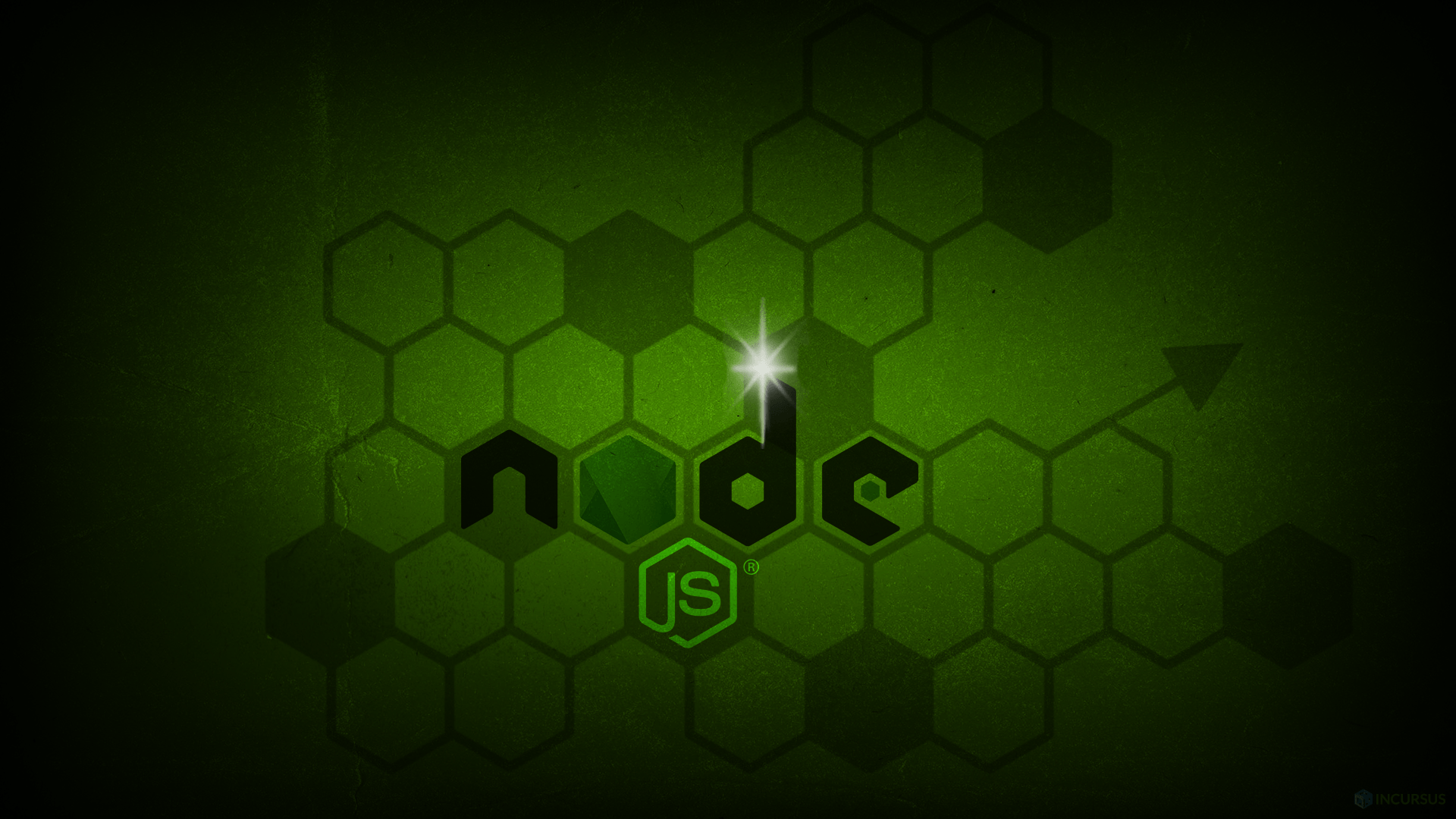
****

**API**

**NOTAS COM**

**NODE.JS**

Iniciar o node.js no projeto, depois de criar a pasta e abri-la no vscode digite no terminal

**npm init -y**

isso deve criar um arquivo chamado package.json

{

  "name": "projeto-09-testes",

  "version": "1.0.0",

  "description": "",

  "main": "index.js",

  "scripts": {

    "test": "echo \"Error: no test specified\" && exit 1"

  },

  "keywords": [],

  "author": "",

  "license": "ISC"

}

Vamos agora fazer a aplicação ser capaz de receber requisições e responder para isso instala o framework express

**npm install express --save**

Com isso na pasta do projeto aparece é instalada uma pasta **node\_modules** e um arquivo **package-lock.json**

GITIGNORE – para evitar o envio dos arquivos do node.js para o repositório porque essa pasta é muito grande

**npx gitignore node**

Com isso será criado um arquivo chamado .gitignore com a lista de todos os arquivos que serão ignorados pelo github

Para recuperar esses arquivos em outro momento (caso baixe o projeto em outra máquina) basta digitar o comando

**npm install**

crie uma pasta **src** na raiz da aplicação, para separar os arquivos da aplicação dos arquivos do framework

dentro da pasta src crie um arquivo server.js

const express = require("express")

const app = express()

const PORT = 3333

app.listen(PORT, () => console.log(`A porta do servidor é ${PORT}`))

para testar se esta tudo funcionado digite no terminal o comando

**node src/server.js**

no terminal deve mostrar a mensagem configurada no passo anterior

para deixar a execução do servidor automática usando um script no package.json

{

  "name": "projeto-09-testes",

  "version": "1.0.0",

  "description": "Api de Notas",

  "main": "index.js",

  "scripts": {

    "start": "node ./src/server.js"

  },

  "keywords": [],

  "author": "",

  "license": "ISC",

  "dependencies": {

    "express": "^4.18.2"

  }

}

Agora para executar a inicialização do servidor basta digitar no terminal

**npm start**

**ctrl + c** para parar o servidor

ROTAS: São os caminhos da aplicação



FAZENDO O GET

no arquivo server.js digite logo após a const: app = express();

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

    response.send("Ola mundo, Node.js")

})

Para testar será necessário reiniciar o servidor

**ctrl + c 🡺** para parar o servidor

**npm start 🡺** para iniciar o servidor

abra o navegador e digite **http://localhost:3333**

deverá ser exibida a mensagem na tela

se por exemplo essa rota fosse message o código ficaria assim:

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

    response.send("Ola mundo, Node.js")

})

**ctrl + c 🡺** para parar o servidor

**npm start 🡺** para iniciar o servidor

abra o navegador e digite **http://localhost:3333/message**

ROUT PARAMS

const { request } = require("express");

const express = require("express");

const app = express();

app.get("/message/:id/:message", (request, response) => {

  response.send(

    `Message id: ${request.params.id}. Message: ${request.params.message}`

  );

});

const PORT = 3333;

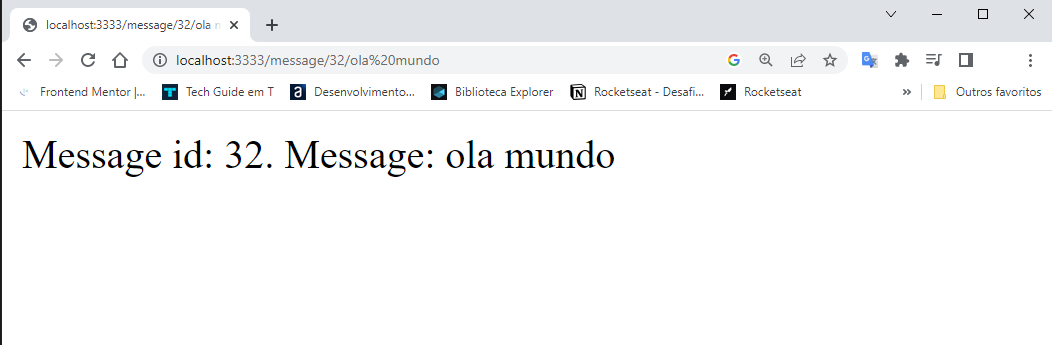
app.listen(PORT, () => console.log(`A porta do servidor é ${PORT}`));

**ctrl + c 🡺** para parar o servidor

**npm start 🡺** para iniciar o servidor

abra o navegador e digite <http://localhost:3333/message/32/ola> mundo!

o resultado deve ser:



Atenção: no ROUT PARAMS os parâmetros são obrigatórios.

