Iniciar o projeto:

$ npx @nestjs/cli new nome\_projeto

seguido para instalar os pacotes:

$ y

qual gerenciador de pacotes usar:

$ npm

acessar a pasta do projeto:

$ cd nome\_projeto

rodar o projeto:

$ npm start

$ npm run start:dev

Inserir a Controller:

Rota Post

Texto

Descrição gerada automaticamente

Simular um BD com Repository:

Texto

Descrição gerada automaticamente

Atualizar a app.module.ts:

Texto

Descrição gerada automaticamente

Refatorando a aplicação:

Tela de computador com letras brancas

Descrição gerada automaticamente

Usuario Module:

Texto

Descrição gerada automaticamente

App Module:

Texto

Descrição gerada automaticamente

Injeção de Dependencia:

Deixar que a aplicação injete o objeto de forma automática, e não manualmente como foi feito anteriormente.

Texto

Descrição gerada automaticamenteTexto

Descrição gerada automaticamente

Transformando o usuário.repository em providers, incluindo o decorator @Injectable

Tela preta com letras brancas

Descrição gerada automaticamente

DTO – Data Transfer Object = Request no Laravel

É um recurso moderno utilizado pra transferir dados entre as camadas da aplicação.  
Ajuda a garantir a integridade dos dados;

Criada uma Pasta, e insere esta classe:

Tela de celular com fundo preto

Descrição gerada automaticamente

Na usuário.controller já identifica o tipo do objeto objeto que será criado, utilizando este recurso DTO.



PIPE (validações)

Parar o projeto e Rodar no terminal os comandos:

$ npm install class-validator class-transformer

Ajustar na Criação da classe DTO os decorators importados nos pacotes de validação.

Texto

Descrição gerada automaticamente

Registrando na class main.ts que a validação será global:

Texto

Descrição gerada automaticamente

\*transform (transforma o json da requisição no objeto para o Nest)

\*whitelist (faz que o pipe desconsidere todas as propriedades que vierem na req que não constam no DTO, não recebendo e não acusando nenhum erro)

\* forbidNonWhitelisted (lança uma excessão se vier na req dados não constante na DTO)

Customizando as mensagens de validação:

Tela de computador com texto preto sobre fundo branco

Descrição gerada automaticamente

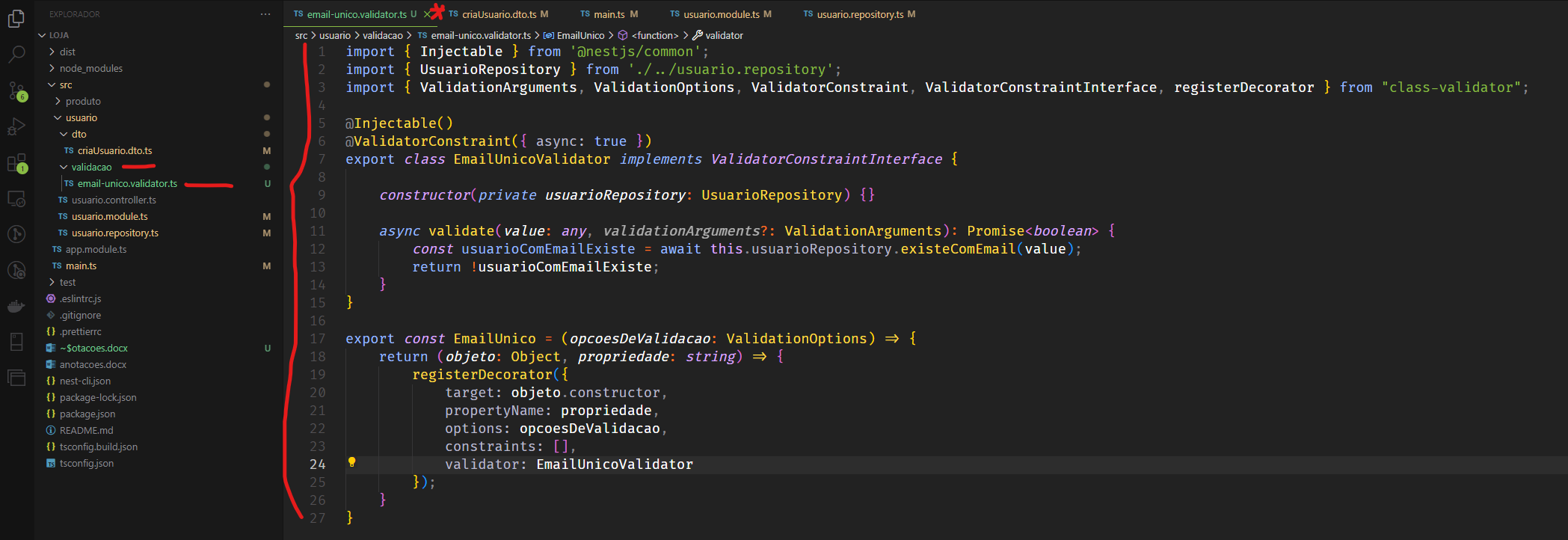
Lista de Decorators:

<https://github.com/typestack/class-validator#validation-decorators>

Customizando Validators/Decorators

Validação Assíncrona:

Criando uma validação personalizada. Indicando no @ValidatorConstraint ser assíncrona, porque o método depende do retorno se já existe ou não este email cadastrado.



Na repository cria o método que faz esta busca:

Texto

Descrição gerada automaticamente

No main inclui uma anotação pro classValidator mande o Nets resolver o problema da injeção de dependência da UsuarioRepository na Class EmailUnicoValidator:  
Texto

Descrição gerada automaticamente

Na UsuarioModule injeta a Class customizada como provider.

