NLW 04 – COMUNIDADE NODE.JS – CONSTRUÍNDO UMA API

Vai uma API aí? (foguetinho)

Nesse projeto é construído um sistema de NPS, constituindo o cadastro de um usuário, a avaliação desse usuário sobre algo e o cálculo de NPS com todas as avaliações já enviadas.

Todo o projeto foi construído durante o evento da NLW 04 feito pela RocketSeat com o intuito de aprender e entender Node.js e como montar uma API com ele e o framework que foi construído para isso, que é o Express. Durante todo o desenvolvimento são usadas ferramentas que facilitam muito e aprendemos a como usá-las. Então isso foi mais um passo para o próximo nível devs.

Meu nome é Felipe e eu montei uma “documentação” do projeto para facilitar mais o aprendizado e deixar público caso alguém queira ver.

AULA 01 – Rumo Ao Próximo Nível

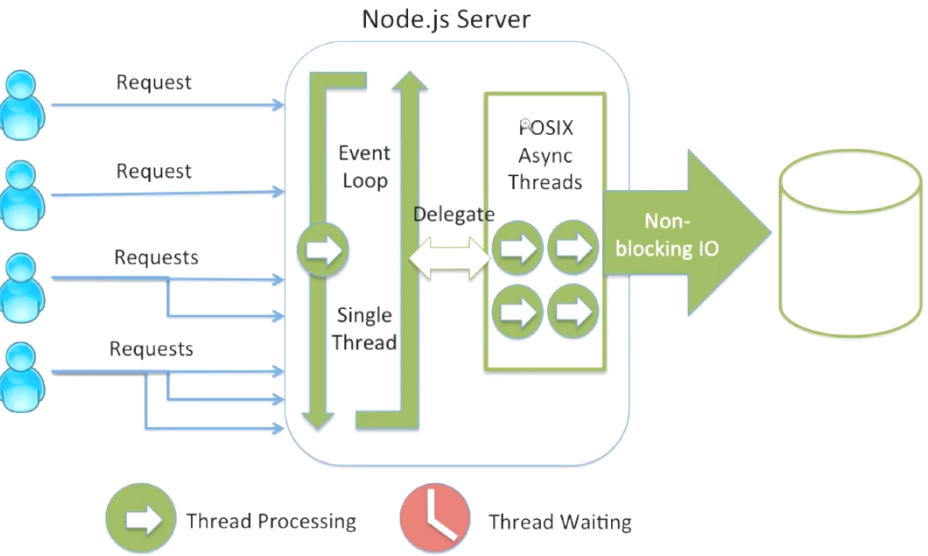
# O QUE É UMA API?

API (Application Programmer Interface)

* Não é uma linguagem
* É um conjunto de aplicações
* Requisição do cliente (FRONTEND, HTML, CSS) ao servidor (BACKEND, BD, autenticação, envio de EMAIL)
* Resposta do servidor ao cliente (resposta em JSON)
  + A API fica disponível para qualquer requisição de qualquer FRONTEND que tenha acesso a isso

# O QUE É O NODE?

É uma plataforma open-source que permite executar o JS do lado do servidor (com um sistema já montado para poder programar de uma forma mais adequada no backend)



# BORA CODAR

Usaremos o TypeScript para desenvolver o projeto.

## O Que é o TypeScript?

// function enviarEmail(destinatario, id, assunto, texto) {

//     // Biblioteca de envio de email

//     console.log(destinatario, id, assunto, texto)

// }

// class EnviarEmailParaUsuario {

//     send() {

//         enviarEmail("dani@gmail.com", 9899, "Opa", "Tudo bem?");

//     }

// }

// UUID (universally unique idenfier)

interface *DadosDeEnvioEmail* {

    destinatario: *String*,

    id: *String*,

    assunto: *String*,

    texto: *String*

}

function enviarEmail({*destinatario*, *id*, *assunto*, *texto*}: *DadosDeEnvioEmail*) {

    console.log(*destinatario*, *id*, *assunto*, *texto*)

}

class EnviarEmailParaUsuario {

    send() {

        enviarEmail({

            destinatario: "dani@gmail.com",

            id: "9899",

            assunto: "Opa",

            texto: "Tudo bem?"

        })

    }

}

# CONFIGURANDO O NOSSO PROJETO

Com uma pasta chamada api criada iremos iniciar o nosso projeto...

Para dar início nas configurações, iremos adicionar os pacotes de configurações no projeto (package.json) adicionando o comando no terminal dentro da nossa pasta de projeto:

$ yarn init -y

Arquivo package.json gerado:

{

  "name": "api",

  "version": "1.0.0",

  "main": "index.js",

  "license": "MIT"

}

## Adicionando As Dependências

Iremos usar o express (um “micro framework” para desenvolver em backend e facilitando muito quanto as rotas e os controllers, sendo o mais utilizado na comunidade node.js) como dependência e desenvolver o nosso servidor, para isso colocamos o seguinte comando:

$ yarn add express

Ele irá gerar a pasta node\_modules e irá ser adicionado ao package.json como dependencies, para indicar nas configurações:

{

  "name": "api",

  "version": "1.0.0",

  "main": "index.js",

  "license": "MIT",

  "dependencies": {

    "express": "^4.17.1"

  }

}

## Criando o Servidor

Antes de criarmos o nosso arquivo de servidor, iremos instalar uma dependência que irá facilitar durante o processo de programar, basicamente irá indicar o que podemos colocar durante esse processo, para isso digitaremos o seguinte comando no terminal:

$ yarn add @types/express -D

Na raiz do nosso projeto será criado uma pasta src e dentro dela o nosso arquivo server.ts

**Agora iremos começar a codar :)**

No nosso arquivo iremos utilizar um recurso importante do node, que é o import, ele possibilita “encaixar” o código que está em um outro arquivo para o arquivo que referirmos ele, então o node já nos facilita com tudo isso e as rotas que serão utilizadas.

Então ficará:

import express from 'express'

const app = express()

Assim já adicionamos o express para desenvolver o nosso servidor. Podemos acessar os atributos que tem dentro dele e moldar algo também.

**Definindo a porta do servidor para ele ser executado:**

app.listen(3333, () => console.log("Server is running"))

Basicamente usamos o atributo listen do express para definir a porta onde ele será executado.

**Hora de executar o server**

Para isso iremos adicionar o typescript como desenvolvimento:

$ yarn add typescript -D

Suas configurações:

$ yarn tsc –init

Então será gerado o arquivo tsconfi.json, nele iremos colocar o strict para false.

Agora como iremos usar alguns comandos em javascript, precisamos usar uma dependência que converte o código para typescript:

$ yarn add ts-node-dev -D

Definindo como iremos executar o server dentro do arquivo package.json, acima de dependencies iremos inserir:

"scripts": {

    "dev": "ts-node-dev src/server.ts"

  },

Agora definimos que ao digitar o comando yarn dev, ele irá consultar o ts-node-dev para executar o server.ts

Então rodando o nosso servidor irá aparecer:

yarn dev

yarn run v1.22.5

$ ts-node-dev src/server.ts

[INFO] 15:48:23 ts-node-dev ver. 1.1.1 (using ts-node ver. 9.1.1, typescript ver. 4.1.5)

Server is running

Vamos melhorar ainda mais o nosso desenvolvimento, para ignorar algumas coisas, como a checagem de erro nas tipagens e alterações no node\_modules. O código ficará da seguinte forma:

"scripts": {

    "dev": "ts-node-dev --transpile-only --ignore-watch node\_modules src/server.ts"

  },

### Métodos Que Podemos Utilizar Dentro Do Servidor Com o Express

* GET (método de busca)
* POST (método de salvar alguma aplicação)
* PUT (método de alterar alguma informação)
* DELETE (método de delete)
* PACTH (método de alteração específica)

**Utilizando o método GET para definir a rota no nosso server**

Método GET:

Ele pegará o que estará dentro da rota e de acordo com a rota solicitada ele chamará a função para realizar algo, podendo pegar os valores enviados ou enviar algo para quem o chamou. Exemplo:

app.get("/users", (*request*, *response*) => {

    return *response*.send("Hello World - NLW04")

})

Também podemos retornar um JSON, como é o caso quando um backend retorna um JSON para o frontend poder usá-lo:

app.get("/users", (*request*, *response*) => {

    return *response*.json({ message: "Hello World - NLW04" })

})

Método POST:

Ele pegará os dados enviados a ele e salvará eles, um exemplo básico sem ter nada funcional ainda:

app.post("/users", (*request*, *response*) => {

    return *response*.json({ message: "Os dados foram salvos com sucesso!" })

})

E como o browser não nos mostrará os responses e requests feitos por um POST, então usaremos o programa Insomnia, que servirá justamente para isso e ainda nos dando detalhes ao você requisitar uma rota ou enviar dados ao requisitá-la.