**Para evitar a duplicidade de códigos podemos simplificar essa instrução com a desestruturação do request.params**

app.get("/message/:id/:message", (request, response) => {

  const { id, message } = request.params;

  response.send(

    `Message id: ${id}

     Message: ${message}`

  );

});

QUERY PARAMS

Atenção: no QUERY PARAMS os parâmetros são OPCIONAIS.

Não tem necessidade de enviar os parâmetros pela rota

const { request } = require("express");

const express = require("express");

const app = express();

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

  const { page, limit } = request.query;

  response.send(

    `Page: ${page}

     qtd máxima de linhas por páginas: ${limit}`

  );

});

const PORT = 3333;

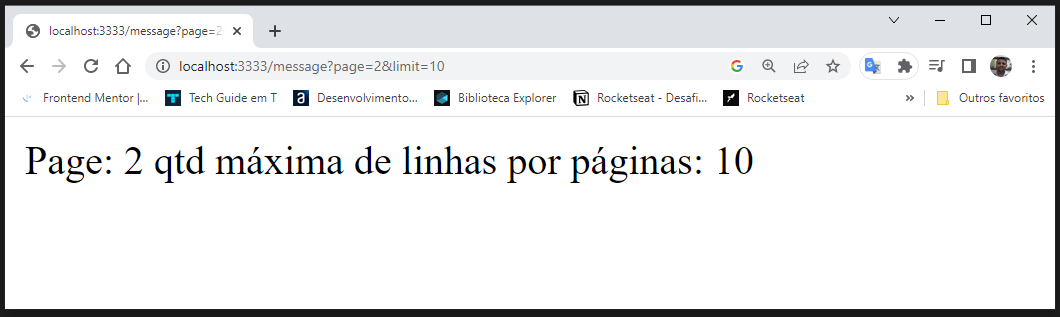
app.listen(PORT, () => console.log(`A porta do servidor é ${PORT}`));

**ctrl + c 🡺** para parar o servidor

**npm start 🡺** para iniciar o servidor

abra o navegador e digite http://localhost:3333/message?page=2&limit=10

o resultado deve ser:



**DEIXAR A REINICIALIZAÇÃO DO SERVIDOR AUTOMATICA**

Para isso podemos colocar no projeto um recurso que fica observando quando o código muda e reinicia automaticamente o servidor, para fazer isso instale uma biblioteca que é o nodemon

**npm install nodemon --save-dev**

**no arquivo package.json adicione a linha**

{

  "name": "projeto-09-testes",

  "version": "1.0.0",

  "description": "Api de Notas",

  "main": "index.js",

  "scripts": {

    "start": "node ./src/server.js",

    "dev": "nodemon ./src/server.js"

  },

  "keywords": [],

  "author": "",

  "license": "ISC",

  "dependencies": {

    "express": "^4.18.2"

  },

  "devDependencies": {

    "nodemon": "^2.0.20"

  }

}

Agora para iniciar o servidor basta digitar no terminal o comando

**npm run dev**

USANDO METODO POST

Para testar o post vamos precisar da ferramenta insominia

Insomnia.rest/download

Depois de instalar o insomnia ainda no insomnia crie uma pasta chamada Users e dentro dessa pasta cria uma

< new HTTP request > com a seguinte configuração

Name: Create

Method: POST

Body: NoBody

Voltando ao código altere o código para:

const { request } = require("express");

const express = require("express");

const app = express();

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

  response.send(`Você chamou o POST`);

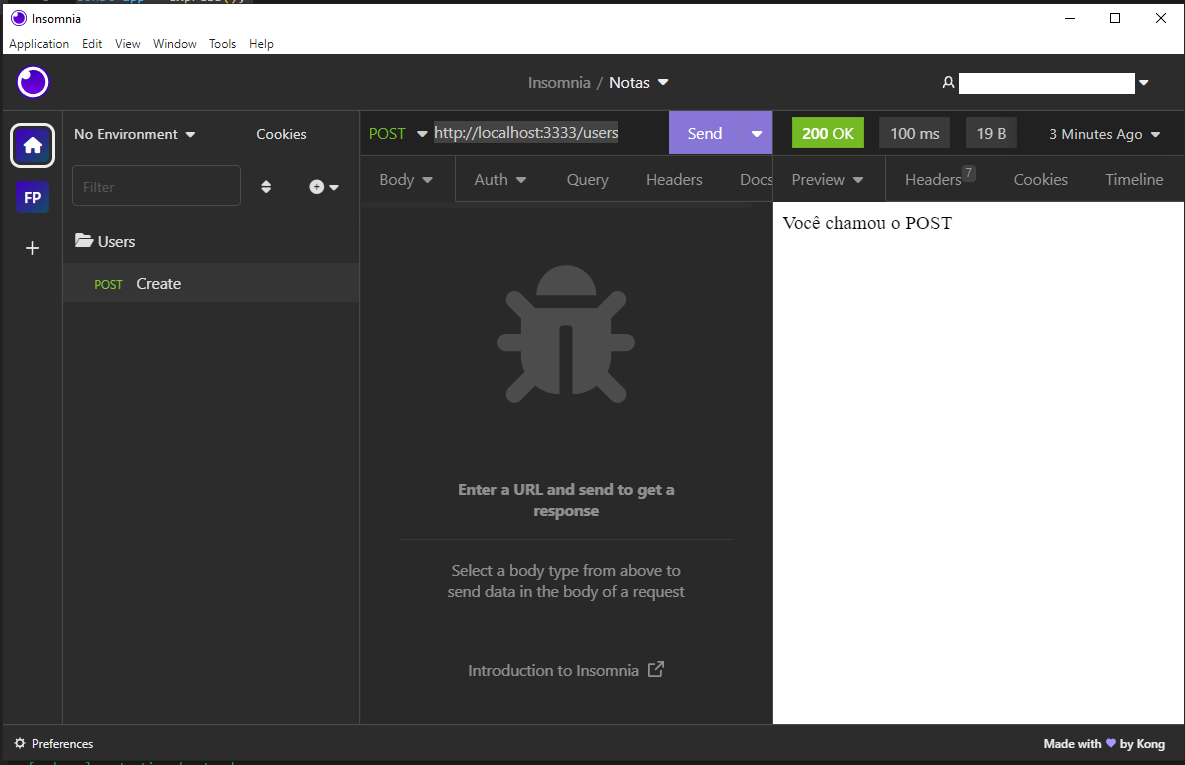
});

const PORT = 3333;

app.listen(PORT, () => console.log(`A porta do servidor é ${PORT}`));

no insomnia digite na barra de endereço: ***http://localhost:3333/users***

o resultado deve ser:



Agora vamos enviar informações para cadastrar os usuários

Mude o insomnia de BODY para JSON e crie um json

{

"name": "Nome do ususrio teste",

"email": "seu@email.com",

"password": "123"

}

No código do arquivo server.js deverá ficar assim

A linha em destaque é para avisar para o node da nossa aplicação que será usado json como resposta a e eu já aproveitei para alterar o **response.send** pelo **responso.json**

const { request } = require("express");

const express = require("express");

const app = express();

app.use(express.json())