Interface gráfica do usuário

Descrição gerada automaticamente com confiança baixa

Injeta o decorator criado na CriaUsuarioDto:

Texto

Descrição gerada automaticamente

Criando a Entidade

Texto

Descrição gerada automaticamente

Na usuário repository é necessário fazer ajustes, que transfere à usuário Entity a modelagem, tipando o objeto usuário:

Tela de computador com texto preto sobre fundo branco

Descrição gerada automaticamente

Já na usuário controller é necessário incluir na modelagem usuário Entity o objeto que vem na request, que já passou pelas validações necessárias. (setas vermelhas)

Texto

Descrição gerada automaticamente

Aproveitamos pra incluir o id, usando a biblioteca uuid, para de rodar o projeto e roda o comando:

$ npm install uuid

Instalando os types, do TS:

$ npm install -D @types/uuid

E reestarta o projeto.

Insere as anotações da seta azul.

E como retorno, restringimos o retorno dos dados, cfe seta verde.

Criando um DTO para listar apenas campos específicos:

Tela de computador com texto preto sobre fundo branco

Descrição gerada automaticamente

Ajustando a controller

Tela de computador com texto preto sobre fundo branco

Descrição gerada automaticamente

Criando Metodo PUT

Copiar o arquivo criaUsuarioDto (Ctrl + Shift + S – cria uma cópia do Arquivo) e realiza uns ajustes.

Atualizando o nome da Classe e colocando o decorator @IsOptional que informa que aquele parâmetro pode ou não ser repassado na requisição.

Tela de computador

Descrição gerada automaticamente

No usuário controller criamos o método atualizaUsuario

Tela de computador com fundo preto

Descrição gerada automaticamente

E na repository:

Texto

Descrição gerada automaticamente

Ajuste do método atualiza pra reaproveitar a lógica que movemos pra função buscaPorId:

Texto

Descrição gerada automaticamente

Criando o método DELETE

Na controller usuário incluímos o método:

Tela de computador com fundo preto

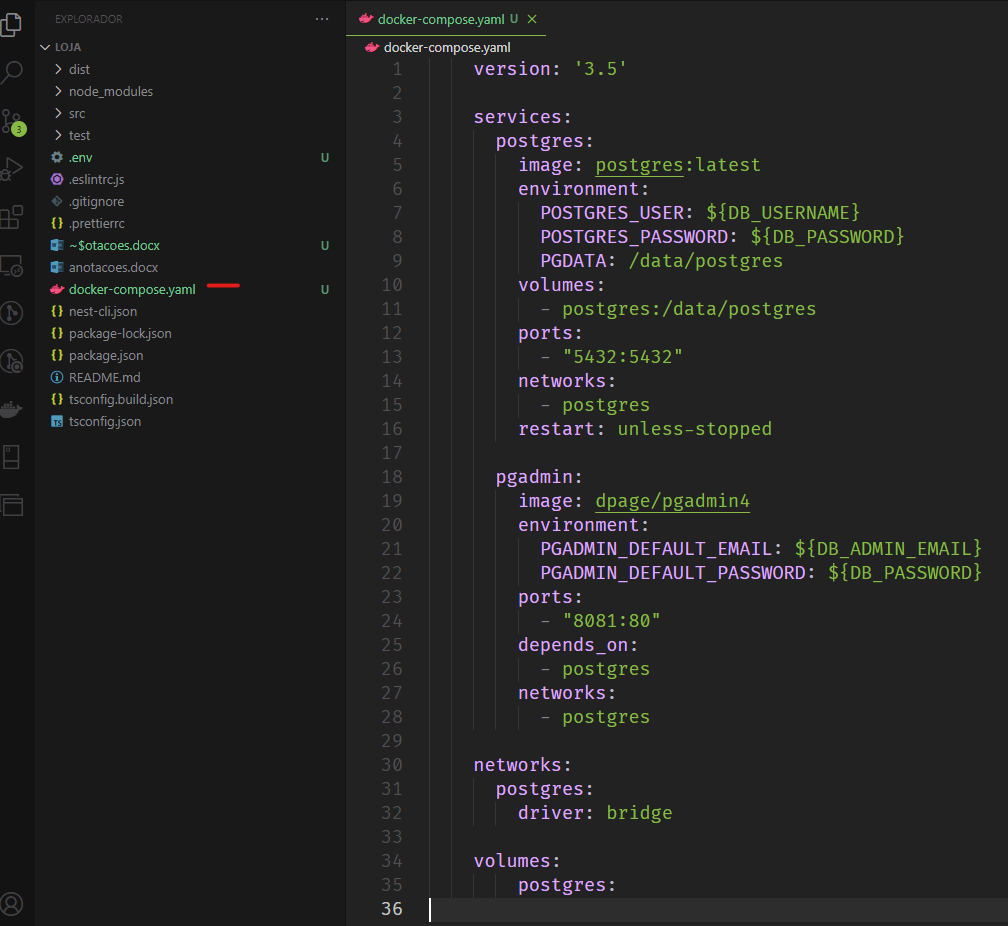
Descrição gerada automaticamente

Na repository do usuário, já ajustada a função atualiza para compartilhar o método que buscaPorId, incluímos o método remove:

Texto

Descrição gerada automaticamente

Configurando o Docker pra trabalhar com o Postgres. Criar o arquivo na raiz do projeto:



Configurar o env, também na raiz do projeto:

Tela de computador com letras e números em fundo preto

Descrição gerada automaticamente

Rodar o comando, pra subir o BD:

$ docker-compose up -d

Instalar a biblioteca de ORM que persistirá os dados no banco. Neste caso utilizaremos o TypeORM, comando:  
$ npm install @nestjs/typeorm typeorm

Por convenção, criar uma pasta na raiz, chamada config, e dentro o arquivo de configuração do BD

Texto

Descrição gerada automaticamente