo está a **primeira semana do plano de estudo de Nest.js**, focando nos fundamentos do framework e na construção da estrutura principal de um projeto real.

**🗂️ Plano de Estudo – Nest.js (Semana 1)**

🎯 **Objetivo:** Entender a arquitetura do Nest.js, saber criar módulos, controladores e serviços, e iniciar uma API RESTful organizada.

**🛠️ Pré-requisitos antes de começar:**

* Node.js instalado
* Conhecimento básico de TypeScript
* Comandos básicos de terminal
* Editor (VS Code)

**📦 Instalação do Nest.js CLI**

npm i -g @nestjs/cli

nest new projeto-tarefas

**🗓️ Dia 1 – Introdução ao Nest.js**

**📚 Conteúdo:**

* O que é Nest.js e por que usá-lo?
* Arquitetura baseada em módulos e injeção de dependência
* Como funciona o padrão MVC no Nest
* Primeira execução de um projeto

**🔧 Atividades:**

* Instalar o Nest CLI e criar projeto com nest new
* Explorar os arquivos gerados (main.ts, app.module.ts, etc.)
* Rodar o servidor: npm run start:dev

**🧪 Exercício:**

* Adicionar um console.log() no main.ts para verificar o ambiente.

**🗓️ Dia 2 – Controladores (Controllers)**

**📚 Conteúdo:**

* O que são controladores
* Decoradores como @Controller() e @Get()
* Rotas e parâmetros de rota

**🔧 Atividades:**

* Criar um controlador tarefas.controller.ts:

nest g controller tarefas

* Criar métodos GET /tarefas, GET /tarefas/:id

**🧪 Exercício:**

* Adicionar mais uma rota: GET /tarefas/status/:status

**🗓️ Dia 3 – Serviços (Services)**

**📚 Conteúdo:**

* O que são serviços
* @Injectable(): como funciona a injeção de dependência
* Separando a lógica de negócio

**🔧 Atividades:**

* Criar serviço tarefas.service.ts:

nest g service tarefas

* Mover lógica de listagem de tarefas para o serviço

**🧪 Exercício:**

* Criar um método getTarefaPorId(id: string) no serviço e usá-lo no controller

**🗓️ Dia 4 – DTOs e Tipagem**

**📚 Conteúdo:**

* Criando interfaces ou classes para tipos
* O que é DTO (Data Transfer Object)
* Validação básica de dados com TypeScript

**🔧 Atividades:**

* Criar DTO create-tarefa.dto.ts com titulo e descricao
* Criar tipo Tarefa com status enum

**🧪 Exercício:**

* Adicionar lógica para POST /tarefas com DTO

**🗓️ Dia 5 – Injeção de dependência e boas práticas**

**📚 Conteúdo:**

* Injeção de serviços com o constructor()
* Boa prática: separar DTOs, entidades, módulos
* Criar status enum para tarefas: ABERTA, EM\_ANDAMENTO, FINALIZADA

**🔧 Atividades:**

* Atualizar o método createTarefa() para usar status padrão ABERTA

**🧪 Exercício:**

* Criar um método filtrarTarefasPorStatus(status: string)

**🗓️ Dia 6 – Módulos**

**📚 Conteúdo:**

* Entendendo o que são módulos
* Dividindo responsabilidades por domínio
* Reutilização de módulos em aplicações grandes

**🔧 Atividades:**

* Criar novo módulo para tarefas (caso não tenha sido criado junto com controller):

nest g module tarefas

* Garantir que o módulo está importando controller e serviço corretamente

**🗓️ Dia 7 – Revisão e prática**

**📚 Conteúdo:**

* Revisar: controller → chama → service → retorna resposta
* Criar mini projeto de API de tarefas (em memória, sem banco de dados)

**🧪 Exercício final da semana:**

Criar uma API de tarefas com os seguintes endpoints:

* GET /tarefas: listar todas
* GET /tarefas/:id: listar por ID
* POST /tarefas: criar tarefa
* DELETE /tarefas/:id: remover tarefa
* PATCH /tarefas/:id/status: atualizar status

**📦 Pastas organizadas no final da semana:**

src/

├── tarefas/

│ ├── dto/

│ │ └── create-tarefa.dto.ts

│ ├── tarefas.controller.ts

│ ├── tarefas.service.ts

│ ├── tarefas.module.ts

├── main.ts

└── app.module.ts

Se quiser, posso montar o **código base da semana 1** em partes ou te guiar a partir do primeiro dia com os comandos e arquivos. Deseja que comecemos isso agora?

Agora quero que crie a segunda semana do plano de estudo do Nest.js

Perfeito! Agora que você já dominou os fundamentos do **Nest.js** (módulos, controllers, services, DTOs), vamos avançar na **segunda semana** focando em:

* Validação
* Pipes
* Middleware
* Guards (autenticação/autorização)
* Filtros de exceção
* Integração com banco de dados (preparando o terreno pro TypeORM)

**🗂️ Plano de Estudo – Nest.js (Semana 2)**

🎯 **Objetivo:** Aprimorar sua API com validações, autenticação básica, tratamento de erros e preparar para integração com banco de dados (TypeORM).

**🗓️ Dia 8 – Pipes e validação de dados**

**📚 Conteúdo:**

* O que são **Pipes** no Nest.js
* Usando **@Body() com DTOs**
* Usando **class-validator e class-transformer**

**🔧 Atividades:**

* Instalar os pacotes:

npm install class-validator class-transformer

* Criar validador no CreateTarefaDto:

import { IsNotEmpty } from 'class-validator';

export class CreateTarefaDto {

@IsNotEmpty()

titulo: string;

@IsNotEmpty()

descricao: string;

}

**🧪 Exercício:**

* Criar um FilterDto para buscar tarefas por status e termo com validação usando @IsOptional().

**🗓️ Dia 9 – Pipes customizados**

**📚 Conteúdo:**

* Criar um **pipe de validação de status**
* Usar @Param() com pipes

**🔧 Atividades:**

* Criar TarefaStatusValidationPipe
* Validar se status recebido é válido (enum)

**🧪 Exercício:**

* Adicionar pipe no endpoint PATCH /tarefas/:id/status

**🗓️ Dia 10 – Exception Filters (tratamento de erros)**

**📚 Conteúdo:**

* O que são filtros de exceção
* Usando HttpException e NotFoundException
* Criar filtro global customizado (opcional)

**🔧 Atividades:**

* Criar exceção personalizada se uma tarefa não for encontrada.

throw new NotFoundException(`Tarefa com ID "${id}" não encontrada`);

**🧪 Exercício:**

* Adicionar exceções personalizadas para os métodos GET, DELETE e UPDATE.

**🗓️ Dia 11 – Middleware**

**📚 Conteúdo:**

* O que é middleware no Nest.js
* Diferença entre middleware, guards e interceptadores
* Aplicar middleware globalmente ou por rota

**🔧 Atividades:**

* Criar middleware de log:

nest g middleware logger

* Aplicar nas rotas de tarefas

**🧪 Exercício:**

* Adicionar timestamp e rota acessada no log.

**🗓️ Dia 12 – Guards (autenticação básica)**

**📚 Conteúdo:**

* O que são guards
* Criar um guard simples (ex: autenticação via token hardcoded)
* Usar @UseGuards() no controller

**🔧 Atividades:**

* Criar AuthGuard que verifica se authorization tem token meutoken123.

**🧪 Exercício:**

* Proteger rota de criação de tarefas com guard.

**🗓️ Dia 13 – Configuração com .env e instalação de TypeORM**

**📚 Conteúdo:**

* Uso do pacote @nestjs/config
* Criar .env para credenciais de banco
* Instalar pacotes do TypeORM

npm install --save @nestjs/typeorm typeorm pg

**🔧 Atividades:**

* Criar .env:

DB\_HOST=localhost

DB\_PORT=5432

DB\_USERNAME=usuario

DB\_PASSWORD=senha

DB\_NAME=nest\_tarefas

* Configurar TypeOrmModule.forRootAsync() em app.module.ts

**🧪 Exercício:**

* Simular a conexão (sem entidades ainda), garantir que app inicia com sucesso.

**🗓️ Dia 14 – Preparar para uso do banco com TypeORM**

**📚 Conteúdo:**

* O que são entidades
* Introdução a decorators do TypeORM
* Diferença entre entidades e DTOs
* Visão geral de repository, migration, relation

**🔧 Atividades:**

* Criar a primeira entidade TarefaEntity com os campos id, titulo, descricao, status, dataCriacao.

@Entity()

export class Tarefa {

@PrimaryGeneratedColumn('uuid')

id: string;

@Column()

titulo: string;

@Column()

descricao: string;

@Column()

status: TarefaStatus;

@CreateDateColumn()

dataCriacao: Date;

}

**🧠 Ao final da semana, você terá:**

✅ API validando dados  
✅ Erros tratados corretamente  
✅ Middleware e autenticação básica funcionando  
✅ Projeto pronto para usar TypeORM com banco de dados real

Se quiser, posso criar o **projeto Nest.js com TypeORM e banco PostgreSQL** como ponto de partida para a próxima etapa. Deseja isso?

Agora quero que crie a terceira semana do plano de estudo do Nest.js

Ótimo! Na terceira semana, você já terá domínio da estrutura do Nest.js e estará pronto para integrar com banco de dados de forma profissional usando o **TypeORM**. Também vamos avançar com autenticação real (com JWT), relacionamento entre entidades e a criação de um projeto completo.

**🗂️ Plano de Estudo – Nest.js (Semana 3)**

🎯 **Objetivo:** Persistência com banco de dados usando TypeORM, criação de entidades, relacionamento, autenticação com JWT e finalização de um projeto completo.

**🗓️ Dia 15 – Integrar TypeORM com Nest.js**

**📚 Conteúdo:**

* Relembrar .env com variáveis de conexão
* Importar TypeOrmModule no AppModule
* Conectar ao PostgreSQL (ou outro)

**🔧 Atividades:**

* Configurar TypeOrmModule.forRoot() com dados do .env
* Testar conexão
* Criar entidade TarefaEntity

@Entity('tarefas')

export class TarefaEntity {

@PrimaryGeneratedColumn('uuid')

id: string;

@Column()

titulo: string;

@Column()

descricao: string;

@Column()

status: TarefaStatus;

}

**🧪 Exercício:**

* Criar repositório de tarefas e salvar tarefa no banco.

**🗓️ Dia 16 – CRUD com banco de dados**

**📚 Conteúdo:**

* Criar TarefaRepository
* Uso do InjectRepository
* Métodos: save, find, findOne, delete

**🔧 Atividades:**

* Refatorar tarefas.service.ts para usar o banco de dados em vez de array

**🧪 Exercício:**

* Implementar os métodos: findAll, findById, create, delete, updateStatus

**🗓️ Dia 17 – Autenticação com JWT (parte 1)**

**📚 Conteúdo:**

* Criar módulo de usuários
* Criar entidade UsuarioEntity
* Instalar pacotes:

npm install @nestjs/jwt @nestjs/passport passport passport-jwt bcryptjs

**🔧 Atividades:**

* Criar UsuarioModule, UsuarioService, UsuarioEntity
* Registrar e salvar novo usuário com senha criptografada (bcrypt)

**🗓️ Dia 18 – Autenticação com JWT (parte 2)**

**📚 Conteúdo:**

* Criar AuthModule e serviço de autenticação
* Validar login, gerar token JWT
* Criar Guard com Passport para proteger rotas

**🔧 Atividades:**

* Criar rota POST /auth/login
* Implementar JwtStrategy para validar token

**🧪 Exercício:**

* Proteger rotas de tarefas com @UseGuards(AuthGuard())

**🗓️ Dia 19 – Relacionamentos com TypeORM**

**📚 Conteúdo:**

* Criar relacionamento User → Tarefa (1:N)
* Usar @ManyToOne() e @OneToMany()

**🔧 Atividades:**

* Alterar entidade TarefaEntity para ter um campo usuario

@ManyToOne(() => UsuarioEntity, usuario => usuario.tarefas, { eager: false })

usuario: UsuarioEntity;

* Ajustar criação de tarefas para registrar o usuário dono da tarefa

**🧪 Exercício:**

* Buscar tarefas de um usuário logado (usando o token JWT)

**🗓️ Dia 20 – Filtros, DTOs avançados e refatorações**

**📚 Conteúdo:**

* Criar DTO para login e registro
* DTOs para atualizar tarefas com validações
* Filtros com parâmetros opcionais

**🔧 Atividades:**

* Refatorar rotas para usar DTOs consistentes
* Criar filtros por status, título e usuário

**🧪 Exercício:**

* Adicionar GET /tarefas?status=ABERTA&busca=texto usando query params

**🗓️ Dia 21 – Finalizando o projeto**

**📚 Conteúdo:**

* Testar todas as rotas
* Documentar API com Swagger
* Preparar deploy local (opcional)

**🔧 Atividades:**

* Instalar Swagger:

npm install --save @nestjs/swagger swagger-ui-express

* Documentar rotas com decorators @ApiTags, @ApiBody, @ApiResponse, etc.

**📦 Estrutura do projeto ao final da 3ª semana:**

src/

├── auth/

│ ├── auth.module.ts

│ ├── auth.service.ts

│ ├── jwt.strategy.ts

│ └── dtos/

├── usuario/

│ ├── usuario.entity.ts

│ ├── usuario.service.ts

├── tarefas/

│ ├── tarefa.entity.ts

│ ├── tarefas.controller.ts

│ ├── tarefas.service.ts

├── common/

│ └── pipes/

├── main.ts

├── app.module.ts

**✅ Ao final da 3ª semana, você terá:**

* API com banco de dados real (PostgreSQL ou outro)
* CRUD completo persistente com autenticação
* Validação com DTOs e Pipes
* Segurança com JWT e Guards
* Relacionamentos com usuários e entidades protegidas
* Documentação com Swagger

Se quiser, posso gerar um repositório base com essa estrutura ou te acompanhar na codificação passo a passo do projeto. Deseja começar por isso agora?

Claro! Abaixo está a **aula completa do Dia 1 – Introdução ao Nest.js**, com explicações teóricas, comandos práticos e exercício final. Ideal para iniciantes e serve como base sólida para os próximos dias.

**📘 Dia 1 – Introdução ao Nest.js**

**📚 Conteúdo**

**✅ O que é o Nest.js?**

* **Nest.js** é um **framework Node.js** para a construção de aplicações **escaláveis**, **eficientes** e **modularizadas**, baseado em **TypeScript**.
* Ele utiliza princípios de **Programação Orientada a Objetos**, **Programação Funcional** e **Arquitetura Modular**.

**✅ Por que usar Nest.js?**

* Código limpo e estruturado (sem caos de arquivos soltos)
* Baseado em **módulos**: cada domínio da aplicação tem seu próprio módulo
* Compatível com bibliotecas do Express e Fastify
* Suporte nativo a **TypeORM**, **JWT**, **Swagger**, **WebSockets** e muito mais
* Fácil escalabilidade para projetos grandes

**✅ Arquitetura e Injeção de Dependência**

* Usa arquitetura **modular**: divide funcionalidades em módulos (ex: TarefasModule, UsuarioModule)
* Usa **Injeção de Dependência (DI)** para desacoplar classes: os serviços são injetados nos controladores via construtor
* Baseado no padrão **MVC**: Controller → Service → Repository (quando usa banco)

**🔧 Atividades Práticas**

**1️⃣ Instalar o Nest CLI**

npm install -g @nestjs/cli

**2️⃣ Criar um novo projeto Nest**

nest new projeto-tarefas

Você será perguntado qual gerenciador de pacotes deseja usar. Pode escolher npm ou yarn.

**3️⃣ Estrutura de arquivos gerada**

Após a criação, observe os principais arquivos:

projeto-tarefas/

├── src/

│ ├── app.controller.ts // Controlador padrão

│ ├── app.service.ts // Serviço padrão

│ ├── app.module.ts // Módulo raiz

│ └── main.ts // Ponto de entrada da aplicação

├── test/ // Testes gerados automaticamente

├── package.json // Dependências e scripts

├── tsconfig.json // Configuração TypeScript

**4️⃣ Rodar o servidor de desenvolvimento**

Acesse a pasta do projeto:

cd projeto-tarefas

npm run start:dev

Você verá algo como:

[Nest] 3456 - 23/06/2025, 14:00:00 [NestFactory] Starting Nest application...

[Nest] 3456 - 23/06/2025, 14:00:01 [RoutesResolver] AppController {/}: +1 route

[Nest] 3456 - 23/06/2025, 14:00:01 [NestApplication] Nest application successfully started on port 3000

Acesse [http://localhost:3000](http://localhost:3000/) e verá a mensagem:

Hello World!

**🧪 Exercício Final**

Abra o arquivo src/main.ts e **adicione um console.log()** para verificar se o ambiente está em desenvolvimento:

async function bootstrap() {

console.log('Ambiente atual:', process.env.NODE\_ENV || 'desenvolvimento');

const app = await NestFactory.create(AppModule);

await app.listen(3000);

}

bootstrap();

Execute novamente com:

npm run start:dev

Verifique no terminal a saída do console.

**✅ O que você aprendeu hoje:**

✔ O que é o Nest.js e por que ele é utilizado  
✔ Como criar um projeto usando Nest CLI  
✔ Como a arquitetura modular funciona  
✔ Como rodar e explorar a estrutura inicial do projeto  
✔ Como modificar o ponto de entrada e testar algo no console

Se quiser, posso te guiar no **Dia 2 – Controllers no Nest.js**, onde você já começará a criar rotas reais na sua API. Deseja seguir?

Claro! Abaixo está a **aula completa do Dia 2 – Controladores (Controllers)** no Nest.js, com explicações didáticas, código comentado, passo a passo e exercícios práticos.

**📘 Dia 2 – Controladores no Nest.js**

**📚 Conteúdo Teórico**

**✅ O que são controladores?**

No Nest.js, os **Controllers** (controladores) são responsáveis por lidar com as **requisições HTTP**. Eles recebem os dados da requisição, **chamam os serviços** responsáveis pela lógica de negócio e devolvem uma resposta ao cliente.

**✅ Decoradores principais**

Nest usa **decoradores** para definir rotas e comportamento:

| **Decorador** | **Descrição** |
| --- | --- |
| @Controller('rota') | Define o prefixo da rota do controller |
| @Get(), @Post(), @Put(), @Delete() | Define o tipo de requisição HTTP |
| @Param() | Captura parâmetros da URL |
| @Query() | Captura parâmetros de query string |
| @Body() | Captura o corpo da requisição (em POST, PUT) |

**🔧 Atividades Práticas**

**1️⃣ Criar um controller chamado tarefas**

No terminal, dentro da pasta do projeto:

nest g controller tarefas

Isso criará um novo controller em src/tarefas/tarefas.controller.ts

**2️⃣ Editar o controller para adicionar rotas GET**

Abra src/tarefas/tarefas.controller.ts e adicione os seguintes métodos:

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

@Controller('tarefas') // prefixo: todas as rotas começam com /tarefas

export class TarefasController {

private tarefas = [

{ id: 1, titulo: 'Estudar Nest.js', status: 'ABERTA' },

{ id: 2, titulo: 'Fazer exercícios', status: 'EM\_ANDAMENTO' },

];

// GET /tarefas

@Get()

getTodasTarefas() {

return this.tarefas;

}

// GET /tarefas/:id

@Get(':id')

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

return this.tarefas.find((tarefa) => tarefa.id === parseInt(id));

}

// GET /tarefas/status/:status

@Get('status/:status')

getTarefasPorStatus(@Param('status') status: string) {

return this.tarefas.filter((tarefa) => tarefa.status === status.toUpperCase());

}

}

**🚀 Testando no navegador ou Postman**

* GET http://localhost:3000/tarefas → retorna todas as tarefas
* GET http://localhost:3000/tarefas/1 → retorna tarefa com ID 1
* GET http://localhost:3000/tarefas/status/aberta → retorna todas com status ABERTA

**🧪 Exercício Final**

Adicione uma nova tarefa ao array e teste as rotas. Exemplo:

{ id: 3, titulo: 'Revisar conteúdo do Dia 2', status: 'FINALIZADA' }

Depois teste:

* GET /tarefas/3
* GET /tarefas/status/finalizada

**✅ O que você aprendeu hoje:**

✔ Criar um controller com o CLI (nest g controller)  
✔ Usar os decoradores @Controller(), @Get(), @Param()  
✔ Criar rotas com parâmetros dinâmicos  
✔ Retornar dados estáticos para testes iniciais  
✔ Preparar o terreno para conectar com serviços (Dia 3)

Se quiser, posso seguir com o **Dia 3 – Serviços (Services)** para ensinar como separar a lógica de negócio do controller e criar um sistema mais organizado. Deseja seguir com isso agora?

Claro! Aqui está a aula completa e detalhada para o **📘 Dia 3 – Serviços (Services)** do seu plano de estudo de Nest.js, com explicação teórica, atividades práticas e exercício final.

**📘 Dia 3 – Serviços no Nest.js**

**📚 Conteúdo Teórico**

**✅ O que são serviços no Nest.js?**

* **Services** (ou **serviços**) são responsáveis por **conter a lógica de negócio** da aplicação.
* Enquanto os **controllers** lidam com as **requisições e respostas HTTP**, os serviços executam o que realmente precisa ser feito.

✨ Eles promovem a separação de responsabilidades (princípio SOLID: Single Responsibility).

**✅ @Injectable() e Injeção de Dependência**

* O decorator @Injectable() diz ao Nest.js que aquela classe pode ser **injetada** em outras classes via o construtor.
* Isso permite que o Nest.js **gere automaticamente instâncias** dos serviços e controle o ciclo de vida deles.

**🔧 Atividades Práticas**

**1️⃣ Criar o serviço de tarefas**

No terminal:

nest g service tarefas

Isso vai gerar o arquivo tarefas.service.ts dentro da pasta src/tarefas.

**2️⃣ Mover a lógica para o serviço**

Abra src/tarefas/tarefas.service.ts e adicione a seguinte implementação:

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

@Injectable()

export class TarefasService {

private tarefas = [

{ id: 1, titulo: 'Estudar Nest.js', status: 'ABERTA' },

{ id: 2, titulo: 'Fazer exercícios', status: 'EM\_ANDAMENTO' },

{ id: 3, titulo: 'Revisar conteúdo do Dia 2', status: 'FINALIZADA' },

];

getTodasTarefas() {

return this.tarefas;

}

getTarefaPorId(id: number) {

return this.tarefas.find((tarefa) => tarefa.id === id);

}

getTarefasPorStatus(status: string) {

return this.tarefas.filter((t) => t.status === status.toUpperCase());

}

}

**3️⃣ Injetar o serviço no controller**

Abra src/tarefas/tarefas.controller.ts e modifique para usar o serviço:

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

import { TarefasService } from './tarefas.service';

@Controller('tarefas')

export class TarefasController {

constructor(private readonly tarefasService: TarefasService) {}

@Get()

getTodasTarefas() {

return this.tarefasService.getTodasTarefas();

}

@Get(':id')

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

return this.tarefasService.getTarefaPorId(parseInt(id));

}

@Get('status/:status')

getTarefasPorStatus(@Param('status') status: string) {

return this.tarefasService.getTarefasPorStatus(status);

}

}

**⚠️ Atenção**

Caso você veja erro como Nest can't resolve dependencies of the TarefasController, certifique-se de que o TarefasService está **declarado no módulo** TarefasModule (ou no AppModule, se ainda não criou um módulo próprio):

@Module({

controllers: [TarefasController],

providers: [TarefasService],

})

export class TarefasModule {}

**🧪 Exercício Final**

Crie o método getTarefaPorId(id: string) no serviço (já feito acima) e:

1. Refatore o controller para usar esse método.
2. Teste a rota GET /tarefas/:id com valores existentes e inexistentes.
3. Tente retornar uma mensagem personalizada se não encontrar nenhuma tarefa (opcional para o próximo dia, quando trataremos erros).

**✅ O que você aprendeu hoje:**

✔ O que é um serviço no Nest.js  
✔ Como criar um serviço com @Injectable()  
✔ Como aplicar **injeção de dependência** no controller  
✔ Como **separar a lógica de negócio** do controller  
✔ Como preparar seu projeto para escalar de forma organizada

Pronto para o **📘 Dia 4 – DTOs e Tipagem**, onde você vai criar classes para entrada de dados e aplicar **validação forte com TypeScript**? Posso montar a aula para você agora, se quiser. Deseja continuar?

Com certeza! Aqui está a aula completa para o **📘 Dia 4 – DTOs e Tipagem** no mesmo formato dos dias anteriores, totalmente em **Markdown** e pronta para ser usada como guia de estudos ou aula.

**📘 Dia 4 – DTOs e Tipagem no Nest.js**

**📚 Conteúdo Teórico**

**✅ O que é um DTO (Data Transfer Object)?**

* DTO é uma **classe que define a forma dos dados** que são **enviados ou recebidos** pela sua API.
* Serve como contrato entre quem envia e quem recebe dados.
* Com Nest.js (e TypeScript), os DTOs são usados junto com validações e decorators.

**✅ Interfaces ou Classes para tipos**

* Você pode definir tipos com interface ou class.
* Nest.js recomenda o uso de **classes**, pois elas funcionam com bibliotecas de validação como class-validator.

**✅ Validação com TypeScript**

* Com TypeScript, é possível **garantir a estrutura dos objetos** e melhorar a **inteligência do editor**.
* Podemos usar **tipos enumerados (enum)** para restringir valores possíveis, como status de uma tarefa.

**🔧 Atividades Práticas**

**1️⃣ Criar um DTO de criação de tarefa**

Crie o arquivo:  
📁 src/tarefas/dto/create-tarefa.dto.ts

export class CreateTarefaDto {

titulo: string;

descricao: string;

}

Essa classe representa os dados esperados na criação de uma tarefa.

**2️⃣ Criar um enum de status de tarefa**

Crie o arquivo:  
📁 src/tarefas/tarefa-status.enum.ts

export enum TarefaStatus {

ABERTA = 'ABERTA',

EM\_ANDAMENTO = 'EM\_ANDAMENTO',

FINALIZADA = 'FINALIZADA',

}

Esse enum define os únicos status válidos para uma tarefa.

**3️⃣ Criar o tipo Tarefa**

Você pode criar uma interface ou classe (vamos usar interface por enquanto):  
📁 src/tarefas/tarefa.model.ts

import { TarefaStatus } from './tarefa-status.enum';

export interface Tarefa {

id: number;

titulo: string;

descricao: string;

status: TarefaStatus;

}

**4️⃣ Atualizar o serviço para usar os tipos**

Abra o arquivo tarefas.service.ts e modifique para usar a nova interface:

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

import { Tarefa } from './tarefa.model';

import { TarefaStatus } from './tarefa-status.enum';

import { CreateTarefaDto } from './dto/create-tarefa.dto';

@Injectable()

export class TarefasService {

private tarefas: Tarefa[] = [];

getTodasTarefas(): Tarefa[] {

return this.tarefas;

}

getTarefaPorId(id: number): Tarefa {

return this.tarefas.find((t) => t.id === id);

}

getTarefasPorStatus(status: string): Tarefa[] {

return this.tarefas.filter((t) => t.status === status.toUpperCase());

}

createTarefa(dto: CreateTarefaDto): Tarefa {

const { titulo, descricao } = dto;

const tarefa: Tarefa = {

id: Date.now(), // simulando ID único

titulo,

descricao,

status: TarefaStatus.ABERTA,

};

this.tarefas.push(tarefa);

return tarefa;

}

}

**5️⃣ Atualizar o controller para receber dados via POST**

Abra tarefas.controller.ts e adicione:

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

import { TarefasService } from './tarefas.service';

import { CreateTarefaDto } from './dto/create-tarefa.dto';

import { Tarefa } from './tarefa.model';

@Controller('tarefas')

export class TarefasController {

constructor(private readonly tarefasService: TarefasService) {}

@Get()

getTodasTarefas(): Tarefa[] {

return this.tarefasService.getTodasTarefas();

}

@Get(':id')

getTarefaPorId(@Param('id') id: string): Tarefa {

return this.tarefasService.getTarefaPorId(parseInt(id));

}

@Get('status/:status')

getTarefasPorStatus(@Param('status') status: string): Tarefa[] {

return this.tarefasService.getTarefasPorStatus(status);

}

@Post()

createTarefa(@Body() createTarefaDto: CreateTarefaDto): Tarefa {

return this.tarefasService.createTarefa(createTarefaDto);

}

}

**🧪 Exercício Final**

1. Faça um POST com JSON no Postman para o endpoint:

POST http://localhost:3000/tarefas

{

"titulo": "Estudar DTOs",

"descricao": "Aprender como usar DTOs no Nest.js"

}

1. Depois acesse:

* GET /tarefas para ver a tarefa criada
* GET /tarefas/ID para ver a tarefa por ID
* GET /tarefas/status/ABERTA para filtrar por status

**✅ O que você aprendeu hoje:**

✔ O que é um DTO e por que usá-lo  
✔ Como definir tipos com classes, interfaces e enums  
✔ Como aplicar DTOs no fluxo de criação de tarefas  
✔ Como usar @Body() para receber dados no Nest.js  
✔ Como preparar seu código para aplicar **validações reais** no próximo dia

Se quiser, posso seguir com o **📘 Dia 5 – Injeção de dependência e boas práticas**, onde refinamos o uso dos serviços e aplicamos organização mais limpa ao código. Deseja que eu monte essa próxima aula?

Claro! Abaixo está a aula elaborada do **📘 Dia 5 – Injeção de Dependência e Boas Práticas** no formato Markdown, seguindo o mesmo padrão dos dias anteriores.

**📘 Dia 5 – Injeção de Dependência e Boas Práticas**

**📚 Conteúdo Teórico**

**✅ Injeção de dependência com constructor()**

* No Nest.js, a **injeção de dependência** acontece automaticamente via o **construtor** das classes.
* Ao declarar um parâmetro no construtor com o tipo de uma classe decorada com @Injectable(), o Nest resolve e fornece a instância para você.

Exemplo:

constructor(private readonly tarefasService: TarefasService) {}

Isso injeta o serviço TarefasService no controller.

**✅ Boas práticas: separar DTOs, entidades, enums e módulos**

Organizar o projeto em **camadas bem definidas** é essencial:

src/

├── tarefas/

│ ├── dto/ → Dados de entrada

│ ├── enums/ → Enums (status, roles, etc.)

│ ├── tarefas.controller.ts → Lida com requisições

│ ├── tarefas.service.ts → Lógica de negócio

│ ├── tarefa.model.ts → Tipo de tarefa

│ └── tarefa-status.enum.ts → Status enum

Isso ajuda a manter o projeto limpo e escalável.

**✅ Status padrão ABERTA no método createTarefa()**

Sempre que uma nova tarefa for criada, ela deve começar com o status **ABERTA**, sem depender do usuário enviar esse valor.

**🔧 Atividades Práticas**

**1️⃣ Revisar e mover o enum para pasta organizada**

Crie a pasta src/tarefas/enums/ e mova o arquivo tarefa-status.enum.ts para lá:

// src/tarefas/enums/tarefa-status.enum.ts

export enum TarefaStatus {

ABERTA = 'ABERTA',

EM\_ANDAMENTO = 'EM\_ANDAMENTO',

FINALIZADA = 'FINALIZADA',

}

Atualize os imports onde for necessário.