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

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

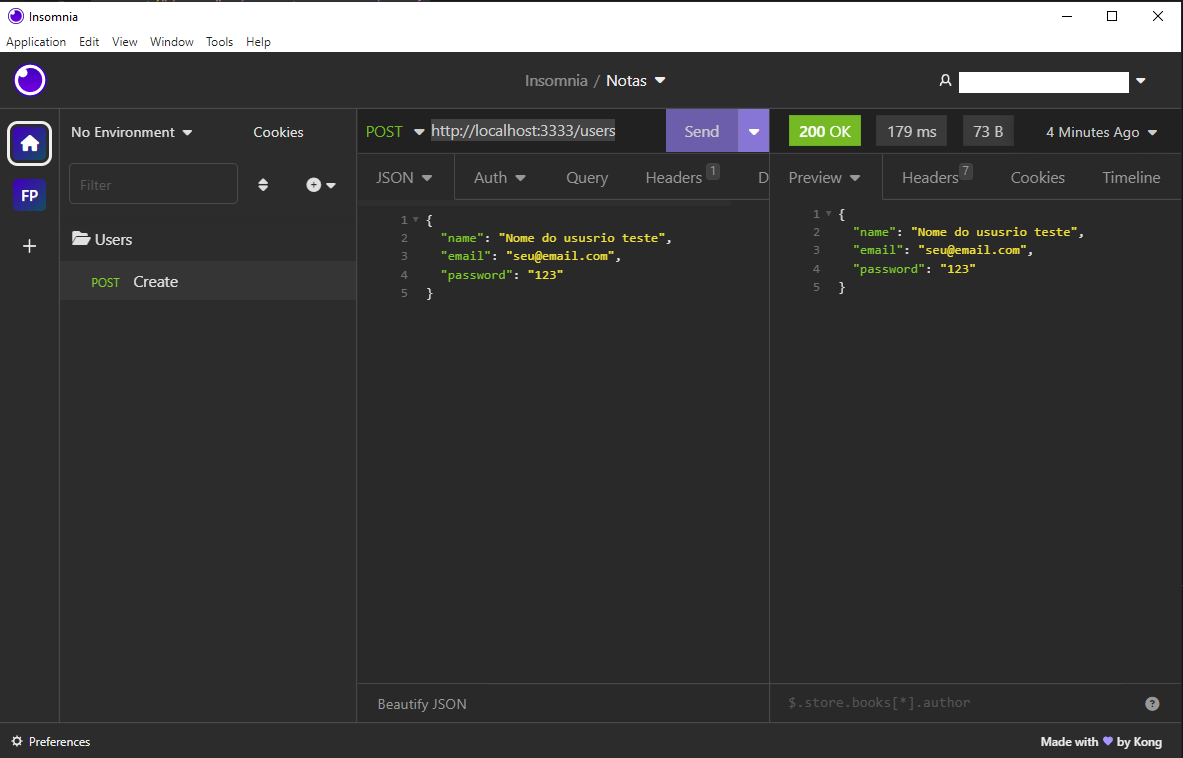
  response.json({ name, email, password });

});

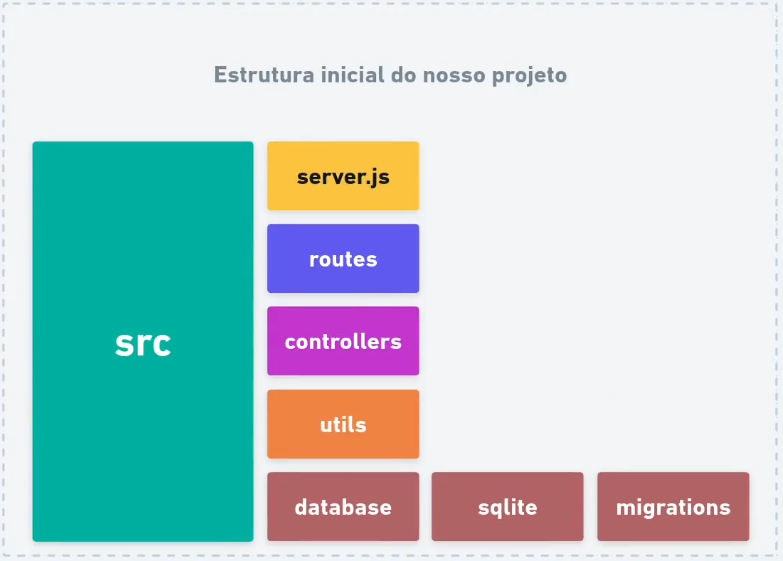
const PORT = 3333;

app.listen(PORT, () => console.log(`A porta do servidor é ${PORT}`));

o resultado deve ser



Estrutura inicial do projeto



Crie uma pasta routes dentro da pasta src

Dentro da pasta routes crie um arquivo users.routes.js

Retira a parte da rota do use recola no arquivo users.routes.js

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

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

  response.json({ name, email, password });

});

Ainda no arquivo users.routes.js

Importe o Router direto do express

const { Router } = require("express")

ai atribua a uma constante chamada usersRoutes o Router

const usersRoutes = Router()

agora é só substituir o app pelo usersRoutes

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

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

    response.json({ name, email, password });

  });

Agora é necessário expor essa rota para o server.js chegar até ela coloca no fim do arquivo

module.exports = usersRoutes;

Crie um arquivo index.js dentro da pasta routes, para importar todas as rotas é esse arquivo que vai agrupar todas

importe o Router para o index.js também

const { Router } = require("express")

criar uma nova constante usersRouter para importar a rota de users

const usersRouter = require("./users.routes")

para o Router() funcionar crie uma constante routes

const routes = Router()

necessário adicionar a rota de users

routes.use("/users", usersRouter);

agora é só exportar essa rota colocando no fim do código

module.exports = routes;

e lá no arquivo users.routes.js não precisa mais deixar a rota complete basta a “/”

usersRoutes.post("/", (request, response) => {

no arquivo **server.js importe o routes** e já pode usa-la

const express = require("express");

const routes = require("./routes");

const app = express();

app.use(express.json());

app.use(routes);

const PORT = 3333;

app.listen(PORT, () => console.log(`A porta do servidor é ${PORT}`));

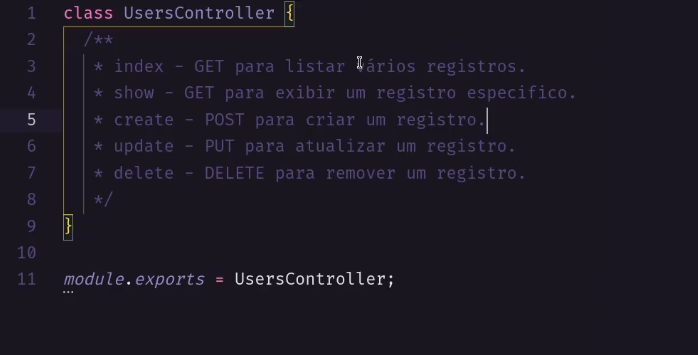
teste no insomnia o resultado é o mesmo do anterior continua funcionado da mesma forma

CONTROLLER

Crie uma pasta controllers dentro da pasta src

Dentro de controllers crie um arquivo UsersController.js (o controller vai ser criado como uma class)

Um controller pode ter no máximo 5 métodos:



No UsersController.js crie o primeiro método dentro da class UsersController

create(request, response){

}

Agora vamos no users.routes.js recorta a parte de dentro das chaves

const { name, email, password } = request.body;

  response.json({ name, email, password });

e cola dentro do create

class UsersController {

  create(request, response) {

    const { name, email, password } = request.body;

    response.json({ name, email, password });

  }

}

module.exports = UsersController;

agora precisamos importar o controller para o routes e como ele é uma class precisa ser instanciado no routes e também chamar o método **create** no lugar do request e response

const { Router } = require("express");

const UsersController = require("../controllers/UsersController");

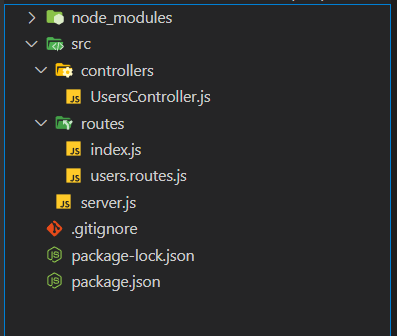
const usersRoutes = Router();

const usersController = new UsersController();

usersRoutes.post("/", usersController.create);

module.exports = usersRoutes;

faça um teste e a aplicação deverá continuar funcionando, como antes.

