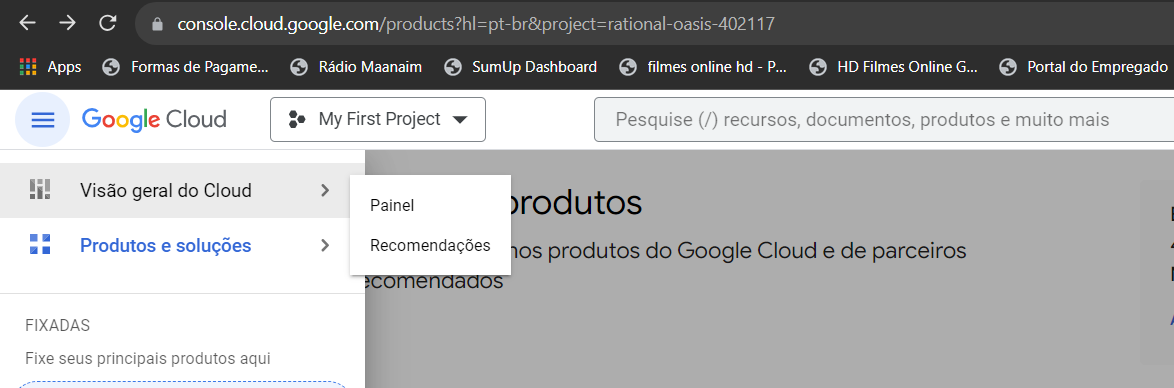
**SEÇÃO 13 – API REST – JAVASCRIPT, NODE, EXPRESS, SQL(MariaDb, MySql)**

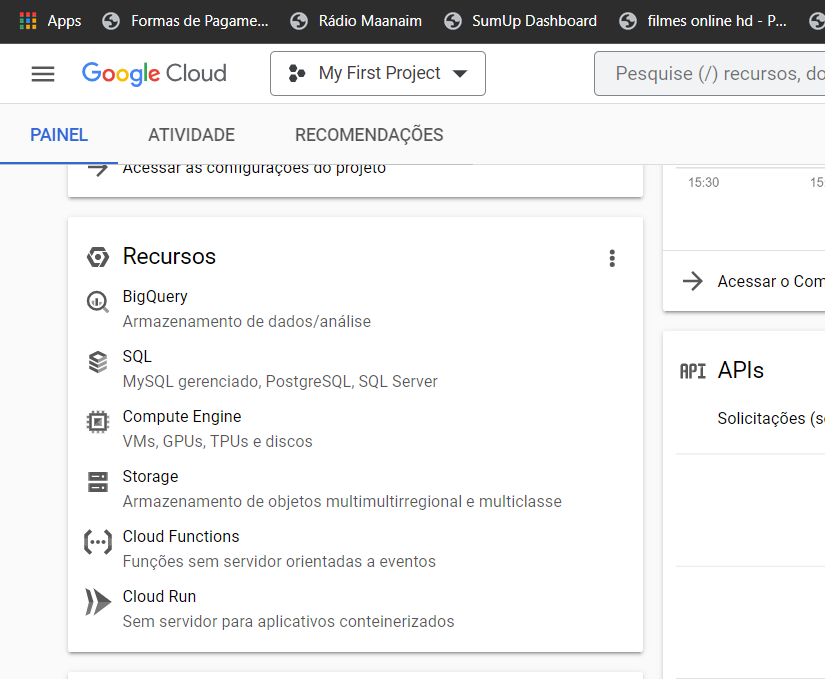
Vamos separa totalmente o backend do front através de uma API para o backend. Iremos trabalhar com o SQL nessa seção.

**168 – Configurando o ambiente, instalando o Insome e mysqlWorkbeanch.**

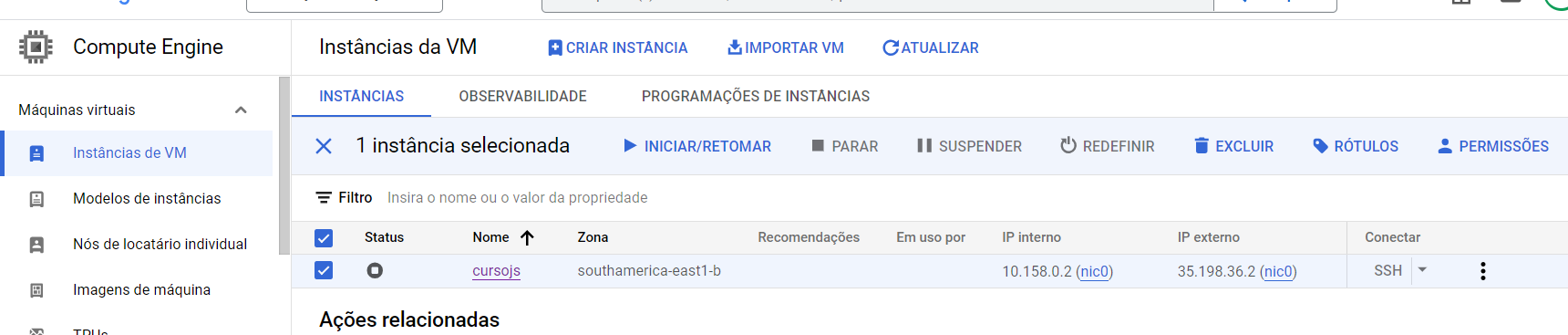
Para começar vamos acessar o painel do google cloud



Em recursos clique em “computer engine”



Selecione a instancia do servidor e clique em inicar/retomar.



Depois vamos copiar o ip externo do servidor e abrir o git bash, vamos abrir o servidor, para isso digite ‘ls’ e veremos o comando criado para inicializar o servidor ‘iniciaServidor.sh’ digite então ‘./iniciaServidor.sh’.

Nele vamos digitar ‘ssh IP\_copiado’, dessa forma iremos iniciar o servidor.

Vamos primeiramente verificar se há alguma versão do Docker instalada.