**2️⃣ Atualizar o método createTarefa() no serviço**

Garanta que o status da tarefa criada seja sempre ABERTA, mesmo que não venha do DTO:

// src/tarefas/tarefas.service.ts

import { TarefaStatus } from './enums/tarefa-status.enum';

createTarefa(dto: CreateTarefaDto): Tarefa {

const { titulo, descricao } = dto;

const tarefa: Tarefa = {

id: Date.now(),

titulo,

descricao,

status: TarefaStatus.ABERTA, // valor padrão

};

this.tarefas.push(tarefa);

return tarefa;

}

**🧪 Exercício Final**

**Criar um método filtrarTarefasPorStatus(status: string)**

No tarefas.service.ts:

filtrarTarefasPorStatus(status: string): Tarefa[] {

return this.tarefas.filter(

(tarefa) => tarefa.status === status.toUpperCase(),

);

}

No tarefas.controller.ts, atualize o método correspondente:

@Get('status/:status')

getTarefasPorStatus(@Param('status') status: string): Tarefa[] {

return this.tarefasService.filtrarTarefasPorStatus(status);

}

**✅ O que você aprendeu hoje:**

✔ Como funciona a **injeção de dependência** no Nest.js  
✔ Como **organizar seu projeto** separando DTOs, enums, controllers e services  
✔ Como definir um **valor padrão** ao criar uma entidade (status ABERTA)  
✔ Como implementar um **filtro por status** usando enums

🎯 Pronto para o **📘 Dia 6 – Módulos no Nest.js**, onde você aprenderá a dividir sua aplicação em partes reutilizáveis e organizadas? Posso montar essa aula para você agora. Deseja seguir?

Sim, **com essas alterações você pode testar normalmente no Postman**! Aqui está um resumo e explicação para garantir que tudo funcione corretamente:

**✅ O que você fez:**

**No tarefas.service.ts:**

filtrarTarefasPorStatus(status: string): Tarefa[] {

return this.tarefas.filter(

(tarefa) => tarefa.status === status.toUpperCase(),

);

}

Esse método filtra as tarefas comparando o status fornecido com o status das tarefas existentes, convertendo para maiúsculas para evitar erros de digitação.

**No tarefas.controller.ts:**

@Get('status/:status')

getTarefasPorStatus(@Param('status') status: string): Tarefa[] {

return this.tarefasService.filtrarTarefasPorStatus(status);

}

Aqui você está definindo uma rota GET com parâmetro de rota status.

**✅ Como testar no Postman**

1. **Certifique-se de que o servidor está rodando**:
2. npm run start:dev
3. **Faça uma requisição GET no Postman para:**
4. GET http://localhost:3000/tarefas/status/ABERTA

Ou teste com:

* + /tarefas/status/em\_andamento
  + /tarefas/status/finalizada

Dica: como o método faz .toUpperCase(), você pode enviar o status em minúsculas, maiúsculas ou misto – tudo será tratado corretamente.

**✅ Exemplo de resposta esperada no Postman:**

[

{

"id": 123456789,

"titulo": "Estudar Nest.js",

"descricao": "Aprender como funciona o framework",

"status": "ABERTA"

}

]

**⚠️ Dica extra: use enum para evitar status inválido**

Para deixar ainda mais seguro, você pode adicionar uma verificação se o status existe no enum TarefaStatus — isso será feito futuramente com **pipes de validação**.

Se quiser, posso te ajudar a implementar essa validação personalizada com enum ainda hoje. Deseja isso agora ou seguimos com o **Dia 6 – Módulos no Nest.js**?

Claro! Aqui está a aula completa e detalhada do **📘 Dia 6 – Módulos no Nest.js**, em formato Markdown, seguindo o padrão dos dias anteriores.

**📘 Dia 6 – Módulos no Nest.js**

**📚 Conteúdo Teórico**

**✅ O que são módulos no Nest.js?**

* Módulos são a **base da organização** de qualquer aplicação Nest.js.
* Cada módulo agrupa **controllers**, **services**, **providers**, e **outros recursos** que compartilham um mesmo propósito.
* A estrutura modular permite dividir a aplicação em **domínios independentes**, facilitando a manutenção e escalabilidade.

O Nest sempre começa com um AppModule (módulo raiz), e você pode criar quantos módulos precisar.

**✅ Por que dividir em módulos?**

* **Isolamento de responsabilidade**: cada módulo trata de uma parte específica da aplicação.
* **Reutilização**: módulos podem ser reutilizados em outras partes ou projetos.
* **Escalabilidade**: facilita adicionar novos domínios e funcionalidades.

**✅ Exemplo de estrutura modular:**

src/

├── app.module.ts

├── tarefas/

│ ├── tarefas.module.ts → Módulo de tarefas

│ ├── tarefas.controller.ts → Controller de tarefas

│ ├── tarefas.service.ts → Serviço de tarefas

│ ├── dto/

│ ├── enums/

│ ├── tarefa.model.ts

**🔧 Atividades Práticas**

**1️⃣ Criar módulo de tarefas (caso não tenha feito ainda)**

No terminal:

nest g module tarefas

Isso criará o arquivo src/tarefas/tarefas.module.ts

**2️⃣ Garantir que o módulo está completo**

Abra o arquivo tarefas.module.ts e **verifique se está importando corretamente o controller e o service**:

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

import { TarefasController } from './tarefas.controller';

import { TarefasService } from './tarefas.service';

@Module({

controllers: [TarefasController],

providers: [TarefasService],

})

export class TarefasModule {}

**3️⃣ Registrar o módulo no módulo raiz (AppModule)**

Abra src/app.module.ts e **adicione o módulo de tarefas no array imports**:

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

import { TarefasModule } from './tarefas/tarefas.module';

@Module({

imports: [TarefasModule],

})

export class AppModule {}

**🧪 Exercício Final**

1. Pare e reinicie o servidor (Ctrl+C e npm run start:dev) para aplicar as alterações.
2. Confirme se as rotas de tarefas continuam funcionando normalmente:
   * GET /tarefas
   * GET /tarefas/1
   * POST /tarefas
3. Teste remover o TarefasService de dentro do AppModule (se ainda estiver lá) para garantir que a **responsabilidade está centralizada no módulo de tarefas**.

**✅ O que você aprendeu hoje:**

✔ O que são e para que servem **módulos** no Nest.js  
✔ Como **organizar o projeto** por domínio funcional usando módulos  
✔ Como criar e registrar **módulos customizados**  
✔ Como garantir o **isolamento de responsabilidade** de cada parte da aplicação

🎯 Pronto para o **📘 Dia 7 – Aplicando lógica de filtros com query params**?  
Você vai aprender a criar rotas como GET /tarefas?status=ABERTA e a usar @Query() para extrair dados da URL. Deseja que eu monte essa próxima aula?

Claro! Abaixo está a **aula completa do 📘 Dia 7 – Revisão e Prática**, fechando a semana 1 do plano de estudos Nest.js com uma atividade integradora prática. A estrutura está em **Markdown**, seguindo o padrão dos dias anteriores.

**📘 Dia 7 – Revisão e Prática: Criando uma API de Tarefas**

**📚 Revisão de Conceitos**

**🔁 Fluxo de funcionamento no Nest.js**

1. **Controller**: recebe a requisição HTTP (GET, POST, etc.)
2. **Service**: contém a **lógica de negócio**
3. **DTOs e Enums**: definem estrutura e regras dos dados
4. **Resposta**: volta para o controller e é enviada para o cliente

**🛠️ Objetivo da Aula**

Criar um mini projeto Nest.js de uma **API de tarefas**, com armazenamento em memória e rotas completas para manipular as tarefas.

Sem banco de dados por enquanto – o foco é praticar controllers, services, DTOs, módulos, enums e boas práticas.

**🧱 Requisitos da API**

**✅ Endpoints:**

| **Método** | **Rota** | **Ação** |
| --- | --- | --- |
| GET | /tarefas | Listar todas as tarefas |
| GET | /tarefas/:id | Obter uma tarefa pelo ID |
| POST | /tarefas | Criar uma nova tarefa |
| DELETE | /tarefas/:id | Remover uma tarefa |
| PATCH | /tarefas/:id/status | Atualizar o status da tarefa |

**🔧 Passo a passo da implementação**

**1️⃣ Criar a estrutura do módulo de tarefas**

Caso ainda não tenha criado:

nest g module tarefas

nest g controller tarefas

nest g service tarefas

**2️⃣ Criar TarefaStatus enum**

📁 src/tarefas/enums/tarefa-status.enum.ts

export enum TarefaStatus {

ABERTA = 'ABERTA',

EM\_ANDAMENTO = 'EM\_ANDAMENTO',

FINALIZADA = 'FINALIZADA',

}

**3️⃣ Criar a interface Tarefa**

📁 src/tarefas/tarefa.model.ts

import { TarefaStatus } from './enums/tarefa-status.enum';

export interface Tarefa {

id: number;

titulo: string;

descricao: string;

status: TarefaStatus;

}

**4️⃣ Criar DTO de criação de tarefa**

📁 src/tarefas/dto/create-tarefa.dto.ts

export class CreateTarefaDto {

titulo: string;

descricao: string;

}

**5️⃣ Criar o TarefasService**

📁 src/tarefas/tarefas.service.ts

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

import { Tarefa } from './tarefa.model';

import { CreateTarefaDto } from './dto/create-tarefa.dto';

import { TarefaStatus } from './enums/tarefa-status.enum';

@Injectable()

export class TarefasService {

private tarefas: Tarefa[] = [];

getTodas(): Tarefa[] {

return this.tarefas;

}

getPorId(id: number): Tarefa {

return this.tarefas.find(t => t.id === id);

}

criar(dto: CreateTarefaDto): Tarefa {

const tarefa: Tarefa = {

id: Date.now(), // gera ID único com timestamp

titulo: dto.titulo,

descricao: dto.descricao,

status: TarefaStatus.ABERTA,

};

this.tarefas.push(tarefa);

return tarefa;

}

remover(id: number): void {

this.tarefas = this.tarefas.filter(t => t.id !== id);

}

atualizarStatus(id: number, status: TarefaStatus): Tarefa {

const tarefa = this.getPorId(id);

tarefa.status = status;

return tarefa;

}

}

**6️⃣ Criar o TarefasController**

📁 src/tarefas/tarefas.controller.ts

import {

Controller,

Get,

Param,

Post,

Delete,

Patch,

Body,

} from '@nestjs/common';

import { TarefasService } from './tarefas.service';

import { CreateTarefaDto } from './dto/create-tarefa.dto';

import { Tarefa } from './tarefa.model';

import { TarefaStatus } from './enums/tarefa-status.enum';

@Controller('tarefas')

export class TarefasController {

constructor(private readonly tarefasService: TarefasService) {}

@Get()

getTodas(): Tarefa[] {

return this.tarefasService.getTodas();

}

@Get(':id')

getPorId(@Param('id') id: string): Tarefa {

return this.tarefasService.getPorId(Number(id));

}

@Post()

criar(@Body() dto: CreateTarefaDto): Tarefa {

return this.tarefasService.criar(dto);

}

@Delete(':id')

remover(@Param('id') id: string): void {

this.tarefasService.remover(Number(id));

}

@Patch(':id/status')

atualizarStatus(

@Param('id') id: string,

@Body('status') status: TarefaStatus,

): Tarefa {

return this.tarefasService.atualizarStatus(Number(id), status);

}

}

**📦 Estrutura final do projeto (sem banco)**

src/

├── tarefas/

│ ├── dto/

│ │ └── create-tarefa.dto.ts

│ ├── enums/

│ │ └── tarefa-status.enum.ts

│ ├── tarefa.model.ts

│ ├── tarefas.controller.ts

│ ├── tarefas.service.ts

│ └── tarefas.module.ts

├── app.module.ts

└── main.ts

**🧪 Exercício Final**

1. Testar os endpoints no Postman:
   * POST /tarefas com:
   * {
   * "titulo": "Estudar para a prova",
   * "descricao": "Focar nos exercícios de Nest.js"
   * }
   * GET /tarefas
   * GET /tarefas/:id
   * PATCH /tarefas/:id/status com:
   * { "status": "EM\_ANDAMENTO" }
   * DELETE /tarefas/:id
2. Tentar criar múltiplas tarefas e alternar status.

**✅ O que você aprendeu hoje:**

✔ Como **juntar tudo que aprendeu na semana** em um mini projeto  
✔ Como aplicar DTOs, enums, services e controllers de forma prática  
✔ Como criar uma API REST funcional sem banco de dados  
✔ Como estruturar corretamente as **pastas e arquivos do projeto**

🎯 Pronto para começar a **Semana 2**, com validações, tratamento de erros e testes automatizados?

Se quiser, posso montar a próxima semana ou um desafio prático com TypeORM e banco de dados! Deseja continuar?

se eu não me engano nós não implementamos a rota patch

Você está absolutamente certo! Até o **Dia 6**, **a rota PATCH /tarefas/:id/status ainda não havia sido implementada**. Ela foi proposta apenas no **Dia 7** como parte do exercício prático.

Mas não se preocupe: abaixo está a **implementação completa da rota PATCH para atualizar o status da tarefa**, com todos os elementos envolvidos:

**✅ Rota PATCH /tarefas/:id/status**

Essa rota atualiza o status de uma tarefa específica.

**1️⃣ Atualizar o TarefasService**

Adicione o método atualizarStatus():

// tarefas.service.ts

import { TarefaStatus } from './enums/tarefa-status.enum';

atualizarStatus(id: number, novoStatus: TarefaStatus): Tarefa {

const tarefa = this.getPorId(id);

tarefa.status = novoStatus;

return tarefa;

}

**2️⃣ Atualizar o TarefasController**

Adicione a rota PATCH:

// tarefas.controller.ts

import { Patch, Body } from '@nestjs/common';

import { TarefaStatus } from './enums/tarefa-status.enum';

@Patch(':id/status')

atualizarStatus(

@Param('id') id: string,

@Body('status') status: TarefaStatus,

): Tarefa {

return this.tarefasService.atualizarStatus(Number(id), status);

}

**3️⃣ Testar no Postman**

**Requisição**:

PATCH http://localhost:3000/tarefas/123456789/status

**Corpo JSON**:

{

"status": "FINALIZADA"

}

✅ **Resposta esperada**:

{

"id": 123456789,

"titulo": "Estudar para a prova",

"descricao": "Focar nos exercícios de Nest.js",

"status": "FINALIZADA"

}

Se quiser deixar mais robusto, posso te mostrar como **validar o status com um pipe customizado**, para garantir que o valor enviado seja um dos definidos no enum. Deseja isso agora ou preferimos seguir com o início da Semana 2?

Com certeza! Aqui está a aula completa e detalhada para o **📘 Dia 8 – Pipes e Validação de Dados no Nest.js**, estruturada em Markdown e seguindo o estilo dos dias anteriores.

**📘 Dia 8 – Pipes e Validação de Dados**

**📚 Conteúdo Teórico**

**✅ O que são Pipes no Nest.js?**

* **Pipes** são classes responsáveis por:
  + Transformar dados de entrada (ex: converter tipos)
  + Validar dados antes que cheguem ao controller
* Funcionam junto com os decoradores @Body(), @Param(), @Query(), etc.

No caso de DTOs, os pipes funcionam junto com bibliotecas como class-validator e class-transformer.

**✅ Validação com class-validator + class-transformer**

* **class-validator**: fornece decorators como @IsString(), @IsNotEmpty(), @IsEnum(), etc.
* **class-transformer**: converte os dados de entrada em instâncias da classe DTO (necessário para que a validação funcione)

**🔧 Atividades Práticas**

**1️⃣ Instalar os pacotes**

npm install class-validator class-transformer

**2️⃣ Habilitar o pipe global de validação**

No main.ts, adicione o **ValidationPipe** global:

// src/main.ts

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

async function bootstrap() {

const app = await NestFactory.create(AppModule);

app.useGlobalPipes(new ValidationPipe());

await app.listen(3000);

}

bootstrap();

**3️⃣ Atualizar o DTO de criação de tarefa**

📁 src/tarefas/dto/create-tarefa.dto.ts

import { IsNotEmpty } from 'class-validator';

export class CreateTarefaDto {

@IsNotEmpty({ message: 'O título não pode estar vazio.' })

titulo: string;

@IsNotEmpty({ message: 'A descrição não pode estar vazia.' })

descricao: string;

}

Agora, se o cliente enviar POST /tarefas com campos vazios, ele receberá uma mensagem de erro automaticamente.

**4️⃣ Criar o FilterTarefasDto com validação**

📁 src/tarefas/dto/filter-tarefas.dto.ts

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

import { TarefaStatus } from '../enums/tarefa-status.enum';

export class FilterTarefasDto {

@IsOptional()

@IsEnum(TarefaStatus, {

message: 'Status inválido. Use ABERTA, EM\_ANDAMENTO ou FINALIZADA',

})

status?: TarefaStatus;

@IsOptional()

@IsString()

termo?: string;

}

Esse DTO poderá ser usado para filtros com query params, como:

GET /tarefas?status=ABERTA&termo=estudar

**5️⃣ Atualizar o controller (para preparar uso do filtro)**

📁 src/tarefas/tarefas.controller.ts

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

import { FilterTarefasDto } from './dto/filter-tarefas.dto';

@Get()

getTarefas(@Query() filtroDto: FilterTarefasDto): Tarefa[] {

if (Object.keys(filtroDto).length) {

return this.tarefasService.filtrarTarefas(filtroDto);

}

return this.tarefasService.getTodas();

}

**(Opcional) Método no service:**

filtrarTarefas(filtroDto: FilterTarefasDto): Tarefa[] {

const { status, termo } = filtroDto;

let tarefasFiltradas = this.tarefas;

if (status) {

tarefasFiltradas = tarefasFiltradas.filter(t => t.status === status);

}

if (termo) {

tarefasFiltradas = tarefasFiltradas.filter(

t => t.titulo.includes(termo) || t.descricao.includes(termo),

);

}

return tarefasFiltradas;

}

**🧪 Exercício Final**

1. Testar POST /tarefas com campos vazios:
2. {
3. "titulo": "",
4. "descricao": ""
5. }

✔️ Esperado: erro 400 com mensagens personalizadas.

1. Testar GET /tarefas?status=INVALIDO  
   ✔️ Esperado: erro 400 com mensagem de status inválido.
2. Testar GET /tarefas?status=EM\_ANDAMENTO&termo=prova  
   ✔️ Esperado: retorno filtrado por status e termo no título/descrição.

**✅ O que você aprendeu hoje:**

✔ O que são e para que servem **pipes** no Nest.js  
✔ Como validar dados com class-validator e class-transformer  
✔ Como criar **DTOs com validação automática**  
✔ Como usar @Query() com validação para filtros avançados  
✔ Como aplicar **boas mensagens de erro** para entrada de dados inválidos

🎯 Pronto para o **📘 Dia 9 – Tratamento de erros com Exceptions**?

Se quiser, posso montar a próxima aula para mostrar como personalizar mensagens de erro e lançar exceções com NotFoundException, BadRequestException, etc. Deseja que eu monte agora?

Ótima pergunta! Vamos resolver isso de forma simples e clara:

Você já tem esse trecho no seu controller atual:

// GET /tarefas

@Get()

getTodasTarefas() {

return this.tarefasService.getTodasTarefas();

}

Agora, você **vai substituí-lo completamente** pelo novo método que usa filtros opcionais com @Query() e FilterTarefasDto.

**✅ Substituição completa**

**Substitua isso:**

// GET /tarefas

@Get()

getTodasTarefas() {

return this.tarefasService.getTodasTarefas();

}

**Por isso:**

// GET /tarefas?status=ABERTA&termo=algo

@Get()

getTarefas(@Query() filtroDto: FilterTarefasDto): Tarefa[] {

if (Object.keys(filtroDto).length) {

return this.tarefasService.filtrarTarefas(filtroDto);

}

return this.tarefasService.getTodasTarefas();

}

**✅ Outras mudanças necessárias**

1. **No topo do controller**, adicione esse import:

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

import { FilterTarefasDto } from './dto/filter-tarefas.dto';

1. **No service**, você já deve ter implementado:

filtrarTarefas(filtroDto: FilterTarefasDto): Tarefa[] {

const { status, termo } = filtroDto;

let tarefasFiltradas = this.tarefas;

if (status) {

tarefasFiltradas = tarefasFiltradas.filter(t => t.status === status);

}

if (termo) {

tarefasFiltradas = tarefasFiltradas.filter(t =>

t.titulo.includes(termo) || t.descricao.includes(termo),

);

}

return tarefasFiltradas;

}

**✅ Exemplo de uso no Postman**

GET http://localhost:3000/tarefas?status=ABERTA&termo=prova

Se quiser, posso revisar seu service e DTO para garantir que tudo esteja 100% pronto para funcionar com filtros. Deseja isso?

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

import { Tarefa } from './tarefa.model';

import { TarefaStatus } from './enums/tarefa-status.enum';

import { CreateTarefaDto } from './dto/create-tarefa.dto';

@Injectable()

export class TarefasService {

private tarefas: Tarefa[] = [];

getTodasTarefas(): Tarefa[] {

return this.tarefas;

}

getTarefaPorId(id: number): Tarefa {

const tarefa = this.tarefas.find((tarefa) => tarefa.id === id);

if (!tarefa) {

throw new Error(`Tarefa com id ${id} não encontrada.`);

}

return tarefa;

}

getTarefasPorStatus(status: string): Tarefa[] {

return this.tarefas.filter((t) => t.status === status.toUpperCase());

}

createTarefa(dto: CreateTarefaDto): Tarefa {

const { titulo, descricao } = dto;

const tarefa: Tarefa = {

id: Date.now(), // simulando ID único

titulo,

descricao,

status: TarefaStatus.ABERTA,

};

this.tarefas.push(tarefa);

return tarefa;

}

filtrarTarefasPorStatus(status: string): Tarefa[] {

return this.tarefas.filter(

(tarefa) => tarefa.status === status.toUpperCase(),

);

}

atualizarStatus(id: number, novoStatus: TarefaStatus): Tarefa {

const tarefa = this.getTarefaPorId(id);

tarefa.status = novoStatus;

return tarefa;

}

}

meu service está assim no momento, mas voce me pedio para modifica lo com o codigo abaixo

filtrarTarefas(filtroDto: FilterTarefasDto): Tarefa[] {

const { status, termo } = filtroDto;

let tarefasFiltradas = this.tarefas;

if (status) {

tarefasFiltradas = tarefasFiltradas.filter(t => t.status === status);

}

if (termo) {

tarefasFiltradas = tarefasFiltradas.filter(

t => t.titulo.includes(termo) || t.descricao.includes(termo),

);

}

return tarefasFiltradas;

}

onde coloco ele ou onde modifico

Excelente! Vamos fazer essa integração da forma mais clara e segura possível.

**✅ Situação atual**

Seu serviço TarefasService já tem dois métodos relacionados a filtros por status:

getTarefasPorStatus(status: string): Tarefa[]

filtrarTarefasPorStatus(status: string): Tarefa[]

Além disso, você **ainda não tem** um método que combine **filtro por status + termo**, como o que eu sugeri.

**✅ O que você deve fazer agora**

**🔁 1. Substitua os dois métodos antigos por este novo:**

Remova:

getTarefasPorStatus(status: string): Tarefa[]

filtrarTarefasPorStatus(status: string): Tarefa[]

E adicione este novo método:

import { FilterTarefasDto } from './dto/filter-tarefas.dto';

filtrarTarefas(filtroDto: FilterTarefasDto): Tarefa[] {

const { status, termo } = filtroDto;

let tarefasFiltradas = this.tarefas;

if (status) {

tarefasFiltradas = tarefasFiltradas.filter(

(t) => t.status === status,

);

}

if (termo) {

tarefasFiltradas = tarefasFiltradas.filter(

(t) =>

t.titulo.toLowerCase().includes(termo.toLowerCase()) ||

t.descricao.toLowerCase().includes(termo.toLowerCase()),

);

}

return tarefasFiltradas;

}

**📁 2. Certifique-se de que o DTO FilterTarefasDto existe:**

📄 src/tarefas/dto/filter-tarefas.dto.ts

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

import { TarefaStatus } from '../enums/tarefa-status.enum';

export class FilterTarefasDto {

@IsOptional()

@IsEnum(TarefaStatus)

status?: TarefaStatus;

@IsOptional()

@IsString()

termo?: string;

}

**💡 3. Atualize seu controller:**

📄 tarefas.controller.ts

@Get()

getTarefas(@Query() filtroDto: FilterTarefasDto): Tarefa[] {

if (Object.keys(filtroDto).length) {

return this.tarefasService.filtrarTarefas(filtroDto);

}

return this.tarefasService.getTodasTarefas();

}

**✅ Resultado final**

Agora você tem **apenas um método de filtro**, bem mais poderoso, e pode buscar com:

GET /tarefas?status=ABERTA&termo=prova

Ou usar individualmente:

* /tarefas?status=FINALIZADA
* /tarefas?termo=estudar

Se quiser, posso te ajudar a mover a lógica de getTarefaPorId para lançar um NotFoundException em vez de Error, como vamos fazer na próxima aula de **tratamento de erros (Dia 9)**. Deseja isso agora?

// GET /tarefas/status/:status

@Get('status/:status')

getTarefasPorStatus(@Param('status') status: string): Tarefa[] {

return this.tarefasService.filtrarTarefas(status);

}

no controller nesse trecho do codigo estou com erro no parametro de filtrarTarefas(status) O tipo 'string' não tem propriedades em comum com o tipo 'FilterTarefasDto'.

o que devo fazer

Excelente observação — e o erro faz total sentido! Vamos esclarecer:

**❌ Problema**

Você está chamando o método filtrarTarefas(status) com um **string**, mas esse método **espera um objeto FilterTarefasDto**, ou seja:

filtrarTarefas(filtroDto: FilterTarefasDto): Tarefa[]

Mas você está passando assim:

this.tarefasService.filtrarTarefas(status) // ❌ incorreto

**✅ Solução**

Você precisa transformar o status (que vem da rota como string) em um **objeto FilterTarefasDto**. Assim:

// GET /tarefas/status/:status

@Get('status/:status')

getTarefasPorStatus(@Param('status') status: string): Tarefa[] {

return this.tarefasService.filtrarTarefas({ status });

}

✔ Isso funciona porque FilterTarefasDto permite campos opcionais (@IsOptional()), e você está passando só o status.

**🧠 Dica extra**

Se quiser que essa rota seja **opcional**, e todo o filtro seja tratado pela rota principal GET /tarefas?status=...&termo=..., você pode **remover essa rota** e manter apenas a versão com @Query().

**✅ Resumo**

* ✅ Corrigir o erro com: this.tarefasService.filtrarTarefas({ status })
* 🧼 Ou melhor ainda: **remova essa rota** e use só a rota com @Query() para manter seu código limpo e consistente.

Se quiser, posso revisar contigo todas as rotas atuais e sugerir uma estrutura RESTful mais organizada com base no que você já fez. Deseja isso?

O tipo 'string' não pode ser atribuído ao tipo 'TarefaStatus | undefined'.ts(2322)

filter-tarefas.dto.ts(10, 5): O tipo esperado vem da propriedade 'status', que é declarada aqui no tipo 'FilterTarefasDto'

agora veio essa menssagem de erro

Ótimo, isso mostra que **o TypeScript está fazendo seu trabalho bem**: protegendo você contra o uso de valores incorretos fora do tipo permitido no DTO.

**📌 O problema**

No DTO FilterTarefasDto, o campo status tem esse tipo:

status?: TarefaStatus;

Ou seja, ele **aceita apenas um valor do enum TarefaStatus ou undefined**.

Mas quando você faz:

this.tarefasService.filtrarTarefas({ status });

Esse status é uma **string vinda da URL**, como "aberta" ou "ABERTA", e **não é reconhecido diretamente como TarefaStatus** pelo TypeScript.

**✅ Soluções possíveis**

**✅ Opção 1 – Fazer cast e validação manual**

Transforme a string em TarefaStatus manualmente, validando antes:

import { TarefaStatus } from '../enums/tarefa-status.enum';

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

@Get('status/:status')

getTarefasPorStatus(@Param('status') status: string): Tarefa[] {

const upperStatus = status.toUpperCase() as TarefaStatus;

const statusValido = Object.values(TarefaStatus).includes(upperStatus);

if (!statusValido) {

throw new BadRequestException(`Status inválido: ${status}`);

}

return this.tarefasService.filtrarTarefas({ status: upperStatus });

}

**✅ Opção 2 – (Recomendada) Remover essa rota específica e centralizar tudo em GET /tarefas?status=...**

Essa abordagem é **mais limpa e RESTful**. Exemplo:

@Get()

getTarefas(@Query() filtroDto: FilterTarefasDto): Tarefa[] {

if (Object.keys(filtroDto).length) {

return this.tarefasService.filtrarTarefas(filtroDto);

}

return this.tarefasService.getTodasTarefas();

}

E o uso no Postman seria:

GET /tarefas?status=ABERTA

GET /tarefas?status=EM\_ANDAMENTO&termo=prova

**✅ Conclusão**

* O erro ocorre porque você está passando uma string onde o TypeScript espera um TarefaStatus.
* 🔧 Corrigir com cast e validação (opção 1), ou
* ✨ Preferir a abordagem com @Query() e FilterTarefasDto (opção 2), que é mais elegante e já cobre todos os filtros.

Se quiser, posso te ajudar a **criar um pipe de validação personalizado** que valida enums automaticamente. Deseja isso?

**# 📘 Dia 9 – Pipes Customizados**

---

**## 📚 Conteúdo Teórico**

**### ✅ O que são Pipes Customizados?**

Pipes customizados são classes que você cria para realizar **\*\*validação ou transformação personalizada\*\*** dos dados antes que eles sejam processados por controllers.

Eles são muito úteis para:

- Validar enums (como status)

- Transformar strings em tipos específicos

- Aplicar regras de negócio em parâmetros de rota, query ou body

---

**## 🎯 Objetivo do Dia**

Criar um **\*\*pipe de validação de status de tarefa\*\*** que:

- Aceita apenas valores válidos definidos no enum `TarefaStatus`

- Rejeita valores inválidos com uma exceção amigável (`BadRequestException`)

---

**## 🔧 Atividades Práticas**

**### 1️⃣ Criar o arquivo do Pipe**

📄 `src/tarefas/pipes/tarefa-status-validation.pipe.ts`

```ts

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

import { TarefaStatus } from '../enums/tarefa-status.enum';

export class TarefaStatusValidationPipe implements PipeTransform {

  readonly statusPermitidos = [

    TarefaStatus.ABERTA,

    TarefaStatus.EM\_ANDAMENTO,

    TarefaStatus.FINALIZADA,

  ];

  transform(value: any) {

    value = value.toUpperCase();

    if (!this.statusValido(value)) {

      throw new BadRequestException(`Status inválido: ${value}`);

    }

    return value;

  }

  private statusValido(status: any) {

    return this.statusPermitidos.includes(status);

  }

}

```

---

**### 2️⃣ Usar o Pipe no controller**

📄 `tarefas.controller.ts`

```ts

import { TarefaStatusValidationPipe } from './pipes/tarefa-status-validation.pipe';

@Patch(':id/status')

atualizarStatus(

  @Param('id') id: string,

  @Body('status', TarefaStatusValidationPipe) status: TarefaStatus,

): Tarefa {

  return this.tarefasService.atualizarStatus(Number(id), status);

}

```

---

**### 3️⃣ Testar com Postman**

**\*\*Rota:\*\***

```

PATCH /tarefas/123456789/status

```

**\*\*Corpo válido:\*\***

```json

{

  "status": "EM\_ANDAMENTO"

}

```

**\*\*Corpo inválido:\*\***

```json

{

  "status": "concluida"

}

```

🔴 Esperado: retorno 400 com mensagem:

```

{

  "statusCode": 400,

  "message": "Status inválido: CONCLUIDA",

  "error": "Bad Request"

}

```

---

**## ✅ O que você aprendeu hoje:**

✔ Como criar pipes customizados

✔ Como usar pipe para validar parâmetros (`@Body`, `@Param`)

✔ Como evitar códigos duplicados de validação

✔ Como lançar exceções personalizadas usando `BadRequestException`

---

🎯 Pronto para o 📘 Dia 10 – Tratamento de exceções com `HttpException` e filtros globais?