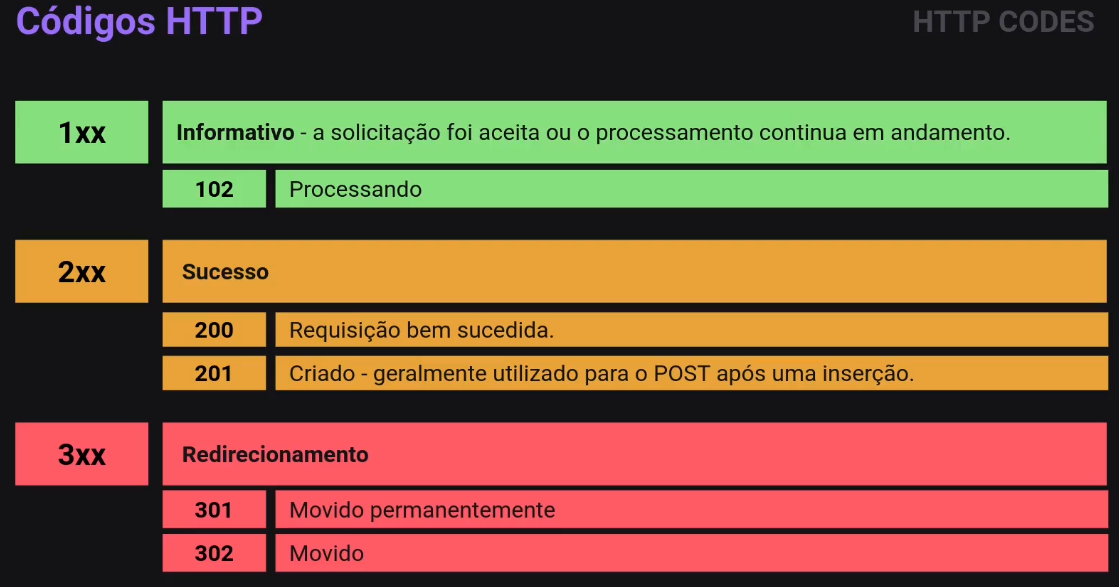
aqui podemos observar que está ficando separado em camadas cada uma com sua responsabilidade

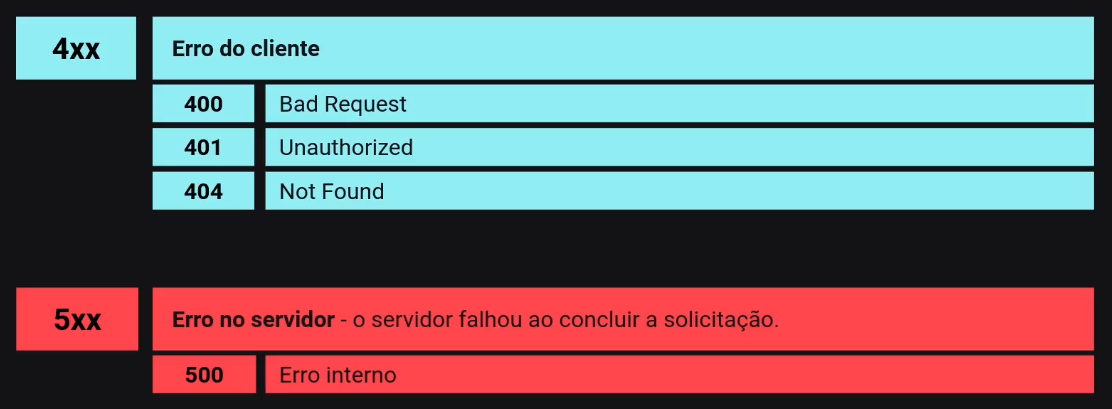
tem uma camada responsável pelas rotas para identificar os endereços as requisições da nossa aplicação que vai redirecionar para o controller correto

e o controller é quem vai lhe dar com esse processamento e trazer a resposta da requisição

e o server devolve a resposta para o cliente

STATUS CODE





COMO UTILIZAR?

No UsersController.js na linha do response

class UsersController {

  create(request, response) {

    const { name, email, password } = request.body;

    response.status(201).json({ name, email, password });

  }

}

module.exports = UsersController;

MIDDLEWARE: São funções que tem acesso ao objeto de solicitação(requisição), o objeto de resposta (resposta), e a próxima função de middleware é comumente denominada de NEXT.

TRATAMENTO DE EXESSÕES

Na pasta src crie uma pasta **utils** e na pasta utils crie um arquivo chamdo de AppError.js

class AppError {

  message;

  statusCode;

  constructor(message, statusCode = 400) {

    this.message = message;

    this.statusCode = statusCode;

  }

}

module.exports = AppError;

utilizando no UsersController.js

1º importe a class AppError para dentro da UsersController.js

const AppError = require("../utils/AppError")

para trabalhar com erros é necessário adicionar ao projeto uma biblioteca chamada **express-async-errors**

no terminal digite o comando **npm install express-async-errors**

depois de instalada a biblioteca vamos em **server.js** porque é lá que vão ser feitos os tratamentos de erros.

Agora importe a **express-async-errors** antes de tudo

require("express-async-errors")

importe também a class AppError

const AppError = "./utils/AppError";

server.js deve ficar assim ao final de todas as alterações:

require("express-async-errors");

const AppError = require("./utils/AppError");

const { request, response } = require("express");

const express = require("express");

const routes = require("./routes");

const app = express();

app.use(express.json());

app.use(routes);

app.use((error, request, response, next) => {

  if (error instanceof AppError) {

    return response.status(error.statusCode).json({

      status: "error",

      message: error.message,

    });

  }

  console.error(error);

  return response.status(500).json({

    status: "error",

    message: "Internal server ERROR.",

  });

});

const PORT = 3333;

app.listen(PORT, () => console.log(`A porta do servidor é ${PORT}`));

para teste tente enviar uma requisição sem informar o nome;

**ORGANIZANDO O INSOMNIA**

criando variáveis de ambiente no insomnia

na parte superior esquerda tem um No Environment e uma seta para baixo clicando nela vai aparecer um menu, escolha Manage Environment , vai abrir uma janela, então, click na cruzinha ao lado de Sub Environments dê um duploclick no New Environment que apareceu ao centro e digite o nome do ambiente, coloque dev para ambiente de desenvolvimento é possível escolher uma cor para esse ambientedo lado direito clicke em color depoi em Chang color e escolha a cor verde entre as chaves ao centro digite “BASE\_URL” : ”localhost:3333” no canto inferior direito clicke em close

agora no canto superio esquerdo aparece um alert click nele e selecione Use dev, para começar a usar o seu ambiente que você acabou de configurar