Digite os comandos abaixo:

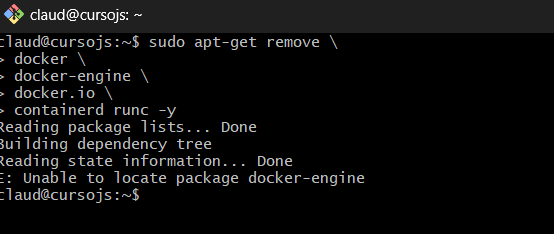
sudo apt-get remove \

docker \

docker-engine \

docker.io \

containerd runc -y

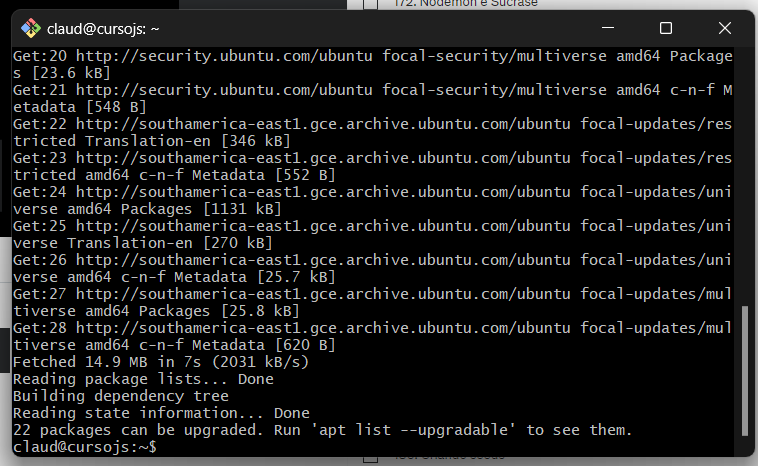


Vemos que o Docker não está instalado na máquina.

Vamos digitar o comando:

sudo apt update

Para verificar se há alguma atualização necessária no servidor.

****

Vemos que existem pacotes para serem atualizados, vamos digitar então :

Sudo apt upgrade

Após a atualização vamos instalar alguns pacotes necessários para o Doker comunity edition que é o gratuido, para isso basta digitar os comandos abaixo no servidor:

sudo apt install \

apt-transport-https \

ca-certificates \

curl \

gnupg-agent \

software-properties-common -y

Agora vamos baixar o pacote do Docker para isso digite:

curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add –

Após baixar vamos atualizar o repositório do servidor com os comandos abaixo:

- sudo add-apt-repository \

"deb [arch=amd64] https://download.docker.com/linux/ubuntu \

$(lsb\_release -cs) \

stable" -y

Antes de realizar a instalação vamos novamente atualizar o servidor:

Sudo apt update

Agora iremos instalar o Docker, para isso digite os comandos:

- sudo apt install docker-ce docker-ce-cli containerd.io -y

Após a instalação do Docker, vamos criar o container do mariaDb utilizando a porta 3306 que é a porta padrão do MySql, vamos dar o nome de ‘bdmariadb1’ ao banco de dados. Vamos então digitar o comando:

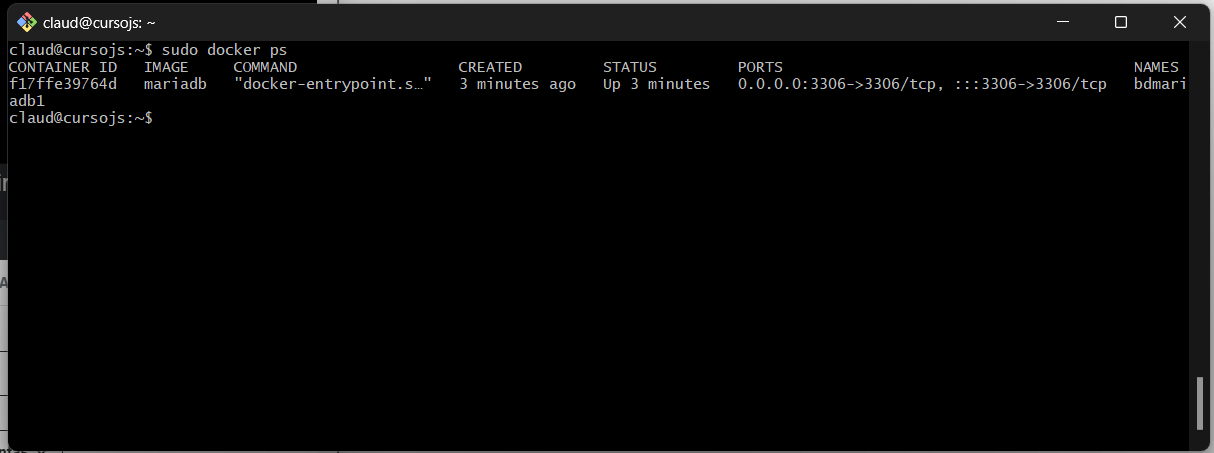
- sudo docker run --restart always -d --name bdmariadb1 -p 3306:3306 \

-e MYSQL\_ROOT\_PASSWORD=cDsc@00003 mariadb

Agora vamos verificar se foi iniciado o container corretamente, digite:

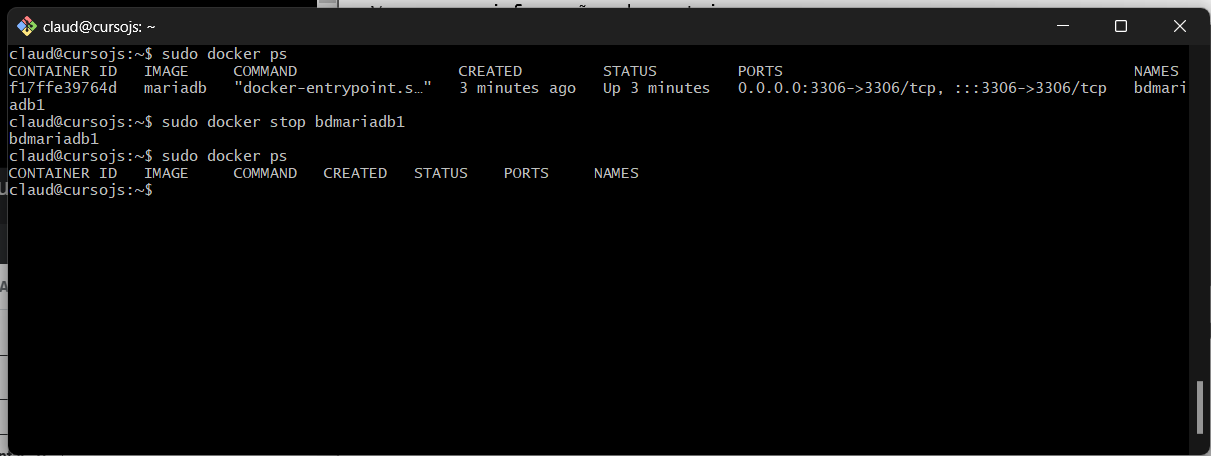
- sudo docker ps

Veremos as informações do container.



Para pararmos este servidor devemos digitar:

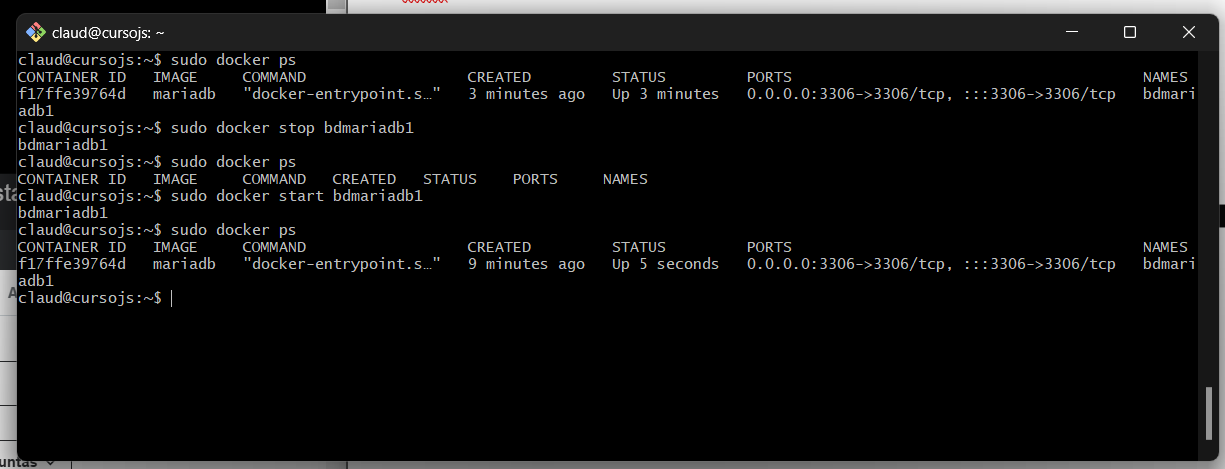
- sudo Docker stop bdmariadb1(nome dado ao banco de dados na criação do container)



Vemos que o servidor foi interrompido e container esvaziado.

Para iniciar novamente digite:

- sudo Docker start dbmariadb1



Vemos novamente o container inicializado.

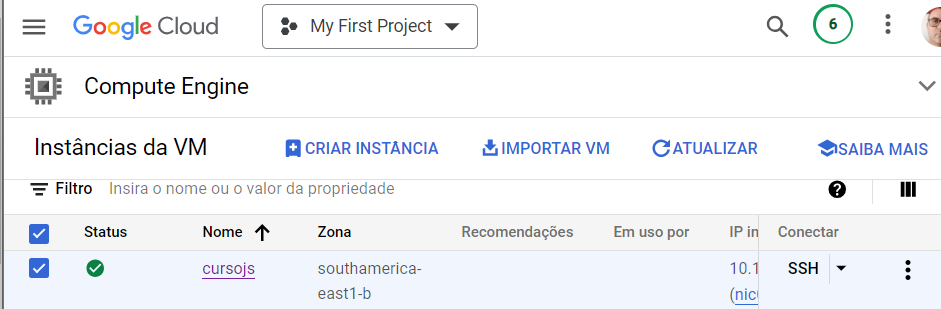
Se quisermos reiniciar o servidor basta utilizar o comando:

- sudo Docker restart bdmariadb1

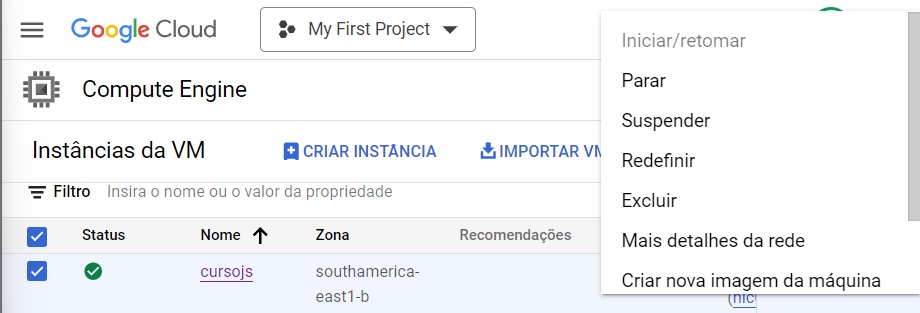
Se quisermos remover(apagar) do servidor o container vamos utilizar o comando:

- sudo Docker rm bdmariadb1

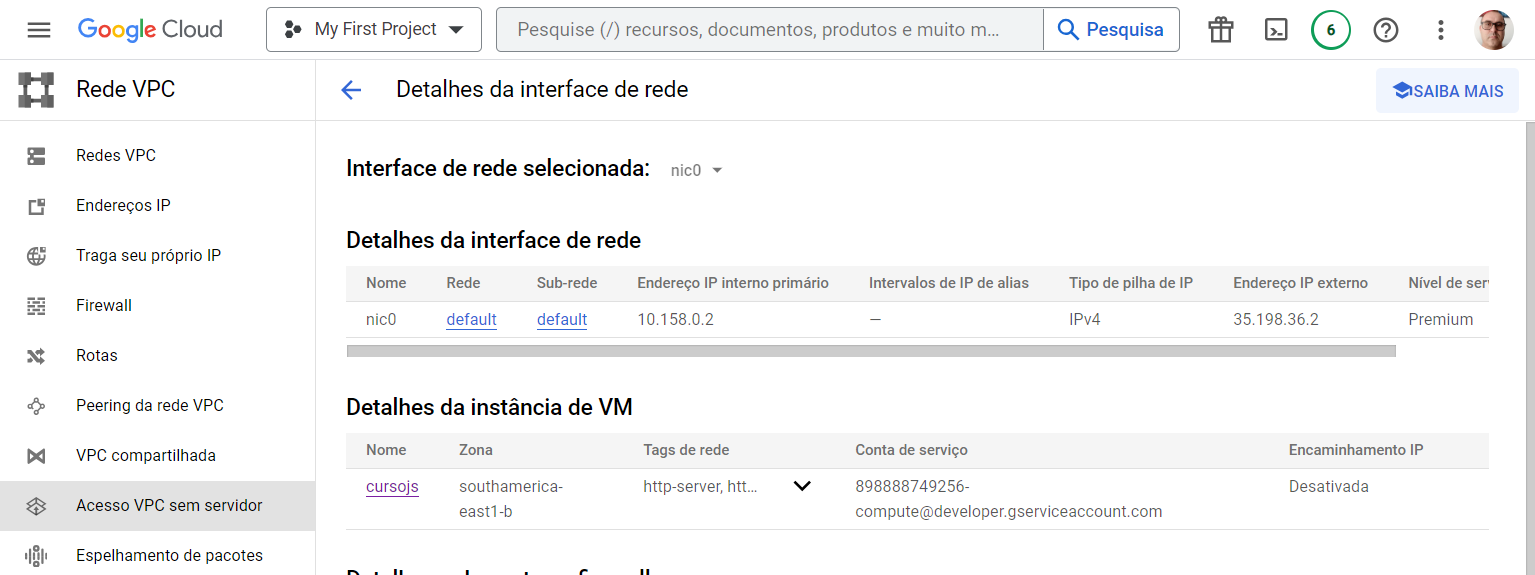
Para podermos acessar o Docker por fora do servidor, temos que ir no google clound e na instância de VM do cursosjs vamos clicar nos três botões à esquerda



e selecionar “Mais detalhes da rede”.



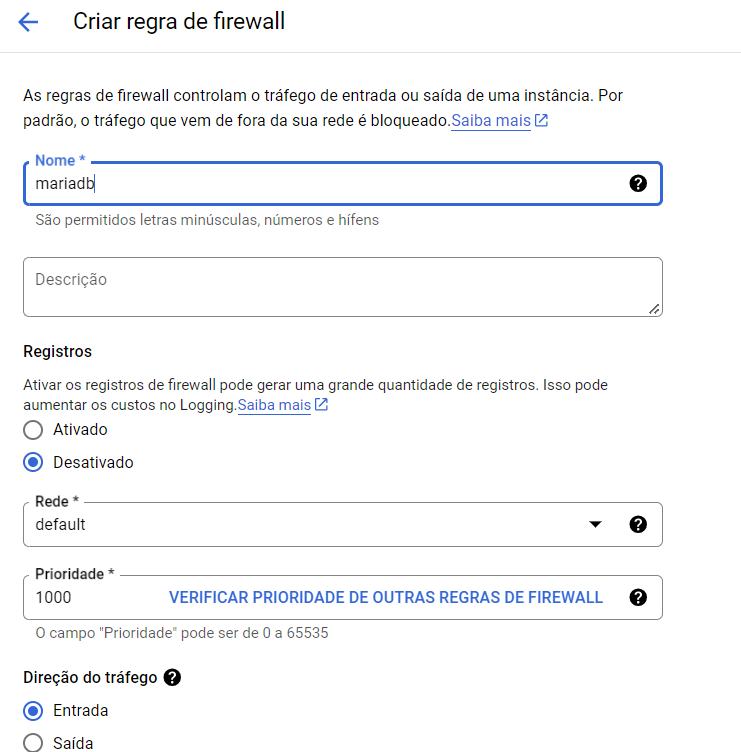
Agora vamos clicar em “Firewall”



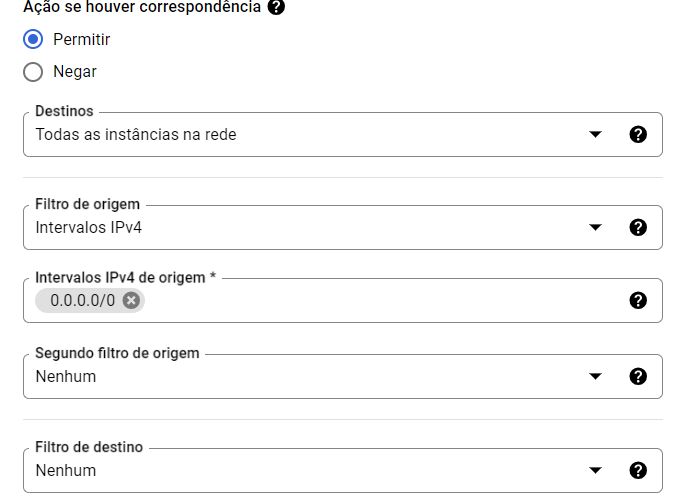
Temos que criar uma nova regra para o firewall permitir o acesso a porta 3306 que é a porta do MySql. Clique em “CRIAR REGRA DE FIREWALL”