Se quiser, posso montar o próximo conteúdo!

@Get() getTarefas(@Query() filtroDto: FilterTarefasDto): Tarefa[] { if (Object.keys(filtroDto).length) { return this.tarefasService.filtrarTarefas(filtroDto); } return this.tarefasService.getTodas(); } nesse codigo de onde voce tira getTodas()

Boa pergunta! O método getTodas() chamado nesse trecho:

return this.tarefasService.getTodas();

… **precisa obrigatoriamente existir no seu serviço (TarefasService)**, senão o TypeScript (ou o Nest.js) vai apontar erro.

**✅ Solução**

Como pelo seu código anterior o nome do método no serviço é este:

getTodasTarefas(): Tarefa[] {

return this.tarefas;

}

Você tem **duas opções** para corrigir:

**✅ Opção 1 – Alterar o nome da chamada no controller:**

@Get()

getTarefas(@Query() filtroDto: FilterTarefasDto): Tarefa[] {

if (Object.keys(filtroDto).length) {

return this.tarefasService.filtrarTarefas(filtroDto);

}

return this.tarefasService.getTodasTarefas(); // <-- nome correto aqui

}

**✅ Opção 2 – Renomear o método no service para getTodas()**

// tarefas.service.ts

getTodas(): Tarefa[] {

return this.tarefas;

}

Isso deixaria o código mais curto, mas pode confundir se você já tem outras variações como getTodasTarefas().

**✍️ Recomendação**

Mantenha consistência. Se você está seguindo o padrão getTodasTarefas(), continue com ele e **corrija o nome no controller**, como mostrado na **opção 1**.

Se quiser, posso revisar toda sua estrutura de nomes e sugerir um padrão limpo para todos os métodos (como getTodasTarefas(), criarTarefa(), removerTarefa(), etc). Deseja isso?

quando criei tanto o controller quanto o service, alem de ser criado o arquivo tarefas.controller.ts e o tarefas.service.ts foram criados os arquivos tarefas.controller.spec.ts e tarefas.service.spec.ts o que são esses arquivos e para que servem

Ótima pergunta! Esses arquivos **.spec.ts** que foram gerados automaticamente são arquivos de **testes unitários**, criados pelo Nest.js para você **testar seu código** de forma automatizada.

**✅ O que são os arquivos .spec.ts?**

* .spec.ts é uma **convenção para arquivos de teste** no Nest.js (e no mundo do TypeScript/Jest).
* Esses arquivos usam o framework de testes **Jest** (instalado por padrão em projetos Nest.js) para **testar as funcionalidades dos seus services e controllers**.
* spec vem de "specification", ou seja, **especificação do comportamento esperado**.

**📦 Arquivos criados**

**🔹 tarefas.controller.spec.ts**

Usado para testar as rotas (endpoints) do controller, por exemplo:

* Se GET /tarefas retorna a lista correta
* Se POST /tarefas cria uma nova tarefa corretamente
* Se PATCH /tarefas/:id/status atualiza o status corretamente

**🔹 tarefas.service.spec.ts**

Usado para testar a lógica da sua camada de serviço (service), por exemplo:

* Se o método getTarefaPorId retorna a tarefa correta
* Se filtrarTarefas retorna corretamente os dados filtrados
* Se atualizarStatus modifica o status como esperado

**🧪 Por que isso é útil?**

* Testes **ajudam a garantir que seu código funciona como esperado**
* Facilitam **refatorações futuras** sem medo de quebrar algo
* Ajudam a prevenir bugs em produção
* São essenciais em projetos grandes e profissionais

**⚙️ Exemplo simples de um .spec.ts**

describe('TarefasService', () => {

let service: TarefasService;

beforeEach(() => {

service = new TarefasService();

});

it('deve retornar todas as tarefas', () => {

expect(service.getTodasTarefas()).toEqual([]);

});

});

Esse teste verifica se ao instanciar o service, a lista de tarefas começa vazia.

**❓Preciso me preocupar com isso agora?**

Se você **está começando**, **não precisa focar nos .spec.ts ainda**. Você pode deixá-los de lado até dominar bem o Nest.js.

👉 Mas quando for avançar para testes automatizados, esses arquivos já estarão prontos como base.

Se quiser, posso te mostrar como **executar os testes .spec.ts com Jest** no terminal ou como **escrever seu primeiro teste real de verdade**. Deseja isso?

**# 📘 Dia 10 – Exception Filters (Tratamento de Erros)**

---

**## 📚 Conteúdo Teórico**

**### ✅ O que são Exception Filters no Nest.js?**

Exception Filters são mecanismos do Nest.js para **\*\*tratar erros de forma centralizada e elegante\*\***.

Você pode:

- Usar exceções pré-definidas como `NotFoundException`, `BadRequestException`, etc.

- Criar filtros globais customizados para lidar com erros de forma padronizada.

---

**### ✅ Exceções mais comuns no Nest.js**

| Classe de Erro         | Código HTTP | Quando usar                            |

|------------------------|-------------|----------------------------------------|

| `NotFoundException`    | 404         | Quando algo não é encontrado           |

| `BadRequestException`  | 400         | Dados inválidos enviados pelo cliente  |

| `UnauthorizedException`| 401         | Acesso não autorizado                  |

| `ForbiddenException`   | 403         | Acesso proibido                        |

---

**## 🔧 Atividades Práticas**

**### 1️⃣ Atualizar método `getTarefaPorId` com exceção personalizada**

📄 `tarefas.service.ts`

Substitua:

```ts

if (!tarefa) {

  throw new Error(\`Tarefa com id \${id} não encontrada.\`);

}

```

Por:

```ts

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

if (!tarefa) {

  throw new NotFoundException(\`Tarefa com ID "\${id}" não encontrada\`);

}

```

---

**### 2️⃣ Repetir para `deleteTarefa` (caso exista)**

Exemplo:

```ts

deleteTarefa(id: number): void {

  const tarefa = this.getTarefaPorId(id);

  this.tarefas = this.tarefas.filter(t => t.id !== tarefa.id);

}

```

Se o `getTarefaPorId` já lança exceção, o método `deleteTarefa()` herdará esse comportamento.

---

**### 3️⃣ Repetir para `atualizarStatus`**

📄 `tarefas.service.ts`

```ts

atualizarStatus(id: number, novoStatus: TarefaStatus): Tarefa {

  const tarefa = this.getTarefaPorId(id); // já valida

  tarefa.status = novoStatus;

  return tarefa;

}

```

---

**## 🧪 Exercício Final**

1. ✅ Garantir que `GET /tarefas/:id` lança `404` se a tarefa não existir

2. ✅ Garantir que `DELETE /tarefas/:id` lança `404` se não for encontrada

3. ✅ Garantir que `PATCH /tarefas/:id/status` lança `404` se o ID for inválido

---

**## ✅ O que você aprendeu hoje:**

✔ Como lançar exceções com `NotFoundException`

✔ Como substituir `throw new Error(...)` por exceções do Nest.js

✔ Como garantir respostas HTTP padronizadas em caso de erro

✔ Como melhorar a clareza e segurança da sua API com tratamento de erros

---

🎯 Pronto para o 📘 Dia 11 – Trabalhar com dados persistentes usando TypeORM?

Se quiser, posso montar o próximo conteúdo para conectar sua API com um banco de dados!

@Controller('tarefas') // prefixo: todas as rotas começam com /tarefas

export class TarefasController {

constructor(private readonly tarefasService: TarefasService) {}

// GET /tarefas?status=ABERTA&termo=algo

@Get()

getTarefas(@Query() filtroDto: FilterTarefasDto): Tarefa[] {

if (Object.keys(filtroDto).length) {

return this.tarefasService.filtrarTarefas(filtroDto);

}

return this.tarefasService.getTodasTarefas();

}

// GET /tarefas/:id

@Get(':id')

getTarefaPorId(@Param('id') id: string): Tarefa {

return this.tarefasService.getTarefaPorId(parseInt(id));

}

@Post()

createTarefa(@Body() createTarefaDto: CreateTarefaDto): Tarefa {

return this.tarefasService.createTarefa(createTarefaDto);

}

@Patch(':id/status')

atualizarStatus(

@Param('id') id: string,

@Body('status', TarefaStatusValidationPipe) status: TarefaStatus,

): Tarefa {

return this.tarefasService.atualizarStatus(Number(id), status);

}

}

meu controller é esse, mas não tenho a rota delete, poderia criar-la e tambem o serviço de delete

Perfeito! Vamos adicionar o endpoint DELETE /tarefas/:id ao seu controller e criar o respectivo método no service.

**✅ 1. Criar o método no service**

Abra o arquivo tarefas.service.ts e adicione:

deleteTarefa(id: number): void {

const tarefa = this.getTarefaPorId(id); // lança NotFoundException se não existir

this.tarefas = this.tarefas.filter(t => t.id !== tarefa.id);

}

Isso reutiliza o método getTarefaPorId para validar a existência da tarefa.

**✅ 2. Criar o endpoint no controller**

Adicione ao seu TarefasController:

@Delete(':id')

deleteTarefa(@Param('id') id: string): void {

this.tarefasService.deleteTarefa(Number(id));

}

Isso define a rota: DELETE /tarefas/:id

**📦 Resultado final da nova rota**

**Endpoint:**

DELETE /tarefas/123456789

**Respostas:**

* ✅ Se a tarefa existir: retorna 204 No Content
* ❌ Se a tarefa **não existir**, você já lançou o erro com:

throw new NotFoundException(`Tarefa com ID "${id}" não encontrada`);

Se quiser, posso atualizar essa rota para retornar um status HTTP apropriado como 204 No Content, ou até adicionar um pequeno feedback com return { message: 'Tarefa removida com sucesso' }.

Deseja isso?

**# 📘 Dia 11 – Middleware**

---

**## 📚 Conteúdo Teórico**

**### ✅ O que é Middleware no Nest.js?**

Middlewares são funções executadas **\*\*antes que o request chegue ao controller\*\***. Eles são úteis para:

- Registrar logs

- Verificar autenticação

- Manipular/transformar dados do request

---

**### 🔁 Diferenças entre Middleware, Guards e Interceptors**

| Conceito     | Executa em...       | Finalidade principal                   |

|--------------|---------------------|----------------------------------------|

| Middleware   | Antes do controller | Pré-processamento, logging             |

| Guard        | Antes do controller | Controle de acesso (autorização)       |

| Interceptor  | Antes e depois      | Modificar request/response, métricas   |

---

**## 🔧 Atividades Práticas**

**### 1️⃣ Gerar middleware**

```bash

nest g middleware logger

```

Cria o arquivo: `src/logger.middleware.ts`

---

**### 2️⃣ Implementar o middleware de log**

📄 `logger.middleware.ts`

```ts

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) {

    const { method, originalUrl } = req;

    const timestamp = new Date().toISOString();

    console.log(`[${timestamp}] ${method} ${originalUrl}`);

    next(); // passa para o próximo middleware ou controller

  }

}

```

---

**### 3️⃣ Aplicar o middleware nas rotas de tarefas**

📄 `app.module.ts`

Importe o módulo:

```ts

import { MiddlewareConsumer, Module, NestModule } from '@nestjs/common';

import { TarefasModule } from './tarefas/tarefas.module';

import { LoggerMiddleware } from './logger.middleware';

@Module({

  imports: [TarefasModule],

})

export class AppModule implements NestModule {

  configure(consumer: MiddlewareConsumer) {

    consumer

      .apply(LoggerMiddleware)

      .forRoutes('tarefas'); // aplica somente às rotas de /tarefas

  }

}

```

---

**## 🧪 Exercício**

✔ Adicione um log personalizado com:

- Timestamp (`new Date().toISOString()`)

- Método HTTP (`GET`, `POST`, etc.)

- URL acessada (`originalUrl`)

---

**## ✅ O que você aprendeu hoje:**

✔ O que são e para que servem os middlewares

✔ Como criar middlewares com o Nest CLI

✔ Como aplicar middlewares globalmente ou em rotas específicas

✔ Como registrar logs com timestamp e rota acessada

---

🎯 Pronto para o 📘 Dia 12 – Guards de Autenticação com JWT?

Se quiser, posso montar o próximo conteúdo!

@Injectable()

export class LoggerMiddleware implements NestMiddleware {

use(req: Request, res: Response, next: NextFunction) {

const { method, originalUrl } = req;

const timestamp = new Date().toISOString();

console.log(`[${timestamp}] ${method} ${originalUrl}`);

next(); // passa para o próximo middleware ou controller

}

}

e essa mudança abaixo

export class AppModule implements NestModule {

configure(consumer: MiddlewareConsumer) {

consumer

.apply(LoggerMiddleware)

.forRoutes('tarefas'); // aplica somente às rotas de /tarefas

}

}

é o mesmo que está pedindo no exercicio abaixo

🧪 Exercício

✔ Adicione um log personalizado com:

- Timestamp (`new Date().toISOString()`)

- Método HTTP (`GET`, `POST`, etc.)

- URL acessada (`originalUrl`)

Sim, exatamente! ✅

O trecho de código que você implementou no LoggerMiddleware:

@Injectable()

export class LoggerMiddleware implements NestMiddleware {

use(req: Request, res: Response, next: NextFunction) {

const { method, originalUrl } = req;

const timestamp = new Date().toISOString();

console.log(`[${timestamp}] ${method} ${originalUrl}`);

next(); // passa para o próximo middleware ou controller

}

}

E o trecho de configuração no AppModule:

export class AppModule implements NestModule {

configure(consumer: MiddlewareConsumer) {

consumer

.apply(LoggerMiddleware)

.forRoutes('tarefas'); // aplica somente às rotas de /tarefas

}

}

✅ **Correspondem exatamente ao que o exercício do Dia 11 propõe:**

**✔️ Requisitos do exercício:**

* **Adicionar log com timestamp:** new Date().toISOString()
* **Capturar método HTTP:** GET, POST, etc. → req.method
* **Capturar a URL acessada:** → req.originalUrl
* **Aplicar apenas nas rotas de tarefas**

🔎 **Conclusão:**  
Você **cumpriu totalmente** o exercício prático do Dia 11. Muito bem!

Se quiser, podemos seguir para o **Dia 12 – Autenticação com JWT**, onde você aprenderá sobre guards e tokens de autenticação. Deseja isso agora?

**# 📘 Dia 12 – Guards (Autenticação Básica)**

---

**## 📚 Conteúdo Teórico**

**### ✅ O que são Guards no Nest.js?**

Guards são mecanismos de segurança que **\*\*controlam se uma requisição pode continuar\*\*** até o controller ou não.

Eles são muito usados para:

- Verificar autenticação (JWT, tokens, etc)

- Checar permissões (roles)

- Restringir acesso por lógica personalizada

---

**## 🧱 Estrutura básica de um Guard**

```ts

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

@Injectable()

export class MeuGuard implements CanActivate {

  canActivate(context: ExecutionContext): boolean {

    // lógica de permissão

    return true;

  }

}

```

---

**## 🔧 Atividades Práticas**

**### 1️⃣ Criar um Guard chamado `AuthGuard`**

```bash

nest g guard auth

```

---

**### 2️⃣ Implementar o guard de autenticação simples**

📄 `auth.guard.ts`

```ts

import {

  CanActivate,

  ExecutionContext,

  Injectable,

  UnauthorizedException,

} from '@nestjs/common';

import { Request } from 'express';

@Injectable()

export class AuthGuard implements CanActivate {

  canActivate(context: ExecutionContext): boolean {

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

    const authHeader = request.headers['authorization'];

    if (authHeader === 'meutoken123') {

      return true;

    }

    throw new UnauthorizedException('Token inválido ou ausente');

  }

}

```

---

**### 3️⃣ Usar o guard na rota protegida**

📄 `tarefas.controller.ts`

```ts

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

import { AuthGuard } from '../auth.guard'; // ou o caminho correspondente

@Post()

@UseGuards(AuthGuard)

createTarefa(@Body() createTarefaDto: CreateTarefaDto): Tarefa {

  return this.tarefasService.createTarefa(createTarefaDto);

}

```

---

**## 🧪 Exercício**

✔ Aplique o guard apenas no método `POST /tarefas`

✔ Teste com o Postman ou Insomnia:

- Enviar sem header `authorization`: ❌ recebe erro 401

- Enviar com `authorization: meutoken123`: ✅ tarefa criada com sucesso

---

**## ✅ O que você aprendeu hoje:**

✔ Como criar guards no Nest.js

✔ Como interceptar requisições antes do controller

✔ Como fazer uma autenticação simples com token hardcoded

✔ Como proteger rotas usando `@UseGuards()`

---

🎯 Em breve: 📘 Dia 13 – Autenticação com JWT real (token dinâmico e login)!

✔ Teste com o Postman ou Insomnia:

- Enviar sem header `authorization`: ❌ recebe erro 401

- Enviar com `authorization: meutoken123`: ✅ tarefa criada com sucesso

não entendi direito como testar no postman

Ótimo! Vamos passo a passo para te mostrar **como testar o AuthGuard com o Postman**. Isso vai garantir que só quem enviar o **token correto** no cabeçalho Authorization consiga criar uma tarefa.

**✅ Pré-requisitos**

Você já tem no seu projeto:

* Um AuthGuard que bloqueia acesso se o token for diferente de meutoken123
* O método POST /tarefas protegido com @UseGuards(AuthGuard)

**🧪 Como testar no Postman**

**📌 1. Abra o Postman e crie uma nova requisição**

* **Método:** POST
* **URL:** http://localhost:3000/tarefas  
  (ou a porta onde seu servidor Nest está rodando)

**📌 2. Vá até a aba Headers**

* Clique na aba Headers
* Adicione um novo header:

| **Key** | **Value** |
| --- | --- |
| Authorization | meutoken123 |

⚠️ Esse valor meutoken123 precisa ser **exatamente** o mesmo usado no AuthGuard.

**📌 3. Vá até a aba Body**

* Selecione a opção raw
* No seletor ao lado, escolha JSON
* Adicione o corpo da requisição com os dados da tarefa:

{

"titulo": "Estudar Nest.js",

"descricao": "Estudar middleware e guards"

}

**🧪 Teste 1: Sem o cabeçalho Authorization**

* Remova o header
* Clique em Send

📉 Resultado esperado:

{

"statusCode": 401,

"message": "Token inválido ou ausente",

"error": "Unauthorized"

}

**🧪 Teste 2: Com o header correto**

* Volte para a aba Headers
* Adicione: Authorization: meutoken123
* Clique em Send

📈 Resultado esperado:

* A tarefa será criada com sucesso
* Você verá a resposta contendo o objeto da tarefa criada

**🛠️ Dica Extra (Debug)**

Se quiser ver no terminal do servidor quais headers estão sendo enviados, adicione um console.log() no seu AuthGuard:

console.log('Header recebido:', request.headers);

Se quiser, posso montar um print ou vídeo passo a passo, ou preparar o **Dia 13 – JWT com login real**. Deseja seguir?

Claro! Aqui está a aula do **📘 Dia 13 – Configuração com .env e instalação de TypeORM**, detalhada e estruturada para você seguir passo a passo.

**📘 Dia 13 – Configuração com .env e Instalação do TypeORM**

**📚 Conteúdo Teórico**

**✅ Por que usar arquivos .env?**

O arquivo .env permite centralizar **credenciais e configurações sensíveis** do projeto, como:

* Dados do banco de dados
* Chaves de API
* URLs de ambientes

Isso torna sua aplicação mais segura e fácil de configurar em diferentes ambientes (desenvolvimento, produção, etc).

**✅ O que é o pacote @nestjs/config?**

É o pacote oficial do NestJS para ler e gerenciar variáveis de ambiente definidas no .env.

**✅ O que é o TypeORM?**

O **TypeORM** é um ORM (Object-Relational Mapper) compatível com NestJS que facilita a conexão com bancos de dados relacionais como PostgreSQL, MySQL etc.

**🔧 Atividades Práticas**

**1️⃣ Instalar os pacotes necessários**

npm install --save @nestjs/config

npm install --save @nestjs/typeorm typeorm pg

pg é o driver para PostgreSQL. Substitua por mysql2 se for usar MySQL.

**2️⃣ Criar o arquivo .env na raiz do projeto**

📄 .env

DB\_HOST=localhost

DB\_PORT=5432

DB\_USERNAME=usuario

DB\_PASSWORD=senha

DB\_NAME=nest\_tarefas

**3️⃣ Configurar @nestjs/config e TypeOrmModule em app.module.ts**

📄 src/app.module.ts

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

import { ConfigModule, ConfigService } from '@nestjs/config';

import { TypeOrmModule } from '@nestjs/typeorm';

import { TarefasModule } from './tarefas/tarefas.module';

@Module({

imports: [

ConfigModule.forRoot({

isGlobal: true, // Torna disponível em toda a aplicação

}),

TypeOrmModule.forRootAsync({

imports: [ConfigModule],

inject: [ConfigService],

useFactory: (configService: ConfigService) => ({

type: 'postgres',

host: configService.get('DB\_HOST'),

port: configService.get<number>('DB\_PORT'),

username: configService.get('DB\_USERNAME'),

password: configService.get('DB\_PASSWORD'),

database: configService.get('DB\_NAME'),

autoLoadEntities: true, // carrega entidades automaticamente

synchronize: true, // ⚠️ usar apenas em desenvolvimento!

}),

}),

TarefasModule,

],

})

export class AppModule {}

**🧪 Exercício**

✅ **Objetivo:** Simular a conexão com o banco, mesmo sem ter entidades criadas ainda.

**Como testar:**

1. Certifique-se que seu PostgreSQL está rodando com os dados do .env
2. Execute o comando:

npm run start:dev

**Esperado no terminal:**

[Nest] ... Successfully connected to the database

**✅ O que você aprendeu hoje:**

✔ Como configurar variáveis de ambiente com .env  
✔ Como usar o @nestjs/config para acessar essas variáveis  
✔ Como instalar e configurar o TypeORM para PostgreSQL  
✔ Como garantir que sua aplicação conecta ao banco com sucesso

🎯 **Próximo passo:**  
📘 Dia 14 – Criando entidades com TypeORM e salvando tarefas no banco.

Se quiser, posso preparar a aula do Dia 14 para você. Deseja isso agora?

depois que rodei o comando npm run start:dev recebi o erro abaixo

[Nest] 113720 - 08/07/2025, 00:18:03 ERROR [TypeOrmModule] Unable to connect to the database. Retrying (6)...

AggregateError [ECONNREFUSED]:

at internalConnectMultiple (node:net:1139:18)

at afterConnectMultiple (node:net:1712:7)

Esse erro:

ERROR [TypeOrmModule] Unable to connect to the database. Retrying...

AggregateError [ECONNREFUSED]

significa que o Nest.js **não conseguiu se conectar ao banco de dados PostgreSQL**.

Vamos resolver isso passo a passo:

**✅ 1. Verifique se o PostgreSQL está rodando**

No terminal, digite:

sudo service postgresql status

ou no Windows (caso esteja usando o pgAdmin ou instalador oficial):

* Abra o **Gerenciador de Tarefas** → guia **Serviços**
* Procure por postgresql
* Veja se está **“Em execução”**

Se não estiver rodando, inicie com:

sudo service postgresql start

ou, no Windows, reinicie o serviço do PostgreSQL.

**✅ 2. Verifique seu arquivo .env**

Confirme se os dados batem com os do seu PostgreSQL local:

DB\_HOST=localhost

DB\_PORT=5432

DB\_USERNAME=usuario <- esse deve ser seu usuário do banco

DB\_PASSWORD=senha <- senha correta desse usuário

DB\_NAME=nest\_tarefas <- o banco de dados precisa existir!

Você pode testar no terminal:

psql -U usuario -d nest\_tarefas

Se der erro: psql: FATAL: database "nest\_tarefas" does not exist, crie o banco com:

createdb -U usuario nest\_tarefas

**✅ 3. Verifique o pg\_hba.conf (opcional avançado)**

Se estiver usando PostgreSQL local e mesmo com tudo certo ainda dá erro, talvez o PostgreSQL esteja configurado para **não aceitar conexões externas** ou **usuário sem senha**. Podemos revisar isso se necessário.

**✅ 4. Teste local com cliente gráfico**

Abra o **pgAdmin**, **DBeaver**, ou outro cliente e tente conectar com os dados do .env.

**✅ 5. Habilite logs de erro detalhados**

Você pode adicionar um console.log() no useFactory do TypeOrmModule para imprimir as variáveis:

useFactory: (configService: ConfigService) => {

console.log('Conectando com:', {

host: configService.get('DB\_HOST'),

user: configService.get('DB\_USERNAME'),

database: configService.get('DB\_NAME'),

});

return {

type: 'postgres',

host: configService.get('DB\_HOST'),

port: configService.get<number>('DB\_PORT'),

username: configService.get('DB\_USERNAME'),

password: configService.get('DB\_PASSWORD'),

database: configService.get('DB\_NAME'),

autoLoadEntities: true,

synchronize: true,

};

}

Se quiser me mostrar o conteúdo atual do seu .env (pode esconder os dados sensíveis), e confirmar se o banco realmente existe, posso te ajudar com um diagnóstico mais direto. Deseja isso?

como reinicia o serviço do PostgreSQL no Windows

Para reiniciar o serviço do **PostgreSQL no Windows**, siga um dos métodos abaixo:

**✅ Método 1: Via Serviços do Windows**

1. Pressione Win + R, digite:
2. services.msc

e pressione **Enter**.

1. Na lista de serviços, procure por algo como:
2. postgresql-x64-15

ou

PostgreSQL Server <versão>

1. Clique com o botão direito no serviço → clique em **"Reiniciar"**.

Se o botão "Reiniciar" estiver desativado, clique primeiro em **Parar** e depois em **Iniciar**.

**✅ Método 2: Via Prompt de Comando (cmd) como administrador**

1. Abra o Prompt de Comando como Administrador:
   * Clique em **Iniciar**, digite cmd
   * Clique com o botão direito e selecione **Executar como administrador**
2. Execute o comando abaixo (substitua pelo nome correto do serviço):

net stop postgresql-x64-15

net start postgresql-x64-15

⚠️ Substitua postgresql-x64-15 pelo nome exato do seu serviço. Para ver o nome correto, vá até o services.msc como no método 1.

**✅ Confirmar se está funcionando**

Abra o terminal e tente acessar com psql:

psql -U seu\_usuario -d nome\_do\_banco

Ou tente novamente executar o Nest.js:

npm run start:dev

Se o erro sumir, a conexão está resolvida ✅

Se quiser, posso te ajudar a identificar o nome correto do serviço PostgreSQL no seu computador. Deseja isso?

como ver a versão do postgre no terminal

Você pode ver a versão do **PostgreSQL** no terminal de várias formas. Abaixo estão os métodos mais comuns:

**✅ 1. Usando o comando psql**

Se o psql estiver instalado e no seu PATH:

psql --version

🔁 Resultado esperado:

psql (PostgreSQL) 15.3

**✅ 2. Dentro do console interativo do PostgreSQL**

Se você entrar no console psql, digite:

SELECT version();

🔁 Resultado esperado:

version

-----------------------------------------------------------------------------------------------------------

PostgreSQL 15.3 on x86\_64-pc-windows, compiled by Visual C++ build ..., 64-bit

Para entrar no psql:

psql -U seu\_usuario -d nome\_do\_banco

Substitua seu\_usuario e nome\_do\_banco pelos dados corretos.

**✅ 3. No Windows (via linha de comando)**

Se estiver usando Windows e instalou via instalador oficial, vá até:

"C:\Program Files\PostgreSQL\<versão>\bin\psql.exe" --version

Substitua <versão> pelo número instalado, como 15, 14, etc.

Se quiser, posso te ajudar a configurar o PATH do PostgreSQL no Windows para você poder rodar psql de qualquer lugar no terminal. Deseja isso?

me diz uma coiza, preciso baixar o postgreSQL na internet para o meu pc

Sim, para usar o PostgreSQL no seu PC, você precisa instalá-lo primeiro, sim — ele não vem instalado por padrão no Windows.

**Como fazer:**

1. **Baixar o instalador oficial**  
   Vá ao site oficial do PostgreSQL:  
   <https://www.postgresql.org/download/windows/>
2. **Escolher o instalador**  
   Normalmente você será direcionado para o instalador da EnterpriseDB, que é fácil de usar.
3. **Instalar**  
   Execute o instalador e siga os passos. Durante a instalação, você vai configurar:
   * Senha do usuário postgres (administrador do banco)
   * Porta (geralmente 5432)
   * Outros detalhes opcionais
4. **Usar**  
   Depois de instalado, você pode acessar o banco usando:
   * O cliente gráfico **pgAdmin** (vem junto)
   * Linha de comando com psql
   * Seu aplicativo Nest.js conectado via TypeORM

**Dica extra**

Se preferir algo mais simples para começar, pode usar serviços como:

* **Docker** (rodar PostgreSQL dentro de um container)
* **Postgres.app** (para Mac) — só para Mac, não Windows

Mas para Windows, o instalador oficial é o mais direto.

Se precisar, posso te ajudar a instalar passo a passo. Quer?

quai são os principais pacotes que preciso instalar do PostgreSQL

Ótima pergunta! 👇 Aqui estão os **principais pacotes que você precisa instalar** para trabalhar com PostgreSQL no seu PC (Windows), especialmente se for usá-lo com o NestJS + TypeORM:

**✅ 1. PostgreSQL Server**

* É o **núcleo do banco de dados**
* Responsável por armazenar e gerenciar os dados

📥 Você instala isso pelo instalador oficial:  
👉 <https://www.postgresql.org/download/windows/>

**✅ 2. pgAdmin (Interface Gráfica)**

* Ferramenta de interface web para gerenciar bancos, rodar queries e visualizar dados
* É opcional, mas **muito útil para iniciantes**

Vem incluído no instalador padrão da EnterpriseDB.

**✅ 3. psql (Client CLI - opcional, mas útil)**

* Ferramenta de terminal para se conectar ao banco com comandos SQL
* Vem junto com o servidor no instalador padrão

Exemplo de uso:

psql -U postgres -d nome\_do\_banco

**✅ 4. Driver do Node.js para PostgreSQL (no seu projeto)**

Este você instala **dentro do seu projeto NestJS**:

npm install --save pg

É o **driver que TypeORM usa** para conectar ao PostgreSQL.

**✅ 5. Bibliotecas de ORM (no seu projeto NestJS)**

Se estiver usando NestJS com TypeORM:

npm install --save @nestjs/typeorm typeorm

**✅ Resumo completo:**

| **O que você instala** | **Onde** | **Para que serve** |
| --- | --- | --- |
| PostgreSQL Server | Instalador oficial | Roda o banco de dados |
| pgAdmin | Instalador oficial | Interface gráfica web |
| psql | Instalador oficial | Linha de comando para SQL |
| pg | No projeto NestJS | Driver PostgreSQL para Node.js |
| @nestjs/typeorm | No projeto NestJS | Integração NestJS + TypeORM |
| typeorm | No projeto NestJS | ORM (mapeamento objeto-relacional) |

Se quiser, posso te passar um passo a passo visual para instalar o PostgreSQL corretamente no seu Windows. Deseja isso?