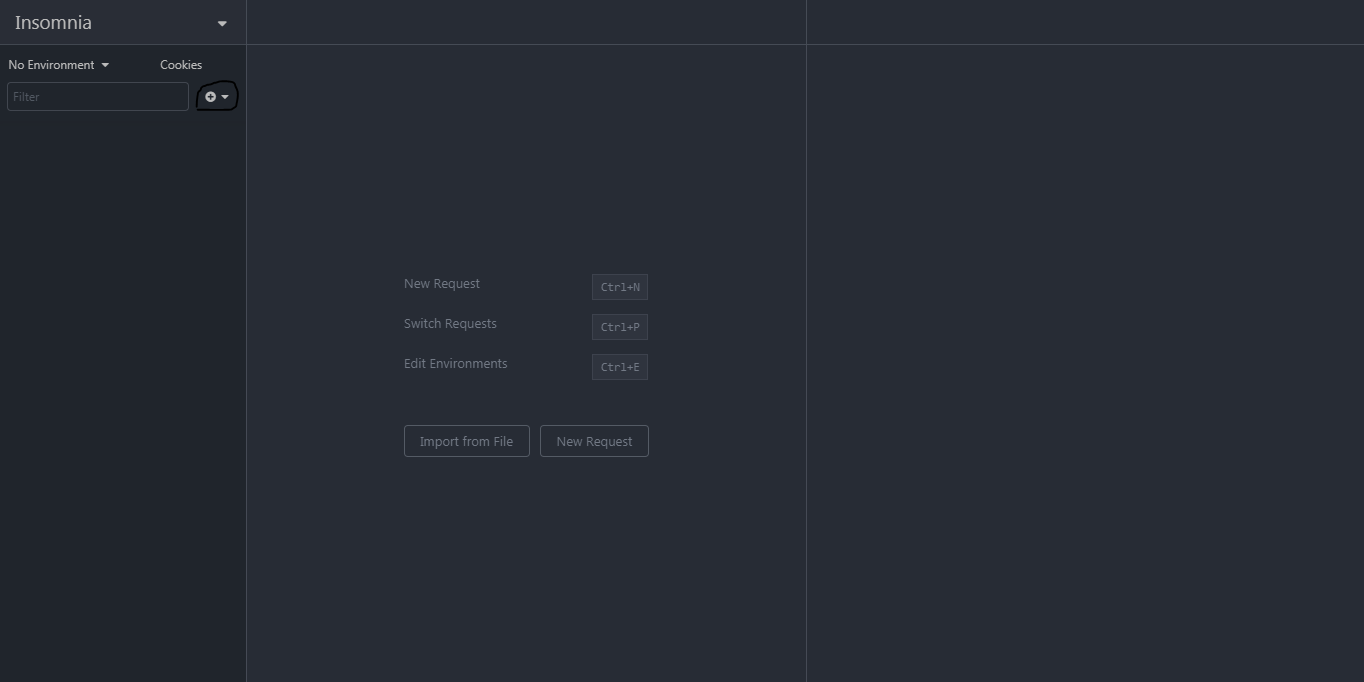
Então para configurá-lo colocando a porta do nosso server iremos acessar: No Environment > Manage Enviroments. Digitando dentro dele:

{

"baseURL": "http://localhost:3333"

}

Dentro dele podemos criar requisições e os seus tipos, como o POST ou GET, então criaremos uma chamada Requisição POST com o método POST como exemplo:



E dentro da rota digitamos baseURL e a rota desejada.

Nele podemos enviar dados como json ou receber dados como json que ele nos envia.

Enfim configuramos e preparamos todo nosso ambiente 😎

Aula 02 – Banco De Dados

Iremos baixar o nosso banco de dados para armazenar os valores que o cliente der em relação ao NPS. O banco de dados será o sqlite3, que servirá mais como uma memória local (para projetos pequenos ou locais).

Para manipular o nosso banco de dados, usaremos o TYPEORM, que é um mapeamento de dados, pegando a nossa classe para mapear para uma tabela específica do banco de dados.

# Instalando o TYPEORM:

$ yarn add typeorm reflect-metadata

# Instalando o SQLITE3

$ yarn add sqlite3

Para especificar onde estará o nosso banco de dados, entidades, migrations ao TYPEORM, precisamos fazer um arquivo de configuração, então adicionaremos o ormconfig.json dentro da pasta raiz.

E criaremos o caminho do nosso banco de dados, dentro da pasta src, adicionaremos a pasta database.

Então o arquivo ormconfig.json ficará assim:

{

    "type": "sqlite",

    "database": "./src/database/database.sqlite"

}

Agora criaremos uma conexão para o nosso banco de dados e importaremos o metadata ao nosso servidor,

então crie um index.ts na pasta database, e lá coloque:

import { createConnection } from "typeorm"

createConnection()

e importe o metada antes do express no nosso server.ts:

import 'reflect-metadata'

e importaremos a nossa conexão de banco de dados ao nosso servidor:

import './database'

e para que seja criado o nosso banco de dados, vamos iniciar o nosso servidor com um $ yarn dev.

**Começando com as migrations**

Para começarmos, iremos configurar o nosso typeorm para códigos typescript, então dentro do arquivo package.json colocamos na parte de scripts:

"typeorm": "ts-node-dev node\_modules/typeorm/cli.js"

e executamos com o comando $ yarn typeorm

Agora criaremos a pasta migrations dentro da pasta database e configuramos para que ela seja criada nessa pasta e especificaremos os arquivos que serão executados como migrations, então no arquivo ormconfig.json

"cli": {

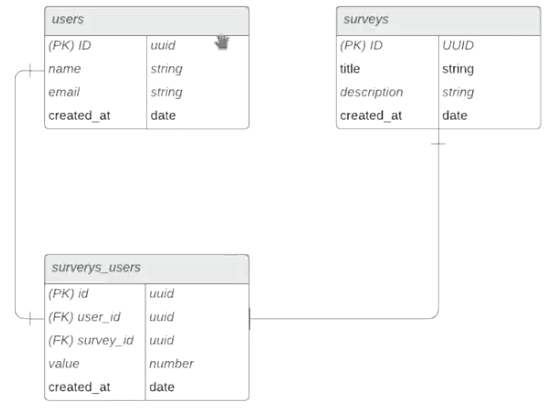
        "migrationsDir": "./src/database/migrations"

    }

e então criaremos ela com o seguinte comando:

$ yarn typeorm migration:create -n CreateUsers

**Criando as nossas tabelas**



**Tabela de usuários**

Então criaremos a tabela usuários e suas conseguintes colunas de acordo com alguns conhecimentos básicos de SQL, o arquivo das migrations deverá ficar assim no momento:

import {MigrationInterface, QueryRunner, Table} from "typeorm";

export class CreateUsers1615235088837 implements *MigrationInterface* {

    public async up(*queryRunner*: *QueryRunner*): *Promise*<*void*> {

        await *queryRunner*.createTable(

**new** Table({

                name: "users",

                columns: [

                    {

                        name: "id",

                        type: "uuid",

                        isPrimary: true

                    },

                    {

                        name: "name",

                        type: "varchar"

                    },

                    {

                        name: "email",

                        type: "varchar"

                    },

                    {

                        name: "created\_at",

                        type: "timestamp",

                        default: "now()"

                    }

                ]

            })

        )

    }

    public async down(*queryRunner*: *QueryRunner*): *Promise*<*void*> {

    }

}

executar a migration:

$ yarn typeorm migration:run

| obs: caso queira desfazer essa execução, realize o seguinte comando:

$ yarn typeorm migration:revert

Está criada a nossa tabela, caso queira ver ela com mais detalhes e testar algumas queries, é recomendado usar o programa Beekeeper Studio.

**Criando métodos para salvar os usuários**

Para isso iremos criar uma pasta chamada controllers dentro de src, onde ele atenderá pelas urls que serão atribuídas quando o nosso servidor for requisitado. Na pasta criaremos o arquivo UserController.ts, então nele colocaremos o método de POST, para criá-lo através da função que iremos criar nessa classe:

import { Request, Response } from 'express'

import { getRepository } from 'typeorm'

import { User } from '../models/User'

class UserController {

    async create(*request*: *Request*, *response*: *Response*) {

        const { name, email } = *request*.body

        const userRepository = getRepository(User)

        const userAlreadyExists = await userRepository.findOne({

            email

        })

        if (userAlreadyExists) {

            return *response*.status(400).json({

                error: "user already exists!"

            })

        }

        const user = userRepository.create({

            name, email

        })

        await userRepository.save(user)

        return *response*.json(user)

    }

}

export { UserController }

Agora precisamos instanciar essa nossa classe dentro do nosso servidor.

Para não ficar tudo muito bagunçado, criaremos um arquivo router.ts, colocando a rota que for requisitada, irá retornar a tal classe instanciada (users no momento).

import { Router } from 'express'

import { UserController } from './controllers/UserController'

const router = Router()

const userController = **new** UserController()

router.post("/users", userController.create)

export { *Router* }

e então ele precisará estar no nosso server:

import 'reflect-metadata'

import express from 'express'

import './database'

import { router } from './router'

const app = express()

app.use(express.json())

app.use(router)

app.listen(3333, () => console.log("Server is running"))

**Passando as informações recolhidas para o nosso banco de dados (migrations)**

Vamos precisar criar nossa entidade, uma pasta chamada models dentro de src, nela iremos criar o arquivo User.ts

Antes de realizarmos isso, precisamos habilitar o “experimentalDecorators”, “emitDecoratorMetadata”, e o "strictPropertyInitialization" definimos como false no tsconfig.ts, para que possamos referenciar as tabelas na qual queremos pegar ou empurrar algum dado.

Agora precisamos importar o uuid, para que tenhamos uma tipagem de valor correta ao inserir o id do usuário:

$ yarn add uuid

$yarn add @types/uuid -D

import { Column, CreateDateColumn, Entity, PrimaryColumn } from "typeorm";

import { *v4* as uuid } from 'uuid'

@Entity('users')

class User {

    @PrimaryColumn()

    readonly id: *String*

    @Column()

    name: *String*

    @Column()

    email: *String*

    @CreateDateColumn()

    created\_at: *Date*

    constructor() {

        if (!*this*.id) { *this*.id = uuid() }

    }

}

export { User }

E especificaremos onde esta entidade está, então no arquivo ormconfig.json coloque:

"entities": ["./src/models/\*\*.ts"]

Agora podemos testar tudo isso enviando um json com o Insomnia e vendo o que há no banco de dados com o beekeeper Studio.