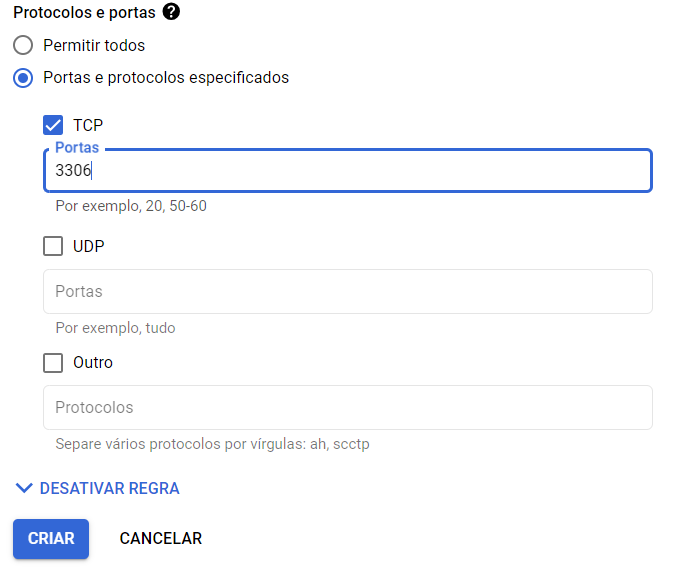
Na tela de configuração vamos dar o nome de “mariadb”



Vamos alterar o campo “Destinos” para “todas as intâncias na rede” temos ainda que alterar o campo “Intevalos de IPv4 de origem” vamos inserir o IP 0.0.0.0/0 que corresponde a qualquer rede.



Temos que selecionar o campo “TCP” informando no campo a porta “3306”. E na sequëncia vamos clicar em “CRIAR”.



Após realizar a configuração da porta no GOOGLE CLOUD vamos ter que testar o acesso, para isso vamos utilizar o “MYSQL WORBENCH” e o “INSOMNIA” no nosso comutador.

Para instalar no UBUNTU basta utilizar os comandos:

- sudo apt install mysql-workbench

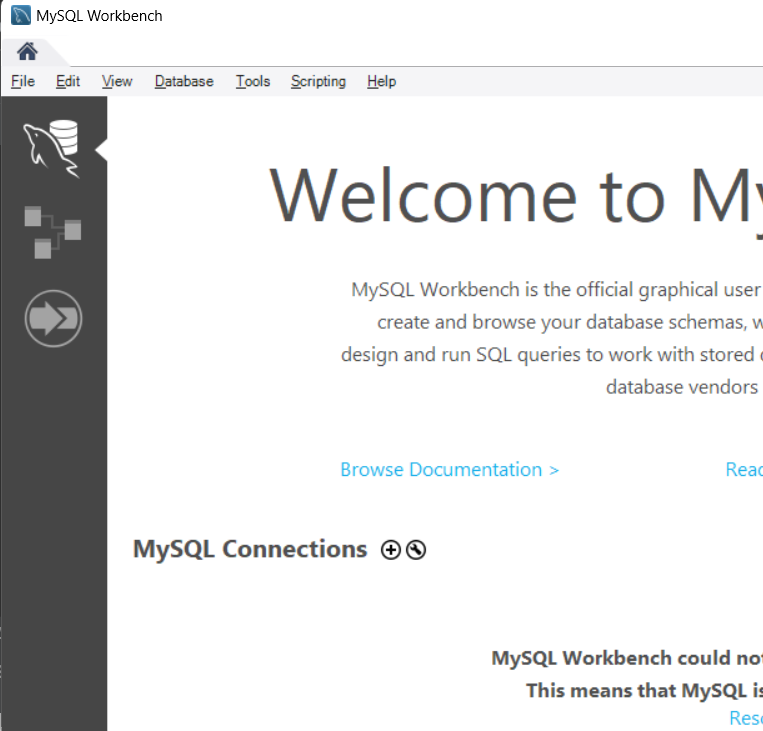
- sudo snap install insomnia

No Windows vamos utilizar o link:

- <https://dev.mysql.com/downloads/workbench/>

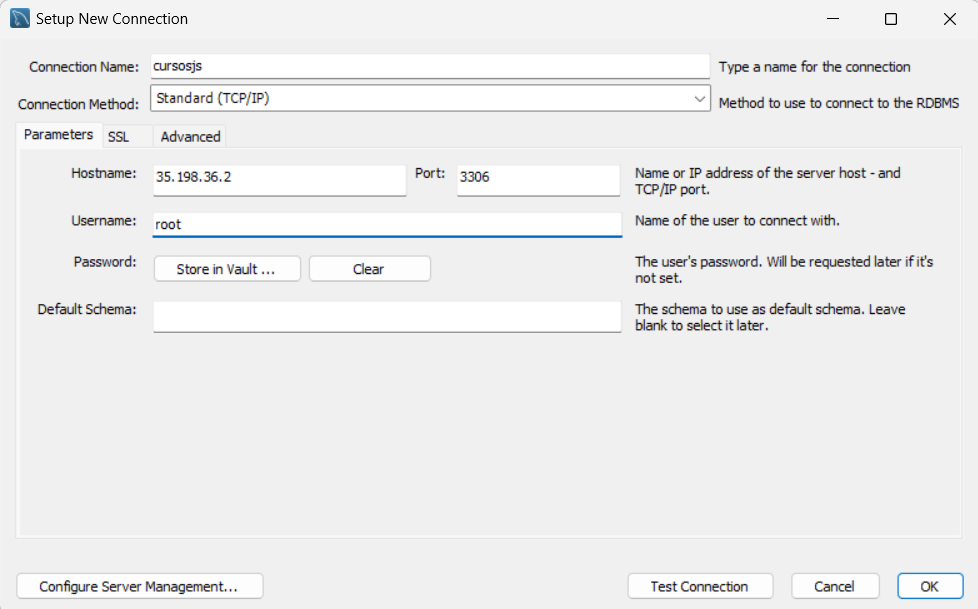
- https://insomnia.rest/download/

Vamos agora abrir o “MySQL Workbench” e clicar no sinal de “mais” em frente à “MySQL Connections”



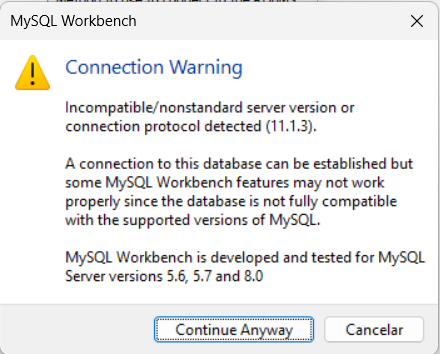
Vamos dar um nome para a conexão igual ao nome da VM do GOOGLE CLOUND, chamado “cursosjs”.