[Nest] 82236 - 08/07/2025, 01:01:13 ERROR [TypeOrmModule] Unable to connect to the database. Retrying (3)...

error: autentica��o do tipo senha falhou para o usu�rio "usuario"

at Parser.parseErrorMessage (C:\meus\_projetos\aulas-nodeJS-nestJS-typeORM\nestJS-aulas\primeira\_semana\projeto-tarefas\node\_modules\pg-protocol\src\parser.ts:369:69)

at Parser.handlePacket (C:\meus\_projetos\aulas-nodeJS-nestJS-typeORM\nestJS-aulas\primeira\_semana\projeto-tarefas\node\_modules\pg-protocol\src\parser.ts:187:21)

at Parser.parse (C:\meus\_projetos\aulas-nodeJS-nestJS-typeORM\nestJS-aulas\primeira\_semana\projeto-tarefas\node\_modules\pg-protocol\src\parser.ts:102:30)

at Socket.<anonymous> (C:\meus\_projetos\aulas-nodeJS-nestJS-typeORM\nestJS-aulas\primeira\_semana\projeto-tarefas\node\_modules\pg-protocol\src\index.ts:7:48)

at Socket.emit (node:events:518:28)

at addChunk (node:internal/streams/readable:561:12)

at readableAddChunkPushByteMode (node:internal/streams/readable:512:3)

at Readable.push (node:internal/streams/readable:392:5)

at TCP.onStreamRead (node:internal/stream\_base\_commons:189:23)

o que é esse erro

Esse erro indica que o NestJS **não conseguiu se conectar ao banco de dados PostgreSQL porque a autenticação falhou**. A mensagem principal está aqui:

❌ autenticação do tipo senha falhou para o usuário "usuario"

**🛠️ O que isso significa?**

Você informou um **usuário e senha incorretos** no seu .env, ou o banco não reconhece esse usuário.

**✅ Como resolver passo a passo**

**1. Verifique seu .env**

Confira se você usou algo assim:

DB\_HOST=localhost

DB\_PORT=5432

DB\_USERNAME=usuario ← ⚠️ Este precisa ser um usuário real do PostgreSQL

DB\_PASSWORD=senha ← ⚠️ Esta senha precisa estar correta

DB\_NAME=nest\_tarefas ← Esse banco precisa existir também

**2. Teste manualmente no terminal**

Abra o terminal (CMD, PowerShell ou Git Bash) e digite:

psql -U usuario -d nest\_tarefas

Se aparecer:

psql: FATAL: autenticação falhou para o usuário "usuario"

➡️ Significa que **o usuário ou a senha estão errados**.

**3. Veja os usuários existentes no PostgreSQL**

Se você tiver acesso ao psql com outro usuário, como o padrão postgres, tente:

psql -U postgres

Depois, rode:

\du

Isso mostra todos os usuários (roles) criados. Exemplo:

Role name | Attributes | Member of

-----------+------------------------------------------------------------+-----------

postgres | Superuser, Create role, Create DB, Replication, Bypass RLS | {}

usuario | | {}

Se o usuário usuario **não aparecer**, você pode criar com:

CREATE USER usuario WITH PASSWORD 'senha';

E dar acesso ao banco:

GRANT ALL PRIVILEGES ON DATABASE nest\_tarefas TO usuario;

**4. Alternativa rápida: use o usuário padrão postgres**

Você pode ajustar seu .env assim:

DB\_USERNAME=postgres

DB\_PASSWORD=sua\_senha\_definida\_na\_instalacao

E testar novamente com:

npm run start:dev

**✅ Conclusão**

Esse erro **não é de código**, e sim de **credenciais erradas no banco**. Para resolver:

* Use um usuário real que exista no PostgreSQL
* Verifique se a senha está correta
* Verifique se o banco nest\_tarefas existe

Se quiser, posso te guiar para criar o usuário certo e o banco passo a passo. Deseja isso?

como crio um banco de dados no pgadmin

Ótimo! Criar um banco de dados no **pgAdmin** é simples e visual. Siga este passo a passo:

**✅ Como criar um banco de dados no pgAdmin**

**1. Abrir o pgAdmin**

* Vá no menu Iniciar do Windows e abra o **pgAdmin 4**
* Aguarde carregar (pode demorar um pouco na primeira vez)

**2. Conectar ao servidor**

* No painel à esquerda, clique no servidor PostgreSQL (pode estar com o nome PostgreSQL 15, PostgreSQL 14, etc.)
* Será pedido a senha do usuário postgres (é a senha que você definiu durante a instalação do PostgreSQL)

**3. Criar novo banco de dados**

1. Clique com o **botão direito** em Databases → **Create** → **Database...**
2. Na janela que abrir:
   * **Database name:** nest\_tarefas (ou o nome que quiser)
   * **Owner:** deixe como postgres (ou selecione outro usuário, se quiser)
3. Clique em **Save**

✅ Pronto! O banco foi criado.

**4. (Opcional) Criar um novo usuário (role) com senha**

Se quiser usar um usuário diferente de postgres:

1. Vá em **Login/Group Roles** (na árvore à esquerda)
2. Clique com o direito → **Create** → **Login/Group Role...**

Na aba **General**:

* **Name:** usuario

Na aba **Definition**:

* Defina a **senha** que será usada no .env

Na aba **Privileges**:

* Marque **Can login?** como ✅

Clique em **Save**

Agora, dê acesso ao banco criado:

1. Vá em Databases > nest\_tarefas > Properties
2. Vá em **Privileges**
3. Clique em **+**, selecione o usuário usuario
4. Marque os privilégios (pelo menos CONNECT, TEMP, SELECT, INSERT, etc.)
5. Clique em **Save**

**5. Atualizar seu .env**

Se usou o usuário postgres:

DB\_USERNAME=postgres

DB\_PASSWORD=sua\_senha

Se criou o usuario:

DB\_USERNAME=usuario

DB\_PASSWORD=senha\_definida

**6. Testar no terminal**

npm run start:dev

Se configurado corretamente, a aplicação NestJS deve conectar ao banco com sucesso 🎉

Se quiser, posso te ajudar agora a criar uma tabela ou entidade com TypeORM dentro desse banco. Deseja continuar com isso?

DB\_HOST=localhost

DB\_PORT=5432

DB\_USERNAME=usuario

DB\_PASSWORD=484368

DB\_NAME=nest\_tarefas

onde defino o usuário

Excelente pergunta! 🙌

Você define o **usuário do banco** de duas formas:

**✅ 1. No seu projeto NestJS (arquivo .env)**

Você **já fez isso corretamente**:

DB\_USERNAME=usuario

DB\_PASSWORD=484368

Esse usuário (usuario) é o que o NestJS tentará usar para se conectar ao PostgreSQL via TypeORM.

**✅ 2. O usuário precisa existir dentro do PostgreSQL**

O problema mais comum é esse: o .env define o usuário, **mas ele ainda não foi criado no banco de dados**.

**Verificar se o usuário existe no PostgreSQL:**

Você pode abrir o **pgAdmin** e seguir estes passos:

1. No painel à esquerda, expanda:
2. Servers > PostgreSQL > Login/Group Roles
3. Veja se o usuário **usuario** está listado ali.

**❌ Se não estiver:**

Você precisa criar esse usuário.

**✅ Criando o usuário no pgAdmin:**

1. Clique com o botão direito em Login/Group Roles → **Create** → **Login/Group Role...**
2. Na aba **General**, digite:
   * **Name:** usuario
3. Na aba **Definition**:
   * **Password:** 484368
4. Na aba **Privileges**:
   * Marque a opção ✅ **Can login?**
5. Clique em **Save**

**✅ Dando acesso ao banco nest\_tarefas:**

1. Vá em:
2. Servers > PostgreSQL > Databases > nest\_tarefas
3. Clique com o botão direito → **Properties**
4. Vá até a aba **Privileges**
5. Clique no botão **+**
6. Escolha o usuário usuario e marque permissões como:
   * ✅ CONNECT
   * ✅ SELECT
   * ✅ INSERT
   * ✅ UPDATE
   * ✅ DELETE
7. Clique em **Save**

**✅ Depois disso:**

Execute novamente seu projeto NestJS:

npm run start:dev

Se tudo estiver certo, o erro de autenticação desaparecerá.

Se quiser, posso te ajudar com a criação da primeira **entidade com TypeORM** agora que seu banco está configurado. Deseja isso?

Excelente observação! 👀

Se a aplicação **iniciou com sucesso**, mas **não apareceu a mensagem**:

[Nest] ... Successfully connected to the database

... então provavelmente **você ainda não criou nenhuma entidade para o TypeORM carregar**, e por isso ele **não chega a tentar se conectar de fato ao banco**. Isso é comum quando usamos autoLoadEntities: true, mas ainda **não há entidades registradas**.

**✅ Como confirmar se o Nest.js se conectou ao banco**

**Opção 1 – Adicionar log manual**

Vá no seu AppModule onde está a configuração do TypeORM, e adicione um log dentro do useFactory:

TypeOrmModule.forRootAsync({

imports: [ConfigModule],

inject: [ConfigService],

useFactory: async (configService: ConfigService) => {

const dbConfig = {

type: 'postgres',

host: configService.get<string>('DB\_HOST'),

port: configService.get<number>('DB\_PORT'),

username: configService.get<string>('DB\_USERNAME'),

password: configService.get<string>('DB\_PASSWORD'),

database: configService.get<string>('DB\_NAME'),

autoLoadEntities: true,

synchronize: true,

};

console.log('[TypeORM] Configuração de conexão:', dbConfig);

return dbConfig;

},

}),

Esse log mostra se ele está **lendo corretamente o .env** e se está tentando conectar.

**Opção 2 – Criar uma entidade simples para forçar conexão**

Crie uma entidade só para teste:

**tarefa.entity.ts (em src/tarefas/):**

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

@Entity()

export class Tarefa {

@PrimaryGeneratedColumn()

id: number;

@Column()

titulo: string;

@Column()

descricao: string;

@Column()

status: string;

}

**Em tarefas.module.ts, adicione:**

import { TypeOrmModule } from '@nestjs/typeorm';

import { Tarefa } from './tarefa.entity';

@Module({

imports: [TypeOrmModule.forFeature([Tarefa])],

...

})

Agora ao iniciar com:

npm run start:dev

➡️ o TypeORM tentará se conectar e **criar a tabela no banco**, e você verá a mensagem:

[Nest] ... Successfully connected to the database

ou receberá algum erro (como falta de permissão ou senha errada).

**✅ Conclusão**

Seu app *provavelmente já está conectado*, só **não há nada ainda que force essa conexão acontecer**. Ao criar **a primeira entidade com o TypeORM**, a conexão será testada e a tabela criada automaticamente se synchronize: true.

Se quiser, posso te ajudar agora a criar a primeira entidade com TypeORM com base nas suas tarefas atuais. Quer seguir com isso?

ConfigModule.forRoot({

isGlobal: true, // Torna disponível em toda a aplicação

}),

TypeOrmModule.forRootAsync({

imports: [ConfigModule],

inject: [ConfigService],

useFactory: (configService: ConfigService) => ({

type: 'postgres',

host: configService.get('DB\_HOST'),

port: configService.get<number>('DB\_PORT'),

username: configService.get('DB\_USERNAME'),

password: configService.get('DB\_PASSWORD'),

database: configService.get('DB\_NAME'),

autoLoadEntities: true, // carrega entidades automaticamente

synchronize: true, // ⚠️ usar apenas em desenvolvimento!

}),

})

o meu appmodule está assim

Perfeito! Seu AppModule está **corretamente configurado** para usar o PostgreSQL com NestJS + TypeORM via .env. 👇 Vamos revisar ponto a ponto:

**✅ O que está funcionando no seu AppModule**

ConfigModule.forRoot({

isGlobal: true,

}),

* ✅ Torna as variáveis do .env acessíveis globalmente com ConfigService

TypeOrmModule.forRootAsync({

imports: [ConfigModule],

inject: [ConfigService],

useFactory: (configService: ConfigService) => ({

type: 'postgres',

host: configService.get('DB\_HOST'),

port: configService.get<number>('DB\_PORT'),

username: configService.get('DB\_USERNAME'),

password: configService.get('DB\_PASSWORD'),

database: configService.get('DB\_NAME'),

autoLoadEntities: true,

synchronize: true,

}),

}),

* ✅ Lê os dados do banco via .env
* ✅ Se autoLoadEntities: true, ele carregará automaticamente entidades registradas nos módulos
* ✅ synchronize: true cria/atualiza as tabelas automaticamente (apenas no ambiente de desenvolvimento)

**⚠️ Atenção**

Se **nenhuma entidade foi criada ainda**, o NestJS não chega a fazer a conexão de fato — mesmo com tudo configurado. Isso significa que você não verá nenhum erro nem sucesso de conexão, **porque ele não tem nada para sincronizar com o banco**.

**✅ O próximo passo recomendado:**

Crie **uma entidade simples**, como:

// src/tarefas/tarefa.entity.ts

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

@Entity()

export class Tarefa {

@PrimaryGeneratedColumn()

id: number;

@Column()

titulo: string;

@Column()

descricao: string;

@Column()

status: string;

}

Depois, registre no módulo:

// tarefas.module.ts

import { TypeOrmModule } from '@nestjs/typeorm';

import { Tarefa } from './tarefa.entity';

@Module({

imports: [TypeOrmModule.forFeature([Tarefa])],

...

})

Ao rodar novamente:

npm run start:dev

➡️ Agora o Nest tentará sincronizar a entidade com o banco e criará a tabela automaticamente.  
Se houver algo errado, o erro será exibido. Caso contrário, a tabela tarefa será criada 🎉

Se quiser, posso gerar a entidade Tarefa completa com enum, DTO e repositório para já começar a usar o banco real. Deseja isso?

[TypeOrmModule.forFeature([Tarefa])],

onde coloca isso

Ótima pergunta! 🎯

O trecho:

TypeOrmModule.forFeature([Tarefa])

deve ser colocado **dentro do imports do módulo onde a entidade será usada**, geralmente no tarefas.module.ts.

**✅ Exemplo completo: tarefas.module.ts**

Se você criou uma entidade chamada Tarefa, o módulo de tarefas (tarefas.module.ts) deve ficar assim:

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

import { TarefasController } from './tarefas.controller';

import { TarefasService } from './tarefas.service';

import { TypeOrmModule } from '@nestjs/typeorm';

import { Tarefa } from './tarefa.entity';

@Module({

imports: [TypeOrmModule.forFeature([Tarefa])],

controllers: [TarefasController],

providers: [TarefasService],

})

export class TarefasModule {}

**🔍 Por que isso é necessário?**

O TypeOrmModule.forFeature([Tarefa]):

* Registra a entidade Tarefa neste módulo.
* Permite que você **injete o repositório da entidade** usando o @InjectRepository(Tarefa) no serviço.

**✅ Próximo passo: usar o repositório**

No TarefasService, agora você pode usar o repositório assim:

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

import { InjectRepository } from '@nestjs/typeorm';

import { Repository } from 'typeorm';

import { Tarefa } from './tarefa.entity';

@Injectable()

export class TarefasService {

constructor(

@InjectRepository(Tarefa)

private readonly tarefaRepository: Repository<Tarefa>,

) {}

// exemplo de uso:

async getTodas(): Promise<Tarefa[]> {

return await this.tarefaRepository.find();

}

}

Se quiser, posso adaptar todo seu TarefasService e TarefasController para usar banco de dados com o TypeORM agora. Deseja isso?

[Nest] 12016 - 08/07/2025, 02:05:29 ERROR [TypeOrmModule] Unable to connect to the database. Retrying (8)...

QueryFailedError: permissão negada para esquema public

at PostgresQueryRunner.query (C:\meus\_projetos\aulas-nodeJS-nestJS-typeORM\nestJS-aulas\primeira\_semana\projeto-tarefas\node\_modules\typeorm\driver\src\driver\postgres\PostgresQueryRunner.ts:325:19)

at process.processTicksAndRejections (node:internal/process/task\_queues:105:5)

at async PostgresQueryRunner.executeQueries (C:\meus\_projetos\aulas-nodeJS-nestJS-typeORM\nestJS-aulas\primeira\_semana\projeto-tarefas\node\_modules\typeorm\query-runner\src\query-runner\BaseQueryRunner.ts:681:13)

at async PostgresQueryRunner.createTable (C:\meus\_projetos\aulas-nodeJS-nestJS-typeORM\nestJS-aulas\primeira\_semana\projeto-tarefas\node\_modules\typeorm\driver\src\driver\postgres\PostgresQueryRunner.ts:616:9)

at async RdbmsSchemaBuilder.createNewTables (C:\meus\_projetos\aulas-nodeJS-nestJS-typeORM\nestJS-aulas\primeira\_semana\projeto-tarefas\node\_modules\typeorm\schema-builder\src\schema-builder\RdbmsSchemaBuilder.ts:635:13)

at async RdbmsSchemaBuilder.executeSchemaSyncOperationsInProperOrder (C:\meus\_projetos\aulas-nodeJS-nestJS-typeORM\nestJS-aulas\primeira\_semana\projeto-tarefas\node\_modules\typeorm\schema-builder\src\schema-builder\RdbmsSchemaBuilder.ts:225:9)

at async RdbmsSchemaBuilder.build (C:\meus\_projetos\aulas-nodeJS-nestJS-typeORM\nestJS-aulas\primeira\_semana\projeto-tarefas\node\_modules\typeorm\schema-builder\src\schema-builder\RdbmsSchemaBuilder.ts:95:13)

at async DataSource.synchronize (C:\meus\_projetos\aulas-nodeJS-nestJS-typeORM\nestJS-aulas\primeira\_semana\projeto-tarefas\node\_modules\typeorm\data-source\src\data-source\DataSource.ts:340:9)

at async DataSource.initialize (C:\meus\_projetos\aulas-nodeJS-nestJS-typeORM\nestJS-aulas\primeira\_semana\projeto-tarefas\node\_modules\typeorm\data-source\src\data-source\DataSource.ts:278:43)

agora está vindo esse erro

Esse erro:

❌ permissão negada para esquema public

significa que o **usuário do banco de dados (usuario) não tem permissão para criar tabelas no schema public** do banco PostgreSQL.

**✅ Como resolver**

Você precisa **dar permissão de uso e criação no schema public** ao usuário usuario.

**🔧 Passo a passo no pgAdmin:**

1. Abra o **pgAdmin**
2. Vá em:
3. Servers > PostgreSQL > Databases > nest\_tarefas > Schemas > public
4. Clique com o **botão direito** em public → **Properties**
5. Vá até a aba **Privileges**
6. Clique no botão ➕ (para adicionar novo role)
7. Em **Grantee**, escolha o usuário usuario
8. Marque os privilégios:
   * ✅ USAGE
   * ✅ CREATE
9. Clique em **Save**

**✅ Alternativa via terminal (psql):**

Se preferir fazer via terminal, conecte-se com o superusuário (ex: postgres) e execute:

GRANT USAGE, CREATE ON SCHEMA public TO usuario;

**⚠️ E se continuar dando erro?**

Verifique também se o usuário tem **privilégio de conexão e escrita no banco**:

GRANT CONNECT ON DATABASE nest\_tarefas TO usuario;

GRANT ALL PRIVILEGES ON ALL TABLES IN SCHEMA public TO usuario;

**✅ Depois disso**

Reinicie o projeto:

npm run start:dev

Agora o NestJS e o TypeORM deverão conseguir criar as tabelas no banco sem erro.

Se quiser, posso te ajudar a criar um script SQL completo que você roda no pgAdmin para já configurar tudo de uma vez. Deseja isso?

Marque os privilégios:

✅ USAGE

✅ CREATE

nessa parte aqui não tem esses privilegios so tem ALL

INSERT

SELECT

UPDATE

DELETE

TRUNCATE

REFERENCES

TRIGGER

MAINTAIN

Ótimo, isso esclarece bastante! O que você está vendo são os **privilégios de tabelas**, mas você precisa **editar os privilégios do *schema***, não das tabelas em si.

Vamos resolver agora mesmo.

**✅ Etapas para conceder USAGE e CREATE no schema public no pgAdmin:**

**🔁 Se você clicou em “public” > Properties > Privileges e só aparecem esses:**

ALL, INSERT, SELECT, UPDATE, DELETE, TRUNCATE, REFERENCES, TRIGGER, MAINTAIN

➡️ **Você está vendo as permissões de uma tabela**, **não do schema**.

**🎯 O que fazer corretamente:**

1. No pgAdmin, vá até:

Servers > PostgreSQL > Databases > nest\_tarefas > Schemas > public

1. Clique com o **botão direito** em public  
   → escolha **Query Tool**
2. Cole o seguinte SQL:

GRANT USAGE, CREATE ON SCHEMA public TO usuario;

Substitua usuario pelo nome real do seu usuário (caso tenha usado outro).

1. Clique no botão de **executar** (ícone de raio) ou pressione **F5**

**✅ Confirmar permissões**

Se você quiser conferir depois, pode rodar:

SELECT \* FROM information\_schema.role\_schema\_grants WHERE grantee = 'usuario';

**📌 Explicação rápida**

* USAGE: permite que o usuário utilize o schema (ex: ler estrutura, navegar)
* CREATE: permite **criar objetos** dentro do schema (public neste caso), como tabelas

Essas permissões são obrigatórias para que o **TypeORM crie automaticamente as tabelas no schema public** quando synchronize: true.

**✅ Próximo passo**

Após rodar o comando, volte ao seu projeto e execute:

npm run start:dev

Se tudo estiver certo, você verá as mensagens de conexão e as tabelas serão criadas automaticamente.

Se quiser, posso montar para você um script SQL completo para:

* Criar o usuário
* Criar o banco
* Conceder todas as permissões necessárias

Quer que eu faça isso pra você?

QueryFailedError: permissão negada para esquema public ainda vem isso

Esse erro persistente:

❌ QueryFailedError: permissão negada para esquema public

confirma que **o usuário ainda não tem permissão no schema public**.

Vamos resolver isso **definitivamente**, com um **script SQL completo**. Rode isso no **pgAdmin (Query Tool)** **como superusuário** (ex: postgres):

**✅ SCRIPT SQL COMPLETO PARA RESOLVER O ERRO**

-- Substitua 'usuario' e '484368' pelo seu usuário e senha, se forem diferentes

-- 1. Criar o usuário (se ainda não existir)

DO $$

BEGIN

IF NOT EXISTS (SELECT FROM pg\_catalog.pg\_roles WHERE rolname = 'usuario') THEN

CREATE ROLE usuario WITH LOGIN PASSWORD '484368';

END IF;

END

$$;

-- 2. Garantir que o usuário tem acesso ao banco

GRANT CONNECT ON DATABASE nest\_tarefas TO usuario;

-- 3. Dar permissões no schema public

GRANT USAGE, CREATE ON SCHEMA public TO usuario;

-- 4. Garantir permissões em tabelas futuras no schema public

ALTER DEFAULT PRIVILEGES IN SCHEMA public

GRANT SELECT, INSERT, UPDATE, DELETE ON TABLES TO usuario;

-- 5. Garantir permissões para SEQUENCES (caso use IDs auto incrementais)

ALTER DEFAULT PRIVILEGES IN SCHEMA public

GRANT USAGE, SELECT ON SEQUENCES TO usuario;

**🔧 Como executar**

1. Abra o **pgAdmin**
2. Clique em nest\_tarefas > Schemas > public
3. Botão direito → **Query Tool**
4. Cole o script acima
5. Execute (ícone do raio ou F5)

**✅ Depois disso**

Reinicie sua aplicação NestJS:

npm run start:dev

➡️ Agora o erro **“permissão negada para esquema public”** deve desaparecer, e a tabela será criada com sucesso.

Se mesmo assim der erro, me diga:

* Qual usuário está no .env (DB\_USERNAME)
* Qual usuário está logado no pgAdmin
* Se você usa o PostgreSQL instalado localmente ou via Docker

Posso adaptar o script para qualquer cenário. Deseja que eu revise o .env com você também?

Obrigado por compartilhar o .env e o AppModule. Vamos resolver passo a passo a **conexão com o PostgreSQL no NestJS**, mesmo sendo sua primeira vez. 👇

**✅ 1. Verifique seu .env (está correto! ✅)**

DB\_HOST=localhost

DB\_PORT=5432

DB\_USERNAME=usuario

DB\_PASSWORD=484368

DB\_NAME=nest\_tarefas

→ Está no formato correto.  
Só falta garantir que esse usuário **tem as permissões certas**.

**✅ 2. Revisando o AppModule (está ótimo! ✅)**

Seu app.module.ts também está configurado corretamente com TypeOrmModule.forRootAsync(...).

**🚨 3. Agora o problema: usuário usuario não tem permissão no banco nest\_tarefas**

Esse é o erro:

QueryFailedError: permissão negada para esquema public

**✅ Solução definitiva**

Vamos rodar um script **dentro do pgAdmin** com um usuário que tenha **permissões de administrador** (geralmente postgres).

**📜 Script SQL para corrigir as permissões do usuário**

1. No pgAdmin, clique com botão direito no banco nest\_tarefas  
   → **Query Tool**
2. Cole esse script e execute:

-- Altere se o usuário for diferente de 'usuario'

DO $$

BEGIN

IF NOT EXISTS (SELECT FROM pg\_catalog.pg\_roles WHERE rolname = 'usuario') THEN

CREATE ROLE usuario WITH LOGIN PASSWORD '484368';

END IF;

END

$$;

-- Dar acesso ao banco

GRANT CONNECT ON DATABASE nest\_tarefas TO usuario;

-- Permissões no schema public

GRANT USAGE, CREATE ON SCHEMA public TO usuario;

-- Permissões padrão para tabelas criadas no futuro

ALTER DEFAULT PRIVILEGES IN SCHEMA public

GRANT SELECT, INSERT, UPDATE, DELETE ON TABLES TO usuario;

-- Permissões para sequências (IDs auto incrementais)

ALTER DEFAULT PRIVILEGES IN SCHEMA public

GRANT USAGE, SELECT ON SEQUENCES TO usuario;

**✅ Depois disso:**

1. Salve
2. Feche o Query Tool
3. Rode novamente sua aplicação:

npm run start:dev

Se tudo estiver certo, você verá algo como:

[Nest] ... Successfully connected to the database

**❓ Dúvidas comuns**

* **Como saber qual usuário estou usando no pgAdmin?**  
  Verifique no canto superior do pgAdmin, ou vá em Login/Group Roles.
* **Posso criar outro usuário com permissões completas?**  
  Sim! Quer que eu gere um script para criar um novo usuário com tudo pronto?

Se quiser, posso:

* revisar sua entidade para garantir que será sincronizada
* testar a conexão com uma tabela real
* criar um novo usuário com permissões administrativas

Deseja que eu siga com alguma dessas opções?

Ótimo! Você seguiu corretamente todos os passos, mas como a **mensagem Successfully connected to the database não apareceu**, mesmo com tudo configurado, isso significa que o NestJS **não chegou a executar nenhuma sincronização** com o banco, por **ainda não existir nenhuma entidade** registrada.

**🎯 Resumo rápido:**

Você **já instalou tudo corretamente** ✅  
Você **configurou o .env e o AppModule corretamente** ✅  
Você \*\*rodou npm run start:dev e **não apareceu nenhum erro de conexão** ❓  
Você **esperava ver “Successfully connected to the database”** ❓

Então...

**🧠 Por que não aparece “conectado com sucesso”?**

O NestJS e o TypeORM **só tentam conectar ao banco de dados** quando alguma **entidade (tabela)** precisa ser sincronizada, ou seja:

❌ Se não tiver nenhuma entidade criada e registrada no módulo, **o NestJS ignora a conexão com o banco.**

**✅ Solução: Criar uma entidade simples para forçar a conexão**

Você pode criar uma entidade chamada Tarefa, mesmo que seja só para teste:

**1️⃣ Criar o arquivo tarefa.entity.ts**

// src/tarefas/tarefa.entity.ts

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

@Entity()

export class Tarefa {

@PrimaryGeneratedColumn()

id: number;

@Column()

titulo: string;

@Column()

descricao: string;

}

**2️⃣ Registrar a entidade no módulo**

No tarefas.module.ts:

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

import { TypeOrmModule } from '@nestjs/typeorm';

import { Tarefa } from './tarefa.entity';

import { TarefasService } from './tarefas.service';

import { TarefasController } from './tarefas.controller';

@Module({

imports: [TypeOrmModule.forFeature([Tarefa])],

controllers: [TarefasController],

providers: [TarefasService],

})

export class TarefasModule {}

**3️⃣ Rodar o projeto novamente**

npm run start:dev

Agora sim, o TypeORM tentará criar a tabela tarefa no banco de dados, e você verá no terminal:

[Nest] ... Successfully connected to the database

**🧪 Confirmando no pgAdmin**

Se tudo deu certo, você verá a tabela tarefa criada no schema public do banco nest\_tarefas.

Se quiser, posso:

* Criar a entidade Tarefa completa com enum de status e DTOs.
* Criar um seed para inserir dados de teste no banco.

Deseja isso agora?

não funcionou recebi como erro a mensagem error: autentica��o do tipo senha falhou para o usu�rio "usuario" quero que saiba que é só uma simulação, sem entidade mesmo, Simular a conexão (sem entidades ainda), garantir que app inicia com sucesso.

Perfeito, obrigado por esclarecer! Como é **apenas uma simulação de conexão (sem entidades ainda)** e você está recebendo este erro:

error: autenticação do tipo senha falhou para o usuário "usuario"

isso **significa que o problema não está no NestJS**, mas sim no **usuário/senha do PostgreSQL** que você colocou no seu .env.

**🧨 O que esse erro significa?**

Esse erro do PostgreSQL quer dizer:

❌ O PostgreSQL **rejeitou a senha informada** para o usuário "usuario".

Mesmo que o usuário exista, se a senha estiver errada, ele não deixará conectar.

**✅ Passo a passo para resolver**

**1. Confirme se o usuário e senha estão corretos**

Abra o **pgAdmin** e siga:

* Vá em:  
  Servers > PostgreSQL > Login/Group Roles
* Verifique se o usuário usuario existe.
* Clique com o botão direito em usuario → **Properties**
* Vá em **Definition** → veja a senha definida.

Se não souber a senha, você pode **alterar a senha** agora:

**2. ✅ Alterar a senha no pgAdmin**

* No menu **Login/Group Roles > usuario**
* Clique com o botão direito > **Properties**
* Vá até a aba **Definition**
* Em **Password**, digite: 484368 (a que você colocou no .env)
* Clique em **Save**

Agora a senha do usuário está sincronizada com seu .env.

**3. (Opcional) Rodar comando SQL no Query Tool**

Se preferir definir ou redefinir a senha via SQL:

ALTER USER usuario WITH PASSWORD '484368';

**4. Reinicie o NestJS**

npm run start:dev

Você deve ver agora:

[Nest]... Starting application...

[Nest]... Successfully connected to the database

**✅ Recapitulando**

| **Verificação** | **Status** |
| --- | --- |
| Usuário usuario existe? | ✅ / ❌ |
| Senha correta no .env? | ✅ / ❌ |
| Porta padrão do PostgreSQL (5432)? | ✅ / ❌ |
| Serviço do PostgreSQL ativo? | ✅ / ❌ |

Se quiser, posso te enviar um script SQL completo para:

* Criar o banco
* Criar o usuário usuario
* Definir a senha correta
* Conceder todas as permissões necessárias

Deseja que eu prepare isso para você agora?

não é para abrir o pgadmin porque é apenas uma simulação, quero apenas Simular a conexão (sem entidades ainda), garantir que app inicia com sucesso.