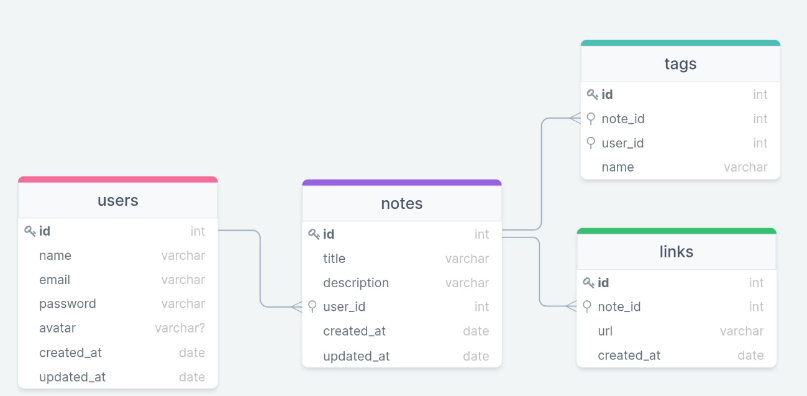
na barra de endereço digite BASE eo insomnia vai sugerir o \_.BASE\_URL criado na configuração anterior click nele

agora na lateral esquerda click na seta ao lado da pasta Users escolha </> Environment

entre as chaves na janela que se abriu digite "RESOURCE" : "users" e dê um click no close no canto inferior direito

na barra de endereço do insomnia digite logo após o \_.BASE\_URL / RESOURCE e da mesma forma como no base\_url o insomnia vai sujerir o resource recém criado e click nele





**CONECTAR COM BANCO DE DADOS**

1º instalar o sqlite3 e o sqlite, no terminal digite o comando:

**npm install sqlite3 sqlite –save**

2º na pasta src crie uma pasta database

3º na pasta database crie uma pasta sqlite para o banco de dados

Na pasta sqlite crie um arquivo index.js

Importe o sqlite3 e o sqlite

O próximo passo é criar uma função assíncrona para que a aplicação aguarde até que a conexão se estabeleça e que o arquivo do banco de dados seja criado.

const sqlite3 = require("sqlite3");

const sqlite = require("sqlite");

const path = require("path");

async function sqliteConnect() {

  const database = await sqlite.open({

    filename: path.resolve(\_\_dirname, "..", "database.db"),

    driver: sqlite3.Database,

  });

  return database;

}

module.exports = sqliteConnect;

Essa função será usada no **server.js** basta importar e executar

require("express-async-errors");

const database = require("./database/sqlite")

const AppError = require("./utils/AppError");

const { request, response } = require("express");

const express = require("express");

const routes = require("./routes");

const app = express();

app.use(express.json());

app.use(routes);

database();

app.use((error, request, response, next) => {

  if (error instanceof AppError) {

    return response.status(error.statusCode).json({

      status: "error",

      message: error.message,

    });

  }

  console.error(error);

  return response.status(500).json({

    status: "error",

    message: "Internal server ERROR.",

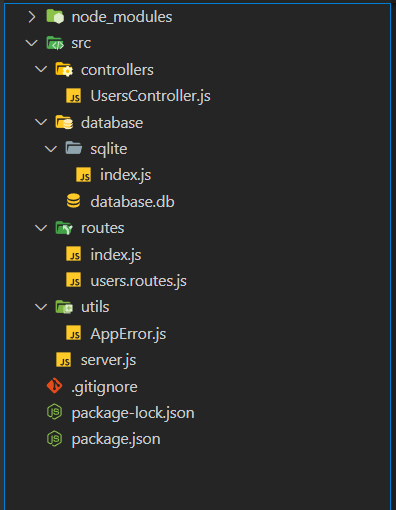
  });

});

const PORT = 3333;

app.listen(PORT, () => console.log(`A porta do servidor é ${PORT}`));

com isso na pasta database deve aparecer um novo arquivo database.db criado pelo sqlite



Para poder acessar o banco e manusear vamos precisar de um SGBD: nesse caso vamos usar o beekeeper Studio

Após a instalação do programa digite

CREATE TABLE users (

id INTEGER PRIMARY KEY AUTOINCREMENT,

name VARCHAR,

email VARCHAR,

password VARCHAR,

avatar VARCHAR NULL,

created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP,

updated\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP

)

Clicke em Run para criar a tabela users

CRIANDO UMA MIGRATION para criar a tabela de usuários no banco de dados

No projeto **criar na pasta sqlite uma pasta chamada de migrations** e nessa pasta um arquivo createUsers.js

const createUsers = `

    CREATE TABLE

    IF NOT EXISTS users (

    id INTEGER PRIMARY KEY AUTOINCREMENT,

    name VARCHAR,

    email VARCHAR,

    password VARCHAR,

    avatar VARCHAR NULL,

    created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP,

    updated\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP

  )

`;

module.exports = createUsers;

na pasta migrations crie um arquico index.js

const sqliteConnect = require("../../sqlite");

const createUsers = require("./createUsers");

async function migrationsRun() {

  const schemas = [createUsers].join("");

  sqliteConnect()

    .then((db) => db.exec())

    .catch((error) => console.error(error));

}

module.exports = migrationsRun;

AGORA VAMOS NO SERVER.JS ao invés de importar o database vamos importar o migrationsRun e execute o migrationsRun() e mudou de lugar para logo após as rotas

require("express-async-errors");

const migrationsRun = require("./database/sqlite/migrations")

const AppError = require("./utils/AppError");

const { request, response } = require("express");

const express = require("express");

const routes = require("./routes");

migrationsRun();

const app = express();

app.use(express.json());

app.use(routes);

app.use((error, request, response, next) => {

  if (error instanceof AppError) {

    return response.status(error.statusCode).json({

      status: "error",

      message: error.message,

    });

  }

  console.error(error);

  return response.status(500).json({

    status: "error",

    message: "Internal server ERROR.",

  });

});

const PORT = 3333;

app.listen(PORT, () => console.log(`A porta do servidor é ${PORT}`));

com isso se apagar o banco de dados ele será recriado automaticamente pela migration e a tabela no banco de dados também

CRIAR A FUNCIONALIDADE DE CADASTRAR USUARIOS.

No userController.js

Apague o if e o response o arquivo deve ficar assim:

const AppError = require("../utils/AppError")

class UsersController {

  create(request, response) {

    const { name, email, password } = request.body;

  }

}

module.exports = UsersController;

sendo capaz de pegar os dados do usuário

agora importe a conexão com o banco de dados

transforme a classe create em assíncrona com o async e logo após o

***const {name, email, password} = response.body;***

crie o const database com um método assíncrono usando o await para criar a conexão do banco de dados.

Fazer a verificação de existência ou não do email do usuário a ser cadastrado.

const AppError = require("../utils/AppError");

const sqliteConnect = require("../database/sqlite");

class UsersController {

  async create(request, response) {

    const { name, email, password } = request.body;

    const database = await sqliteConnect();

    const checkUserExistsEmail = await database.get(

      "SELECT \* FROM users WHERE email = (?)",

      [email]

    );

    if(checkUserExistsEmail){

      throw new AppError("Este email já está em uso!")