No campo “Hostname” vamos adicionar o IP externo da VM “cursosjs” do GOOGLE CLOUD”.

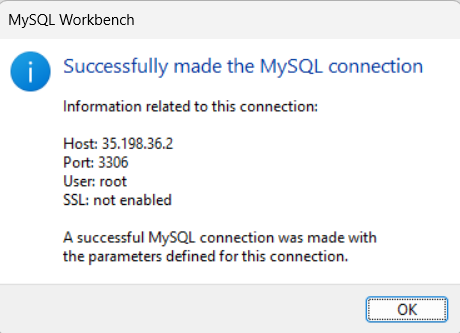


Agora vamos clicar em “Test connection”. Será solicitado a inserção de uma senha.Neste caso será a mesma senha utilizada para criar o container do “bdmariadb” no Docker.

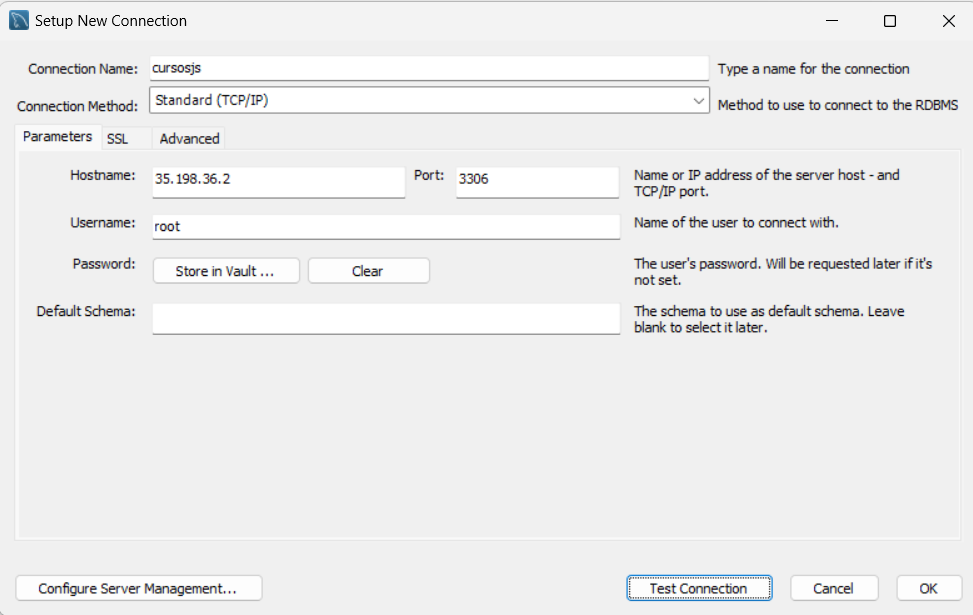
Vemos que foi lançado um warning porém funciona corretamente.



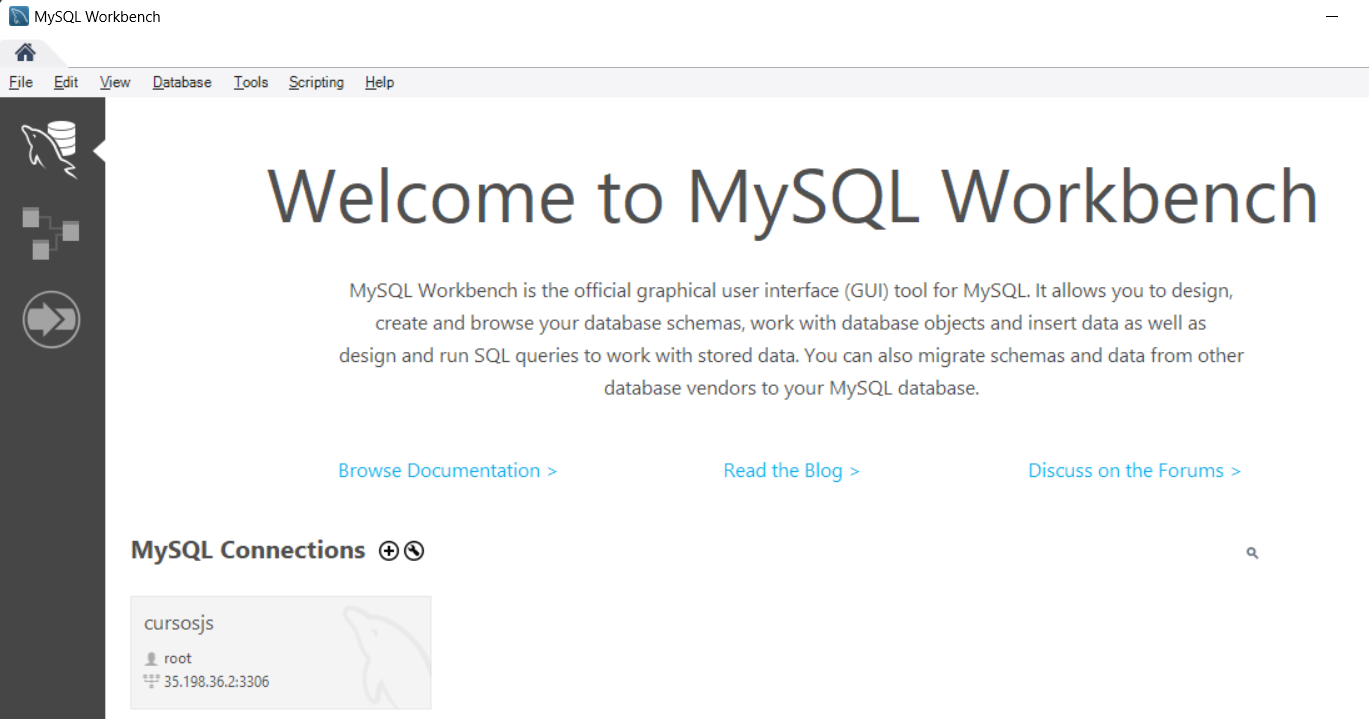
Clique em “Continue Anyway” e será exibida a mensagem de conexão realizada com sucesso.