Entendi agora com total clareza:  
✅ **Você quer simular a conexão com o PostgreSQL sem usar o pgAdmin, sem criar entidade, e só garantir que a aplicação NestJS inicia com sucesso.**

**🧠 O problema atual:**

Mesmo sem entidades, o NestJS está tentando se conectar ao PostgreSQL com:

DB\_USERNAME=usuario

DB\_PASSWORD=484368

E como **esse usuário não existe de verdade no PostgreSQL**, ele retorna:

error: autenticação do tipo senha falhou para o usuário "usuario"

**✅ Como simular a conexão sem precisar de banco real**

Você tem 3 opções:

**✅ Opção 1: Usar o usuário real do PostgreSQL**

Se você instalou o PostgreSQL no Windows, ele criou um usuário padrão chamado:

* postgres (nome de usuário)
* senha: a que você definiu na instalação

Atualize seu .env:

DB\_USERNAME=postgres

DB\_PASSWORD=senha\_que\_você\_usou\_na\_instalação

Essa é a forma mais direta de “simular a conexão” — o Nest vai tentar conectar, e se o banco estiver rodando, já será suficiente.

**✅ Opção 2: Usar synchronize: false e banco fictício**

Se não quiser nem conectar de verdade, você pode alterar seu AppModule para algo assim:

TypeOrmModule.forRoot({

type: 'postgres',

host: 'localhost',

port: 5432,

username: 'fake',

password: 'fake',

database: 'fake',

autoLoadEntities: false,

synchronize: false,

}),

Mas atenção: isso **ainda vai tentar conectar**. Então o PostgreSQL precisa estar rodando, mesmo que o usuário/banco não existam de verdade, **senão dá erro de autenticação**.

**✅ Opção 3 (ideal): Usar banco em memória para testes com SQLite**

Para simular a conexão **sem precisar de PostgreSQL**, use SQLite temporariamente:

npm install sqlite3

No seu AppModule:

TypeOrmModule.forRoot({

type: 'sqlite',

database: ':memory:',

synchronize: true,

autoLoadEntities: true,

}),

✅ Agora o NestJS inicia normalmente sem PostgreSQL, sem pgAdmin, sem entidades, e você pode continuar desenvolvendo.

**✅ Qual o melhor caminho pra você agora?**

Se você **está só treinando Nest + TypeORM e ainda não vai usar banco de verdade**, **recomendo usar SQLite** (Opção 3). Quer que eu configure isso no seu projeto?

Posso te enviar:

* O AppModule pronto com SQLite
* .env adequado
* Um exemplo mínimo funcional

Quer que eu faça isso?

Claro! Aqui está a aula do **🗓️ Dia 14 – Preparar para uso do banco com TypeORM**, elaborada em formato **Markdown** para você usar em seus estudos ou como material de aula:

# 🗓️ Dia 14 – Preparar para uso do banco com TypeORM

## 🎯 Objetivo do Dia

Entender o conceito de \*\*entidades\*\* no TypeORM, suas diferenças em relação aos DTOs e como mapear classes para tabelas usando decorators do TypeORM.

---

## 📚 Conteúdo

### 🔹 O que são Entidades?

Entidades são \*\*representações de tabelas no banco de dados\*\*. Cada instância de uma entidade representa uma linha da tabela.

- Exemplo: `TarefaEntity` → tabela `tarefa`

- Cada propriedade → uma coluna da tabela

---

### 🔹 Introdução aos Decorators do TypeORM

Decorators são usados para dizer ao TypeORM como mapear as propriedades:

| Decorador | Função |

|------------------------|---------------------------------------------------------|

| `@Entity()` | Marca a classe como uma entidade (uma tabela) |

| `@PrimaryGeneratedColumn()` | Gera uma coluna primária com valor automático |

| `@Column()` | Marca a propriedade como uma coluna da tabela |

| `@CreateDateColumn()` | Armazena a data de criação automaticamente |

---

### 🔹 Diferença entre Entidades e DTOs

| Entidade | DTO |

|----------------------------------|-------------------------------------------|

| Reflete estrutura da base | Reflete dados que entram ou saem da API |

| Usada com TypeORM | Usada com validação (class-validator) |

| Pode ter regras de banco (chaves, timestamp) | Só possui os campos necessários |

---

### 🔹 Visão geral:

- \*\*Repository\*\*: camada de acesso ao banco de dados

- \*\*Migration\*\*: controle de alterações de estrutura do banco

- \*\*Relation\*\*: relacionamento entre entidades (1:1, 1:N, N:N)

> \*Esses temas serão aprofundados nas próximas aulas.\*

---

## 🔧 Atividade

Crie a entidade `Tarefa` com os seguintes campos:

```ts

// src/tarefas/tarefa.entity.ts

import {

Entity,

PrimaryGeneratedColumn,

Column,

CreateDateColumn,

} from 'typeorm';

import { TarefaStatus } from './enums/tarefa-status.enum';

@Entity()

export class Tarefa {

@PrimaryGeneratedColumn('uuid')

id: string;

@Column()

titulo: string;

@Column()

descricao: string;

@Column()

status: TarefaStatus;

@CreateDateColumn()

dataCriacao: Date;

}

⚠️ Lembre-se de importar corretamente o TarefaStatus (enum que representa o status da tarefa).

**✅ Verificação**

1. Adicione a entidade em tarefas.module.ts:

TypeOrmModule.forFeature([Tarefa])

1. Rode o projeto:

npm run start:dev

1. Verifique no console se a tabela foi criada (se synchronize: true estiver ativado)

**🧪 Exercício (opcional)**

* Tente acessar a tabela no banco via pgAdmin ou SQLite (dependendo do banco que estiver usando).
* Adicione um campo extra opcional como @Column({ default: false }) arquivada: boolean.

**Próximos passos**

* Conectar o service com o banco usando o Repository
* Refatorar os métodos create, find, update, delete para usarem banco de dados

---

Se quiser, posso gerar esse arquivo `.md` para você baixar diretamente. Deseja isso?

o servidor estava rodando tudo certo mas deois dessas mudanças recebi o erro QueryFailedError: permissão negada para esquema public

Esse erro:

QueryFailedError: permissão negada para esquema public

significa que o **usuário configurado no seu banco de dados não tem permissão para criar ou acessar tabelas no schema public**, o que o TypeORM tenta fazer automaticamente ao sincronizar entidades.

**✅ Solução definitiva: dar permissão no schema public**

Você precisa **garantir que o usuário usado no seu .env tem as permissões certas no banco** — principalmente:

* USAGE
* CREATE
* Permissão em **default privileges** para futuras tabelas/sequências

**💡 Como resolver sem usar o pgAdmin?**

**1. Acesse o terminal do seu sistema**

**2. Conecte-se ao PostgreSQL com o usuário administrador (geralmente postgres):**

psql -U postgres -d nest\_tarefas

Ele vai pedir a senha do usuário postgres.

Se estiver no Windows e o psql não estiver reconhecido, vá até o terminal do PostgreSQL (pode estar em algo como SQL Shell (psql)).

**3. Execute os comandos abaixo para dar as permissões ao usuário usuario:**

-- Se o usuário ainda não existe:

CREATE ROLE usuario WITH LOGIN PASSWORD '484368';

-- Conceder conexão ao banco

GRANT CONNECT ON DATABASE nest\_tarefas TO usuario;

-- Entrar no banco de dados

\c nest\_tarefas

-- Dar permissão no schema public

GRANT USAGE, CREATE ON SCHEMA public TO usuario;

-- Permissão padrão para tabelas criadas no futuro

ALTER DEFAULT PRIVILEGES IN SCHEMA public

GRANT SELECT, INSERT, UPDATE, DELETE ON TABLES TO usuario;

-- Permissão para uso de sequências (ID auto-incremental)

ALTER DEFAULT PRIVILEGES IN SCHEMA public

GRANT USAGE, SELECT ON SEQUENCES TO usuario;

**4. Reinicie o servidor NestJS**

npm run start:dev

Agora o erro **permissão negada para esquema public** deve desaparecer, e o NestJS conseguirá sincronizar sua entidade com o banco de dados.

**❗ Importante**

Esse erro **não é causado pelo código do NestJS**, mas sim pelas **restrições do PostgreSQL para o usuário configurado**.

Se quiser, posso gerar um **script .sql pronto para executar** e resolver isso automaticamente.

Quer que eu gere esse script?

GRANT USAGE, CREATE ON SCHEMA public TO usuario;

onde está escrito usuario é pra eu escrever o usuario que está no .env

✅ **Sim, exatamente!**

No comando:

GRANT USAGE, CREATE ON SCHEMA public TO usuario;

Você deve substituir usuario pelo **nome do usuário que você colocou no seu .env**, na variável DB\_USERNAME.

**🔍 Exemplo prático:**

Se no seu .env está assim:

DB\_USERNAME=meuusuario

DB\_PASSWORD=484368

Então o comando correto seria:

GRANT USAGE, CREATE ON SCHEMA public TO meuusuario;

**💡 Dica: veja o usuário atual do seu projeto**

Abra o terminal e rode:

echo %DB\_USERNAME%

Ou verifique diretamente no seu .env.

Se quiser, posso montar um script .sql completo com os comandos de permissões já substituídos com o seu usuário atual. Deseja?

Excelente pergunta! 👇

Se você já está usando **NestJS com TypeORM**, a maneira **mais simples e automática** de adicionar um campo ao banco de dados é modificar diretamente sua entidade com o decorator @Column() e deixar que o **TypeORM sincronize** isso com o PostgreSQL (caso synchronize: true esteja ativado).

**✅ 1. No código (NestJS + TypeORM)**

**🧱 Entidade Tarefa**

No seu arquivo tarefa.entity.ts, adicione a nova coluna assim:

@Column({ default: false })

arquivada: boolean;

Exemplo completo:

@Entity()

export class Tarefa {

@PrimaryGeneratedColumn('uuid')

id: string;

@Column()

titulo: string;

@Column()

descricao: string;

@Column()

status: TarefaStatus;

@CreateDateColumn()

dataCriacao: Date;

@Column({ default: false })

arquivada: boolean; // ✅ nova coluna opcional

}

**🧪 2. Sincronização com o banco**

Se no AppModule você tiver:

synchronize: true

Então, ao iniciar o NestJS com npm run start:dev, ele fará automaticamente:

* Adição da nova coluna arquivada
* Com valor padrão false para registros antigos

**🛠️ 3. Fazer manualmente no PostgreSQL (caso precise)**

Se você quiser **executar diretamente no PostgreSQL**, você pode rodar:

ALTER TABLE tarefa ADD COLUMN arquivada BOOLEAN DEFAULT false;

Isso adiciona a coluna com valor padrão false.

**⚠️ Aviso importante**

synchronize: true **NÃO DEVE SER USADO EM PRODUÇÃO**, pois pode causar perda de dados. Em ambientes reais, use **migrations**.

Se quiser, posso te mostrar como criar uma **migration oficial** do TypeORM para adicionar essa coluna com segurança. Deseja isso?

Claro! Aqui está a **Aula do Dia 15 – Integrar TypeORM com Nest.js**, elaborada no formato de plano de aula em Markdown, conforme seus dias anteriores:

# 🗓️ Dia 15 – Integrar TypeORM com Nest.js

## 🎯 Objetivo do Dia

Aprender a integrar o NestJS com o banco de dados usando TypeORM, conectando via variáveis de ambiente e utilizando uma entidade real.

---

## 📚 Conteúdo

### 🔹 .env com variáveis de conexão

O arquivo `.env` armazena as credenciais de acesso ao banco. Exemplo:

```env

DB\_HOST=localhost

DB\_PORT=5432

DB\_USERNAME=postgres

DB\_PASSWORD=senha\_do\_postgres

DB\_NAME=nest\_tarefas

**🔹 Importando TypeOrmModule no AppModule**

No app.module.ts usamos TypeOrmModule.forRootAsync() para ler o .env dinamicamente:

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

import { ConfigModule, ConfigService } from '@nestjs/config';

import { TypeOrmModule } from '@nestjs/typeorm';

@Module({

imports: [

ConfigModule.forRoot({ isGlobal: true }),

TypeOrmModule.forRootAsync({

imports: [ConfigModule],

inject: [ConfigService],

useFactory: (configService: ConfigService) => ({

type: 'postgres',

host: configService.get('DB\_HOST'),

port: configService.get<number>('DB\_PORT'),

username: configService.get('DB\_USERNAME'),

password: configService.get('DB\_PASSWORD'),

database: configService.get('DB\_NAME'),

autoLoadEntities: true,

synchronize: true, // ⚠️ apenas em desenvolvimento!

}),

}),

],

})

export class AppModule {}

**🔧 Atividades**

**✅ 1. Instale os pacotes do TypeORM**

npm install --save @nestjs/typeorm typeorm pg @nestjs/config

**✅ 2. Crie a entidade TarefaEntity**

Crie o arquivo src/tarefas/tarefa.entity.ts:

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

import { TarefaStatus } from './enums/tarefa-status.enum';

@Entity('tarefas')

export class TarefaEntity {

@PrimaryGeneratedColumn('uuid')

id: string;

@Column()

titulo: string;

@Column()

descricao: string;

@Column()

status: TarefaStatus;

}

**✅ 3. Adicione a entidade no módulo**

No arquivo tarefas.module.ts:

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

import { TypeOrmModule } from '@nestjs/typeorm';

import { TarefaEntity } from './tarefa.entity';

import { TarefasService } from './tarefas.service';

import { TarefasController } from './tarefas.controller';

@Module({

imports: [TypeOrmModule.forFeature([TarefaEntity])],

controllers: [TarefasController],

providers: [TarefasService],

})

export class TarefasModule {}

**🧪 Exercício**

**🔹 Criar o repositório e salvar no banco**

No tarefas.service.ts, injete o repositório:

import { InjectRepository } from '@nestjs/typeorm';

import { Repository } from 'typeorm';

import { TarefaEntity } from './tarefa.entity';

@Injectable()

export class TarefasService {

constructor(

@InjectRepository(TarefaEntity)

private tarefasRepository: Repository<TarefaEntity>,

) {}

async createTarefa(dto: CreateTarefaDto): Promise<TarefaEntity> {

const tarefa = this.tarefasRepository.create({

titulo: dto.titulo,

descricao: dto.descricao,

status: TarefaStatus.ABERTA,

});

return this.tarefasRepository.save(tarefa);

}

}

No controller:

@Post()

async create(@Body() dto: CreateTarefaDto): Promise<TarefaEntity> {

return this.tarefasService.createTarefa(dto);

}

**✅ Verificação final**

* Rode o projeto com npm run start:dev
* Use Postman ou Insomnia:
  + Rota: POST /tarefas
  + Body:
  + {
  + "titulo": "Estudar NestJS",
  + "descricao": "Aula de integração com banco"
  + }
* Confirme se a tarefa foi salva no banco

**🚀 Próximo passo**

* Implementar métodos de listagem (find, findOne, delete, update) usando o repositório do TypeORM

---

Se quiser, posso gerar esse conteúdo como um arquivo `.md` para você baixar. Deseja isso?

Claro! Aqui está a **Aula do Dia 16 – CRUD com banco de dados**, elaborada em formato de plano de aula em Markdown, seguindo o padrão dos dias anteriores:

# 🗓️ Dia 16 – CRUD com banco de dados

## 🎯 Objetivo do Dia

Aprender a usar o `Repository` do TypeORM no `TarefasService` para realizar as operações CRUD diretamente no banco de dados, substituindo o array em memória.

---

## 📚 Conteúdo

### 🔹 Criar o repositório de tarefas

No NestJS, não é necessário criar uma classe `TarefaRepository` manualmente. O próprio TypeORM já expõe um repositório automaticamente ao usar `@InjectRepository()`.

### 🔹 O que é `InjectRepository`?

É um decorator que injeta a instância do repositório da entidade para realizar operações no banco de dados.

### 🔹 Principais métodos usados:

| Método | Função |

|--------------|----------------------------------------------------|

| `save()` | Cria ou atualiza uma entidade |

| `find()` | Lista todos os registros |

| `findOne()` | Busca um registro por critério (ex: ID) |

| `delete()` | Deleta um registro |

| `update()` | Atualiza campos de um registro (ou usar `save()`) |

---

## 🔧 Atividades

### ✅ Refatorar o `TarefasService`

1. \*\*Importe os pacotes:\*\*

```ts

import { InjectRepository } from '@nestjs/typeorm';

import { Repository } from 'typeorm';

import { TarefaEntity } from './tarefa.entity';

1. **Injete o repositório no construtor:**

constructor(

@InjectRepository(TarefaEntity)

private readonly tarefasRepository: Repository<TarefaEntity>,

) {}

1. **Implemente os métodos:**

async findAll(): Promise<TarefaEntity[]> {

return this.tarefasRepository.find();

}

async findById(id: string): Promise<TarefaEntity> {

const tarefa = await this.tarefasRepository.findOne({ where: { id } });

if (!tarefa) {

throw new NotFoundException(`Tarefa com ID "${id}" não encontrada`);

}

return tarefa;

}

async create(dto: CreateTarefaDto): Promise<TarefaEntity> {

const nova = this.tarefasRepository.create({

...dto,

status: TarefaStatus.ABERTA,

});

return this.tarefasRepository.save(nova);

}

async delete(id: string): Promise<void> {

const result = await this.tarefasRepository.delete(id);

if (result.affected === 0) {

throw new NotFoundException(`Tarefa com ID "${id}" não encontrada`);

}

}

async updateStatus(id: string, status: TarefaStatus): Promise<TarefaEntity> {

const tarefa = await this.findById(id);

tarefa.status = status;

return this.tarefasRepository.save(tarefa);

}

**🧪 Exercício**

**📌 Refatore o Controller para chamar os métodos atualizados:**

@Get()

findAll(): Promise<TarefaEntity[]> {

return this.tarefasService.findAll();

}

@Get(':id')

findById(@Param('id') id: string): Promise<TarefaEntity> {

return this.tarefasService.findById(id);

}

@Post()

create(@Body() dto: CreateTarefaDto): Promise<TarefaEntity> {

return this.tarefasService.create(dto);

}

@Delete(':id')

delete(@Param('id') id: string): Promise<void> {

return this.tarefasService.delete(id);

}

@Patch(':id/status')

updateStatus(

@Param('id') id: string,

@Body('status', TarefaStatusValidationPipe) status: TarefaStatus,

): Promise<TarefaEntity> {

return this.tarefasService.updateStatus(id, status);

}

**✅ Verificação**

1. Execute a aplicação com:

npm run start:dev

1. Teste os endpoints com Postman/Insomnia:

* GET /tarefas
* GET /tarefas/:id
* POST /tarefas com body:

{

"titulo": "Estudar NestJS",

"descricao": "Aula de CRUD com banco"

}

* PATCH /tarefas/:id/status
* DELETE /tarefas/:id

**🔄 Comparação**

| **Antes (em memória)** | **Agora (com banco)** |
| --- | --- |
| Armazenava as tarefas em um array | Usa banco de dados PostgreSQL |
| Não persistia os dados | Os dados ficam salvos entre execuções |

**🚀 Próximo passo**

* Implementar filtros por status e termo diretamente via banco (usando find com where)
* Criar paginação e ordenação com TypeORM (opcional)

---

Se quiser, posso gerar esse conteúdo em um arquivo `.md` para você baixar diretamente. Deseja isso?

async findAll(): Promise<TarefaEntity[]> {

return this.tarefasRepository.find();

}

async findById(id: string): Promise<TarefaEntity> {

const tarefa = await this.tarefasRepository.findOne({ where: { id } });

if (!tarefa) {

throw new NotFoundException(`Tarefa com ID "${id}" não encontrada`);

}

return tarefa;

}

async create(dto: CreateTarefaDto): Promise<TarefaEntity> {

const nova = this.tarefasRepository.create({

...dto,

status: TarefaStatus.ABERTA,

});

return this.tarefasRepository.save(nova);

}

async delete(id: string): Promise<void> {

const result = await this.tarefasRepository.delete(id);

if (result.affected === 0) {

throw new NotFoundException(`Tarefa com ID "${id}" não encontrada`);

}

}

async updateStatus(id: string, status: TarefaStatus): Promise<TarefaEntity> {

const tarefa = await this.findById(id);

tarefa.status = status;

return this.tarefasRepository.save(tarefa);

}

voce fala pra eu implementar esses codigos em tarefas.service.ts mas e o codigo que ja existe nele como o abaixo

private tarefas: Tarefa[] = [];

getTodasTarefas(): Tarefa[] {

return this.tarefas;

}

getTarefaPorId(id: number): Tarefa {

const tarefa = this.tarefas.find((tarefa) => tarefa.id === id);

if (!tarefa) {

throw new NotFoundException(`Tarefa com id ${id} não encontrada.`);

}

return tarefa;

}

async createTarefa(dto: CreateTarefaDto): Promise<TarefaEntity> {

const tarefa = this.tarefasRepository.create({

titulo: dto.titulo,

descricao: dto.descricao,

status: TarefaStatus.ABERTA,

});

return this.tarefasRepository.save(tarefa);

}

atualizarStatus(id: number, novoStatus: TarefaStatus): Tarefa {

const tarefa = this.getTarefaPorId(id);

tarefa.status = novoStatus;

return tarefa;

}

deleteTarefa(id: number): void {

const tarefa = this.getTarefaPorId(id); // lança NotFoundException se não existir

this.tarefas = this.tarefas.filter((t) => t.id !== tarefa.id);

}

filtrarTarefas(filtroDto: FilterTarefasDto): Tarefa[] {

const { status, termo } = filtroDto;

let tarefasFiltradas = this.tarefas;

if (status) {

tarefasFiltradas = tarefasFiltradas.filter((t) => t.status === status);

}

if (termo) {

tarefasFiltradas = tarefasFiltradas.filter(

(t) =>

t.titulo.toLowerCase().includes(termo.toLowerCase()) ||

t.descricao.toLowerCase().includes(termo.toLowerCase()),

);

}

return tarefasFiltradas;

}

Ótima pergunta — **sim, você deve substituir os métodos antigos que usam array por versões que utilizam o banco de dados com TypeORM.**

**🔁 O que fazer com o código antigo:**

Abaixo está um **guia de substituição**, explicando quais partes você pode **remover** e **quais deve manter ou refatorar**:

**✅ REMOVER completamente:**

Você não precisa mais do array em memória:

private tarefas: Tarefa[] = [];

E também pode apagar todos os métodos que usam esse array, como:

getTodasTarefas(): Tarefa[] { ... }

getTarefaPorId(id: number): Tarefa { ... }

deleteTarefa(id: number): void { ... }

filtrarTarefas(filtroDto: FilterTarefasDto): Tarefa[] { ... }

Esses métodos só funcionavam porque você armazenava tarefas em memória — **isso não é necessário mais**, pois agora os dados estão persistidos no banco.

**✅ SUBSTITUIR pelos novos métodos com TypeORM:**

Aqui estão os métodos atualizados para usar no banco de dados:

async findAll(): Promise<TarefaEntity[]> {

return this.tarefasRepository.find();

}

async findById(id: string): Promise<TarefaEntity> {

const tarefa = await this.tarefasRepository.findOne({ where: { id } });

if (!tarefa) {

throw new NotFoundException(`Tarefa com ID "${id}" não encontrada`);

}

return tarefa;

}

async create(dto: CreateTarefaDto): Promise<TarefaEntity> {

const nova = this.tarefasRepository.create({

...dto,

status: TarefaStatus.ABERTA,

});

return this.tarefasRepository.save(nova);

}

async delete(id: string): Promise<void> {

const result = await this.tarefasRepository.delete(id);

if (result.affected === 0) {

throw new NotFoundException(`Tarefa com ID "${id}" não encontrada`);

}

}

async updateStatus(id: string, status: TarefaStatus): Promise<TarefaEntity> {

const tarefa = await this.findById(id);

tarefa.status = status;

return this.tarefasRepository.save(tarefa);

}

**🚧 *E o método filtrarTarefas com status e termo?***

Você pode implementá-lo assim usando **query builder do TypeORM**:

async filtrarTarefas(filtroDto: FilterTarefasDto): Promise<TarefaEntity[]> {

const { status, termo } = filtroDto;

const query = this.tarefasRepository.createQueryBuilder('tarefa');

if (status) {

query.andWhere('tarefa.status = :status', { status });

}

if (termo) {

query.andWhere(

'(LOWER(tarefa.titulo) LIKE LOWER(:termo) OR LOWER(tarefa.descricao) LIKE LOWER(:termo))',

{ termo: `%${termo}%` },

);

}

return query.getMany();

}

**✅ Resultado final:**

Seu novo tarefas.service.ts será limpo, organizado e **totalmente baseado em banco de dados**, como deve ser numa aplicação real.

Se quiser, posso te ajudar a montar o tarefas.service.ts final já com tudo substituído. Deseja isso?

Claro! Aqui está a **Aula do Dia 17 – Autenticação com JWT (Parte 1)** elaborada em formato de plano de aula, pronta para te guiar passo a passo:

# 🗓️ Dia 17 – Autenticação com JWT (Parte 1)

## 🎯 Objetivo do Dia

Aprender a criar usuários e armazenar senhas criptografadas, preparando o projeto para autenticação com JWT.

---

## 📚 Conteúdo

### 🔐 Introdução à Autenticação

Antes de autenticar, precisamos de:

- Um \*\*módulo de usuários\*\*

- Uma \*\*entidade de usuário\*\*

- Um serviço para \*\*registrar e salvar\*\* o usuário no banco

- Senha \*\*criptografada\*\* (nunca salvar texto puro!)

---

## 📦 Instalar pacotes necessários

Execute:

```bash

npm install @nestjs/jwt @nestjs/passport passport passport-jwt bcryptjs

Esses pacotes serão usados nas próximas aulas para login, geração e validação de tokens JWT.

**🧱 Estrutura**

Crie o módulo e o service de usuários:

nest g module usuario

nest g service usuario

**🧬 Criar entidade UsuarioEntity**

**🗂️ src/usuario/usuario.entity.ts**

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

@Entity('usuarios')

export class UsuarioEntity {

@PrimaryGeneratedColumn('uuid')

id: string;

@Column({ unique: true })

email: string;

@Column()

senha: string;

@Column()

nome: string;

}

**🛠️ Configurar UsuarioService**

**🗂️ src/usuario/usuario.service.ts**

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

import { InjectRepository } from '@nestjs/typeorm';

import { Repository } from 'typeorm';

import \* as bcrypt from 'bcryptjs';

import { UsuarioEntity } from './usuario.entity';

@Injectable()

export class UsuarioService {

constructor(

@InjectRepository(UsuarioEntity)

private readonly usuarioRepository: Repository<UsuarioEntity>,

) {}

async registrar(nome: string, email: string, senha: string): Promise<UsuarioEntity> {

const salt = await bcrypt.genSalt();

const senhaCriptografada = await bcrypt.hash(senha, salt);

const usuario = this.usuarioRepository.create({

nome,

email,

senha: senhaCriptografada,

});

return this.usuarioRepository.save(usuario);

}

async buscarPorEmail(email: string): Promise<UsuarioEntity | null> {

return this.usuarioRepository.findOne({ where: { email } });

}

}

**🧪 Exercício**

1. Crie o controller de usuários:

nest g controller usuario

1. Implemente uma rota para cadastro:

**🗂️ src/usuario/usuario.controller.ts**

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

import { UsuarioService } from './usuario.service';

@Controller('usuarios')

export class UsuarioController {

constructor(private readonly usuarioService: UsuarioService) {}

@Post('registrar')

async registrar(

@Body('nome') nome: string,

@Body('email') email: string,

@Body('senha') senha: string,

) {

return this.usuarioService.registrar(nome, email, senha);

}

}

**✅ Testando no Postman**

* Endpoint: POST http://localhost:3000/usuarios/registrar
* Body (JSON):

{

"nome": "João",

"email": "joao@email.com",

"senha": "minhasenha123"

}

* Esperado: retorna objeto com ID, nome e email. Senha vem criptografada no banco.

**🧩 Próximos Passos**

Na próxima aula (Dia 18), vamos:

* Criar o processo de **login**
* Gerar **JWT**
* Proteger rotas usando o token

Se quiser, posso gerar esse conteúdo como arquivo `.md` para você baixar. Deseja isso?

node:internal/modules/cjs/loader:1228

throw err;

^

Error: Cannot find module './../../node\_modules/bcryptjs/index.d'

Require stack:

- C:\meus\_projetos\aulas-nodeJS-nestJS-typeORM\nestJS-aulas\primeira\_semana\projeto-tarefas\dist\usuario\usuario.service.js

- C:\meus\_projetos\aulas-nodeJS-nestJS-typeORM\nestJS-aulas\primeira\_semana\projeto-tarefas\dist\usuario\usuario.module.js

- C:\meus\_projetos\aulas-nodeJS-nestJS-typeORM\nestJS-aulas\primeira\_semana\projeto-tarefas\dist\app.module.js

- C:\meus\_projetos\aulas-nodeJS-nestJS-typeORM\nestJS-aulas\primeira\_semana\projeto-tarefas\dist\main.js

at Function.\_resolveFilename (node:internal/modules/cjs/loader:1225:15)

at Function.\_load (node:internal/modules/cjs/loader:1055:27)

at TracingChannel.traceSync (node:diagnostics\_channel:322:14)

at wrapModuleLoad (node:internal/modules/cjs/loader:220:24)

at Module.require (node:internal/modules/cjs/loader:1311:12)

at require (node:internal/modules/helpers:136:16)

at Object.<anonymous> (C:\meus\_projetos\aulas-nodeJS-nestJS-typeORM\nestJS-aulas\primeira\_semana\projeto-tarefas\src\usuario\usuario.service.ts:5:1)

at Module.\_compile (node:internal/modules/cjs/loader:1554:14)

at Object..js (node:internal/modules/cjs/loader:1706:10)

at Module.load (node:internal/modules/cjs/loader:1289:32) {

code: 'MODULE\_NOT\_FOUND',

requireStack: [

'C:\\meus\_projetos\\aulas-nodeJS-nestJS-typeORM\\nestJS-aulas\\primeira\_semana\\projeto-tarefas\\dist\\usuario\\usuario.service.js',

'C:\\meus\_projetos\\aulas-nodeJS-nestJS-typeORM\\nestJS-aulas\\primeira\_semana\\projeto-tarefas\\dist\\usuario\\usuario.module.js',

'C:\\meus\_projetos\\aulas-nodeJS-nestJS-typeORM\\nestJS-aulas\\primeira\_semana\\projeto-tarefas\\dist\\app.module.js',

'C:\\meus\_projetos\\aulas-nodeJS-nestJS-typeORM\\nestJS-aulas\\primeira\_semana\\projeto-tarefas\\dist\\main.js'

]

}

o que significa esse erro

Esse erro significa que o Node.js **não conseguiu encontrar o módulo bcryptjs** ao tentar rodar seu projeto Nest.js, especificamente no usuario.service.ts.

**🧨 Erro:**

Error: Cannot find module './../../node\_modules/bcryptjs/index.d'

Esse trecho indica que o Node tentou importar o bcryptjs, mas o pacote não está disponível no diretório node\_modules.

