**Criando uma API**

**npm init -y** = cria o JSON da API.

**npm install express -save** = Instala a Framework Express. A partir desta instalção nodejs cria a pasta node\_modules e o arquivo package-lock.json que carrega todas as configurações da aplicação.

Crie o arquivo .gitignore, dentro dele coloque node\_modules, assim quando um comit for feito a pasta node\_modules não vai ser comitada, pois se ela for apagada basta colocar npm install no terminal que ela volta puxando as configuralções dos arquivos json.

Crie uma pasta src, em seguida crie um arquivo server.js dentro da mesma e faça as seguintes configurações:

const express = require("express");//Importa o express

const app = express();//Inicializa o Express

const PORT = 3333;

app.listen(PORT, () => console.log(`Server is running on Port ${PORT}`));

**Para inicializar a API** basta executar o server.js com o node no terminal:

**node src/server.js**

**Se quiser encerrar a aplicação** basta apertar Contrl + C no terminal.

Ou então podemos fazer uma alteração na configuração scripts do arquivo package.json para inicializarmos :

 "scripts": {

    "start": "node ./src/server.js"

 },

Deste modo não precisamos digitar o comando "node src/server" e sim "npm start"

**Para dar agilidade no processo de desenvolvimento**, podemos instalar o **nodemon**, que é uma ferramenta do Vscode que serve para atualizar o servidor local automaticamente na máquina, ao invés de ter que parar o servidor e reiniciar em toda alteração.

Para instalar o nodemon executamos o seguinte comando no terminal:

**npm install nodemon –save-dev**

Após a sua instalação podemos notar que um novo parâmetro foi acrescentado no arquivo package.json, o devDependecies.

"devDependencies": {

    "nodemon": "^2.0.22"

}

Para rodarmos nossa api com o nodemon temos que fazer uma alteração manualmente no arquivo package.json:

"scripts": {

    "start": "node ./src/server.js",

    "dev": "nodemon ./src/server.js"

  },

Agora para a aplicação rodar através do nodemon, abrimos o terminal e digitamos o seguinte comando:

**npm run dev**

**Metodos de requisição:**

GET = leitura

POST = criação

PUT = atualização

DELETE = deleção

PATCH = atualização parcial

**Metodo GET route params** exemplo:

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

    response.send(`

        Mensagem ID: ${request.params.id}.

        Para o usuário: ${request.params.user}.

    `);

});

Podemos abreviar o request da seguinte forma:

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

    const { id, user } = request.params;

    response.send(`

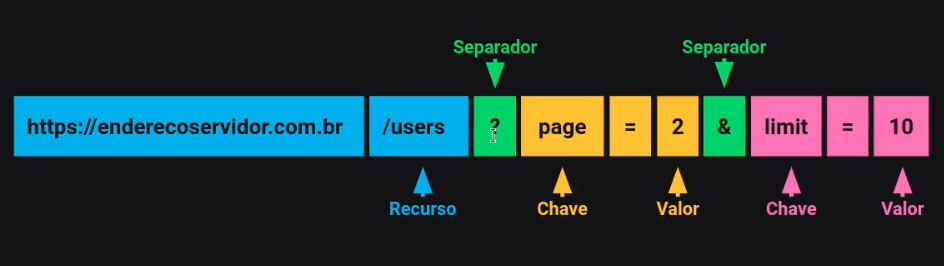
        Mensagem ID: ${id}.

        Para o usuário: ${user}.

    `);

});

**Metodo GET query params** estrutura:



Exemplo:

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

    const { page, limit } = request.query;

    response.send(`Página: ${page}. Mostrar: ${limit}`);

})

O **async-errors** é uma biblioteca que trata erros. Para instala-la basta digitar o seguinte comando no terminal:

**npm install express-async-errors --save**

O SQLite é um banco de dados relacional. Para instala-lo basta digitar o seguinte comando no termianal:

**npm install sqlite3 sqlite --save**

Para **criptografar senha** precisamos instalar um pacote no terminal:

**npm install bcryptjs**

Um Query Builder é um construtor de consulta, que nos permite construir instruções SQL independente do banco de dados utilizado. No nosso caso usaremos a biblioteca Knex do node:

**npm install knex --save**

Para a configuração do knex é necessário criar um arquivo através do node. Para isso usamos o seguinte comando:

**npx knex init**

Após criar o arquivo knex.js podemos abri-lo e começar a configura-lo. Ao abrir, podemos apagar os comentários e deixar apenas as seguintes instruções:

const path = require("path");//padroniza a navegação entre pastas independete do sistema operacional.

module.exports = {

  development: {

    client: 'sqlite3',

    connection: {

      filename: path.resolve(\_\_dirname, "src", "database", "database.db")//Aqui colocamos o caminho do nosso db.