    }

    return response.status(201).json();

  }

}

module.exports = UsersController;

após a verificação de email coloque o código para cadastrar no banco

const AppError = require("../utils/AppError");

const sqliteConnect = require("../database/sqlite");

class UsersController {

  async create(request, response) {

    const { name, email, password } = request.body;

    const database = await sqliteConnect();

    const checkUserExistsEmail = await database.get(

      "SELECT \* FROM users WHERE email = (?)",

      [email]

    );

    if (checkUserExistsEmail) {

      throw new AppError("Este email já está em uso!");

    }

    await database.run(

      "INSERT INTO users (name, email, password) VALUES (?, ?, ?)",

      [name, email, password]

    );

    return response.status(201).json();

  }

}

module.exports = UsersController;

para testar faça uma requisição usando o insomnia o resultado deve ser um novo usuário cadastrado na tabela do banco de dados

CRIPTOGRAFANDO A SENHA

No terminal digite o comando

**npm install bcryptjs --save**

para instalar a biblioteca bcryptjs

então pegamos o hash do bcrypt para gerar a criptografia essa importação fica na 1ª linha

const { hash } = require("bcryptjs")

então antes do await database.run(

crie uma variável para criptografar a senha

const hashedPassword = await hash(password, 8)

antes desse hash temos que colocar um await

e ao invés de salvar a senha que o user informou salvamos essa criptografada

const { hash } = require("bcryptjs")

const AppError = require("../utils/AppError");

const sqliteConnect = require("../database/sqlite");

class UsersController {

  async create(request, response) {

    const { name, email, password } = request.body;

    const database = await sqliteConnect();

    const checkUserExistsEmail = await database.get(

      "SELECT \* FROM users WHERE email = (?)",

      [email]

    );

    if (checkUserExistsEmail) {

      throw new AppError("Este email já está em uso!");

    }

    const hashedPassword = hash(password, 8)

    await database.run(

      "INSERT INTO users (name, email,password) VALUES (?, ?, ?)",

      [name, email, hashedPassword]

    );

    return response.status(201).json();

  }

}

module.exports = UsersController;

para teste cadastre um novo usuário a senha deverá estar criptografada

**CRIAR ROTA PARA ATUALIZAR OS DADOS DO USUÁRIO**

No insomnia crie uma nova requisição

<new HTTP request>

Name: Update

Method: PUT

Body: JSON

use O \_BASE\_URL

e o \_RESOURCE

por enquanto vamos passar o id do usuário como parâmetro pela barra de endereço e pegar por parâmetro

agora crie um json com o nome e email dados que substituirão o cadastro

{

"name": "Francisco",

"email": "rosendo@hotmail.com"

}

Logo abaixo da função de create crie uma nova função assíncrona chamada de update

async update(request, response) {

    const { name, email } = request.body;

    const { id } = request.params;

    const database = await sqliteConnect();

    const user = await database.get("SELECT \* FROM users WHERE id = (?)", [id]);

    if (!user) {

      throw new AppError("Usuário não encontrado");

    }

    const userWithUpdatedEmail = await database.get(

      "SELECT \* FROM users WHERE email = (?)", [email]);

    if (userWithUpdatedEmail && userWithUpdatedEmail.id !== user.id) {

      throw new AppError("Este email já está em uso!");

    }

    user.name = name ?? user.name;

    user.email = email ?? user.email;

    await database.run(`

    UPDATE users SET

    name = ?,

    email = ?,

    updated\_at = DATETIME('now')

    WHERE id = ?`,[user.name, user.email, id]

    );

    return response.status(200).json();

  }

Depois disso vá em users.routes.js e acrescente uma nova rota

const { Router } = require("express");

const UsersController = require("../controllers/UsersController");

const usersRoutes = Router();

const usersController = new UsersController();

usersRoutes.post("/", usersController.create);

usersRoutes.put("/:id", usersController.update);

module.exports = usersRoutes;

Para teste tente atualizar algum usuário já cadastrado passando o id por parâmetro

UPDATE DE SENHA

async update(request, response) {

    const { name, email, avatar, password, old\_password } = request.body;

    const { id } = request.params;

    const database = await sqliteConnect();

    const user = await database.get("SELECT \* FROM users WHERE id = (?)", [id]);

    if (!user) {

      throw new AppError("Usuário não encontrado");

    }

    const userWithUpdatedEmail = await database.get(

      "SELECT \* FROM users WHERE email = (?)",

      [email]

    );

    if (userWithUpdatedEmail && userWithUpdatedEmail.id !== user.id) {

      throw new AppError("Este email já está em uso!");

    }

    user.name = name ?? user.name;

    user.email = email ?? user.email;

    user.avatar = avatar ?? user.avatar;

    if (password && !old\_password) {

      throw new AppError("Necessário informar a senha antiga.");

    }

    if (password && old\_password) {

      const checkOldPassword = await compare(old\_password, user.password);

      if (!checkOldPassword) {

        throw new AppError("Senha antiga informada não confere.");

      }

      user.password = await hash(password, 8);

    }

    await database.run(

      `

    UPDATE users SET

    name = ?,

    email = ?,

    avatar = ?,

    password = ?,

    updated\_at = DATETIME('now')

    WHERE id = ?`,

      [user.name, user.email, user.avatar, user.password, id]

    );

    return response.status(200).json();

  }

Para teste altere a senha de um usuário já cadastrado

QUERY BUILDER

Estratégia para gerar códigos sql independente do bando de dados

INSTALANDO O KNEX

Site do knex = kinex.org

npm install knex –save

após a instalação vamos gerar o arquivo de configuração do knex, para isso digite o comando

npx knex init

isso vai gerar um novo arquivo na raiz do projeto **knexfile.js** a parte que interessa é só essa

module.exports = {

  development: {

    client: 'sqlite3',

    connection: {

      filename: './dev.sqlite3'

    }

  }

};

Importe o paht 🡺 para evitar erros em caso de mudança de sistema operacional usaremos o path para resolver esse endereço

const path = require("path");

module.exports = {

  development: {

    client: 'sqlite3',

    connection: {

      filename: path.resolve(

        \_\_dirname, "src", "database", "database.db"

      )

    },

    useNullAsDefault: true

  }

};

Crie uma pasta dentro da pasta database chamada de knex e dentro dessa pasta um arquivo index.js

Vamos importar as configurações do knex para esse arquivo

const config = require("../../../knexfile")

após isso importe o knex

const knex = require("knex")

agora crie uma conexão usando o knex

const connection = knex(config.development)

e por ultimo fazer o exports para que essa connection possa ser usada em outros lugares.

const config = require("../../../knexfile");

const knex = require("knex");

const connection = knex(config.development);

module.exports = connection;

MIGRATIONS 🡺 é uma forma de versionar o banco de dados



Vá em knexfile.js para indicar em que lugar serão guardadas as informações das migrations

const path = require("path");

module.exports = {

  development: {

    client: "sqlite3",

    connection: {

      filename: path.resolve(\_\_dirname, "src", "database", "database.db"),

    },

    migrations: {

      directory: path.resolve(

        \_\_dirname,

        "src",

        "database",

        "knex",

        "migrations"

      ),

    },

    useNullAsDefault: true,

  },

}

Ai é só digitar o comando no terminal

**npx knex migrate:make createNotes**

com isso será criada dentro da pasta knex uma pasta migrations com um arquivo chamado createNotes.js e deveremos alterar para que fique assim

exports.up = (knex) => knex.schema.createTable("notes", (table) => {});

exports.down = (knex) => knex.schema.dropTable("notes");

agora no método up colocaremos como deveram ser os campos da tabela e isso é basicamente o que precisamos para criar nossa migration

exports.up = (knex) => knex.schema.createTable("notes", (table) => {

    table.increments("id")

    table.text("title")

    table.text("description")

    table.integer("user\_id").references("id").inTable("users")

    table.timestamp("created\_at").default(knex.fn.now())

    table.timestamp("updated\_at").default(knex.fn.now())

});

exports.down = (knex) => knex.schema.dropTable("notes");

Para rodar uma migration use no terminal o comando:

**npx knex migrate:latest**

para que fique mais simples podemos transformar esse comando de criação de migrations em um script

vamos em pacage.json onde já havíamos criado uns scripts de start, de dev, criaremos outro de migrate

 "migrate" : "knex migrate:latest"

Agora crie o migration “crateTags”

No terminal use o comando:

**npx knex migrate:make createTags**

será gerado um arquivo dentro da pasta migrations dentro da pasta knex xxxxxxxxxxxxxx\_createTags.js

volte ao xxxxxxxxxxxxxx\_crateNotes.js e copie todo o conteúdo

abra o arquivo xxxxxxxxxxxxxx\_createTags.js e cole

então ajuste para que fique assim

exports.up = knex => knex.schema.createTable("tags", table => {

    table.increments("id");

    table.text("name").notNullable();

    table.integer("note\_id").references("id").inTable("notes").onDelete("CASCADE");

    table.integer("user\_id").references("id").inTable("users");

  });

exports.down = (knex) => knex.schema.dropTable("tags");

no terminal digite o comando

$ npx knex migrate:make createLinks para criar a migration de Links

exports.up = (knex) =>

  knex.schema.createTable("links", (table) => {

    table.increments("id");

    table.text("url").notNullable;

    table.integer("note\_id").references("id").inTable("notes")

.onDelete(“CASCADE”);

    table.timestamp("created\_at").default(knex.fn.now());

 });

exports.down = (knex) => knex.schema.dropTable("links");

no terminal use o comando:

**$ npm run migrate**

O resultado deve ser:

> projeto-09-api@1.0.0 migrate

> knex migrate:latest

Using environment: development

Batch 2 run: 2 migrations

E as duas tabelas devem ser criadas no banco de dados.