Para finalizar a criação clique em “ok”.



Agora já podemos ver nossa conexão no painel principal do “workbench”.

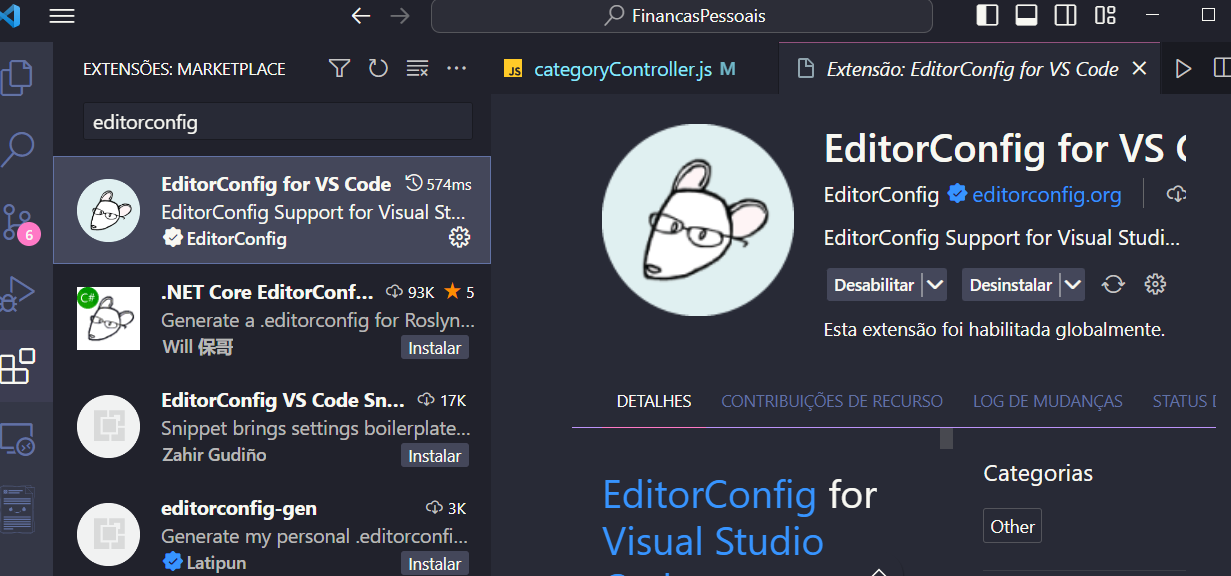


Agora vamos instalar o “Insomnia”.

- https://insomnia.rest/download/

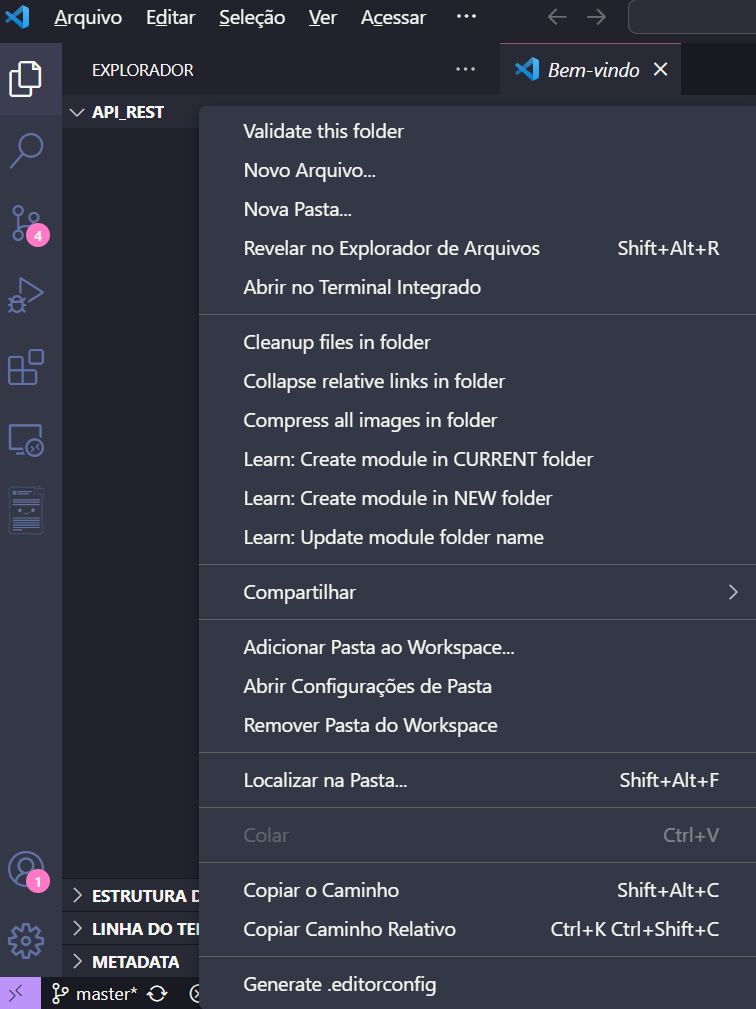
**169 – Editorconfig**

No vsCode vamos localizar uma extenção chamada “editorconfig for vscode”

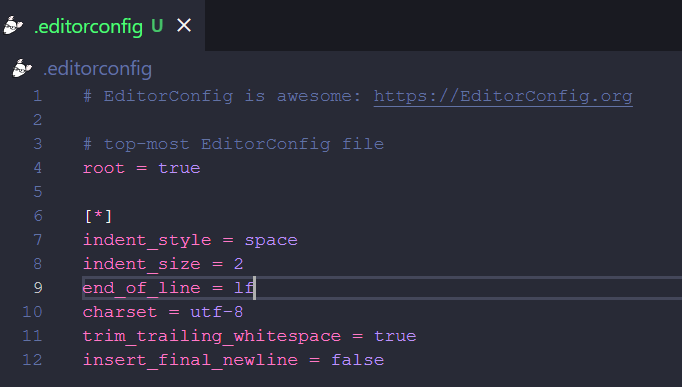


Após a instalação da extenção vamos criar a pasta do projeto “API\_REST”.