Está aí o nosso banco de dados

AULA 03 – TESTANDO NOSSA APLICAÇÃO

**Refatorando Nosso UserController**

Como nosso UserController não é adequado para ficar acessando o banco de dados (ele está acessando com o getRepository que está configurado para acessar as tabelas e colunas do nosso banco de dados), então criaremos uma pasta dentro de src, chamada repositories, e criaremos um arquivo para agora acessar o banco de dados, chamado UserRepository.ts.

Esse nosso arquivo vai poder acessar os atributos e métodos do nosso banco de dados, por isso iremos usar o extends Repository do typeorm para que a nossa classe UserRepository dentro do arquivo, tenha como esses acessos da tabela do nosso banco de dados que iremos especificar, que no caso estamos queremos a entidade User. Confira como está ficando o seguinte código:

import { EntityRepository, Repository } from "typeorm";

import { User } from "../models/User";

@EntityRepository(User)

class UserRepository extends Repository<User> {

}

export { UserRepository }

e como o nosso UserController não tem mais acesso direto ao banco de dados, então ele ficará da seguinte maneira:

import { Request, Response } from 'express'

import { getCustomRepository } from 'typeorm'

import { UserRepository } from '../repositories/UserRepository'

class UserController {

    async create(*request*: *Request*, *response*: *Response*) {

        const { name, email } = *request*.body

        const userRepository = getCustomRepository(UserRepository)

        const userAlreadyExists = await userRepository.findOne({

            email

        })

        if (userAlreadyExists) {

            return *response*.status(400).json({

                error: "user already exists!"

            })

        }

        const user = userRepository.create({

            name, email

        })

        await userRepository.save(user)

return *response*.status(201).json(user)

    }

}

export { UserController }

**Criando A Tabela De Pesquisas**

Primeiramente, igual à criação da tabela de usuários, iremos executar o comando do typeorm para criar a migration:

$ yarn typeorm migration:creation -n CreateSurveys

e a migration ficará com as seguintes colunas:

import {MigrationInterface, QueryRunner, Table} from "typeorm";

export class CreateSurveys1617329865962 implements *MigrationInterface* {

    public async up(*queryRunner*: *QueryRunner*): *Promise*<*void*> {

        await *queryRunner*.createTable(

**new** Table ({

                name: "surveys",

                columns: [

                    {

                        name: "id",

                        type: "uuid",

                        isPrimary: true

                    },

                    {

                        name: "title",

                        type: "varchar"

                    },

                    {

                        name: "description",

                        type: "varchar"

                    },

                    {

                        name: "created\_at",

                        type: "timestamp",

                        default: "now()"

                    }

                ]

            })

        )

    }

    public async down(*queryRunner*: *QueryRunner*): *Promise*<*void*> {

        await *queryRunner*.dropTable("surveys")

    }

}

agora é só executar a migration e teremos a nossa nova tabela:

$ yarn typeorm migration:run

e criaremos o nosso model do Survey, então dentro da pasta models, criamos o arquivo Survey.ts:

import { Column, Entity, PrimaryColumn } from "typeorm"

import { *v4* as uuid } from "uuid"

@Entity('surveys')

class Survey {

    @PrimaryColumn()

    readonly id: *String*

    @Column()

    title: *String*

    @Column()

    description: *String*

    @Column()

    created\_at: *Date*

    constructor() {

        if (!*this*.id) { *this*.id = uuid() }

    }

}

export { Survey }

Agora o nosso controller para que seja salvo as nossas pesquisas, dentro da pasta controllers teremos o nosso novo arquivo SurveyController.ts, que no momento ficará assim:

import { Request, Response } from "express"

class SurveyController {

    async create(*Response*: *Response*, *Request*: *Request*) {

    }

}

export { SurveyController }

e como o nosso controller precisa do repositório para ter acesso à tabela do banco de dados, então criaremos o no repositório SurveyRepository.ts dentro de repositories, que no caso irá usar o nosso model do Survey para indicar a tabela e os dados que serão passados para ela:

import { EntityRepository, Repository } from "typeorm";

import { Survey } from "../models/Survey";

@EntityRepository(Survey)

class SurveyRepository extends Repository<Survey> {}

export { SurveyRepository }

Agora podemos alterar o nosso SurveyController para salvar as pesquisas:

import { Request, Response } from "express"

import { getCustomRepository } from "typeorm"

import { SurveyRepository } from "../repositories/SurveyRepository"

class SurveyController {

    async create(*response*: *Response*, *request*: *Request*) {

        const { title, description } = *request*.body

        const surveyRepository = getCustomRepository(SurveyRepository)

        const survey = surveyRepository.create({

            title, description

        })

        await surveyRepository.save(survey)

        return *response*.status(201).json(survey)

    }

}

export { SurveyController }

agora é necessário colocar esse SurveyController no nosso Router, para que quando for requisitado uma URL, ele execute tudo isso:

import { Router } from 'express'

import { SurveyController } from './controllers/SurveyController'

import { UserController } from './controllers/UserController'

const router = Router()

const userController = **new** UserController()

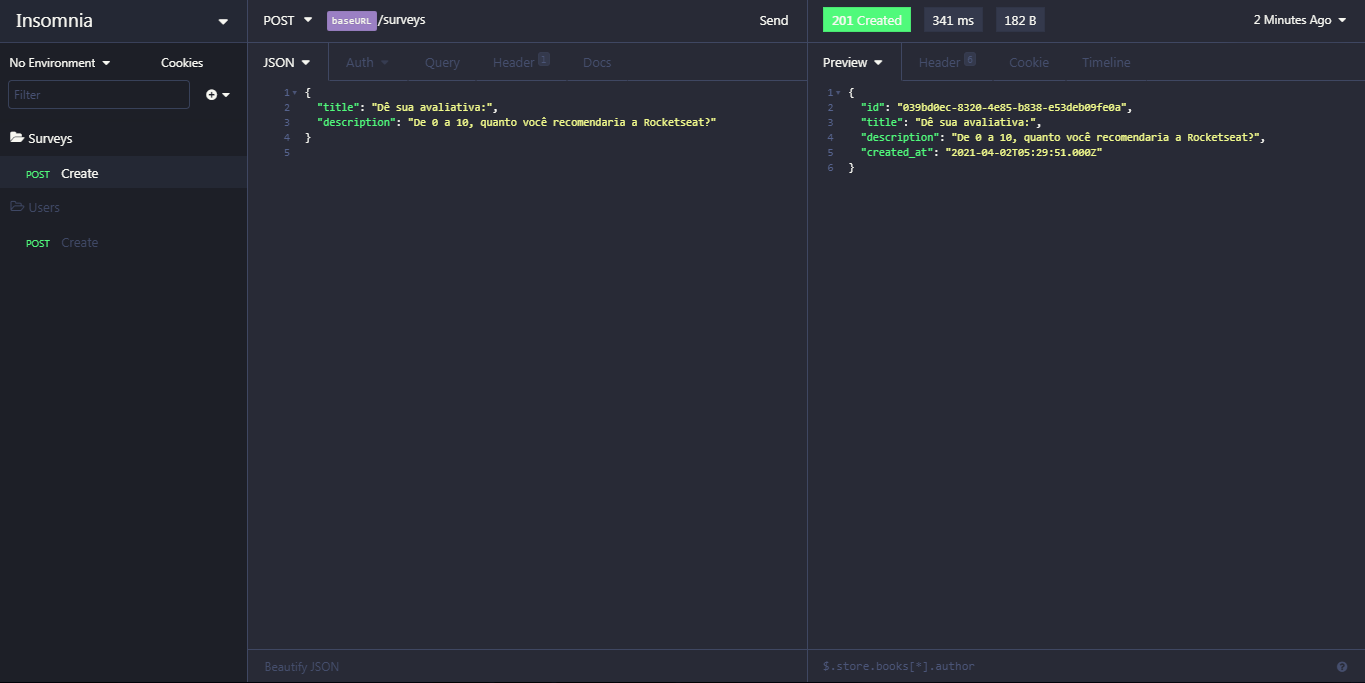
const surveyController = **new** SurveyController()

router.post("/users", userController.create)

router.post("/surveys", surveyController.create)

export { router }

Agora podemos testar essa aplicação pelo Insomnia:



E já podemos adicionar um método para listar todas as pesquisas que temos no nosso SurveyController, o código ficará da seguinte maneira:

abaixo do método create...

async show(*request*: *Request*, *response*: *Response*) {

        const surveyRepository = getCustomRepository(SurveyRepository)

        const all = await surveyRepository.find()

        return *response*.json(all)

    }

e precisamos informar a rota requisitada para executar esse método:

no arquivo routes.ts

router.get("/surveys", surveyController.show)

**Sobre Testes Automatizados**

1 – Testes Unitários: determinada funcionalidade da nossa aplicação,

testes com dados e aplicações fakes, para que não dê alterações no nosso projeto

2 – Testes De Integração: funcionalidade completa de integração,

-> request -> routes -> controllers -> repository (models)

<- repository (models) <- controllers <- routes <- response

3 – Testes De Ponta a Ponta (E2E): ação do usuário na aplicação,

mais para frontend

Dando início aos nossos testes

instalar a ferramenta jest (utilizar para fazer os nossos testes):

$ yarn add jest @types/jest -D

arquivo de configuração do jest:

$ yarn jest –init

e ele nos perguntará se queremos iniciar nosso teste de aplicação, então basta colocar y;

a gente quer um arquivo de teste em typescript, então a outra resposta é y;

usaremos node para os testes;

não será usado o coverage reports;

usaremos o v8;

e será limpado automaticamente para cada teste, então y.