Por padrão a funcionalidade de apagar em cascata no sqlite é desabilitada por padrão, para resolver isso,

Vá no arquivo knexfile.js

Logo após a connection{...}, adicione um **pool**

pool: {

      afterCreate: (conn, cb) => conn.run("PRAGMA foreign\_keys = ON", cb)

    },

FUNCIONALIDADE DE CRIAR UMA NOVA NOTA

No insomnia crie uma pasta para Notes

Dentro já cria uma nova rota

<new HTTP request>

Name: Create

Method: POST

Body: JSON

use O \_BASE\_URL

e cria o \_RESOURCE

<clica em environment Overrides (JSON Format)>

{

“RESOURCE” : “notes”

}

Clica em DONE

Então já cria a nota:

{

"title":"Introdução ao nodejs",

"description" : "Essa é uma nota de exemplo 1.",

"tags" : ["node", "express"],

"links" : ["link1", "link2"]

}

Não existe essa rota ainda mas antes de fazer ela crie um novo controller

Clique na pasta controllers

Crie um novo arquivo: NotesController.js

1º importa o knex do database

const knex = require("../database/knex")

2º cria a class NotesController

OBS.: O id do user por enquanto vai ser passado manualmente na barra de endereço lá no insomnia

Vamos utilizar a função map para percorrer cada item da lista

const knex = require("../database/knex");

class NotesController {

  async create(request, response) {

    const { title, description, tags, links } = request.body;

    const { user\_id } = request.params;

    const note\_id = await knex("notes").insert({

      title,

      description,

      user\_id,

    });

    const linksInsert = links.map((link) => {

      return {

        note\_id,

        url: link,

      };

    });

    await knex("links").insert(linksInsert);

    const tagsInsert = tags.map((name) => {

      return {

        note\_id,

        name,

        user\_id,

      };

    });

    await knex("tags").insert(tagsInsert);

    response.json();

  }

}

module.exports = NotesController;

3º vai na pasta routes

Copia o user.routes.js para aproveitar a estrutura

Cria o notes.routes.js

const { Router } = require("express");

const NotesController = require("../controller/NotesController")

notesRoutes = Router();

const notesController = new NotesController()

notesRoutes.post("/:user\_id", notesController.create);

module.exports = notesRoutes;

volta no código e importa a ./notes.routes no index.js de dentro da pasta routes

const { Router } = require("express");

const usersRouter = require("./users.routes")

const notesRouter = require("./notes.routes")

const routes = Router()

routes.use("/users", usersRouter)

routes.use("/notes", notesRouter)

module.exports = routes;

para testar crie uma nova nota passando o user\_id como parâmetro na barra de endereço do insomnia

{

"title":"Introdução ao nodejs",

"description" : "Essa é uma nota de exemplo 1.",

"tags" : ["node", "express"],

"links" : ["link1", "link2"]

}

CRIAR UMA ROTA PARA MOSTRAR OS DADOS DA NOTA

No insomnia <new HTTP request>

Name: Show

Method: GET

Body: NoBody

No notesController.js

Cria outro método logo abaixo do async create(request, response){...}

async show(request, response) {

    const { id } = request.params;

    const note = await knex("notes").where({ id }).first();

    return response.json(note);

  }

Para poder usar essa rota tem que ir no notes.routes.js e acrescenta-la logo após o

notesRoutes.post("/:user\_id", notesController.create);

notesRoutes.get("/:id", notesController.show);

para pegar as tags e os links adiciona dentro do método show

const tags = await knex("tags").where({note\_id : id}).orderBy("name")

const links = await knex("links").where({note\_id : id}).orderBy("created\_at")

e adiciona no return também

return response.json({

        ...note, tags, links

    });

CRIAR ROTA PARA DELETAR ALGUMA NOTA

No insomnia <new HTTP request>

Name: Delete

Method: DELETE

Body: NoBody

Agora lá no notesController.js cria um novo método clamado delete

 async delete(request, response) {

    const { id } = request.params;

    await knex("notes").where({ id }).delete();

    return response.json();

  }

Agora vamos em notes.routes.js e adiciona a rota para delatar notas

const { Router } = require("express");

const NotesController = require("../controller/NotesController")

notesRoutes = Router();

const notesController = new NotesController()

notesRoutes.post("/:user\_id", notesController.create);

notesRoutes.get("/:id", notesController.show);

notesRoutes.delete("/:id", notesController.delete);

module.exports = notesRoutes;

LISTANDO AS NOTAS DE UM USUÁRIO ESPECIFICO PESQUISANDO PELO ID

No insomnia <new HTTP request>

Name: Index

Method: GET

Body: NoBody

Adiciona a \_BASE\_URL/\_RESOURCE

Agora lá no código no arquivo notesController.js

Cria uma nova função chamada index

async index(request, response) {

    const { user\_id } = request.query;

    const notes = await knex("notes")

    .where({ user\_id })

    .orderBy("title");

    return response.json(notes);

  }

Para conseguir usar esse método vá em notes.routes.js

const { Router } = require("express");

const NotesController = require("../controller/NotesController")

notesRoutes = Router();

const notesController = new NotesController()

notesRoutes.get("/", notesController.index); //adiciona essa linha

notesRoutes.post("/:user\_id", notesController.create);

notesRoutes.get("/:id", notesController.show);

notesRoutes.delete("/:id", notesController.delete);

module.exports = notesRoutes;

FUNCIONALIDADE PESQUISAR PELO TITULO DA NOTA

Continua no Notes.Controller.js adiciona o .whererLike(...) logo após o .where({ user\_id })

async index(request, response) {

    const { user\_id } = request.query;

    const notes = await knex("notes")

    .where({ user\_id })

    .whereLike("title", `%${title}%`);

    .orderBy("title");

    return response.json(notes);

  }

APLICAR FILTRO PELAS TAGS

No insomnia na aba Query

Adiciona mais um item agora o tags

No código do NotesController.js

crie a variável let notes.

e retire o const do: **const notes = await knex(“notes”)...**

vamos fazer um if(...){...} e o código do index fica assim:

async index(request, response) {

    const { title, user\_id, tags } = request.query;

    let notes;

    if (tags) {

      const filterTags = tags.split(',').map(tag => tag);

      notes = await knex("tags").whereIn("name", filterTags)

    } else {

      notes = await knex("notes")

        .where({ user\_id })

        .whereLike("title", `%${title}%`)

        .orderBy("title");

    }

    return response.json(notes);

  }

Com isso é possível trazes apenas as tags sem informações

USANDO INNER JOIN

Array com quais campos eu quero selecionar de cada tabela

Para não causar nenhuma confusão note que cada campo tem também o nome da tabela a qual ela pertence

        .select(["notes.id", "notes.title", "notes.user\_id"])

Agora adiciona um novo filtro para filtrar pelas tags que sejam do id deste usuário que esta sendo utilizado.