    },

    migrations: {

      directory: path.resolve(\_\_dirname, "src", "database", "knex", "migrations")//Aqui colocamos o caminho da nossa migration, que é responsavel pelo versionamento do nosso banco.

    },

    useNullAsDeFault: true

  },

};

Na sequencia criamos uma pasta com o nome knex dentro da pasta database, dentro da pasta knex criamos um arquivo index.js para importar as configurações do arquivo knex. Dentro deste index.js colocamos as seguintes configurações:

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

const knex = require("knex");

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

const knex = require("knex");

const connection = knex(config.development);

module.exports = connection;

**Migration:** é uma forma de versionar a base de dados. Ela trabalha na manipulação da base de dados, criando, alterando ou removendo algo sem impactar o que não foi alterado.

Uma **migration possui dois métodos:**

**UP:** método responsável por criar ou alterar algo no banco de dados.

**DOWN:** responsável pelo rollback, ou seja, desfazer as alterações realizadas pela migration.

Para criar o arquivo de configuração da migration temos que primeiro criar a pasta migrations dentro da nossa pasta knex e na sequencia executar o seguinte comando no terminal:

**npx knex migrate:make createNotes** (obs: createNotes é o nome do arquivo)

Após criarmos este arquivo, configuramos ele da seguinte forma**:**

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

  //Aqui criamos a tabela. Exemplo:

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");

Podemos criar uma tabela vinculada a outra. Ao fazer isso também podemos usar uma função no campo referenciado a tabela para que se a primeira tabela for apagada a vinculada também seja. A função é **“.onDelete(“CASCADE”)**. Exemplo:

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

Para esta função funcionar temos que habilita-la, pois por padrão ela vem desabilitada no sqlite. Para isso temos que ir no arquivo knex.js na raiz do projeto e acrescentar o seguinte comando:

    pool: {

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

    }, //Colocamos entre connection e migrations.

Após criar a tabela dentro do campo rodamos o knex da seguinte maneira:

**npx knex migrate:latest**

Ou então podemos fazer uma alteração na configuração scripts do arquivo package.json para rodarmos a migration:

"scripts": {

    "start": "node ./src/server.js",

    "dev": "nodemon ./src/server.js",

    "migrate": "knex migrate:latest"

  },

E rodar o seguinte comando:

**npm run migrate**

**Para gerenciarmos o Banco de Dados,** usamos Beekeeper baixado no github, que é a versão gratuita deste excelente **SGBD** (Sistema de Gerenciamento de Banco de Dados ou Data Base Management System)

**SQL – Structured Query Language, ou Linguagem de Consulta Estruturada.** É a linguagem padrão para banco de dados relacionais. Dentro da linguagem SQL temos grupos de comandos. Comandos DDL, comandos DML**.**

**SQL – Comandos DDL** (Data Definition Language) são os comandos que definem a estrutura de armazenamento de dados, exemplos: CREATE, DROP, ALTER.

**SQL – Comandos DML** (Data Manipulation Language) são os comandos usados para manipular os dados de uma tabela, exmplos: INSERT, READ, UPDATE, DELETE

**Tipos de dados SGBD:**

**INTEGER** = número inteiro

**VARCHAR** = texto

**TIMESTAMP** = data

**Comandos para SGBD:**

**CREATE TABLE** = cria tabela

**DROP** = deleta

**ALTER** = atualiza informações

**PRIMARY KEY** = cria uma chave primária

**AUTOINCREMENT** = preenche automaticamente

**NULL** = cria um campo não obrigatório

**DEFAULT CURRENT\_TIMESTAMP** = por padrão preenche com a data atual.

**RENAME TO** = renomear para

**ADD** = adiciona um campo

Exemplo de manipulação de dados:

**INSERT INTO** users **/\*Adiciona dados a uma tabela\*/**

(name, email, password)

**VALUES**

('João', 'joao@email.com', '456');

**SELECT** \* **FROM** users; **/\*Seleciona toda a tabela\*/**

**SELECT** id, name, password **FROM** users **/\*Especifica quais colunas selecionar\*/**

**UPDATE** users **SET**

name = 'Juninho',

email = 'juninho@email.com',

avatar = 'junin.png'

**WHERE** id = 1 **/\*Sempre lembrar de especificar pela chave primária\*/**

**DELETE FROM** users

**WHERE** id = 2 **/\*Se não colocar where e especificar todos os dados da tabela são apagados\*/**

Arquitetura Back End