Certo, agora o arquivo jest.config.ts foi criado, podemos então configurá-lo,

na linha 11 habilitaremos o bail para true, para que ele não execute mais a checagem de erros após encontrar um;

na linha 140 desabilitaremos o testEnvironment, comentando a linha;

na linha 149 será habilitado o testMatch, onde é o caminho que será colocado os nossos testes:

testMatch: [

    "\*\*/\_\_tests\_\_/\*.test.ts"

],

e a pasta \_\_tests\_\_ dentro de src poderá ser criada

para desenvolvermos precisaremos do jest como dependência de desenvolvimento:

$ yarn add ts-jest -D

e para especificarmos que trabalharemos com typescript dentro dos testes, iremos no arquivo do jest.config.ts, na linha 96, habilitaremos o preset e colocaremos:

preset: "ts-jest",

Fazendo os testes agora...

no nosso arquivo de teste começaremos descrevendo o nosso teste, então temos como exemplo:

describe("First", () => {

    it("deve ser possível somar dois números", () => {

        expect(2 + 2).toBe(4)

    })

})

nele a gente está descrevendo o nosso teste e está fazendo uma espera de um valor que seja igual à um determinado, e para executarmos o teste iremos dar o comando:

$ yarn test

também temos o seguinte método:

it("deve ser possível somar dois números", () => {

        expect(2 + 2).not.toBe(5)

})

então agora podemos apagar este arquivo...

Criando de fato os nossos testes de integração:

criando o arquivo User.test.ts na nossa pasta de \_\_tests\_\_

e para simularmos um servidor para teste, usaremos uma ferramenta chamada supertest:

$ yarn add supertest @types/supertest -D

Então para não termos de usar o nosso server nos tests, iremos abstrai-lo para reaproveitar o seu código, sem que causemos alguma interferência nele,

criamos um arquivo app.ts e colocaremos nele parte do código do server:

import 'reflect-metadata'

import express from 'express'

import './database'

import { router } from './router'

const app = express()

app.use(express.json())

app.use(router)

export { app }

e o arquivo do nosso server.ts:

import { app } from "./app";

app.listen(3333, () => console.log("Server is running 🚀"))

e o nosso arquivo de teste ficará da seguinte forma no momento:

import request from "supertest"

import { app } from "../app"

describe("Users", () => {

    request(app).post("/users").send({

        email: "user@email.com",

        name: "User Example"

    })

})

como queremos ter um ambiente de teste, não podemos usar o mesmo banco de dados também, então vamos refatorar o código, mais especificamente a conexão dele no arquivo index.ts na pasta database...

import { Connection, createConnection } from "typeorm"

export default async (): *Promise*<Connection> => {

    return createConnection()

}

basicamente o que estamos fazendo, é indicando se a execução que estamos fazendo em nossa api é para testes ou para executar o nosso banco de dados funcional mesmo. Ou seja, “verificar qual o ambiente que estamos utilizando”,

por isso é necessário configurar o nosso comando de test, indicando a variável NODE\_ENV para ele executar o banco de dados do ambiente que referirmos, ficando assim no arquivo package.json na linha de test:

"test": "env NODE\_ENV=test jest"

(caso o sistema operacional não for Windows, então apenas retire o env)

Então ele identificará quando formos usar a banco de dados de teste, quando dermos o comando yarn test, atribuirá o valor do NODE\_ENV para test, e caso ele seja igual a teste, então a partir dessa condição, ele mudará a conexão do banco de dados para o nosso novo banco de dados teste, o database.test.sqlite e caso não seja dado o comando teste, o NODE\_ENV não terá esse valor test, o que resultará em uma conexão com o nosso banco de dados funcional:

import { Connection, createConnection, getConnectionOptions } from "typeorm"

export default async (): *Promise*<Connection> => {

    const defaultOptions = await getConnectionOptions()

    return createConnection(

*Object*.assign(defaultOptions, {

            database:

                process.env.NODE\_ENV === "test"

                    ? "./src/database/database.test.sqlite"

                    : defaultOptions.database

        })

    )

}

agora precisamos mudar a conexão feita no nosso arquivo app.ts:

import 'reflect-metadata'

import express from 'express'

import createConnection from './database'

import { router } from './router'

createConnection()

const app = express()

app.use(express.json())

app.use(router)

export { app }

Agora vamos voltar para o nosso ambiente de teste,

nele será colocado a conexão com o banco de dados e executaremos as migrations nele para que ele seja estruturado antes que usemos algo dele, pois ele ainda não foi usado:

import request from "supertest"

import { app } from "../app"

import createConnection from "../database"

describe("Users", () => {

    beforeAll(async () => {

        const connection = await createConnection()

await connection.runMigrations()

    })

    request(app).post("/users").send({

        email: "user@email.com",

        name: "User Example"

    })

})

e iremos descrever o que deverá ser retornado nesse teste (o status de concluído) para que ele mostre um status a nós:

import request from "supertest"

import { app } from "../app"

import createConnection from "../database"

describe("Users", () => {

    beforeAll(async () => {

        const connection = await createConnection()

await connection.runMigrations()

    })

    it("Should be able to create a new user", async () => {

        const response = await request(app).post("/users").send({

            email: "user@email.com",

            name: "User Example"

        })

        expect(response.status).toBe(201)

    })

})

Executemos o comando de teste:

$ yarn test

Agora vamos fazer um teste de validação de um usuário sendo salvo com um mesmo email:

it("Should not be able to create a user with exists email", async () => {

        const response = await request(app).post("/users").send({

            name: "User Example",

            email: "user@email.com"

        })

        expect(response.status).toBe(400)

})

Antes de executarmos o teste novamente, precisamos especificar que queremos excluir o banco de dados teste para cada vez que terminar um teste, já que ele ficará dando o comando das migrations após executarmos um teste, então no package.json, adicionaremos abaixo de test:

"posttest": "rm ./src/database/database.test.sqlite"

e como ele só executará caso algum teste dê um resultado concluído, então precisamos apenas essa vez executar o comando manualmente, já que o test não poderá ser concluído com o banco de dados ainda existente:

$ rm ./src/database/database.test.sqlite

agora o comando de test

$ yarn test

Então por agora o teste de pesquisa, sendo quase a mesma coisa que o do usuário, criando o arquivo Survey.test.ts:

import request from "supertest"

import { app } from "../app"

import createConnection from "../database"

describe("Surveys", () => {

    beforeAll(async () => {

        const connection = await createConnection()

        await connection.runMigrations()

    })

    it("Should be able to create a new survey", async () => {

        const response = await request(app).post("/surveys").send({

            title: "Title Example",

            description: "Description Example"

        })

        expect(response.status).toBe(201)

        expect(response.body).toHaveProperty("id")

    })

    it("Should be able to get all surveys", async () => {

        await request(app).post("/surveys").send({

            title: "Title Example2",

            description: "Description Example2"

        })

        const response = await request(app).get("/surveys")

        expect(response.body.length).toBe(2)

    })

})

e também podemos executar

$ yarn test

Agora já fizemos toda a integração de testes :)

AULA 04 – ENVIO DE EMAIL

**Criando a tabela createSurveysUsers**

Essa tabela guardará a avaliação do usuário de uma certa pesquisa, então como padrão, para criar a nossa migration, vamos colocar o comando:

$ yarn typeorm migration:create -n createSurveysUsers

e no arquivo criaremos as seguintes tabelas:

import {MigrationInterface, QueryRunner, Table} from "typeorm";

export class CreateSurveysUsers1617736513174 implements *MigrationInterface* {

    public async up(*queryRunner*: *QueryRunner*): *Promise*<*void*> {

        await *queryRunner*.createTable(

**new** Table({

                name: "surveys\_users",

                columns: [

                    {

                        name: "id",

                        type: "uuid",

                        isPrimary: true

                    },

                    {

                        name: "user\_id",

                        type: "uuid"

                    },

                    {

                        name: "survey\_id",

                        type: "uuid"

                    },

                    {

                        name: "value",

                        type: "number",

                        isNullable: true

                    },

                    {

                        name: "created\_at",

                        type: "timesamp",

                        default: "now()"

                    }

                ]

            })

        )

    }

    public async down(*queryRunner*: *QueryRunner*): *Promise*<*void*> {

*queryRunner*.dropTable("surveys\_users")

    }

}

no mesmo arquivo, precisamos indicar de qual tabelas os dados do usuário e da pesquisa está, então indicaremos as chaves estrangeiras, ou seja, dados que virão de outra tabela para ser colocada nessa,

abaixo das colunas criadas:

foreignKeys: [

                    {

                        name: "fk\_user",

                        referencedTableName: "users",

                        referencedColumnNames: ["id"],

                        columnNames: ["user\_id"],

                        onDelete: "CASCADE",

                        onUpdate: "CASCADE"

                    },

                    {

                        name: "fk\_survey",

                        referencedTableName: "surveys",

                        referencedColumnNames: ["id"],

                        columnNames: ["survey\_id"],

                        onDelete: "CASCADE",

                        onUpdate: "CASCADE"

                    }

]

apenas indicamos um nome, a tabela na qual queremos buscar o valor, a coluna dessa tabela e a coluna da tabela na qual estamos que é para onde esse valor será enviado. No onDelete e onUpdate indicamos que quando algum usuário for removido por exemplo, então ele irá excluir os dados da pesquisa dele, ou quando for criado, ele já crie automaticamente.

podemos executar a nossa migration...