**✅ Como resolver**

**1. Verifique se você realmente instalou o pacote:**

Execute:

npm install bcryptjs

⚠️ **Importante**: às vezes achamos que instalamos, mas esquecemos de confirmar ou ocorreu erro na instalação.

**2. Verifique a importação no seu arquivo**

No usuario.service.ts, verifique se está assim:

import \* as bcrypt from 'bcryptjs';

Essa é a forma correta para importar bcryptjs.

**3. Recompile e reinicie o projeto**

Após instalar corretamente:

npm run build

npm run start:dev

**❗ Obs. sobre index.d**

O index.d.ts é o arquivo de **tipagem TypeScript**. O erro menciona isso porque o Nest (via TypeScript) tenta encontrar o tipo ao compilar. Não se preocupe com esse .d.ts diretamente — ao instalar corretamente o pacote, isso será resolvido automaticamente.

Se o problema persistir, você pode forçar uma reinstalação:

rm -rf node\_modules

rm package-lock.json

npm install

Isso apaga e reinstala todas as dependências.

Se quiser, posso verificar com você o package.json para garantir que tudo está certo. Deseja isso?

[Nest] 93508 - 08/07/2025, 15:22:10 ERROR [ExceptionHandler] UnknownDependenciesException [Error]: Nest can't resolve dependencies of the UsuarioService (?). Please make sure that the argument "UsuarioEntityRepository" at index [0] is available in the UsuarioModule context.

Potential solutions:

- Is UsuarioModule a valid NestJS module?

- If "UsuarioEntityRepository" is a provider, is it part of the current UsuarioModule?

- If "UsuarioEntityRepository" is exported from a separate @Module, is that module imported within UsuarioModule?

@Module({

imports: [ /\* the Module containing "UsuarioEntityRepository" \*/ ]

})

at Injector.lookupComponentInParentModules (C:\meus\_projetos\aulas-nodeJS-nestJS-typeORM\nestJS-aulas\primeira\_semana\projeto-tarefas\node\_modules\@nestjs\core\injector\injector.js:262:19)

at process.processTicksAndRejections (node:internal/process/task\_queues:105:5)

at async Injector.resolveComponentInstance (C:\meus\_projetos\aulas-nodeJS-nestJS-typeORM\nestJS-aulas\primeira\_semana\projeto-tarefas\node\_modules\@nestjs\core\injector\injector.js:215:33)

at async resolveParam (C:\meus\_projetos\aulas-nodeJS-nestJS-typeORM\nestJS-aulas\primeira\_semana\projeto-tarefas\node\_modules\@nestjs\core\injector\injector.js:129:38)

at async Promise.all (index 0)

at async Injector.resolveConstructorParams (C:\meus\_projetos\aulas-nodeJS-nestJS-typeORM\nestJS-aulas\primeira\_semana\projeto-tarefas\node\_modules\@nestjs\core\injector\injector.js:144:27)

at async Injector.loadInstance (C:\meus\_projetos\aulas-nodeJS-nestJS-typeORM\nestJS-aulas\primeira\_semana\projeto-tarefas\node\_modules\@nestjs\core\injector\injector.js:70:13)

at async Injector.loadProvider (C:\meus\_projetos\aulas-nodeJS-nestJS-typeORM\nestJS-aulas\primeira\_semana\projeto-tarefas\node\_modules\@nestjs\core\injector\injector.js:98:9)

at async C:\meus\_projetos\aulas-nodeJS-nestJS-typeORM\nestJS-aulas\primeira\_semana\projeto-tarefas\node\_modules\@nestjs\core\injector\instance-loader.js:56:13

at async Promise.all (index 3) {

type: 'UsuarioService',

context: {

index: 0,

dependencies: [

'UsuarioEntityRepository'

],

name: 'UsuarioEntityRepository'

},

metadata: {

id: 'ad8e8b079e9b0ca340e4e'

},

moduleRef: {

id: '14b2b013688554a5cf36e'

}

}

agora veio esse erro

Esse erro:

Nest can't resolve dependencies of the UsuarioService (?). Please make sure that the argument "UsuarioEntityRepository" at index [0] is available in the UsuarioModule context.

significa que **o Nest.js não conseguiu injetar o repositório da entidade UsuarioEntity no seu UsuarioService**, porque o módulo onde está esse repositório **não está configurado corretamente** no UsuarioModule.

**✅ Como resolver**

Você precisa garantir 3 coisas:

**1. Sua entidade UsuarioEntity está anotada com @Entity()**

// usuario.entity.ts

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

@Entity('usuarios')

export class UsuarioEntity {

@PrimaryGeneratedColumn('uuid')

id: string;

@Column()

nome: string;

@Column({ unique: true })

email: string;

@Column()

senha: string;

}

**2. Você está registrando o repositório com TypeOrmModule.forFeature() no UsuarioModule**

// usuario.module.ts

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

import { TypeOrmModule } from '@nestjs/typeorm';

import { UsuarioService } from './usuario.service';

import { UsuarioController } from './usuario.controller';

import { UsuarioEntity } from './usuario.entity';

@Module({

imports: [TypeOrmModule.forFeature([UsuarioEntity])], // ✅ necessário

providers: [UsuarioService],

controllers: [UsuarioController],

exports: [UsuarioService],

})

export class UsuarioModule {}

**3. Você está injetando o repositório corretamente no UsuarioService**

// usuario.service.ts

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

import { InjectRepository } from '@nestjs/typeorm';

import { Repository } from 'typeorm';

import { UsuarioEntity } from './usuario.entity';

@Injectable()

export class UsuarioService {

constructor(

@InjectRepository(UsuarioEntity)

private readonly usuarioRepository: Repository<UsuarioEntity>,

) {}

async criar(nome: string, email: string, senha: string): Promise<UsuarioEntity> {

const novoUsuario = this.usuarioRepository.create({ nome, email, senha });

return this.usuarioRepository.save(novoUsuario);

}

}

**🛠️ Dica de Teste**

Se você já fez tudo isso e ainda assim o erro aparecer:

1. **Pare o servidor**
2. Apague os arquivos compilados (dist) e recompile:

rm -rf dist

npm run build

npm run start:dev

Se quiser, posso revisar seu usuario.module.ts e usuario.service.ts se você colar aqui. Deseja isso?

{

"message": "Cannot POST /usuarios/registrar",

"error": "Not Found",

"statusCode": 404

}

no postman veio a mensagem acima

[2025-07-08T19:30:19.280Z] POST /tarefas/registrar

e no terminal veio a mensagem acima o que está faltando

O erro:

"Cannot POST /usuarios/registrar"

significa que o NestJS **não encontrou a rota POST /usuarios/registrar** definida no seu projeto. E o fato de no terminal aparecer:

POST /tarefas/registrar

mostra que você provavelmente **enviou a requisição para o endpoint errado** (talvez tarefas/registrar ao invés de usuarios/registrar).

**✅ Verificações para resolver**

**1. Você criou um UsuarioController?**

No seu controller, deve existir algo como:

// usuario.controller.ts

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

import { UsuarioService } from './usuario.service';

import { CreateUsuarioDto } from './dto/create-usuario.dto';

@Controller('usuarios') // <-- prefixo da rota

export class UsuarioController {

constructor(private readonly usuarioService: UsuarioService) {}

@Post('registrar') // <-- POST /usuarios/registrar

async registrar(@Body() dto: CreateUsuarioDto) {

return this.usuarioService.criar(dto.nome, dto.email, dto.senha);

}

}

**2. O UsuarioController está declarado no UsuarioModule?**

@Module({

imports: [TypeOrmModule.forFeature([UsuarioEntity])],

controllers: [UsuarioController], // ✅ obrigatório

providers: [UsuarioService],

})

export class UsuarioModule {}

**3. O UsuarioModule está importado no AppModule?**

@Module({

imports: [

ConfigModule.forRoot({ isGlobal: true }),

TypeOrmModule.forRootAsync({ /\*...\*/ }),

UsuarioModule, // ✅ isso aqui precisa existir

],

})

export class AppModule {}

**4. Você está enviando para a URL correta no Postman?**

* Método: POST
* URL: http://localhost:3000/usuarios/registrar ✅
* Body (raw JSON):

{

"nome": "João",

"email": "joao@email.com",

"senha": "123456"

}

Se depois de tudo isso ainda não funcionar, envie aqui o conteúdo de:

* usuario.controller.ts
* usuario.module.ts
* app.module.ts

Que eu reviso para você! Deseja isso?

eu não tenho um CreateUsuarioDto

Ótimo! Então esse é o ponto exato que está faltando: você precisa criar o CreateUsuarioDto, que é o **objeto de transferência de dados** que define como os dados do usuário serão recebidos no corpo da requisição.

**✅ Criando o CreateUsuarioDto**

1. Crie a pasta dto dentro da pasta usuario (caso ainda não tenha).
2. Dentro dela, crie o arquivo create-usuario.dto.ts com o seguinte conteúdo:

// src/usuario/dto/create-usuario.dto.ts

import { IsEmail, IsNotEmpty, MinLength } from 'class-validator';

export class CreateUsuarioDto {

@IsNotEmpty()

nome: string;

@IsEmail()

email: string;

@MinLength(6)

senha: string;

}

🔒 Aqui usamos o class-validator para garantir que o nome não está vazio, o e-mail tem formato válido e a senha tem pelo menos 6 caracteres.

**✅ Atualize o UsuarioController**

Garanta que o seu controller está assim:

// src/usuario/usuario.controller.ts

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

import { UsuarioService } from './usuario.service';

import { CreateUsuarioDto } from './dto/create-usuario.dto';

@Controller('usuarios')

export class UsuarioController {

constructor(private readonly usuarioService: UsuarioService) {}

@Post('registrar')

async registrar(@Body() dto: CreateUsuarioDto) {

return this.usuarioService.criar(dto.nome, dto.email, dto.senha);

}

}

**✅ Teste no Postman**

* **Método**: POST
* **URL**: http://localhost:3000/usuarios/registrar
* **Headers**:
  + Content-Type: application/json
* **Body**:

{

"nome": "Maria",

"email": "maria@email.com",

"senha": "senha123"

}

Se após isso ainda houver erro, me mande o conteúdo completo de:

* usuario.service.ts
* usuario.module.ts
* usuario.controller.ts

E te ajudo a revisar ponto a ponto.

Deseja que eu gere o arquivo .ts pronto para o DTO para você baixar?

agora registrou o usuario, mas ele vveio com id está correto assim

Sim, **está absolutamente correto** que o usuário venha com um id ao ser registrado. Vou te explicar por quê:

**✅ Por que o id aparece ao registrar?**

Quando você registra um usuário com TypeORM e faz:

return this.usuarioRepository.save(usuario);

O TypeORM:

1. **Insere o usuário no banco**.
2. **Retorna o objeto completo salvo**, incluindo o id que foi gerado automaticamente pelo banco (se estiver usando @PrimaryGeneratedColumn()).

**🧠 Exemplo prático:**

Imagine sua entidade assim:

@Entity('usuarios')

export class UsuarioEntity {

@PrimaryGeneratedColumn('uuid') // ou sem 'uuid' se for numérico

id: string;

@Column()

nome: string;

@Column()

email: string;

@Column()

senha: string;

}

Ao fazer o registro via controller:

POST /usuarios/registrar

{

"nome": "João",

"email": "joao@email.com",

"senha": "senha123"

}

Você deve receber algo como:

{

"id": "f3a817b2-4534-4d8b-bc8f-5ec31465d542",

"nome": "João",

"email": "joao@email.com",

"senha": "$2a$10$..."

}

**⚠️ Recomendações importantes:**

1. **Nunca retorne a senha (mesmo criptografada)** em respostas da API.
   * Você pode usar uma função que exclui o campo antes de retornar:
2. const { senha, ...usuarioSemSenha } = usuarioSalvo;
3. return usuarioSemSenha;
4. Esse id (UUID ou numérico) será importante depois para:
   * Associar tarefas ao usuário
   * Fazer login e autenticação com JWT
   * Identificar quem criou o quê

Se quiser, posso te ajudar a:

* Esconder a senha da resposta da API
* Criar um retorno customizado (DTO de resposta)
* Criar o login com JWT (próximo passo)

Quer que prepare o **Dia 18: Autenticação com JWT (parte 2 - login e geração de token)**?

Claro! Aqui está a **aula completa do Dia 18 – Autenticação com JWT (parte 2)** em formato Markdown, pronta para você baixar, estudar ou implementar:

# 🗓️ Dia 18 – Autenticação com JWT (parte 2)

## 📚 Conteúdo

Neste dia, você irá:

- Criar o módulo de autenticação (`AuthModule`)