        .where("notes.user\_id", user\_id)

Agora aplicando o like para poder selecionar os títulos das notas sem necessidade de digitar exatamente como foi cadastrado:

        .whereLike("notes.title", `%${title}%`)

Logo após o .whereIn(“name”, filterTags) que já tinha sido feito antes adiciona o INNER JOIN, para conectar a tabela

Aqui coloca o nome da tabela a qual quer conectar e os campos que serão usados para conectar elas dentro de notas tem o notas.id e dentro de tags tem tags.nota\_id

        .innerJoin("notes", "notes.id", "tags.note\_id")

        .orderBy("notes.title");

Código final

async index(request, response) {

    const { title, user\_id, tags } = request.query;

    let notes;

    if (tags) {

      const filterTags = tags.split(",").map((tag) => tag);

      notes = await knex("tags")

        .select(["notes.id", "notes.title", "notes.user\_id"])

        .where("notes.user\_id", user\_id)

        .whereLike("notes.title", `%${title}%`)

        .whereIn("name", filterTags)

        .innerJoin("notes", "notes.id", "tags.note\_id")

        .orderBy("notes.title");

    } else {

      notes = await knex("notes")

        .where({ user\_id })

        .whereLike("title", `%${title}%`)

        .orderBy("title");

    }

    return response.json(notes);

  }

CONCEITO DE MAP E FILTER

Map é uma função do Java script para percorrer cada elemento do array eo o map retorna outro array apenas com o resultado

const tags = [

  {id: 1, name: "node", note\_id: 1},

  {id: 2, name: "express", note\_id: 1},

  {id: 3, name: "react", note\_id: 1},

  {id: 4, name: "javascript", note\_id: 2},

  {id: 5, name: "frontend", note\_id: 2},

  ];

  const mewArray = tags;

é possível criar um novo array a partir desse:

  const newArray = tags.map(*tag* => {

return{

      name: tag.name

    }

  });

  console.log(newArray);

resultado deve ser um objeto para cada tag somente com o nome:

(5) [{...}, {...}, {...}, {...}, {...}]

0:{name: "node"}

1:{name: "express"}

2:{name: "react"}

3:{name: "javascript"}

4:{name: "frontend"}

Porque a partir de tags fiz um newArray com o map e percorrendo cada item peguei só o nome

Se quiser retornar todos os itens dentro de tegs, não precisa nem colocar chaves

 const newArray = tags.map(*tag* => tag);

  console.log(newArray);

ou se preferir usar as chaves pode ser assim:

const newArray = tags.map(*tag* => {

    return tag;

    });

  console.log(newArray);

nos dois casos o resultado deve ser:

(5) [{...}, {...}, {...}, {...}, {...}]

0:(3) {id: 1, name: "node", note\_id: 1}

id:1

name:"node"

note\_id:1

1:(3) {id: 2, name: "express", note\_id: 1}

id:2

name:"express"

note\_id:1

2:(3) {id: 3, name: "react", note\_id: 1}

id:3

name:"react"

note\_id:1

3:(3) {id: 4, name: "javascript", note\_id:...}

id:4

name:"javascript"

note\_id:2

4:(3) {id: 5, name: "frontend", note\_id: 2...}

id:5

name:"frontend"

note\_id:2

sintax reduzida

const newArray = tags.map(*tag* => tag.name);

  console.log(newArray);

só que agora o resultado é um novo vetor só com os nomes de cada elemento de tags:

(5) ["node", "express", "react", "javasc...]

0:"node"

1:"express"

2:"react"

3:"javascript"

4:"frontend"

E se quiser retornar tudo o que já tem dentro de cada tag e adicionar mais alguma coisa usa o spred operator

 const newArray = tags.map(*tag* => {

    return{

      ...tag,

      date: new Date()

    }

  });

  console.log(newArray);

o resultado deve ser dentro de cada elementos dentro dos itens que já tinha agora tem a data dentro de cada um deles

(5) [{...}, {...}, {...}, {...}, {...}]

0:(4) {id: 1, name: "node", note\_id: 1, da...}

id:1

name:"node"

note\_id:1

date:Fri Dec 16 2022 15:34:29 GMT-0300 (Horário Padrão de Brasília)

1:(4) {id: 2, name: "express", note\_id: 1,...}

2:(4) {id: 3, name: "react", note\_id: 1, d...}

3:(4) {id: 4, name: "javascript", note\_id:...}

4:(4) {id: 5, name: "frontend", note\_id: 2...}

O MAP é para manipular um array e retornar um novo array, funciona como uma estrutura de repetição;

O FILTER é uma função do Java script usada para quando você quer filtrar o conteúdo do array, retorna um novo array só que filtrada.

const newArray = tags.filter(*tag* => tag.note\_id === 1);

  console.log(newArray);

o resultado deve ser somente as tags de note\_id = 1

(3) [{...}, {...}, {...}]

0:(3) {id: 1, name: "node", note\_id: 1}

id:1

name:"node"

note\_id:1

1:(3) {id: 2, name: "express", note\_id: 1}

id:2

name:"express"

note\_id:1

2:(3) {id: 3, name: "react", note\_id: 1}

id:3

name:"react"

note\_id:1

...voltando a aula...

Queremos que quando enviar a consulta retornar as tags com mais informações vinculadas

Logo após o else crie uma constante e fazer um filtro que retorne todas as tags onde o id\_user seja o mesmo

 const userTags = await knex("tags").where({ user\_id });

cria outra constante de notas com as tags e vai percorrer todas as notas e filtar as tags da nota

 const userTags = await knex("tags").where({ user\_id });

    const noteWithTags = notes.map((note) => {

      const noteTags = userTags.filter((tag) => tag.note\_id === note.id);

      return {

        ...note,

        tags: noteTags,

      };

    });

    return response.json(noteWithTags);

  }

LISTAR TAGS

No insomnia

Criar uma nova pasta chamada Tags

No insomnia <new HTTP request>

Name: Index

Method: GET

Body: NoBody

Usa a \_.BASE\_URL/

Cria um <environment> chamado \_.RESOURCE

Por enquanto o id do user vai ser passado por parâmetro na barra de endereço

Voltando ao código dentro da pasta controllers crie um novo arquivo TagsController.js

const knex = require("../database/knex");

class TagsController {

  async index(request, response) {

    const { user\_id } = request.params;

    const tags = await knex("tags").where({ user\_id });

    return response.json(tags);

  }

}

module.exports = TagsController;

na pasta routes cria um arquivo tags.routes.js

const { Router } = require("express");

const TagsController = require("../controller/TagsController");

tagsRoutes = Router();

const tagsController = new TagsController();

tagsRoutes.get("/:user\_id", tagsController.index);

module.exports = tagsRoutes;

e no index.js da pasta routes cria a nova rota

const { Router } = require("express");

const usersRouter = require("./users.routes")

const notesRouter = require("./notes.routes")

const tagsRouter = require("./tags.routes")

const routes = Router()

routes.use("/users", usersRouter)

routes.use("/notes", notesRouter)

routes.use("/tags", notesRouter)

module.exports = routes;

REVIÃO

Passo 1 – npm run dev 🡺 ao iniciar o servidor o banco de dados é criado com a tabela de users (porque a migration do sqlite é executada de forma automática)

Passo 2 – npm run migrate 🡺 roda a migration e cria as outras tabelas

Passo 3 – cria um usuário pelo insomnia

Passo 4 – ao tentar cadastrar novamente o mesmo user o app não pode deixar informando **email já esta em uso**

Passo 5 – atualizar o usuário passando só nome e email diferentes do cadastrado

Passo 6 – tentar atualizar a senha passando só a senha nova, o app deve requerer que seja passada a antiga também

Passo 7 – tentar atualizar a senha passando senha antiga diferente da que esta cadastrada

Passo 8 – atualizar usuário usando a senha antiga correta nesse caso deve funcionar

Passo 9 – teste de criação de notas