$ yarn typeorm migration:run

**Criando SurveyUser models**

na pasta models iremos criar o arquivo SurveyUser.ts

import { Column, CreateDateColumn, Entity, PrimaryColumn } from "typeorm";

import { *v4* as uuid } from "uuid";

@Entity("surveys\_users")

class SurveyUser {

    @PrimaryColumn()

    readonly id: *String*

    @Column()

    user\_id: *String*

    @Column()

    survey\_id: *String*

    @Column()

    value: *Number*

    @CreateDateColumn()

    created\_at: *Date*

    constructor() {

        if (*this*.id) { *this*.id = uuid() }

    }

}

export { SurveyUser }

**Criando nosso repositório dos surveys users**

na pasta repositories cria-se uma pasta SurveyUserRepository.ts:

import { EntityRepository, Repository } from "typeorm";

import { SurveyUser } from "../models/SurveyUser";

@EntityRepository(SurveyUser)

class SurveyUserRepository extends Repository<SurveyUser> {}

export { SurveyUserRepository }

**Criando nosso SendMailController**

Na pasta controllers criamos o arquivo SendMailController.ts onde iremos enviar o email pedindo a avaliativa e depois salvar ela, veja como fica até a parte de verificar se o email e a pesquisa solicitada existe e salvando o user\_id e o survey\_id:

import { Request, Response } from "express"

import { getCustomRepository } from "typeorm"

import { SurveyRepository } from "../repositories/SurveyRepository"

import { SurveyUserRepository } from "../repositories/SurveyUserRepository"

import { UserRepository } from "../repositories/UserRepository"

class SendMailController {

    async execute(*request*: *Request*, *response*: *Response*) {

        const { email, survey\_id } = *request*.body // we will send an email with the amount received from request.body and send the survey to it according to survey\_id

        const userRepository = getCustomRepository(UserRepository)

        const surveyRepository = getCustomRepository(SurveyRepository)

        const surveyUserRepository = getCustomRepository(SurveyUserRepository)

        // checking if the user exists, so we send an email to him

        const userAlreadyExists = await userRepository.findOne({ email })

        if (!userAlreadyExists) {

            return *response*.status(400).json({

                error: "User does not exists!"

            })

        }

        // checking if the survey exists so that it can be sent to email

        const surveyAlreadyExists = await surveyRepository.findOne({ id: survey\_id })

        if (!surveyAlreadyExists) {

            return *response*.status(400).json({

                error: "Survey does not exists!"

            })

        }

        // save the information in the surveys\_users tables

        const surveyUser = surveyUserRepository.create({

            user\_id: userAlreadyExists.id,

            survey\_id

        })

        await surveyUserRepository.save(surveyUser)

    }

}

export { SendMailController }

e colocaremos essa rota no router.ts:

import { Router } from 'express'

import { SendMailController } from './controllers/SendMailController'

import { SurveyController } from './controllers/SurveyController'

import { UserController } from './controllers/UserController'

const router = Router()

const userController = **new** UserController()

const surveyController = **new** SurveyController()

const sendMailController = **new** SendMailController()

router.post("/users", userController.create)

router.post("/surveys", surveyController.create)

router.get("/surveys", surveyController.show)

router.post("/sendMail", sendMailController.execute)

export { router }

**Fazendo o envio de email**

import nodemailer, { Transporter } from "nodemailer"

class SendMailService {

    private client: *Transporter*

    constructor() {

        nodemailer.createTestAccount().then(*account* => {

            const transporter = nodemailer.createTransport({

                host: *account*.smtp.host,

                port: *account*.smtp.port,

                secure: *account*.smtp.secure,

                auth: {

                    user: *account*.user,

                    pass: *account*.pass

                }

            });

*this*.client = transporter

        })

    }

    async execute(*to*: *string*, *subject*: *string*, *body*: *string*) {

        const message = await *this*.client.sendMail({

            to,

            subject,

            html: *body*,

            from: "NPS <noreplace@nps.com.br>"

        })

        console.log('Message sent: %s', message.messageId);

        console.log('Preview URL: %s', nodemailer.getTestMessageUrl(message));

    }

}

export default **new** SendMailService()

Na função execute, colocamos para quem enviaremos, qual o assunto e a descrição (pesquisa), então exportamos a classe já instanciada.

**Criando um template para envio de email**

Vamos criar um template para que o envio de email fique mais apresentável, não conter apenas textos :)

Utilizando o Handlebars para construir o nosso template, nele é possível colocar variáveis dentro do template (HTML):

$ yarn add handlebars

Criando o diretório para o nosso template. Dentro da pasta src, criaremos uma pasta chamada views e dentro dela uma outra pasta, chamada emails, nela conterá o nosso template para emails feito com o Handlebars.

o arquivo que colocaremos se chamará npsMail.hbs, um arquivo de extensão handlebars no qual podemos colocar html...

Para colocarmos variáveis dentro do html, colocamos da seguinte maneira: {{ nomeVariável }}

<style>

*.mail-container* {

        width: 800px;

        display: flex;

        justify-content: center;

        align-items: center;

        align-content: center;

        flex-direction: column;

        font-family: Arial, Helvetica, sans-serif;

        color: #202024;

    }

*.mail-container* h2 {

        color: #4c4c4c;

    }

*.mail-container* *.subject* {

        display: flex;

        justify-content: center;

        align-items: center;

        flex-direction: column;

        margin: .6rem 0 1.5rem 0;

    }

*.mail-container* *.subject* h2 {

        margin-bottom: 0;

    }

*.mail-container* *.subject* strong {

        color: #555;

    }

*.mail-container* *.level* {

        display: flex;

        justify-content: space-between;

        width: 310px;

        margin-bottom: .3rem;

    }

*.mail-container* *.level* span*:first-child* {

        color: #f36565;

    }

*.mail-container* *.level* span*:last-child* {

        color: #29d476;

    }

*.mail-container* *.answers* {

        display: flex;

        justify-content: center;

        width: 450px;

    }

*.mail-container* *.answers* *.value* {

        display: flex;

        justify-content: center;

        align-items: center;

        padding: 10px;

        margin: 0 2px;

        width: 1rem;

        border-radius: 50%;

        background-color: #8257e6;

        color: #FFF;

        text-decoration: none;

        transition: .2s;

    }

*.mail-container* *.answers* *.value:hover* {

        background-color: #744ad8;

    }

*.mail-container* footer {

        display: flex;

        justify-content: center;

        align-items: center;

        flex-direction: column;

        width: 60%;

        padding: .8rem;

        font-size: .9rem;

        border-top: 1px #2929304b solid;

        color: #9292ad;

    }

*.mail-container* footer h3 {

        margin-top: 0;

    }

</style>

<div *class=*"mail-container">

    <img *width=*"130" *src=*"https://rocketseat.gallerycdn.vsassets.io/extensions/rocketseat/rocketseatreactnative/3.0.1/1588456740326/Microsoft.VisualStudio.Services.Icons.Default" *alt=*"Rocketseat">

    <label>Olá <strong>{{ *name* }}</strong>! Tudo bem? Aqui é a Rocketseat</label>

    <div *class=*"subject">

        <h2>{{ *title* }}</h2>

        <br>

        <strong>{{ *description* }}</strong>

    </div>

    <div *class=*"level">

        <span>Pouco provável &#9785;</span>

        <span>Muito provável &#9786;</span>

    </div>

    <div *class=*"answers">

        <a *class=*"value" *href=*"">1</a>

        <a *class=*"value" *href=*"">2</a>

        <a *class=*"value" *href=*"">3</a>

        <a *class=*"value" *href=*"">4</a>

        <a *class=*"value" *href=*"">5</a>

        <a *class=*"value" *href=*"">6</a>

        <a *class=*"value" *href=*"">7</a>

        <a *class=*"value" *href=*"">8</a>

        <a *class=*"value" *href=*"">9</a>

        <a *class=*"value" *href=*"">10</a>

    </div>

    <br>

    <br>

    <footer>

        <strong>&#9734; Sua opinião é muito importante para nós &#9734;</strong>

        <h3>Equipe &#10084; <strong>NLW</strong></h3>

    </footer>

</div>

Neste código estamos colocando o título, a descrição, o campo para o usuário enviar sua avaliação e um footer.