- Implementar um serviço que valide o login e gere um token JWT

- Criar a estratégia `JwtStrategy` com o Passport

- Proteger rotas usando `@UseGuards(AuthGuard())`

---

## 📦 Instalações necessárias

Caso ainda não tenha instalado os pacotes:

```bash

npm install @nestjs/jwt @nestjs/passport passport passport-jwt bcryptjs

**🔧 Atividades**

**1. Criar AuthModule e AuthService**

nest g module auth

nest g service auth

**2. Criar DTO de login**

Crie o arquivo login-auth.dto.ts:

export class LoginAuthDto {

email: string;

senha: string;

}

**3. Configurar JwtModule em auth.module.ts**

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

import { JwtModule } from '@nestjs/jwt';

import { PassportModule } from '@nestjs/passport';

import { AuthService } from './auth.service';

import { JwtStrategy } from './jwt.strategy';

import { UsuarioModule } from 'src/usuario/usuario.module';

@Module({

imports: [

PassportModule,

JwtModule.register({

secret: 'jwtseguro', // ou use process.env.JWT\_SECRET

signOptions: { expiresIn: '1h' },

}),

UsuarioModule,

],

providers: [AuthService, JwtStrategy],

exports: [AuthService],

})

export class AuthModule {}

**4. Criar AuthService**

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

import { JwtService } from '@nestjs/jwt';

import { UsuarioService } from 'src/usuario/usuario.service';

import \* as bcrypt from 'bcryptjs';

import { LoginAuthDto } from './dto/login-auth.dto';

@Injectable()

export class AuthService {

constructor(

private usuarioService: UsuarioService,

private jwtService: JwtService,

) {}

async login(loginDto: LoginAuthDto): Promise<{ access\_token: string }> {

const { email, senha } = loginDto;

const usuario = await this.usuarioService.findByEmail(email);

if (!usuario) {

throw new UnauthorizedException('Usuário não encontrado');

}

const senhaValida = await bcrypt.compare(senha, usuario.senha);

if (!senhaValida) {

throw new UnauthorizedException('Senha incorreta');

}

const payload = { id: usuario.id, email: usuario.email };

const token = this.jwtService.sign(payload);

return { access\_token: token };

}

}

**5. Criar AuthController**

nest g controller auth

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

import { AuthService } from './auth.service';

import { LoginAuthDto } from './dto/login-auth.dto';

@Controller('auth')

export class AuthController {

constructor(private readonly authService: AuthService) {}

@Post('login')

login(@Body() loginDto: LoginAuthDto) {

return this.authService.login(loginDto);

}

}

**6. Criar JwtStrategy**

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

import { PassportStrategy } from '@nestjs/passport';

import { ExtractJwt, Strategy } from 'passport-jwt';

@Injectable()

export class JwtStrategy extends PassportStrategy(Strategy) {

constructor() {

super({

jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(),

ignoreExpiration: false,

secretOrKey: 'jwtseguro', // ou process.env.JWT\_SECRET

});

}

async validate(payload: any) {

return { id: payload.id, email: payload.email };

}

}

**7. Criar JwtAuthGuard**

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

import { AuthGuard } from '@nestjs/passport';

@Injectable()

export class JwtAuthGuard extends AuthGuard('jwt') {}

**🧪 Exercício**

**✅ Proteger rotas da tarefa:**

No seu TarefasController:

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

import { JwtAuthGuard } from 'src/auth/jwt-auth.guard';

@UseGuards(JwtAuthGuard)

@Controller('tarefas')

export class TarefasController {

// ... rotas protegidas

}

**🔍 Testando no Postman**

1. Faça login:

POST /auth/login

{

"email": "usuario@email.com",

"senha": "senha123"

}

→ Copie o access\_token retornado.

1. Use o token nas demais requisições protegidas:

* Vá até **Headers**
* Adicione:

Authorization: Bearer SEU\_TOKEN

**✅ Conclusão**

Agora seu sistema possui autenticação JWT completa e rotas protegidas!  
No **Dia 19**, vamos **associar tarefas ao usuário autenticado** e começar a construir um sistema multiusuário.

Deseja que eu já prepare o conteúdo do **Dia 19 – Relacionamento entre usuário e tarefas (OneToMany)**?

---

Se quiser, posso gerar esse conteúdo como um arquivo `.md` para você baixar. Deseja o arquivo?

const usuario = await this.usuarioService.findByEmail(email);

nesse trecho do codigo recebo a mensage dizendo que A propriedade 'findByEmail' não existe no tipo 'UsuarioService'.

@Injectable()

export class UsuarioService {

constructor(

@InjectRepository(UsuarioEntity)

private readonly usuarioRepository: Repository<UsuarioEntity>,

) {}

async registrar(

nome: string,

email: string,

senha: string,

): Promise<UsuarioEntity> {

const salt = await bcrypt.genSalt();

const senhaCriptografada = await bcrypt.hash(senha, salt);

const novoUsuario = this.usuarioRepository.create({

nome,

email,

senha: senhaCriptografada,

});

return this.usuarioRepository.save(novoUsuario)

}

async buscarPorEmail(email: string): Promise<UsuarioEntity | null> {

return this.usuarioRepository.findOne({ where: { email } })

}

}

meu usuario service é esse que propriedade coloco no lugar de findByEmail ou como crio essa propriedade

Ótimo! A mensagem de erro está correta: o método findByEmail() não existe no seu UsuarioService. Mas você já tem um método com a mesma função, só com nome diferente: buscarPorEmail().

**✅ Duas opções para resolver:**

**✅ Opção 1: Alterar o nome no auth.service.ts**

No arquivo onde está usando findByEmail(), troque pelo método real:

const usuario = await this.usuarioService.buscarPorEmail(email);

**✅ Opção 2 (opcional): Criar um alias findByEmail()**

Se você quiser manter o nome findByEmail() (que é mais comum em projetos internacionais), você pode simplesmente criar um alias no seu UsuarioService:

findByEmail(email: string): Promise<UsuarioEntity | null> {

return this.buscarPorEmail(email);

}

Assim, você pode usar ambos os nomes se quiser.

**🔚 Resumo**

Se você **não quer mudar o nome do método**, **adicione o findByEmail**.  
Se você **quer usar o que já existe**, **substitua por buscarPorEmail()**.

Deseja que eu revise todo o seu auth.service.ts com base nisso?

@Controller('auth')

export class AuthController {

constructor(private readonly authService: AuthService) {}

@Post('login')

login(@Body() loginDto: LoginAuthDto) {

return this.authService.login(loginDto);

}

}

no meu AuthController no retorno estou recebendo a menssagem que A propriedade 'login' não existe no tipo 'AuthService'

@Injectable()

export class AuthService {

constructor(

private usuarioService: UsuarioService,

private jwtService: JwtService,

) {}

async login(loginDto: LoginAuthDto): Promise<{ access\_token: string }> {

const { email, senha } = loginDto;

const usuario = await this.usuarioService.findByEmail(email);

if (!usuario) {

throw new UnauthorizedException('Usuário não encontrado');

}

const senhaValida = await bcrypt.compare(senha, usuario.senha);

if (!senhaValida) {

throw new UnauthorizedException('Senha incorreta');

}

const payload = { id: usuario.id, email: usuario.email };

const token = this.jwtService.sign(payload);

return { access\_token: token };

}

}

essa é minha AuthService

A mensagem de erro **"A propriedade 'login' não existe no tipo 'AuthService'"** acontece provavelmente por **inconsistência de importações** ou **problema de build/descompasso entre o código e a versão transpilada**. Seu método login() está implementado corretamente na AuthService.

Aqui está um checklist para resolver isso:

**✅ 1. Verifique se salvou o arquivo auth.service.ts**

Pode parecer simples, mas se você estiver usando um editor como o VS Code, e o arquivo ainda estiver com alterações não salvas, o NestJS pode não reconhecer o método.

**✅ 2. Recompile o projeto**

Se está usando npm run start:dev, apenas salve tudo e ele reinicia. Mas se o erro persistir:

# Pare o servidor e limpe os arquivos compilados:

Ctrl + C

rm -rf dist

# Recompile tudo

npm run start:dev

**✅ 3. Verifique se AuthService está no providers de AuthModule**

No auth.module.ts:

@Module({

imports: [UsuarioModule, JwtModule.register({...})],

providers: [AuthService],

controllers: [AuthController],

})

export class AuthModule {}

**✅ 4. Verifique se o arquivo auth.service.ts está corretamente nomeado e exportado**

O seu AuthService precisa estar assim:

@Injectable()

export class AuthService {

// ...

}

E a exportação do arquivo deve ser correta. Certifique-se de que não há nenhum erro de digitação no nome do arquivo ou na importação.

**✅ 5. Verifique se a importação no AuthController está correta**

Em auth.controller.ts:

import { AuthService } from './auth.service'; // Caminho correto

**✅ 6. Confirme que não há outro AuthService vazio ou duplicado**

Às vezes um AuthService duplicado ou incompleto pode ser importado por engano.

**Conclusão**

Com base no seu código, o método login() **existe sim**, então esse erro é reflexo de um problema de build, importação errada ou arquivo salvo incorretamente.

Se quiser, posso te ajudar a revisar o auth.module.ts também para garantir que tudo esteja no lugar certo. Deseja isso?

Criar JwtStrategy

onde crio esse arquivo

Você deve criar o arquivo da **JwtStrategy** dentro do módulo de autenticação, geralmente no diretório auth/. O nome do arquivo normalmente é:

src/auth/jwt.strategy.ts

**✅ Como criar**

Você pode criar manualmente ou com o comando Nest CLI:

nest g service auth/jwt --flat

O --flat evita que seja criada uma subpasta chamada jwt.

Em seguida, edite o arquivo jwt.strategy.ts assim:

**📄 Exemplo de jwt.strategy.ts**

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

import { PassportStrategy } from '@nestjs/passport';

import { ExtractJwt, Strategy } from 'passport-jwt';

import { ConfigService } from '@nestjs/config';

@Injectable()

export class JwtStrategy extends PassportStrategy(Strategy) {

constructor(configService: ConfigService) {

super({

jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(),

secretOrKey: configService.get<string>('JWT\_SECRET'),

});

}

async validate(payload: any) {

// payload é o que foi definido em `jwtService.sign(payload)`

return { id: payload.id, email: payload.email };

}

}

**✅ Atualize o auth.module.ts**

Certifique-se de adicionar a JwtStrategy no providers do módulo:

@Module({

imports: [

PassportModule,

JwtModule.register({

secret: process.env.JWT\_SECRET,

signOptions: { expiresIn: '1h' },

}),

UsuarioModule,

],

providers: [AuthService, JwtStrategy],

controllers: [AuthController],

})

export class AuthModule {}

**🧪 Testando**

Depois disso, você poderá proteger rotas com:

@UseGuards(AuthGuard('jwt'))

Se quiser, posso gerar o conteúdo completo dos arquivos jwt.strategy.ts, auth.module.ts e um exemplo de proteção de rota. Deseja isso?

[Nest] 42012 - 08/07/2025, 18:12:49 ERROR [ExceptionHandler] TypeError: passport\_jwt\_1.ExtractJwt.fromAuthHeaderAsBeareToken is not a function

at new JwtStrategy (C:\meus\_projetos\aulas-nodeJS-nestJS-typeORM\nestJS-aulas\primeira\_semana\projeto-tarefas\src\auth\jwt.strategy.ts:9:40)

at Injector.instantiateClass (C:\meus\_projetos\aulas-nodeJS-nestJS-typeORM\nestJS-aulas\primeira\_semana\projeto-tarefas\node\_modules\@nestjs\core\injector\injector.js:373:19)

at callback (C:\meus\_projetos\aulas-nodeJS-nestJS-typeORM\nestJS-aulas\primeira\_semana\projeto-tarefas\node\_modules\@nestjs\core\injector\injector.js:65:45)

at process.processTicksAndRejections (node:internal/process/task\_queues:105:5)

at async Injector.resolveConstructorParams (C:\meus\_projetos\aulas-nodeJS-nestJS-typeORM\nestJS-aulas\primeira\_semana\projeto-tarefas\node\_modules\@nestjs\core\injector\injector.js:145:24)

at async Injector.loadInstance (C:\meus\_projetos\aulas-nodeJS-nestJS-typeORM\nestJS-aulas\primeira\_semana\projeto-tarefas\node\_modules\@nestjs\core\injector\injector.js:70:13)

at async Injector.loadProvider (C:\meus\_projetos\aulas-nodeJS-nestJS-typeORM\nestJS-aulas\primeira\_semana\projeto-tarefas\node\_modules\@nestjs\core\injector\injector.js:98:9)

at async C:\meus\_projetos\aulas-nodeJS-nestJS-typeORM\nestJS-aulas\primeira\_semana\projeto-tarefas\node\_modules\@nestjs\core\injector\instance-loader.js:56:13

at async Promise.all (index 4)

at async InstanceLoader.createInstancesOfProviders (C:\meus\_projetos\aulas-nodeJS-nestJS-typeORM\nestJS-aulas\primeira\_semana\projeto-tarefas\node\_modules\@nestjs\core\injector\instance-loader.js:55:9)

quando rodei o servidor veio esse erro

Esse erro ocorre por um **erro de digitação no nome da função** do passport-jwt.

No seu arquivo jwt.strategy.ts, provavelmente você escreveu:

ExtractJwt.fromAuthHeaderAsBeareToken()

⚠️ **Observe que está escrito Beare em vez de Bearer.**

**✅ Correção**

Altere para:

ExtractJwt.fromAuthHeaderAsBearerToken()

✅ fromAuthHeaderAsBearerToken é o nome correto da função do passport-jwt.

**📌 Exemplo correto do trecho:**

super({

jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(),

secretOrKey: configService.get<string>('JWT\_SECRET'),

});

Depois de corrigir isso, salve o arquivo e reinicie seu servidor com:

npm run start:dev

Se quiser, posso revisar seu arquivo completo jwt.strategy.ts. Deseja que eu verifique?

POST /auth/login

{

"email": "usuario@email.com",

"senha": "senha123"

}

quando rodei isso no postman

{

"message": "Usuário não encontrado",

"error": "Unauthorized",

"statusCode": 401

}

veio essa mensagem

A mensagem:

{

"message": "Usuário não encontrado",

"error": "Unauthorized",

"statusCode": 401

}

significa que o email enviado no corpo da requisição ("usuario@email.com") **não foi encontrado no banco de dados**.

**✅ Verificações que você deve fazer:**

**1. ✅ Verifique se o usuário foi registrado com esse e-mail**

Execute uma consulta no banco (via pgAdmin ou terminal):

SELECT \* FROM usuario WHERE email = 'usuario@email.com';

Se **nenhum resultado for retornado**, isso confirma que o usuário **ainda não existe no banco**.

**2. ✅ Verifique se você usou exatamente esse e-mail ao cadastrar**

Durante o registro (POST /usuarios/registrar), veja se o corpo da requisição tinha o mesmo e-mail:

{

"nome": "Usuario Teste",

"email": "usuario@email.com",

"senha": "senha123"

}

Cuidado com erros de digitação, letras maiúsculas ou espaços no e-mail.

**3. ✅ Verifique o método buscarPorEmail**

Seu UsuarioService deve ter esse método implementado assim:

async buscarPorEmail(email: string): Promise<UsuarioEntity | null> {

return this.usuarioRepository.findOne({ where: { email } });

}

E no AuthService, você deve estar chamando assim:

const usuario = await this.usuarioService.buscarPorEmail(email);

Se estiver tentando usar findByEmail, e o método real for buscarPorEmail, você vai receber undefined sempre.

**💡 Se o usuário ainda não foi cadastrado, faça o seguinte:**

1. Envie no Postman:

POST /usuarios/registrar

Content-Type: application/json

{

"nome": "Usuario Teste",

"email": "usuario@email.com",

"senha": "senha123"

}

1. Depois disso, tente novamente o login:

POST /auth/login

{

"email": "usuario@email.com",

"senha": "senha123"

}

Se mesmo assim não funcionar, posso revisar sua UsuarioEntity, UsuarioService, AuthService, e AuthController. Deseja isso?

{

"statusCode": 500,

"message": "Internal server error"

}veio essa mensagem

A resposta:

{

"statusCode": 500,

"message": "Internal server error"

}

significa que **algo quebrou internamente no seu backend**, provavelmente um erro de lógica ou execução no Nest.js. Como o erro é genérico, vamos te ajudar a depurar.

**✅ Passos para identificar o problema**

**1. ✅ Veja o log do terminal**

Logo acima dessa mensagem, o terminal deve ter mostrado a **stack trace** com algo parecido com:

[Nest] 12345 - ERROR [ExceptionHandler] ...

TypeError: ...

Por favor, copie e cole aqui a mensagem do terminal **completa**, para eu te ajudar com precisão.

**2. ✅ Verifique se você está chamando this.usuarioService.buscarPorEmail(...) corretamente**

Lembra que antes estava chamando findByEmail, que **não existia**? Se você não corrigiu isso, o erro interno pode ser:

TypeError: this.usuarioService.findByEmail is not a function

**✅ Correto:**

const usuario = await this.usuarioService.buscarPorEmail(email);

**3. ✅ Confirme se sua UsuarioEntity tem os campos esperados**

@Entity('usuario')

export class UsuarioEntity {

@PrimaryGeneratedColumn('uuid')

id: string;

@Column()

nome: string;

@Column({ unique: true })

email: string;

@Column()

senha: string;

}

**4. ✅ Verifique se o campo senha está vindo corretamente e compare com o hash**

Talvez o erro esteja no bcrypt.compare(...) se a senha no banco for null, por exemplo.

**🔧 Como capturar mais detalhes dos erros**

Adicione um console.log em auth.service.ts, no método login, por exemplo:

async login(loginDto: LoginAuthDto): Promise<{ access\_token: string }> {

const { email, senha } = loginDto;

const usuario = await this.usuarioService.buscarPorEmail(email);

console.log('Usuário encontrado:', usuario);

if (!usuario) {

throw new UnauthorizedException('Usuário não encontrado');

}

const senhaValida = await bcrypt.compare(senha, usuario.senha);

if (!senhaValida) {

throw new UnauthorizedException('Senha incorreta');

}

const payload = { id: usuario.id, email: usuario.email };

const token = this.jwtService.sign(payload);

return { access\_token: token };

}

**✅ Me envie:**

1. O erro exato que aparece no terminal
2. A versão final da sua função login(...) no AuthService

Assim consigo te ajudar com precisão.

Quer que eu te envie um exemplo 100% funcional do módulo de autenticação com JWT para comparar?

[Nest] 9016 - 08/07/2025, 18:24:14 ERROR [ExceptionsHandler] Error: secretOrPrivateKey must have a value

at module.exports [as sign] (C:\meus\_projetos\aulas-nodeJS-nestJS-typeORM\nestJS-aulas\primeira\_semana\projeto-tarefas\node\_modules\jsonwebtoken\sign.js:111:20)

at JwtService.sign (C:\meus\_projetos\aulas-nodeJS-nestJS-typeORM\nestJS-aulas\primeira\_semana\projeto-tarefas\node\_modules\@nestjs\jwt\dist\jwt.service.js:41:20)

at AuthService.login (C:\meus\_projetos\aulas-nodeJS-nestJS-typeORM\nestJS-aulas\primeira\_semana\projeto-tarefas\src\auth\auth.service.ts:28:35)

at async C:\meus\_projetos\aulas-nodeJS-nestJS-typeORM\nestJS-aulas\primeira\_semana\projeto-tarefas\node\_modules\@nestjs\core\router\router-execution-context.js:46:28

at async C:\meus\_projetos\aulas-nodeJS-nestJS-typeORM\nestJS-aulas\primeira\_semana\projeto-tarefas\node\_modules\@nestjs\core\router\router-proxy.js:9:17

no terminal vei esse erro

Esse erro:

Error: secretOrPrivateKey must have a value

significa que o **Nest.js não conseguiu encontrar a chave secreta (secret)** usada para **gerar o token JWT** com o JwtService.

**✅ Solução**

Você **precisa definir a chave secreta do JWT** no seu .env e configurar corretamente o JwtModule.

**✅ 1. Adicione isso ao seu .env:**

JWT\_SECRET=minha\_chave\_super\_secreta

JWT\_EXPIRES\_IN=3600s

**✅ 2. No AuthModule, configure o JwtModule assim:**

// auth.module.ts

import { JwtModule } from '@nestjs/jwt';

import { ConfigModule, ConfigService } from '@nestjs/config';

@Module({

imports: [

ConfigModule,

JwtModule.registerAsync({

imports: [ConfigModule],

useFactory: async (configService: ConfigService) => ({

secret: configService.get<string>('JWT\_SECRET'),

signOptions: {

expiresIn: configService.get<string>('JWT\_EXPIRES\_IN') || '1h',

},

}),

inject: [ConfigService],

}),

],

providers: [AuthService, JwtStrategy],

controllers: [AuthController],

})

export class AuthModule {}

Isso garante que o Nest vai usar a chave do .env para assinar os tokens.

**✅ 3. Reinicie seu servidor:**

Depois de fazer essas alterações e salvar o .env, reinicie com:

npm run start:dev

**⚠️ Importante**

Verifique se o ConfigModule está importado globalmente no AppModule:

ConfigModule.forRoot({

isGlobal: true,

}),

Se quiser, posso te passar um **exemplo completo de AuthModule com JWT** para comparar com o seu. Deseja?

[Nest] 115996 - 08/07/2025, 18:37:37 ERROR [ExceptionHandler] UnknownDependenciesException [Error]: Nest can't resolve dependencies of the AuthService (?, JwtService). Please make sure that the argument UsuarioService at index [0] is available in the AuthModule context.

Potential solutions:

- Is AuthModule a valid NestJS module?

- If UsuarioService is a provider, is it part of the current AuthModule?

- If UsuarioService is exported from a separate @Module, is that module imported within AuthModule?

@Module({

imports: [ /\* the Module containing UsuarioService \*/ ]

})

at Injector.lookupComponentInParentModules (C:\meus\_projetos\aulas-nodeJS-nestJS-typeORM\nestJS-aulas\primeira\_semana\projeto-tarefas\node\_modules\@nestjs\core\injector\injector.js:262:19)

at process.processTicksAndRejections (node:internal/process/task\_queues:105:5)

at async Injector.resolveComponentInstance (C:\meus\_projetos\aulas-nodeJS-nestJS-typeORM\nestJS-aulas\primeira\_semana\projeto-tarefas\node\_modules\@nestjs\core\injector\injector.js:215:33)

at async resolveParam (C:\meus\_projetos\aulas-nodeJS-nestJS-typeORM\nestJS-aulas\primeira\_semana\projeto-tarefas\node\_modules\@nestjs\core\injector\injector.js:129:38)

at async Promise.all (index 0)

at async Injector.resolveConstructorParams (C:\meus\_projetos\aulas-nodeJS-nestJS-typeORM\nestJS-aulas\primeira\_semana\projeto-tarefas\node\_modules\@nestjs\core\injector\injector.js:144:27)

at async Injector.loadInstance (C:\meus\_projetos\aulas-nodeJS-nestJS-typeORM\nestJS-aulas\primeira\_semana\projeto-tarefas\node\_modules\@nestjs\core\injector\injector.js:70:13)

at async Injector.loadProvider (C:\meus\_projetos\aulas-nodeJS-nestJS-typeORM\nestJS-aulas\primeira\_semana\projeto-tarefas\node\_modules\@nestjs\core\injector\injector.js:98:9)

at async C:\meus\_projetos\aulas-nodeJS-nestJS-typeORM\nestJS-aulas\primeira\_semana\projeto-tarefas\node\_modules\@nestjs\core\injector\instance-loader.js:56:13

at async Promise.all (index 3) {

type: 'AuthService',

context: {

index: 0,

dependencies: [

[class UsuarioService],

[class JwtService]

],

name: [class UsuarioService]

},

metadata: {

id: '8ec96714cc1e06a69de13'

},

moduleRef: {

id: 'cdad5f712e4e240ac4c38'

}

}

agora recebi esse erro

Esse erro significa que o **AuthService está tentando injetar o UsuarioService, mas o NestJS não sabe onde encontrar esse UsuarioService**.

**✅ Solução: importar o módulo onde o UsuarioService está declarado**

Você precisa **importar o UsuarioModule dentro do AuthModule**, pois é lá que o UsuarioService está registrado como provider.

**🔧 Passos para corrigir**

**1. Verifique se o UsuarioService está dentro de UsuarioModule:**

// usuario.module.ts

@Module({

imports: [TypeOrmModule.forFeature([UsuarioEntity])],

providers: [UsuarioService],

exports: [UsuarioService], // 👈 ESSENCIAL para que outros módulos possam usá-lo

})

export class UsuarioModule {}

**2. Importe o UsuarioModule dentro do AuthModule:**

// auth.module.ts

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

import { AuthService } from './auth.service';

import { UsuarioModule } from '../usuario/usuario.module'; // ✅ caminho correto para seu projeto

import { JwtModule } from '@nestjs/jwt';

import { JwtStrategy } from './jwt.strategy';

@Module({

imports: [

UsuarioModule, // ✅ necessário para usar UsuarioService

JwtModule.register({ secret: 'seuSegredo', signOptions: { expiresIn: '1h' } }),

],

providers: [AuthService, JwtStrategy],

controllers: [],

})

export class AuthModule {}

**✅ Explicação**

* exports: [UsuarioService] no UsuarioModule: permite que outros módulos usem o serviço.
* imports: [UsuarioModule] no AuthModule: torna o serviço disponível.

Depois de fazer isso, **reinicie seu servidor** com:

npm run start:dev

Se quiser, posso revisar seus arquivos auth.module.ts e usuario.module.ts se você colar aqui. Deseja?

Use o token nas demais requisições protegidas:

Vá até Headers

Adicione:

makefile

Copiar

Editar

Authorization: Bearer SEU\_TOKEN

me ajuda com isso

Claro! Aqui está um passo a passo simples para **usar o token JWT no Postman** (ou Insomnia) e acessar as **rotas protegidas com @UseGuards(AuthGuard())**:

**✅ Como usar o token JWT nas requisições protegidas**

**1. Faça login para obter o token**

* Envie um POST para:
* http://localhost:3000/auth/login
* Corpo da requisição (Body → raw → JSON):
* {
* "email": "usuario@email.com",
* "senha": "senha123"
* }
* A resposta será algo assim:
* {
* "access\_token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9..."
* }

**2. Copie o token (valor de access\_token)**

**3. Faça requisição para uma rota protegida (ex: GET /tarefas)**

* Método: GET
* URL:
* http://localhost:3000/tarefas

**4. Vá até a aba "Headers" no Postman**

* Adicione um novo cabeçalho (Header):

| **Chave** | **Valor** |
| --- | --- |
| Authorization | Bearer **cole\_seu\_token\_aqui** |

* Exemplo:
* Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...

**5. Envie a requisição**

Se o token for válido, a resposta virá normalmente.  
Caso o token esteja ausente ou inválido, você verá:

{

"statusCode": 401,

"message": "Unauthorized"

}

Se quiser, posso verificar se sua rota protegida está corretamente anotada com @UseGuards(AuthGuard()). Deseja?

Envie a requisição que requisição

Ótima pergunta!

A "requisição" mencionada no **passo 5** se refere a **qualquer rota da sua API que esteja protegida com autenticação**, ou seja, que esteja usando:

@UseGuards(AuthGuard())

**✅ Exemplos de requisições protegidas que você pode testar:**

**🔹 GET /tarefas**

Para listar todas as tarefas (caso tenha protegido essa rota).

* Método: GET
* URL:
* http://localhost:3000/tarefas

**🔹 POST /tarefas**

Para criar uma nova tarefa.

* Método: POST
* URL:
* http://localhost:3000/tarefas
* Body → raw → JSON:
* {
* "titulo": "Nova tarefa protegida",
* "descricao": "Descrição da tarefa"
* }

**Como enviar a requisição:**

Depois de preencher a rota e colar o token JWT no cabeçalho:

1. Vá ao Postman.
2. Escolha o método (GET, POST, etc.).
3. Coloque a URL (ex: http://localhost:3000/tarefas).
4. No menu **"Headers"**, adicione:
5. Authorization: Bearer SEU\_TOKEN\_AQUI
6. Clique em **"Send"**.

Se quiser, me diga qual rota você já protegeu com o @UseGuards(AuthGuard()) que te ajudo a testá-la diretamente.

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

import { AuthService } from './auth.service';

import { PassportModule } from '@nestjs/passport';

import { JwtModule, JwtService } from '@nestjs/jwt';

import { UsuarioModule } from 'src/usuario/usuario.module';

import { AuthController } from './auth.controller';

import { JwtStrategy } from './jwt.strategy';

import { ConfigModule, ConfigService } from '@nestjs/config';

@Module({

imports: [

UsuarioModule,

ConfigModule,

JwtModule.registerAsync({

imports: [ConfigModule],

useFactory: async (configService: ConfigService) => ({

secret: configService.get<string>('JWT\_SECRET'),

signOptions: {

expiresIn: configService.get<string>('JWT\_EXPIRES\_IN'),

},

}),

inject: [ConfigService],

}),

],

providers: [AuthService, JwtStrategy],

controllers: [AuthController],

})

export class AuthModule {}

aqui no alth.module.ts não precisa do PassportModule?

Sim, **precisa** sim!

Você deve importar o PassportModule no seu AuthModule, pois ele é necessário para usar os **Guards baseados em estratégias**, como o AuthGuard() com a JwtStrategy.

**✅ Atualização recomendada no seu AuthModule:**

Adicione PassportModule dentro do array imports:

@Module({

imports: [

UsuarioModule,

ConfigModule,

PassportModule, // ✅ Adicione isso aqui

JwtModule.registerAsync({

imports: [ConfigModule],

useFactory: async (configService: ConfigService) => ({

secret: configService.get<string>('JWT\_SECRET'),

signOptions: {

expiresIn: configService.get<string>('JWT\_EXPIRES\_IN'),

},

}),

inject: [ConfigService],

}),

],

providers: [AuthService, JwtStrategy],

controllers: [AuthController],

})

export class AuthModule {}

**💡 Por que precisa do PassportModule?**

Porque o AuthGuard('jwt') internamente usa o Passport, e o PassportModule precisa estar carregado no contexto do módulo para o Nest.js conseguir injetar corretamente.

Se você já tem o JwtStrategy criado e está usando o @UseGuards(AuthGuard('jwt')), o PassportModule é **obrigatório**.

Se quiser, posso revisar o seu JwtStrategy também para garantir que tudo esteja certo. Deseja?

export class LoginAuthDto {

email: string;

senha: string;

}

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

import { AuthService } from './auth.service';

import { LoginAuthDto } from './dto/login-auth.dto';

@Controller('auth')

export class AuthController {

constructor(private readonly authService: AuthService) {}

@Post('login')

login(@Body() loginDto: LoginAuthDto) {

return this.authService.login(loginDto);

}

}

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

import { Request } from 'express';

import { Observable } from 'rxjs';

@Injectable()

export class AuthGuard implements CanActivate {

canActivate(

context: ExecutionContext,

): boolean | Promise<boolean> | Observable<boolean> {

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

const authHeader = request.headers['authorization'];

if (authHeader === 'meutoken123') {

return true;

}

throw new UnauthorizedException('Token inválido ou ausente');

}

}

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

import { AuthService } from './auth.service';

import { PassportModule } from '@nestjs/passport';

import { JwtModule, JwtService } from '@nestjs/jwt';

import { UsuarioModule } from 'src/usuario/usuario.module';

import { AuthController } from './auth.controller';

import { JwtStrategy } from './jwt.strategy';

import { ConfigModule, ConfigService } from '@nestjs/config';

@Module({

imports: [

UsuarioModule,

ConfigModule,

PassportModule,

JwtModule.registerAsync({

imports: [ConfigModule],

useFactory: async (configService: ConfigService) => ({

secret: configService.get<string>('JWT\_SECRET'),

signOptions: {

expiresIn: configService.get<string>('JWT\_EXPIRES\_IN'),

},

}),

inject: [ConfigService],

}),

],

providers: [AuthService, JwtStrategy],

controllers: [AuthController],

})

export class AuthModule {}

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

import { JwtService } from '@nestjs/jwt';

import { UsuarioService } from 'src/usuario/usuario.service';

import \* as bcrypt from 'bcryptjs';

import { LoginAuthDto } from './dto/login-auth.dto';

@Injectable()

export class AuthService {

constructor(

private usuarioService: UsuarioService,

private jwtService: JwtService,

) {}

async login(loginDto: LoginAuthDto): Promise<{ access\_token: string }> {

const { email, senha } = loginDto;

const usuario = await this.usuarioService.findByEmail(email);

if (!usuario) {

throw new UnauthorizedException('Usuário não encontrado');

}

const senhaValida = await bcrypt.compare(senha, usuario.senha);

if (!senhaValida) {

throw new UnauthorizedException('Senha incorreta');

}

const payload = { sub: usuario.id, email: usuario.email };

const token = this.jwtService.sign(payload);

return { access\_token: token };

}

}

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

import { AuthGuard } from '@nestjs/passport';

@Injectable()

export class JwtAuthGuard extends AuthGuard('jwt') {}

import { Injectable } from "@nestjs/common";

import { PassportStrategy } from "@nestjs/passport";

import { ExtractJwt, Strategy } from 'passport-jwt';

@Injectable()

export class JwtStrategy extends PassportStrategy(Strategy) {

constructor() {

super({

jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(),

ignoreExpiration: false,

secretOrKey: 'jwtseguro', // ou process.env.JWT\_SECRET

});

}

async validate(payload: any) {

return { id: payload.id, email: payload.email }

}

}

estou te enviando todas os arquivos da pasta auth para você analizar e me dizer se está faltando alguma coisa

Ótimo! Você organizou muito bem a estrutura da pasta auth. Abaixo, vou revisar **cada parte do seu código** e te dizer:

* ✅ O que está **correto**
* ⚠️ O que **precisa de ajuste**
* 💡 Sugestões para melhorar ou evitar erros no futuro

**✅ LoginAuthDto**

export class LoginAuthDto {

email: string;

senha: string;

}

**✅ Correto.**  
Se quiser deixar mais robusto, pode adicionar validações com decorators do class-validator:

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

export class LoginAuthDto {

@IsEmail()

email: string;

@IsString()

@MinLength(6)

senha: string;

}

E lembre-se de usar o ValidationPipe global em main.ts:

app.useGlobalPipes(new ValidationPipe());

**✅ AuthController**

@Post('login')

login(@Body() loginDto: LoginAuthDto) {

return this.authService.login(loginDto);

}

**✅ Correto.** Vai funcionar bem se o método login() do AuthService estiver configurado.

**⚠️ AuthGuard personalizado**

export class AuthGuard implements CanActivate {

canActivate(...) {

...

}

}

**⚠️ Aqui está o guard antigo baseado em token fixo (meutoken123)**.

Você **não precisa mais disso** agora que está usando **JWT**.  
➡️ **Remova esse AuthGuard personalizado** se já estiver usando o JwtAuthGuard.

**✅ JwtAuthGuard**

@Injectable()

export class JwtAuthGuard extends AuthGuard('jwt') {}

**✅ Perfeito.** Isso que você deve usar para proteger suas rotas com JWT:

@UseGuards(JwtAuthGuard)

@Get('protegida')

getSegredo() {

return { segredo: 'Você está autenticado!' };

}

**⚠️ JwtStrategy**

super({

jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(),

secretOrKey: 'jwtseguro',

});

**⚠️ Problema:** Você está **colocando o segredo fixo** no código.

**✅ Solução:** Use o .env com ConfigService.

constructor(configService: ConfigService) {

super({

jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(),

secretOrKey: configService.get<string>('JWT\_SECRET'),

ignoreExpiration: false,

});

}

Não esqueça de **importar ConfigService** e **injetar no construtor**:

constructor(private configService: ConfigService) { ... }

E também de adicionar ConfigModule no AuthModule.

**⚠️ AuthService**

const usuario = await this.usuarioService.findByEmail(email);

Mas no UsuarioService o nome do método é buscarPorEmail.

**✅ Corrija para:**

const usuario = await this.usuarioService.buscarPorEmail(email);

**✅ AuthModule**

@Module({

imports: [

UsuarioModule,

ConfigModule,

PassportModule,

JwtModule.registerAsync({ ... }),

],

providers: [AuthService, JwtStrategy],

controllers: [AuthController],

})

**✅ Tudo certo aqui.**  
Só garanta que seu .env tem essas duas variáveis:

JWT\_SECRET=jwtseguro

JWT\_EXPIRES\_IN=3600s

**✅ Finalizando**

**✅ Para proteger rotas, use:**

@UseGuards(JwtAuthGuard)

@Get('tarefas')

findAllTarefas() { ... }

**✅ Verificação final de arquivos .env**

Garanta que seu .env contém:

JWT\_SECRET=jwtseguro

JWT\_EXPIRES\_IN=1h

**💡 Resumo:**

| **Item** | **Status** | **Ação** |
| --- | --- | --- |
| DTO de login | ✅ | Ok |
| Controller | ✅ | Ok |
| AuthService | ⚠️ | Corrigir nome do método (buscarPorEmail) |
| JwtStrategy | ⚠️ | Usar ConfigService ao invés de string fixa |
| Guard antigo | ❌ | Remover o guard com meutoken123 |
| JwtAuthGuard | ✅ | Usar para proteger as rotas |
| AuthModule | ✅ | Está bom |
| .env | ⚠️ | Verifique se JWT\_SECRET e JWT\_EXPIRES\_IN estão lá |

Se quiser, posso te ajudar a aplicar o @UseGuards(JwtAuthGuard) nas suas rotas de tarefas também. Deseja isso?

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

import { AuthService } from './auth.service';

import { PassportModule } from '@nestjs/passport';

import { JwtModule, JwtService } from '@nestjs/jwt';

import { UsuarioModule } from 'src/usuario/usuario.module';

import { AuthController } from './auth.controller';

import { JwtStrategy } from './jwt.strategy';

import { ConfigModule, ConfigService } from '@nestjs/config';

@Module({

imports: [

UsuarioModule,

ConfigModule,

PassportModule,

JwtModule.registerAsync({

imports: [ConfigModule],

useFactory: async (configService: ConfigService) => ({

secret: configService.get<string>('JWT\_SECRET'),

signOptions: {

expiresIn: configService.get<string>('JWT\_EXPIRES\_IN'),

},

}),

inject: [ConfigService],

}),

],

providers: [AuthService, JwtStrategy],

controllers: [AuthController],

})

export class AuthModule {}

aqui em auth.module não pecisa usar JwtService?

Ótima pergunta! ✅

No seu AuthModule, **você não precisa adicionar JwtService manualmente** em providers porque o JwtModule já **fornece** automaticamente o JwtService.

**✅ Ou seja:**

Este trecho já resolve tudo:

JwtModule.registerAsync({

imports: [ConfigModule],

useFactory: async (configService: ConfigService) => ({

secret: configService.get<string>('JWT\_SECRET'),

signOptions: {

expiresIn: configService.get<string>('JWT\_EXPIRES\_IN'),

},

}),

inject: [ConfigService],

}),

**📌 Conclusão:**

// Isto está certo:

@Module({

imports: [

UsuarioModule,

ConfigModule,

PassportModule,

JwtModule.registerAsync({ ... }),

],

providers: [AuthService, JwtStrategy], // 🔥 Não precisa colocar JwtService aqui

controllers: [AuthController],

})

export class AuthModule {}

Você pode usar o JwtService normalmente via injeção de dependência no AuthService:

constructor(

private usuarioService: UsuarioService,

private jwtService: JwtService, // <-- isso funciona pois foi injetado pelo JwtModule

) {}

Se quiser, posso te mostrar como usar esse token JWT em uma rota protegida com @UseGuards(JwtAuthGuard) e acessar os dados do usuário autenticado. Deseja isso?

Sim, exatamente! ✅

Quando você coloca @UseGuards(JwtAuthGuard) **acima do controlador inteiro**, como neste exemplo:

@UseGuards(JwtAuthGuard)

@Controller('tarefas')

export class TarefasController {

// ...

}

👉 **Todas as rotas dentro desse controller serão protegidas automaticamente**, ou seja:

* GET /tarefas
* GET /tarefas/:id
* POST /tarefas
* PATCH /tarefas/:id/status
* DELETE /tarefas/:id

Todas exigirão um **JWT válido no header Authorization: Bearer TOKEN**.

**✅ Alternativa: proteger rotas individualmente**

Se você quiser proteger **apenas algumas rotas específicas**, pode fazer assim:

@Controller('tarefas')

export class TarefasController {

constructor(private readonly tarefasService: TarefasService) {}

@UseGuards(JwtAuthGuard)

@Get()

findAll(): Promise<TarefaEntity[]> {