Aplicando o template para ele ser enviado no email...

no arquivo sendMailService.ts

import nodemailer, { Transporter } from "nodemailer"

import handlebars from "handlebars"

import { resolve } from "path"

import { readFileSync } from "fs"

class SendMailService {

    private client: *Transporter*

    constructor() {

        nodemailer.createTestAccount().then(*account* => {

            const transporter = nodemailer.createTransport({

                host: *account*.smtp.host,

                port: *account*.smtp.port,

                secure: *account*.smtp.secure,

                auth: {

                    user: *account*.user,

                    pass: *account*.pass

                }

            });

*this*.client = transporter

        })

    }

    async execute(*to*: *string*, *subject*: *string*, *body*: *string*) {

        const npsMailPath = resolve(\_\_dirname, "..", "views", "emails", "npsMail.hbs") // get project directory

        const templateFileContent = readFileSync(npsMailPath).toString("utf-8") // reads file from directory

        const mailTemplateParse = *handlebars*.compile(templateFileContent) // compiles the template file to be sent as an email by passing the variables that are present in it

        const html = mailTemplateParse({

            name: *to*,

            desption: *subject*,

            html: *body*

        })

        const message = await *this*.client.sendMail({

            to,

            subject,

            html,

            from: "NPS <noreplace@nps.com.br>"

        })

        console.log('Message sent: %s', message.messageId);

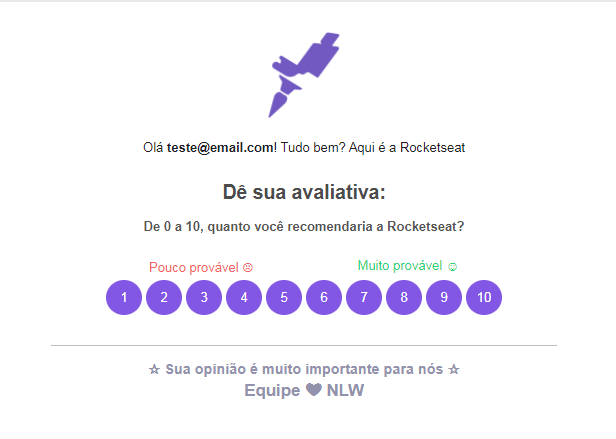
        console.log('Preview URL: %s', nodemailer.getTestMessageUrl(message));

    }

}

export default **new** SendMailService()

Olha só como ficou o nosso email :)



Refatorando o nosso serviço de envio de email para que o arquivo de template recebido seja mais versátil, para que ele seja mandado mais abrangentemente, passando o arquivo desejado pela pessoa que usar a api.

Para isso vamos colocar a rota do arquivo do template no nosso controller, para que ele receba o arquivo e passe para o serviço de envio de email, então o retiramos do serviço e o colocamos acima da execução do sendMailService no controller e importando método resolve para fazer a busca desse arquivo, ficando da seguinte maneira agora:

SendMailService.ts:

import nodemailer, { Transporter } from "nodemailer"

import handlebars from "handlebars"

import { readFileSync } from "fs"

class SendMailService {

    private client: *Transporter*

    constructor() {

        nodemailer.createTestAccount().then(*account* => {

            const transporter = nodemailer.createTransport({

                host: *account*.smtp.host,

                port: *account*.smtp.port,

                secure: *account*.smtp.secure,

                auth: {

                    user: *account*.user,

                    pass: *account*.pass

                }

            });

*this*.client = transporter

        })

    }

    async execute(*to*: *string*, *subject*: *string*, *body*: *string*, *path*: *string*) {

        const templateFileContent = readFileSync(*path*).toString("utf-8")

        const mailTemplateParse = *handlebars*.compile(templateFileContent)

        const html = mailTemplateParse({

            name: *to*,

            title: *subject*,

            description: *body*

        })

        const message = await *this*.client.sendMail({

            to,

            subject,

            html,

            from: "NPS <noreplace@nps.com.br>"

        })

        console.log('Message sent: %s', message.messageId)

        console.log('Preview URL: %s', nodemailer.getTestMessageUrl(message))

    }

}

export default **new** SendMailService()

SendMailController.ts:

import { Request, Response } from "express"

import { getCustomRepository } from "typeorm"

import { SurveyRepository } from "../repositories/SurveyRepository"

import { UserRepository } from "../repositories/UserRepository"

import { SurveyUserRepository } from "../repositories/SurveyUserRepository"

import { resolve } from "path"

import SendMailService from "../services/SendMailService"

class SendMailController {

    async execute(*request*: *Request*, *response*: *Response*) {

        const { email, survey\_id } = *request*.body

        const userRepository = getCustomRepository(UserRepository)

        const surveyRepository = getCustomRepository(SurveyRepository)

        const surveyUserRepository = getCustomRepository(SurveyUserRepository)

        const user = await userRepository.findOne({ email })

        if (!user) {

            return *response*.status(400).json({

                error: "User does not exists!"

            })

        }

        const survey = await surveyRepository.findOne({ id: survey\_id })

        if (!survey) {

            return *response*.status(400).json({

                error: "Survey does not exists!"

            })

        }

        const surveyUser = surveyUserRepository.create({

            user\_id: user.id,

            survey\_id

        })

        await surveyUserRepository.save(surveyUser)

        const npsMailPath = resolve(\_\_dirname, "..", "views", "emails", "npsMail.hbs")

        await SendMailService.execute(email, survey.title, survey.description)

        return *response*.json(surveyUser)

    }

}

export { SendMailController }

Mas para que a refatoração aconteça de forma correta, necessitamos receber a rota do arquivo desejado e arrumar os parâmetros para que seja enviado para o nosso SendMailService.execute(). Para isso iremos mudar o parâmetro body para variables, que serão as variáveis que estão adentradas no template de email.

no sendMailController.ts:

const variables = {

            name: user.name,

            title: survey.title,

            description: survey.description

        }

        const npsMailPath = resolve(\_\_dirname, "..", "views", "emails", "npsMail.hbs")

        await SendMailService.execute(email, survey.title, variables, npsMailPath)

no sendMailService.ts:

async execute(*to*: *string*, *subject*: *string*, *variables*: *object*, *path*: *string*) {

        const templateFileContent = readFileSync(*path*).toString("utf-8")

        const mailTemplateParse = *handlebars*.compile(templateFileContent)

        const html = mailTemplateParse(*variables*)

        const message = await *this*.client.sendMail({

            to,

            subject,

            html,

            from: "NPS <noreplace@nps.com.br>"

        })

        console.log('Message sent: %s', message.messageId)

        console.log('Preview URL: %s', nodemailer.getTestMessageUrl(message))

    }

Recebendo a avaliação do usuário para completar o surveyUser (então temos a pesquisa, que é enviada para o usuário e precisamos receber sua avaliação. Survey -> User -> Value),

Então no template, naqueles links que tem das notas da avaliação do usuário, ao clicar em um destes links conforme sua avaliação desejada, será encaminhada uma url informando a nota que ele deu e o id desse usuário, como neste esquema:

[http://localhost:3333/answers/${nota}?userId=${id\_usuario}](http://localhost:3333/answers/$%7bnota%7d?userId=$%7bid_usuario%7d)

Então receberemos a variável do link para onde o valor da nota será redirecionado, para isso citaremos a nota e o id do usuário.

A parte de resposta do template de envio de email ficará assim conforme a url:

<div *class=*"answers">

        <a *class=*"value" *href=*"{{ *link* }}/1?userId={{ *user\_id* }}">1</a>

        <a *class=*"value" *href=*"{{ *link* }}/2?userId={{ *user\_id* }}">2</a>

        <a *class=*"value" *href=*"{{ *link* }}/3?userId={{ *user\_id* }}">3</a>

        <a *class=*"value" *href=*"{{ *link* }}/4?userId={{ *user\_id* }}">4</a>

        <a *class=*"value" *href=*"{{ *link* }}/5?userId={{ *user\_id* }}">5</a>

        <a *class=*"value" *href=*"{{ *link* }}/6?userId={{ *user\_id* }}">6</a>

        <a *class=*"value" *href=*"{{ *link* }}/7?userId={{ *user\_id* }}">7</a>

        <a *class=*"value" *href=*"{{ *link* }}/8?userId={{ *user\_id* }}">8</a>

        <a *class=*"value" *href=*"{{ *link* }}/9?userId={{ *user\_id* }}">9</a>

        <a *class=*"value" *href=*"{{ *link* }}/10?userId={{ *user\_id* }}">10</a>

</div>

Então precisamos indicar o link e o id do usuário nas variables (feita para indicar as variáveis do nosso template de envio de email) do controller que automatizamos:

para indicar o link, como não queremos algo estático, ou seja, colocamos uma variável de link que pode alternar para cada tipo de envio de email, então precisamos tornar isso uma variável de ambiente, para que ela seja indicada para cada solicitação de envio de email. Com esta questão temos um arquivo que se chama .env, nele podemos colocar variáveis, então na raiz do projeto o criaremos:

USER\_ANSWER\_URL\_EMAIL=http://localhost:3333/answers

e agora podemos colocar este env como variável para o link e colocaremos também o id do usuário para que seja especificado no seu envio de nota.

const variables = {

            name: user.name,

            title: survey.title,

            description: survey.description,

            link: process.env.USER\_ANSWER\_URL\_EMAIL,

            user\_id: user.id

}

Então a aplicação pode estar funcionando normalmente

Antes de continuarmos com qualquer coisa vamos refatorar um código, pois temos um problema de que um mesmo email pode ser enviado mais de uma vez para um mesmo usuário, então se caso já houver uma pesquisa para um mesmo usuário então iremos apenas mandar um email com essas mesmas informações,

no nosso SendMailController.ts adicionamos o nosso padrão alreadyExists antes de salvá-lo para verificar se ele já existe, ou seja, se o email já foi enviado:

import { Request, Response } from "express"

import { getCustomRepository } from "typeorm"

import { SurveyRepository } from "../repositories/SurveyRepository"

import { UserRepository } from "../repositories/UserRepository"

import { SurveyUserRepository } from "../repositories/SurveyUserRepository"

import { resolve } from "path"

import SendMailService from "../services/SendMailService"

class SendMailController {

    async execute(*request*: *Request*, *response*: *Response*) {

        const { email, survey\_id } = *request*.body

        const userRepository = getCustomRepository(UserRepository)

        const surveyRepository = getCustomRepository(SurveyRepository)

        const surveyUserRepository = getCustomRepository(SurveyUserRepository)

        const user = await userRepository.findOne({ email })

        if (!user) {

            return *response*.status(400).json({

                error: "User does not exists!"

            })

        }

        const survey = await surveyRepository.findOne({ id: survey\_id })

        if (!survey) {

            return *response*.status(400).json({

                error: "Survey does not exists!"

            })

        }

        const variables = {

            name: user.name,

            title: survey.title,

            description: survey.description,

            link: process.env.USER\_ANSWER\_URL\_MAIL,

            user\_id: user.id

        }

        const npsMailPath = resolve(\_\_dirname, "..", "views", "emails", "npsMail.hbs")

        const surveyUserAlreadyExists = await surveyUserRepository.findOne({

            where: [{ user\_id: user.id }, { value: null }]

        })

        if (surveyUserAlreadyExists) {

            await SendMailService.execute(email, survey.title, variables, npsMailPath)

            return *response*.json(surveyUserAlreadyExists)

        }

        const surveyUser = surveyUserRepository.create({

            user\_id: user.id,

            survey\_id

        })

        await surveyUserRepository.save(surveyUser)

        await SendMailService.execute(email, survey.title, variables, npsMailPath)

        return *response*.json(surveyUser)

    }

}

export { SendMailController }

Agora iremos fazer uma relação da tabela surveysUsers do nosso banco de dados, pois como há o id do usuário e da pesquisa, então podemos trazer um objeto buscando as informações do usuário na tabela de usuários e a pesquisa na tabela de pesquisas, para isso mudaremos o nosso model do SurveyUser.ts, iremos fazer um tipo de relação para que ele busque na tabela de usuários, onde tenha uma coluna id igual ao user\_id da tabela do surveysUsers e a mesma coisa para o survey. Sabemos que podemos ter muitas pesquisas para um usuário, ou seja, podemos enviar uma ou mais pesquisas para um mesmo usuário:

SurveyUser.ts:

import { Column, CreateDateColumn, Entity, JoinColumn, ManyToOne, PrimaryColumn } from "typeorm";

import { *v4* as uuid } from "uuid";

import { Survey } from "./Survey";

import { User } from "./User";

@Entity("surveys\_users")

class SurveyUser {

    @PrimaryColumn()

    readonly id: *String*

    @Column()

    user\_id: *String*

    @ManyToOne(() => User)

    @JoinColumn({ name: "user\_id" })

    user: User

    @Column()

    survey\_id: *String*

    @ManyToOne(() => Survey)

    @JoinColumn({ name: "survey\_id" })

    survey: Survey

    @Column()

    value: *Number*

    @CreateDateColumn()

    created\_at: *Date*

    constructor() {

        if (!*this*.id) { *this*.id = uuid() }

    }

}

export { SurveyUser }

SendMailController.ts:

const surveyUserAlreadyExists = await surveyUserRepository.findOne({

            where: [{ user\_id: user.id }, { value: null }],

            relations: ["user", "survey"]

})

Agora finalmente terminamos o nosso envio de email...

AULA 05 – FINALIZANDO API COM VALIDAÇÕES

**Refatorando códigos**

Começaremos refatorando código, no SendMailController.ts na validação de existir um surveyUser está validando errado, pois ele verifica se tem um user\_id igual ao id de usuário ou se tem um value igual a null, também queremos especificar qual a pesquisa que ele quer que verifique em relação ao usuário, para isso iremos modificar da seguinte maneira para que as duas condições tenham que ser verdadeiras:

const surveyUserAlreadyExists = await surveyUserRepository.findOne({

            where: { user\_id: user.id, survey\_id, value: null },

            relations: ["user", "survey"]

})

Agora precisamos especificar a resposta que o usuário nos der clicando no link de acordo com a sua avaliativa desejada, nela estamos especificando quem ele é através do seu id e o valor de sua avaliativa, porém esse valor tem que estar de acordo com uma tal pesquisa, ou seja, precisamos especificar o survey\_id para que seja atribuído um valor do usuário de acordo com a pesquisa enviada a ele, para isso iremos mudar a url do nosso template de envio de email e as variables do surveyController para que ele especifique por meio do template qual a pesquisa que ele estará respondendo, para que especifique o usuário e a pesquisa, apenas temos que referenciar o id do surveyUser, especificaremos ele para caso ele já exista (buscando na tabela já existente) e caso não exista depois da parte de criar um novo pegaremos o id que será criado para que seja enviado no email:

SendMailController.ts:

import { Request, Response } from "express"

import { getCustomRepository } from "typeorm"

import { SurveyRepository } from "../repositories/SurveyRepository"

import { UserRepository } from "../repositories/UserRepository"

import { SurveyUserRepository } from "../repositories/SurveyUserRepository"

import { resolve } from "path"

import SendMailService from "../services/SendMailService"

class SendMailController {

    async execute(*request*: *Request*, *response*: *Response*) {

        const { email, survey\_id } = *request*.body

        const userRepository = getCustomRepository(UserRepository)

        const surveyRepository = getCustomRepository(SurveyRepository)

        const surveyUserRepository = getCustomRepository(SurveyUserRepository)

        const user = await userRepository.findOne({ email })

        if (!user) {

            return *response*.status(400).json({

                error: "User does not exists!"

            })

        }

        const survey = await surveyRepository.findOne({ id: survey\_id })

        if (!survey) {

            return *response*.status(400).json({

                error: "Survey does not exists!"

            })

        }

        const npsMailPath = resolve(\_\_dirname, "..", "views", "emails", "npsMail.hbs")

        const surveyUserAlreadyExists = await surveyUserRepository.findOne({

            where: { user\_id: user.id, survey\_id, value: null },

            relations: ["user", "survey"]

        })

        const variables = {

            name: user.name,

            title: survey.title,

            description: survey.description,

            link: process.env.USER\_ANSWER\_URL\_MAIL,

            id: surveyUserAlreadyExists.id

        }

        if (surveyUserAlreadyExists) {

            await SendMailService.execute(email, survey.title, variables, npsMailPath)

            return *response*.json(surveyUserAlreadyExists)

        }

        const surveyUser = surveyUserRepository.create({

            user\_id: user.id,

            survey\_id

        })

        await surveyUserRepository.save(surveyUser)

        variables.id = surveyUser.id

        await SendMailService.execute(email, survey.title, variables, npsMailPath)

        return *response*.json(surveyUser)

    }

}

export { SendMailController }

npsMail.hbs:

<div *class=*"answers">

        <a *class=*"value" *href=*"{{ *link* }}/1?su={{ *id* }}">1</a>

        <a *class=*"value" *href=*"{{ *link* }}/2?su={{ *id* }}">2</a>

        <a *class=*"value" *href=*"{{ *link* }}/3?su={{ *id* }}">3</a>

        <a *class=*"value" *href=*"{{ *link* }}/4?su={{ *id* }}">4</a>

        <a *class=*"value" *href=*"{{ *link* }}/5?su={{ *id* }}">5</a>

        <a *class=*"value" *href=*"{{ *link* }}/6?su={{ *id* }}">6</a>

        <a *class=*"value" *href=*"{{ *link* }}/7?su={{ *id* }}">7</a>

        <a *class=*"value" *href=*"{{ *link* }}/8?su={{ *id* }}">8</a>

        <a *class=*"value" *href=*"{{ *link* }}/9?su={{ *id* }}">9</a>

        <a *class=*"value" *href=*"{{ *link* }}/10?su={{ *id* }}">10</a>

    </div>

Então podemos receber a reposta do usuário de acordo com a pesquisa (email) enviada para ele, então basicamente iremos upar o valor da resposta dele na coluna de value do surveyUser, ou seja, estamos adicionando sua avaliativa para a tabela...

então criaremos mais um controller para lidar com esse request,

na pasta controllers adicionaremos um arquivo chamado AnswerController.ts, onde será adicionado a valor da avaliativa do usuário de acordo com a pesquisa mandada para ele (o dado será upado para a coluna value da tabela surveysUsers de acordo com o usuário e pesquisa). Sabemos que sua resposta será enviada de acordo com a url que ele solicitou, ou seja, estaremo recebendo um Route Params (Parâmetros que compõem a rota). Portanto ele será “pegado” a partir do nosso router, usando o método get, no qual ficará atribuído mais ou menos da seguinte forma:

router.get(“/answers/:value”)

o value indica um dos números que o usuário escolheu para atribuir sua avaliativa quanto a pesquisa,

e para identificarmos qual é a nossa pesquisa enviada para o usuário (SurveyUser) iremos usar os Query Params, no qual é o valor que vem depois do “?” na url, no qual não é obrigatório, porém podemos filtrar dados a partir dele, ou seja, iremos fazer a busca do SurveyUser a partir dele.

Para pegarmos os nossos params vindo da url atribuiremos para um objeto chamado value o request.params, assim ele receberá o valor que está vindo da url quando ela for requisitada, também pegaremos o nosso su (survey user) vindo da url usando o request.query, onde identificaremos o id do SurveyUser...

import { Request, Response } from "express"

class AnswerController {

    async execute(*request*: *Request*, *response*: *Response*) {

        const { value  } = *request*.params

        const { su } = *request*.query

    }

}

export { AnswerController }

e agora temos que chamar o nosso SurveyUserRepository para ter acesso para salvar esse value de acordo com o id da SurveyUser e verificaremos se essa id é válida:

import { Request, Response } from "express"

import { getCustomRepository } from "typeorm"

import { SurveyUserRepository } from "../repositories/SurveyUserRepository"

class AnswerController {

    async execute(*request*: *Request*, *response*: *Response*) {

        const { value  } = *request*.params

        const { su } = *request*.query

        const surveyUserRepository = getCustomRepository(SurveyUserRepository)

        const surveyUser = await surveyUserRepository.findOne({

            id: *String*(su)

        })

        if (!surveyUser) {

            return *response*.status(400).json({

                error: "Survey User does not exists!"