return this.tarefasService.findAll();

}

// outras rotas públicas ou protegidas

}

**📌 Testando com o Postman**

1. Faça login: POST /auth/login  
   Corpo:
2. {
3. "email": "seu@email.com",
4. "senha": "suaSenha123"
5. }
6. Copie o access\_token retornado.
7. Vá no **Postman**, na aba **Headers**, adicione:
8. Key: Authorization
9. Value: Bearer SEU\_TOKEN
10. Faça a requisição protegida.

Se quiser, posso te mostrar como acessar os dados do usuário autenticado dentro do controller ou service (usando @Request() ou @CurrentUser() com um decorator). Deseja isso?

no postman estou conseguindo fazer login, porque ja fiz o registro de um usuario e qundo cololocona rota post /auth/login recebo no body o token. Depois pego esse token e coloco-o em Authorization Berer, mas quando coloco na rota post /tarefa para criar uma tarefa recebo no body uma json com a mensagem Unauthorized, o que pode ser?

Ótimo! Você já está bem encaminhado. Se você está conseguindo fazer o login e recebendo o token corretamente, mas ao usá-lo em uma rota protegida (POST /tarefa) recebe Unauthorized, isso geralmente está relacionado a **alguns pontos específicos**. Vamos resolver juntos.

**✅ Verificações passo a passo**

**1. Você está usando o guard certo na rota protegida?**

A rota POST /tarefa precisa estar protegida com o @UseGuards(JwtAuthGuard):

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

import { JwtAuthGuard } from 'src/auth/jwt-auth.guard';

@UseGuards(JwtAuthGuard)

@Controller('tarefa')

export class TarefaController {

@Post()

criar(@Body() dto: CreateTarefaDto) {

// lógica

}

}

**2. Você registrou o JwtAuthGuard corretamente?**

Você já mostrou que criou o JwtAuthGuard, que está certo:

@Injectable()

export class JwtAuthGuard extends AuthGuard('jwt') {}

Só certifique-se de que está realmente usando **esse guard** e não o AuthGuard genérico ou um custom.

**3. Seu JwtStrategy está configurado corretamente?**

Você precisa garantir que está usando a estratégia correta com passport-jwt:

import { PassportStrategy } from '@nestjs/passport';

import { ExtractJwt, Strategy } from 'passport-jwt';

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

import { ConfigService } from '@nestjs/config';

@Injectable()

export class JwtStrategy extends PassportStrategy(Strategy) {

constructor(private configService: ConfigService) {

super({

jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(),

ignoreExpiration: false,

secretOrKey: configService.get<string>('JWT\_SECRET'),

});

}

async validate(payload: any) {

return { id: payload.sub, email: payload.email };

}

}

⚠️ Se você **não usar ConfigService**, o secretOrKey precisa bater com o valor usado no .env.

**4. Token está mesmo no formato correto no Postman?**

No **Postman**:

* Vá até a aba **Headers**
* Adicione (ou confirme):
* Key: Authorization
* Value: Bearer SEU\_TOKEN\_AQUI

⚠️ Certifique-se de que está com **"Bearer"** e **espaço antes do token**.

**5. O módulo Auth está funcionando corretamente?**

No seu AuthModule, certifique-se de que:

* PassportModule está importado
* JwtStrategy está no providers
* JwtModule está configurado com o mesmo JWT\_SECRET

**✅ Resumo final**

Se **tudo acima estiver ok** e ainda assim estiver dando erro, envie os seguintes pontos para eu analisar e corrigir rapidamente:

* TarefaController.ts
* JwtStrategy.ts
* JwtAuthGuard.ts
* Conteúdo do seu .env (esconda a senha se quiser)
* Resultado da rota /auth/login (token retornado)
* Print do header do Postman (ou o valor do Authorization que está usando)

Quer que eu analise algum desses agora?

Claro! Vamos montar a aula do **🗓️ Dia 19 – Relacionamentos com TypeORM**, com base no seu cronograma.

**🗓️ Dia 19 – Relacionamentos com TypeORM**

**🎯 Objetivo:**

Criar relacionamento entre as entidades Usuario e Tarefa, onde um usuário pode ter várias tarefas (1:N), e garantir que cada tarefa pertence a um único usuário autenticado.

**📚 Conteúdo Teórico**

**✅ O que é um relacionamento 1:N?**

No contexto de banco de dados relacional e ORM (como o TypeORM), uma relação 1:N significa:

* **Um usuário** pode ter **várias tarefas** (OneToMany).
* **Cada tarefa** pertence a **um único usuário** (ManyToOne).

**✅ Decorators TypeORM**

* @OneToMany() → Define o lado **um para muitos** (usuário → tarefas).
* @ManyToOne() → Define o lado **muitos para um** (tarefa → usuário).
* { eager: false } → Define se o dado relacionado deve ser carregado automaticamente.

**🧱 Estrutura das Entidades**

**🧑 Entidade UsuarioEntity**

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

import { TarefaEntity } from 'src/tarefa/tarefa.entity';

@Entity('usuarios')

export class UsuarioEntity {

@PrimaryGeneratedColumn('uuid')

id: string;

@Column()

nome: string;

@Column({ unique: true })

email: string;

@Column()

senha: string;

@OneToMany(() => TarefaEntity, tarefa => tarefa.usuario)

tarefas: TarefaEntity[];

}

**✅ Entidade TarefaEntity**

import {

Entity,

PrimaryGeneratedColumn,

Column,

CreateDateColumn,

ManyToOne,

} from 'typeorm';

import { TarefaStatus } from './tarefa-status.enum';

import { UsuarioEntity } from 'src/usuario/usuario.entity';

@Entity('tarefas')

export class TarefaEntity {

@PrimaryGeneratedColumn('uuid')

id: string;

@Column()

titulo: string;

@Column()

descricao: string;

@Column()

status: TarefaStatus;

@CreateDateColumn()

dataCriacao: Date;

@ManyToOne(() => UsuarioEntity, usuario => usuario.tarefas, { eager: false })

usuario: UsuarioEntity;

}

**🔧 Atividades Práticas**

**1. Criar relacionamento entre tarefas e usuário**

* Atualize as entidades conforme acima.

**2. Atualizar método de criação de tarefa (tarefas.service.ts)**

async create(dto: CreateTarefaDto, usuario: UsuarioEntity): Promise<TarefaEntity> {

const nova = this.tarefasRepository.create({

...dto,

status: TarefaStatus.ABERTA,

usuario, // atribuindo o usuário dono da tarefa

});

return this.tarefasRepository.save(nova);

}

**3. Extrair usuário do token JWT (tarefas.controller.ts)**

import { Controller, Post, Body, UseGuards, Req } from '@nestjs/common';

import { JwtAuthGuard } from 'src/auth/jwt-auth.guard';

import { Request } from 'express';

@UseGuards(JwtAuthGuard)

@Controller('tarefas')

export class TarefasController {

constructor(private readonly tarefaService: TarefaService) {}

@Post()

async criar(@Body() dto: CreateTarefaDto, @Req() req: Request) {

const usuario = req.user;

return this.tarefaService.create(dto, usuario);

}

}

📌 **Dica:** o req.user é preenchido automaticamente pelo JwtStrategy.

**4. Buscar tarefas por usuário logado**

async findByUsuario(usuario: UsuarioEntity): Promise<TarefaEntity[]> {

return this.tarefasRepository.find({

where: { usuario: { id: usuario.id } },

});

}

No controller:

@Get()

async listar(@Req() req: Request) {

const usuario = req.user;

return this.tarefaService.findByUsuario(usuario);

}

**🧪 Exercício**

**Implementar** o endpoint GET /tarefas para listar somente as tarefas do usuário autenticado, usando JwtAuthGuard.

**✅ Conclusão**

Ao final dessa aula, você terá:

* Criado relacionamento entre UsuarioEntity e TarefaEntity
* Vinculado cada nova tarefa ao usuário autenticado via token
* Garantido que as tarefas listadas sejam apenas as do usuário logado

**💡 Próximo passo sugerido (Dia 20):**

**Upload de arquivos ou avatar do usuário** (usando @nestjs/platform-express + multer)

Deseja que eu elabore a aula do dia 20 também?

4. Buscar tarefas por usuário logado

ts

Copiar

Editar

async findByUsuario(usuario: UsuarioEntity): Promise<TarefaEntity[]> {

return this.tarefasRepository.find({

where: { usuario: { id: usuario.id } },

});

}

essa inclusão é no arquivo tarefas.service

Sim, exatamente! ✅

Esse método findByUsuario deve ser incluído dentro do arquivo tarefas.service.ts. Ele permite buscar todas as tarefas vinculadas ao usuário autenticado (baseado no usuario.id).

**📄 Exemplo completo no tarefas.service.ts:**

@Injectable()

export class TarefasService {

constructor(

@InjectRepository(TarefaEntity)

private readonly tarefasRepository: Repository<TarefaEntity>,

) {}

// Buscar todas as tarefas de um usuário

async findByUsuario(usuario: UsuarioEntity): Promise<TarefaEntity[]> {

return this.tarefasRepository.find({

where: { usuario: { id: usuario.id } },

});

}

// (outros métodos como create, updateStatus etc. ficam aqui também)

}

**E no tarefas.controller.ts, você usa assim:**

@UseGuards(JwtAuthGuard)

@Get()

async listar(@Req() req: Request) {

const usuario = req.user as UsuarioEntity;

return this.tarefaService.findByUsuario(usuario);

}

📌 **Importante:** Para que req.user tenha o id e email, sua JwtStrategy precisa retornar esses dados no validate:

validate(payload: any) {

return { id: payload.sub, email: payload.email };

}

Se precisar revisar essa parte ou montar o Dia 20, é só me avisar!

@Get()

async listar(@Req() req: Request) {

const usuario = req.user;

return this.tarefasService.findByUsuario(usuario);

}

@Post()

create(@Body() dto: CreateTarefaDto, @Req() req: Request) {

const usuario = req.user;

return this.tarefasService.create(dto, usuario);

}

tanto no get quanto no post no trecho de codigo const usuario = req.user; estou recebendo uma mensagem de que A propriedade 'user' não existe no tipo 'Request<ParamsDictionary, any, any, ParsedQs, Record<string, any>>'.

Excelente pergunta — esse erro ocorre porque o TypeScript não sabe que você estendeu o objeto Request para incluir a propriedade user, que foi adicionada pela JwtStrategy com o Passport.

**✅ Solução 1 (Recomendada): Criar um tipo customizado para Request**

Crie um tipo que estende o tipo do Express e adiciona o campo user:

**1. Crie um novo arquivo chamado user-request.interface.ts:**

import { Request } from 'express';

import { UsuarioEntity } from 'src/usuario/usuario.entity';

export interface UserRequest extends Request {

user: UsuarioEntity;

}

**2. No seu tarefas.controller.ts, importe e use esse tipo:**

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

import { TarefasService } from './tarefas.service';

import { JwtAuthGuard } from 'src/auth/jwt-auth.guard';

import { UserRequest } from 'src/auth/user-request.interface';

import { CreateTarefaDto } from './dto/create-tarefa.dto';

@Controller('tarefas')

@UseGuards(JwtAuthGuard)

export class TarefasController {

constructor(private readonly tarefasService: TarefasService) {}

@Get()

async listar(@Req() req: UserRequest) {

const usuario = req.user;

return this.tarefasService.findByUsuario(usuario);

}

@Post()

async create(@Body() dto: CreateTarefaDto, @Req() req: UserRequest) {

const usuario = req.user;

return this.tarefasService.create(dto, usuario);

}

}

**✅ Solução 2 (rápida, mas não ideal): usar *type assertion***

Se quiser uma solução rápida apenas para testes, você pode forçar o cast com as any:

const usuario = (req as any).user;

Mas **não é recomendado** para produção, pois você perde a verificação de tipo e segurança do TypeScript.

Se quiser, posso te ajudar agora a montar o Dia 20 – Filtros e busca avançada ou qualquer outra parte do projeto. Deseja continuar?

Sim ✅, o exercício **"Implementar o endpoint GET /tarefas para listar somente as tarefas do usuário autenticado, usando JwtAuthGuard"** é exatamente **o que essa implementação abaixo faz**, com uma pequena observação:

@Get()

async listar(@Req() req: Request) {

const usuario = req.user;

return this.tarefaService.findByUsuario(usuario);

}

**⚠️ Observação:**

O seu código está correto **em lógica**, **mas ainda precisa de duas pequenas melhorias** para funcionar 100% com segurança e tipagem correta:

**✅ 1. Proteger com o guard JwtAuthGuard**

Você precisa garantir que só usuários autenticados possam acessar essa rota, então adicione o decorador @UseGuards:

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

import { JwtAuthGuard } from 'src/auth/jwt-auth.guard';

@UseGuards(JwtAuthGuard)

@Get()

async listar(@Req() req: Request) {

const usuario = req.user;

return this.tarefaService.findByUsuario(usuario);

}

**✅ 2. Tipagem correta com UserRequest**

Você precisa usar o tipo personalizado (interface que estende Request) para que o TypeScript reconheça a propriedade user:

import { UserRequest } from 'src/auth/user-request.interface';

@UseGuards(JwtAuthGuard)

@Get()

async listar(@Req() req: UserRequest) {

const usuario = req.user;

return this.tarefaService.findByUsuario(usuario);

}

**✅ Resumo:**

O seu código:

@Get()

async listar(@Req() req: Request) {

const usuario = req.user;

return this.tarefaService.findByUsuario(usuario);

}

🔁 **É o mesmo exercício** do enunciado:  
**"GET /tarefas para listar tarefas do usuário autenticado com JwtAuthGuard"**.

💡 Só precisa:

* Garantir o uso de @UseGuards(JwtAuthGuard)
* E usar o tipo UserRequest para evitar erro de tipagem.

Se quiser, posso revisar sua função findByUsuario ou te ajudar a implementar o @Post() também usando o usuário logado. Deseja seguir?

como testa no postman se deu certo as tarefas terem sido salvas apenas para o usuario logado

Ótima pergunta! Para testar no **Postman** se as tarefas estão sendo salvas **somente para o usuário logado**, você pode seguir este **passo a passo prático**:

**✅ Etapas para testar corretamente no Postman**

**1. Registrar dois usuários diferentes**

Envie uma requisição POST para /usuarios/registrar:

**🔹 Usuário A**

POST http://localhost:3000/usuarios/registrar

Body (JSON):

{

"nome": "João",

"email": "joao@email.com",

"senha": "senha123"

}

**🔹 Usuário B**

POST http://localhost:3000/usuarios/registrar

Body (JSON):

{

"nome": "Maria",

"email": "maria@email.com",

"senha": "senha123"

}

**2. Fazer login com cada usuário e obter o token**

**🔹 Login Usuário A**

POST http://localhost:3000/auth/login

Body:

{

"email": "joao@email.com",

"senha": "senha123"

}

🔁 Copie o access\_token retornado.

**🔹 Login Usuário B**

POST http://localhost:3000/auth/login

Body:

{

"email": "maria@email.com",

"senha": "senha123"

}

🔁 Copie também o token da Maria.

**3. Criar tarefas usando cada token**

**🔹 Criar tarefa com token de João**

POST http://localhost:3000/tarefas

Headers:

Authorization: Bearer TOKEN\_DO\_JOAO

Body:

{

"titulo": "Estudar NestJS",

"descricao": "Aprofundar em banco de dados",

"status": "ABERTA"

}

**🔹 Criar tarefa com token de Maria**

POST http://localhost:3000/tarefas

Headers:

Authorization: Bearer TOKEN\_DA\_MARIA

Body:

{

"titulo": "Fazer compras",

"descricao": "Comprar legumes",

"status": "ABERTA"

}

**4. Buscar tarefas de cada usuário**

**🔹 Com token de João**

GET http://localhost:3000/tarefas

Headers:

Authorization: Bearer TOKEN\_DO\_JOAO

✅ Deve **retornar apenas as tarefas criadas por João**.

**🔹 Com token de Maria**

GET http://localhost:3000/tarefas

Headers:

Authorization: Bearer TOKEN\_DA\_MARIA

✅ Deve **retornar apenas as tarefas criadas por Maria**.

**🧪 Conclusão**

Se cada usuário só conseguir **ver suas próprias tarefas**, e ao criar tarefas elas só forem listadas quando **autenticado com o mesmo token**, então ✅ está funcionando corretamente!

Se quiser, posso te ajudar a implementar o **Dia 20 – Filtros e busca por status ou título** para refinar ainda mais a aplicação. Deseja continuar?

não deu certo a lógica, porque registrei dois usuários, loguei nos dois usuarios e criei tarefas para cada um deles, mas na hora de mostrar as tarefas de cada um deles não mostra separadas as tarefas tanto para joão quanto para maria vem todas as tarefas juntas

Se ao listar as tarefas **todos os usuários veem todas as tarefas**, isso significa que o relacionamento entre **Tarefa** e **Usuário** não está funcionando corretamente — provavelmente as tarefas **não estão sendo associadas ao usuário logado na hora da criação**, ou o campo usuarioId não está sendo salvo corretamente no banco.

Vamos resolver isso passo a passo ✅

**✅ PASSO 1 – Verifique se a entidade TarefaEntity tem o relacionamento**

// tarefa.entity.ts

@ManyToOne(() => UsuarioEntity, usuario => usuario.tarefas, { eager: false })

usuario: UsuarioEntity;

E **no banco de dados**, isso deve gerar a coluna usuarioId como chave estrangeira.

**✅ PASSO 2 – Garanta que o usuário está sendo atribuído corretamente na criação da tarefa**

**📄 tarefas.service.ts**

async create(dto: CreateTarefaDto, usuario: UsuarioEntity): Promise<TarefaEntity> {

const nova = this.tarefasRepository.create({

...dto,

status: TarefaStatus.ABERTA,

usuario: usuario, // <- aqui é essencial

});

return this.tarefasRepository.save(nova);

}

Se isso estiver faltando, o campo usuarioId não será preenchido.

**✅ PASSO 3 – Verifique o método de listagem de tarefas**

async findByUsuario(usuario: UsuarioEntity): Promise<TarefaEntity[]> {

return this.tarefasRepository.find({

where: { usuario: { id: usuario.id } },

relations: ['usuario'], // opcional, mas útil para debug

});

}

**✅ PASSO 4 – Verifique o uso correto de @Request ou @User**

**📄 tarefas.controller.ts**

Se estiver usando isso:

@Get()

@UseGuards(AuthGuard('jwt'))

async listar(@Req() req: Request) {

const usuario = req.user as UsuarioEntity; // type assertion

return this.tarefasService.findByUsuario(usuario);

}

**ou melhor ainda**, crie um decorator personalizado @User() para pegar o usuário diretamente do token decodificado:

**✅ Crie um arquivo: usuario.decorator.ts**

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

export const UsuarioLogado = createParamDecorator(

(data: unknown, ctx: ExecutionContext) => {

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

return request.user;

},

);

**E no controller:**

@Get()

@UseGuards(AuthGuard('jwt'))

listar(@UsuarioLogado() usuario: UsuarioEntity) {

return this.tarefasService.findByUsuario(usuario);

}

**✅ PASSO 5 – Verifique o token JWT e o JwtStrategy**

Na sua JwtStrategy, o payload precisa ter sub (ID do usuário):

async validate(payload: any) {

return { id: payload.sub, email: payload.email };

}

**🧪 Teste novamente:**

1. Faça login com cada usuário
2. Crie tarefas com cada token
3. Liste tarefas com cada token

Agora deve funcionar corretamente: **cada usuário verá apenas suas próprias tarefas**.

Se quiser, posso revisar o código completo da entidade Tarefa, Usuario, serviço e controller para garantir que esteja tudo certo. Deseja que eu revise?

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

export const UsuarioLogado = createParamDecorator(

(data: unknown, ctx: ExecutionContext) => {

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

return request.user;

},

);

onde uso esse decorator

Você usa o decorator @UsuarioLogado() **nos controladores** para obter o usuário autenticado diretamente da requisição (sem precisar acessar manualmente req.user), desde que a rota esteja protegida com o AuthGuard.

**✅ Exemplo de uso no TarefasController**

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

import { TarefasService } from './tarefas.service';

import { CreateTarefaDto } from './dto/create-tarefa.dto';

import { UsuarioLogado } from 'src/auth/usuario.decorator';

import { UsuarioEntity } from 'src/usuario/usuario.entity';

import { AuthGuard } from '@nestjs/passport';

@Controller('tarefas')

@UseGuards(AuthGuard('jwt')) // garante que só usuários autenticados acessem

export class TarefasController {

constructor(private readonly tarefasService: TarefasService) {}

@Get()

async listar(@UsuarioLogado() usuario: UsuarioEntity) {

return this.tarefasService.findByUsuario(usuario);

}

@Post()

async create(

@Body() dto: CreateTarefaDto,

@UsuarioLogado() usuario: UsuarioEntity,

) {

return this.tarefasService.create(dto, usuario);

}

}

**📝 Explicação**

* O NestJS, via o AuthGuard('jwt'), injeta o user (decodificado do token JWT) na requisição.
* O decorator @UsuarioLogado() recupera esse user do request.
* Assim, você acessa o usuário autenticado com mais clareza e segurança no código.

Se ainda não estiver funcionando corretamente, envie seu JwtStrategy e o que está sendo retornado no validate() para garantir que o request.user tenha as informações corretas (por exemplo: o id do usuário).

Deseja que eu revise também isso?

fiz todas as mudanças acima mas ainda está listando tudo junto tanto as tarefas de maria quanto de joão

Se mesmo com o decorator @UsuarioLogado() as tarefas continuam sendo listadas para **todos os usuários**, o problema está provavelmente no **relacionamento entre TarefaEntity e UsuarioEntity** ou na forma como você está **salvando** e **buscando** as tarefas.

Vamos verificar ponto a ponto para garantir que tudo esteja correto:

**✅ 1. Relacionamento em TarefaEntity**

Verifique se sua entidade TarefaEntity tem o relacionamento com o usuário **corretamente definido**:

import { ManyToOne } from 'typeorm';

import { UsuarioEntity } from 'src/usuario/usuario.entity';

@ManyToOne(() => UsuarioEntity, usuario => usuario.tarefas, { eager: false })

usuario: UsuarioEntity;

**✅ 2. Relacionamento em UsuarioEntity (opcional, mas bom ter)**

@OneToMany(() => TarefaEntity, tarefa => tarefa.usuario)

tarefas: TarefaEntity[];

**✅ 3. Ao criar a tarefa, vincular o usuário**

No tarefas.service.ts, a função create **deve** associar o usuário corretamente:

async create(dto: CreateTarefaDto, usuario: UsuarioEntity): Promise<TarefaEntity> {

const tarefa = this.tarefasRepository.create({

...dto,

status: TarefaStatus.ABERTA,

usuario, // <-- importante!

});

return this.tarefasRepository.save(tarefa);

}

**✅ 4. Buscar apenas as tarefas do usuário logado**

A função findByUsuario **deve** filtrar corretamente:

async findByUsuario(usuario: UsuarioEntity): Promise<TarefaEntity[]> {

return this.tarefasRepository.find({

where: { usuario: { id: usuario.id } },

});

}

❗ Atenção: isso **só funciona** se o campo usuario estiver corretamente populado no banco e com os relacionamentos ativados no TypeORM.

**✅ 5. Verificar se a tabela tarefas no banco tem a coluna usuarioId preenchida**

Execute esse SQL no seu PostgreSQL:

SELECT id, titulo, usuarioId FROM tarefas;

Se a coluna usuarioId estiver **vazia (null)**, quer dizer que ao salvar a tarefa você **não associou o usuário corretamente**.

**✅ 6. Dica extra: use console.log(usuario) no create() e no findByUsuario()**

Exemplo:

console.log('Usuário recebido:', usuario);

Isso ajuda a garantir que você está recebendo o usuário corretamente no serviço.

Se você me mostrar:

1. O conteúdo da sua entidade TarefaEntity
2. O trecho onde a tarefa é criada (create)
3. O retorno do SQL: SELECT id, titulo, "usuarioId" FROM tarefas;

Eu consigo verificar e corrigir rapidamente com você.

Deseja me mostrar esses trechos?

Obrigado pelas informações detalhadas. Pelo que você enviou, o problema **está claro**: sua tabela tarefas no banco de dados **não existe** ou **tem outro nome**, por isso não conseguimos verificar se o campo usuarioId está sendo salvo corretamente — e é bem possível que ele **não esteja mesmo sendo persistido**.

Vamos resolver isso em **3 etapas rápidas**:

**✅ 1. Verifique se a tabela foi criada com nome diferente**

No seu TarefaEntity, você **não definiu o nome da tabela**, então o TypeORM vai usar o nome padrão: tarefa\_entity (nome da classe, em snake\_case e pluralizado se configurado).

🔍 Rode o seguinte comando no seu PostgreSQL para listar as tabelas existentes:

\dt

Ou se estiver usando DBeaver, PgAdmin ou similar, veja no painel esquerdo em "Tables".

Procure um nome como:

* tarefa\_entity
* tarefa
* tarefas (só se você colocou @Entity('tarefas'), o que **você não fez**)

Se achar a tabela correta, rode o SQL assim:

SELECT id, titulo, "usuarioId" FROM tarefa\_entity;

**✅ 2. (Opcional, mas recomendado) Renomeie explicitamente a tabela**

Você pode forçar o nome da tabela com o @Entity('tarefas'), como abaixo:

@Entity('tarefas')

export class TarefaEntity {

...

}

Depois disso, **recrie a tabela** ou use migrations para aplicar a mudança no banco.

**✅ 3. Garanta que o campo usuario está sendo salvo**

Se você rodar o SQL corretamente e ainda ver que usuarioId está vindo NULL, então o problema está na criação da tarefa.

Nesse caso, adicione um console.log antes de salvar:

console.log('Criando tarefa para usuário:', usuario);

const novaTarefa = this.tarefasRepository.create({

...dto,

status: TarefaStatus.ABERTA,

usuario,

});

return this.tarefasRepository.save(novaTarefa);

Verifique se o usuario está com id válido.

**✅ 4. Teste o filtro no serviço**

E a função findByUsuario() está correta?

async findByUsuario(usuario: UsuarioEntity): Promise<TarefaEntity[]> {

return this.tarefasRepository.find({

where: {

usuario: {

id: usuario.id,

},

},

relations: ['usuario'], // só se precisar dos dados do usuário

});

}

**✅ 5. Teste final no Postman**

1. Faça login com o usuário A → receba o token.
2. Crie 2 tarefas usando esse token.
3. Faça login com o usuário B → receba outro token.
4. Crie 1 tarefa com esse segundo token.
5. Use o endpoint GET /tarefas com o token do usuário A → deve retornar só as 2 tarefas dele.
6. Use o mesmo endpoint com o token do usuário B → deve retornar só a tarefa dele.

Se mesmo após isso não funcionar, me envie:

* O nome exato da tabela onde as tarefas estão sendo salvas (via \dt ou SHOW TABLES)
* O resultado da query SELECT \* FROM <nome\_da\_tabela> LIMIT 10;

Assim resolvemos com precisão. Deseja que eu te ajude com a criação das migrations ou exclusão/recriação da tabela?

depois que rodei o comando SELECT id, titulo, usuarioId FROM tarefa\_entity; no postgresql, veio esse erro:

ERROR: coluna "usuarioid" não existe

LINE 1: SELECT id, titulo, usuarioId FROM tarefa\_entity;

^

HINT: Talvez você queira fazer referência à coluna "tarefa\_entity.usuarioId".

ERRO: coluna "usuarioid" não existe

SQL state: 42703

Character: 20

Esse erro significa que a coluna usuarioId **existe sim**, mas o PostgreSQL é **sensível a maiúsculas e minúsculas** quando você **não usa aspas duplas**. Ou seja:

* usuarioId ≠ usuarioid
* Se a coluna foi criada com usuarioId (com "I" maiúsculo), então você **precisa usar aspas duplas**.

**✅ Como resolver**

Rode o comando assim:

SELECT id, titulo, "usuarioId" FROM tarefa\_entity;

⚠️ As aspas duplas são **obrigatórias** nesse caso, porque o TypeORM criou a coluna com letra maiúscula (usuarioId), e o PostgreSQL trata identificadores com maiúsculas de forma especial.

**🧠 Explicação extra (caso queira entender melhor)**

No PostgreSQL:

* Sem aspas → tudo é transformado para **minúsculo**.
* Com aspas → respeita exatamente como está.

Então:

SELECT usuarioid FROM tarefa\_entity; -- erro

SELECT "usuarioId" FROM tarefa\_entity; -- funciona

**🎯 O que fazer agora**

1. Rode o comando corrigido:
2. SELECT id, titulo, "usuarioId" FROM tarefa\_entity;
3. Veja se os valores de "usuarioId" estão aparecendo corretos (devem ter os **ids dos usuários**).
4. Se a coluna "usuarioId" estiver **vazia (NULL)**, isso confirma que você **não está salvando o usuário ao criar a tarefa**, e precisará corrigir o método create no TarefasService.

Se quiser, posso analisar novamente seu método create() para garantir que o usuário está sendo salvo corretamente. Deseja me mostrar esse trecho?

quando tento trocar o nome da tabela no postgresql recebo a mensagem relaação"tarefas" já existe

Essa mensagem:

relação "tarefas" já existe

significa que você tentou **renomear uma tabela para um nome que já está em uso**.

**✅ Como resolver**

Você **não pode renomear** uma tabela para um nome que **já existe**. Para resolver, você tem 3 opções:

**🔁 Opção 1: Deletar a tabela antiga "tarefas" (⚠️ PERDE os dados)**

Se você **não precisa da tabela antiga**, pode deletá-la:

DROP TABLE tarefas;

Depois disso, você pode renomear a nova tabela:

ALTER TABLE tarefa\_entity RENAME TO tarefas;

**📦 Opção 2: Escolher outro nome**

Se você quer manter as duas tabelas, use outro nome:

ALTER TABLE tarefa\_entity RENAME TO tarefas\_nova;

**🛠️ Opção 3: Verificar com segurança se existe e só então deletar**

DROP TABLE IF EXISTS tarefas;

**❗ Dica importante para o TypeORM**

Se você estiver usando TypeORM com:

@Entity('tarefas')

export class TarefaEntity { ... }

E a tabela tarefas **já existe no banco**, o TypeORM **vai tentar recriar** se synchronize: true, e isso pode causar conflitos.

**Solução:**

* Use synchronize: false em produção.
* Use migration para controle mais seguro de mudanças.

Se quiser, posso revisar suas entidades e o AppModule para garantir que a estrutura está correta. Deseja?

SELECT id, titulo, "usuarioId" FROM tarefas; com esse comando verifiquei que os ids que vem dos usuários são diferentes mas mesmo assim no postman quando coloco para listar elas vem de todos os usuários

Ótimo! Se os usuarioId estão aparecendo corretamente na tabela tarefas, então **o relacionamento entre tarefa e usuário está salvo corretamente no banco**.  
Ou seja, **o problema está no código que busca as tarefas**, que provavelmente **não está filtrando pelo usuarioId corretamente**.

**✅ Verificações e correções necessárias:**

**1. Sua entidade TarefaEntity precisa estar assim:**

@ManyToOne(() => UsuarioEntity, (usuario) => usuario.tarefas, { eager: false })

usuario: UsuarioEntity;

Isso garante que a tarefa guarda referência ao usuário.

**2. O método findByUsuario em TarefasService deve estar assim:**

async findByUsuario(usuario: UsuarioEntity): Promise<TarefaEntity[]> {

return this.tarefasRepository.find({

where: {

usuario: { id: usuario.id },

},

relations: ['usuario'], // opcional, se quiser carregar os dados do usuário junto

});

}

⚠️ **Se você não colocar where: { usuario: { id: ... } }**, o Nest/TypeORM vai trazer **todas as tarefas**, inclusive de outros usuários.

**3. Use o decorador @UsuarioLogado() no controller**

Evite usar @Req() manualmente. Use o decorador que você já criou:

@Get()

@UseGuards(JwtAuthGuard)

listar(@UsuarioLogado() usuario: UsuarioEntity) {

return this.tarefasService.findByUsuario(usuario);

}

@Post()

@UseGuards(JwtAuthGuard)

create(@Body() dto: CreateTarefaDto, @UsuarioLogado() usuario: UsuarioEntity) {

return this.tarefasService.create(dto, usuario);

}

**4. No método create do TarefasService**

Garanta que a tarefa salva tem o usuário atribuído:

async create(dto: CreateTarefaDto, usuario: UsuarioEntity): Promise<TarefaEntity> {

const nova = this.tarefasRepository.create({

...dto,

status: TarefaStatus.ABERTA,

usuario,

});

return this.tarefasRepository.save(nova);

}

**5. Token do usuário correto no Postman**

Verifique se:

* Você está usando **um token diferente** para cada usuário.
* O token está no cabeçalho como:

Authorization: Bearer SEU\_TOKEN\_AQUI

Faça o login com João, copie o token e teste /tarefas.  
Depois, faça o login com Maria e teste /tarefas com o token dela.  
Os resultados **devem ser diferentes**.

Se quiser, posso revisar seus arquivos tarefas.controller.ts e tarefas.service.ts para garantir que tudo está certo. Deseja?

@UseGuards(JwtAuthGuard)

@Controller('tarefas') // prefixo: todas as rotas começam com /tarefas

export class TarefasController {

constructor(private readonly tarefasService: TarefasService) {}

// GET /tarefas?status=ABERTA&termo=algo

@Get()

findAll(): Promise<Tarefas[]> {

return this.tarefasService.findAll();

}

// GET /tarefas/:id

@Get(':id')

findById(@Param('id') id: string): Promise<Tarefas> {

return this.tarefasService.findById(id);

}

@Get()

async listar(@UsuarioLogado() usuario: UsuarioEntity) {

return this.tarefasService.findByUsuario(usuario);

}

@Post()

async create(

@Body() dto: CreateTarefaDto,

@UsuarioLogado() usuario: UsuarioEntity,

) {

return this.tarefasService.create(dto, usuario);

}

@Patch(':id/status')

updateStatus(

@Param('id') id: string,

@Body('status', TarefaStatusValidationPipe) status: TarefaStatus,

): Promise<Tarefas> {

return this.tarefasService.updateStatus(id, status);

}

@Delete(':id')

delete(@Param('id') id: string): Promise<void> {

return this.tarefasService.delete(id);

}

}

existe algum problema na minhas rotas que pode está gerando esse erro

Sim, há um **erro importante nas suas rotas** que está impactando diretamente na listagem filtrada por usuário.

**❌ Problema:**

Você tem **dois métodos @Get() na mesma rota**:

// Rota 1:

@Get()

findAll(): Promise<Tarefas[]> {

return this.tarefasService.findAll();

}

// Rota 2:

@Get()

async listar(@UsuarioLogado() usuario: UsuarioEntity) {

return this.tarefasService.findByUsuario(usuario);

}

📛 Isso faz com que **apenas o primeiro seja executado** (o findAll() que retorna todas as tarefas, ignorando o usuário).

**✅ Solução:**

Você deve **remover um dos dois métodos**. Como agora você quer **listar tarefas do usuário logado**, o ideal é **manter apenas o listar()**, assim:

@Get()

async listar(@UsuarioLogado() usuario: UsuarioEntity) {

return this.tarefasService.findByUsuario(usuario);

}

E remova completamente o findAll() da controller:

// ❌ Remover este:

@Get()

findAll(): Promise<Tarefas[]> {

return this.tarefasService.findAll();

}

**🛠️ Alternativa avançada (caso queira manter filtro por query string):**

Se quiser implementar um GET /tarefas?status=ABERTA&termo=algumaCoisa **mas apenas para o usuário logado**, você pode adaptar o método listar para aceitar query params:

@Get()

async listar(

@UsuarioLogado() usuario: UsuarioEntity,

@Query() filtroDto: FilterTarefasDto,

) {

return this.tarefasService.filtrar(filtroDto, usuario);

}

E no tarefas.service.ts você filtra por usuario.id + os filtros opcionais (status, termo).

**✅ Resumo**

* **Remova o método duplicado @Get()** que retorna todas as tarefas sem filtro.
* Mantenha somente a versão com @UsuarioLogado().
* Confirme que a findByUsuario() filtra corretamente no tarefas.service.ts.

Se quiser, posso revisar o TarefasService para garantir que o método findByUsuario() está correto. Deseja?

E no tarefas.service.ts você filtra por usuario.id + os filtros opcionais (status, termo). me mostra isso no servisse

Claro! Vamos adaptar o seu método no tarefas.service.ts para que ele:

* Filtre as tarefas **do usuário logado** (usuario.id)
* E **aplique os filtros opcionais**: status e termo

**✅ 1. DTO de filtro (filter-tarefas.dto.ts)**

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

import { TarefaStatus } from '../tarefa-status.enum';

export class FilterTarefasDto {

@IsOptional()

@IsEnum(TarefaStatus)

status?: TarefaStatus;

@IsOptional()

@IsString()

termo?: string;

}

**✅ 2. Método no tarefas.service.ts**

async filtrar(

filtroDto: FilterTarefasDto,

usuario: UsuarioEntity,

): Promise<TarefaEntity[]> {

const { status, termo } = filtroDto;

const query = this.tarefasRepository.createQueryBuilder('tarefa');

query.where('tarefa.usuarioId = :usuarioId', { usuarioId: usuario.id });

if (status) {

query.andWhere('tarefa.status = :status', { status });

}

if (termo) {

query.andWhere(

'(LOWER(tarefa.titulo) LIKE LOWER(:termo) OR LOWER(tarefa.descricao) LIKE LOWER(:termo))',

{ termo: `%${termo}%` },

);

}

return query.getMany();

}

**✅ 3. Controlador (controller)**

No seu TarefasController, o método @Get() agora fica assim:

@Get()

async listar(

@UsuarioLogado() usuario: UsuarioEntity,

@Query() filtroDto: FilterTarefasDto,

) {

return this.tarefasService.filtrar(filtroDto, usuario);

}

**✅ 4. Exemplo de uso no Postman**

Rota GET /tarefas com parâmetros de filtro:

GET /tarefas?status=ABERTA&termo=importante

Header:

Authorization: Bearer SEU\_TOKEN

Se quiser, posso revisar também o TarefaEntity para garantir que a relação com o usuário está correta. Deseja?

@Get()

async listar(@UsuarioLogado() usuario: UsuarioEntity) {

return this.tarefasService.findByUsuario(usuario);

}

eu vou usar essa rota no controller, quero saber se devo tirar tambem do service o metodo findAll e criar um listar para cada usuario

@Injectable()

export class TarefasService {

constructor(

@InjectRepository(Tarefas)

private readonly tarefasRepository: Repository<Tarefas>,

) {}

async findAll(): Promise<Tarefas[]> {

return this.tarefasRepository.find();

}

async findById(id: string): Promise<Tarefas> {

const tarefa = await this.tarefasRepository.findOne({ where: { id } });

if (!tarefa) {

throw new NotFoundException(`Tarefa com ID "${id}" não encontrada`);

}

return tarefa;

}

// Buscar todas as tarefas de um usuário

async findByUsuario(usuario: UsuarioEntity): Promise<Tarefas[]> {

return this.tarefasRepository.find({

where: { usuario: { id: usuario.id } },

});

}

async create(dto: CreateTarefaDto, usuario: UsuarioEntity): Promise<Tarefas> {

console.log('Criando tarefa para usuário:', usuario);

const novaTarefa = this.tarefasRepository.create({

...dto,

status: TarefaStatus.ABERTA,

usuario: usuario, // atribuindo o usuário dono da tarefa

});

return this.tarefasRepository.save(novaTarefa);

}

async delete(id: string): Promise<void> {

const result = await this.tarefasRepository.delete(id);

if (result.affected === 0) {

throw new NotFoundException(`Tarefa com ID "${id}" não encontrada`);

}

}

async updateStatus(id: string, status: TarefaStatus): Promise<Tarefas> {

const tarefa = await this.findById(id);

tarefa.status = status;

return this.tarefasRepository.save(tarefa);

}

}

Sim, **você deve remover o método findAll()** do seu TarefasService e **usar apenas o findByUsuario()**, pois:

**❗ Por que remover findAll()?**

* O findAll() **retorna todas as tarefas de todos os usuários**, o que **viola a lógica de segurança do seu sistema**.
* O correto é **cada usuário ver apenas as tarefas que ele criou**, e isso já está sendo feito pelo método findByUsuario().

**✅ O que fazer:**

1. **Remova findAll()** do TarefasService:

// Remova isso:

async findAll(): Promise<Tarefas[]> {

return this.tarefasRepository.find();

}

1. **Mantenha somente findByUsuario()**, que já está correto:

async findByUsuario(usuario: UsuarioEntity): Promise<Tarefas[]> {

return this.tarefasRepository.find({

where: { usuario: { id: usuario.id } },

});

}

1. **No controller**, use apenas esta rota:

@Get()

async listar(@UsuarioLogado() usuario: UsuarioEntity) {

return this.tarefasService.findByUsuario(usuario);

}

**💡 Melhor ainda: implementar os filtros**

Se quiser dar um passo além, você pode **usar o DTO de filtros**, como status e termo, dentro do findByUsuario, e unificar tudo em um só método.

Deseja que eu te ajude a fazer isso agora?