            })

        }

        surveyUser.value = *Number*(value)

        await surveyUserRepository.save(surveyUser)

        return *response*.json( surveyUser )

    }

}

export { AnswerController }

e agora precisamos indicar qual a rota requisitada irá executar este método da nossa classe AnswerController:

import { Router } from 'express'

import { AnswerController } from './controllers/AnswerController'

import { SendMailController } from './controllers/SendMailController'

import { SurveyController } from './controllers/SurveyController'

import { UserController } from './controllers/UserController'

const router = Router()

const userController = **new** UserController()

const surveyController = **new** SurveyController()

const sendMailController = **new** SendMailController()

const answerController = **new** AnswerController()

router.post("/users", userController.create)

router.post("/surveys", surveyController.create)

router.get("/surveys", surveyController.show)

router.post("/sendMail", sendMailController.execute)

router.get("/answers/:value", answerController.execute)

export { router }

Finalmente a resposta do usuário pode ser salva!!!!!!!!!!!!!!

**Calculando o NPS**

Com as avaliativas do usuário de acordo com a pesquisa salvas, podemos calcular o NPS de fato...

como de costume, para ter a manipulação de requests e dados criaremos um controller, na pasta de controllers será criado um arquivo NpsController.ts,

antes de calcularmos vamos entender como funciona um cálculo de NPS, então acompanhe os tipos de NPS de acordo com a avaliativa do usuário:

Detratores => 0 – 6

Passivos => 7 – 8

Promotores => 9 – 10

Para o seu cálculo, o que realmente irá importar será as notas mais baixas (Detratores) e as notas mais altas (Promotores), fazendo o seguinte cálculo relacionando a quantidade dos tipos de notas que o usuário deu com a quantidade de avaliativas. Então formularemos o cálculo na seguinte estrutura:

(Número de promotores - Número de Detratores) / (Número de respondentes) \* 100

Agora podemos manipular todos os dados necessários para o seu cálculo dentro do nosso controller para que consigamos realizá-lo por completo, então os dados que pegaremos na nossa aplicação serão os seguintes:

import { Request, Response } from "express"

import { getCustomRepository, IsNull, Not } from "typeorm"

import { SurveyUserRepository } from "../repositories/SurveyUserRepository"

class NpsController {

    async execute(*request*: *Request*, *response*: *Response*) {

        const { survey\_id } = *request*.params

        const surveyUserRepository = getCustomRepository(SurveyUserRepository)

        const surveysUsers = await surveyUserRepository.find({

            survey\_id,

            value: Not(IsNull())

        })

        const numberOfDetractors = surveysUsers.filter(

            (*surveyAnswer*) => *surveyAnswer*.value >= 0 && *surveyAnswer*.value <= 6

        ).length

        const numberOfPassives = surveysUsers.filter(

            (*surveyAnswer*) => *surveyAnswer*.value >= 7 && *surveyAnswer*.value <= 8

        ).length

        const numberOfPromoters = surveysUsers.filter(

            (*surveyAnswer*) => *surveyAnswer*.value >= 9 && *surveyAnswer*.value <= 10

        ).length

        const totalAnswers = surveysUsers.length

        const calculate = ((numberOfPromoters - numberOfDetractors) / totalAnswers) \* 100

        return *response*.json({

            surveysUsers,

            numberOfDetractors,

            numberOfPassives,

            numberOfPromoters,

            totalAnswers,

            nps: calculate

        })

    }

}

export { NpsController }

e precisamos de uma rota para requisitar o cálculo de nps conforme a survey requisitada pela url, então no routes.ts:

import { Router } from 'express'

import { AnswerController } from './controllers/AnswerController'

import { NpsController } from './controllers/NpsController'

import { SendMailController } from './controllers/SendMailController'

import { SurveyController } from './controllers/SurveyController'

import { UserController } from './controllers/UserController'

const router = Router()

const userController = **new** UserController()

const surveyController = **new** SurveyController()

const sendMailController = **new** SendMailController()

const answerController = **new** AnswerController()

const npsController = **new** NpsController()

router.post("/users", userController.create)

router.post("/surveys", surveyController.create)

router.get("/surveys", surveyController.show)

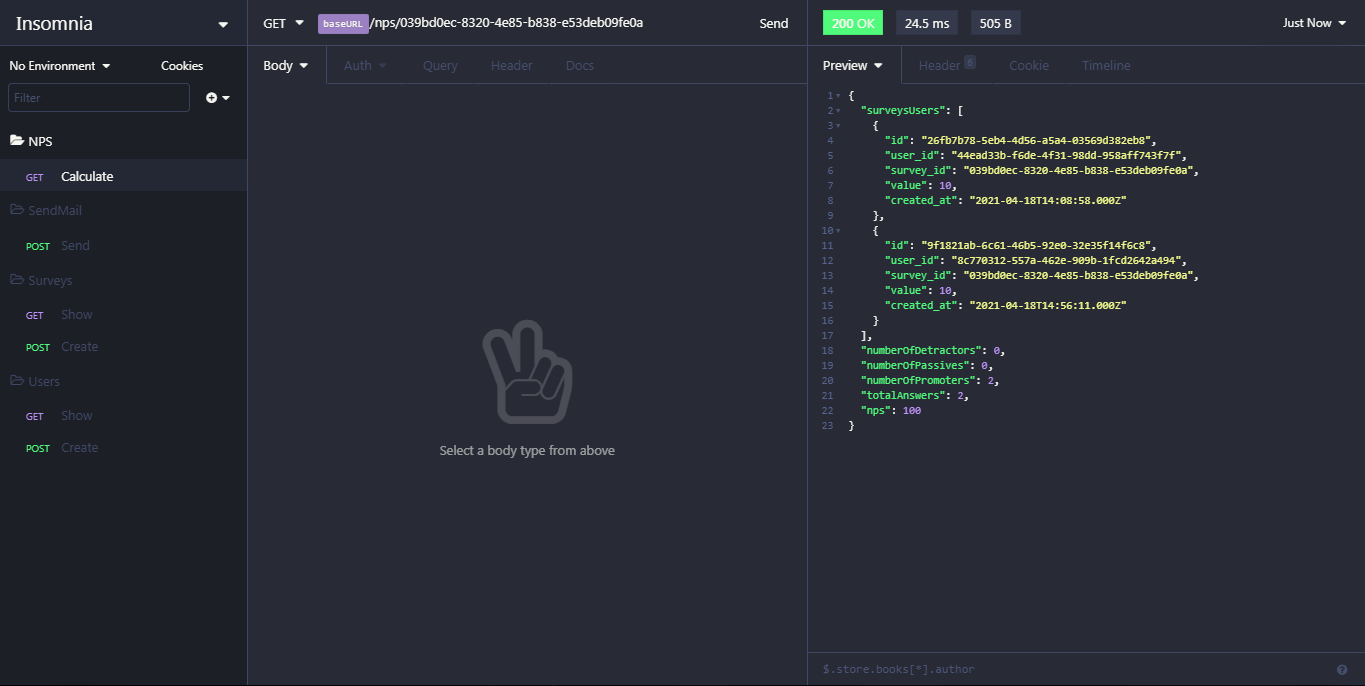
router.post("/sendMail", sendMailController.execute)

router.get("/answers/:value", answerController.execute)

router.get("/nps/:survey\_id", npsController.execute)

export { router }

e podemos testar no insmomnia criando uma nova request com esta url e passando o id de alguma survey que já foi enviada para algum user, veja um resultado:



Como o esperado ele calculou de acordo com a pesquisa na qual requisitamos através request params, retornando todas as informações que queríamos, principalmente o cálculo do nps.

**Criando as validações da api**

então precisamos instalar essa aplicação:

$ yarn add yup

e iremos importá-lo no nosso primeiro controller para fazer as validações dos dados que estão em sua “responsabilidade”.

*ps: explicação no código...*

import { Request, Response } from "express"

import { getCustomRepository } from "typeorm"

import { UserRepository } from "../repositories/UserRepository"

import \* as yup from "yup"

class UserController {

    async create(*request*: *Request*, *response*: *Response*) {

        const { name, email } = *request*.body

        const schema = yup.object().shape({

            name: yup.string().required(),

            email: yup.string().email().required()

        }) // defining what data format the request must be for it to be validated

        try {

       await schema.validate(*request*.body, { abortEarly: false })

  } catch(err) {

       return *response*.status(400).json({ error: err })

  } // if the request data format is not valid, it will return an error

        const userRepository = getCustomRepository(UserRepository)

        const userAlreadyExists = await userRepository.findOne({ email })

        if (userAlreadyExists) {

            return *response*.status(400).json({

                error: "user already exists!"

            })

        }

        const user = userRepository.create({

            name, email

        })

        await userRepository.save(user)

        return *response*.status(201).json(user)

    }

}

export { UserController }

Agora você pode testar com requests do insomnia.

**Refatorando nossa aplicação**

Um ponto importante é refatorar a aplicação para que execute em qualquer ambiente, para isso começaremos alterando o comando posttest