Vamos abrir a pasta no Vscode e dentro dela vamos clicar com o botão direito e selecionar “Generate.editorconfig”.



Será criado o arquivo “.editorconfig” nele iremos realizar as configurações abaixo:



170- Instalando o ESLINT

Com o vscode aberto na pasta do projeto vamos iniciar o projeto com :

- npm init -y

Agora vamos configurar o “eslint” ele irá nos forçar a digitar um código melhor, digite:

- npx eslint –-init

Vamos selecionar as opções:

- “To check syntax, find problems, and enforce code style”

- “JavaScript modules(import/export)”

Teremos que configurar o “SUCRESE” para utilizar “import/export” dentro do node.

Como iremos construir uma API\_REST não vamos utilizar nenhum framework, por esse motivo vamos selecionar.

- NOne of these

Náo iremos utilizar typescript nesse projeto por isso selecione:

- Does your Project use typescritp – NO

Nessa seção iremos utilizar o node, por isso selecione:

- Node.

- Use a popular style guide.

- Airbnb (<http://github.com/airbnb/javascript>)

- Javascript

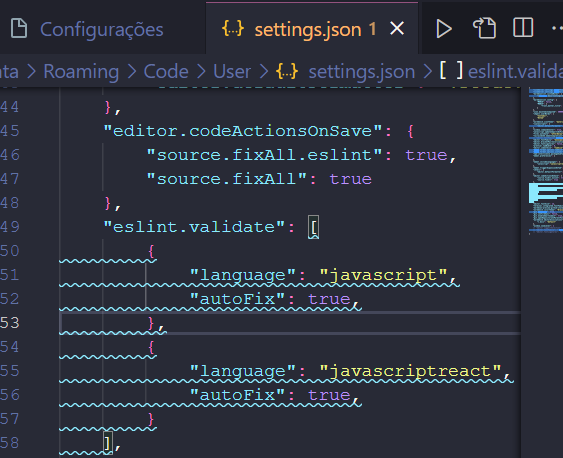
- Selecione yes para instalar o airbnb.

- selecione npm

Será instalado as dependências necessárias.

Vemos que também foi criado o arquivo .eslint que possui alguns atributos inclusive o “rule” onde iremos posteriormente criar regras.

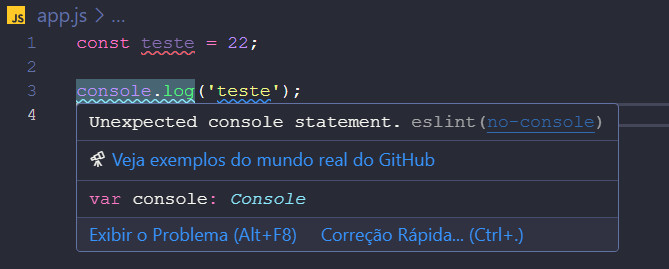
Abra as configurações do vscode e Adicione:



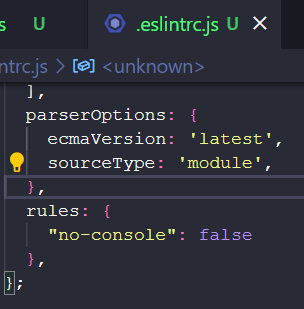
Vamos agora criar o arquivo “app.js” e verificar se as alterações no eslint estão funcionando.

Vemos que agora as regras do eslint estão funcionando. Mas tem algumas regras que poderemos alterar para isso basta copiar qual a regra que se quer mudar e inserir no atributo rules do arquivo “.eslintrc.js”

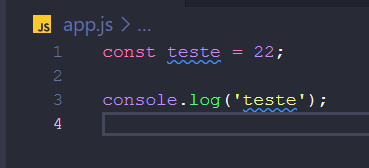
Por exemplo, se tentarmos utilizar o “console.log” o “eslint” irá reclamar.



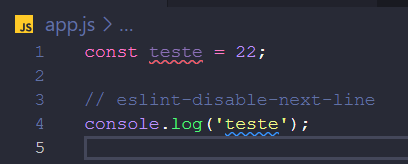
Para resolver isso vamos copiar a regra que está entre parênteses que nesse caso é “no-console” e vamos inserir no atributo rules do arquivo “eslintrc.js”.



Voltando no arquivo “app.js” vemos que o console.log agora está sendo aceito.



Se não quisermos alterar a configuração no eslint, podemos ainda neutralizar a ação do eslint utilizando um comentário sobre a linha ou no final da linha.



171 – Nodemon e Sucrase

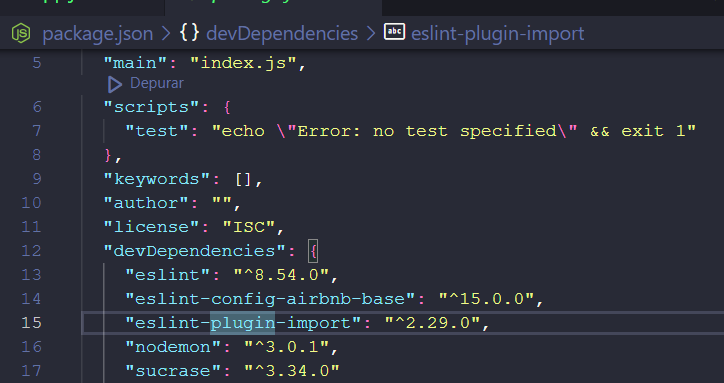
Nessa aula iremos configurar o nodemon e sucrase.

O sucrase irá permitir que utilizemos import e exports dos módulos do ES6 do Javascript.

Para isso vamos realizar algumas instalações para desenvolvimento, digite no terminal:

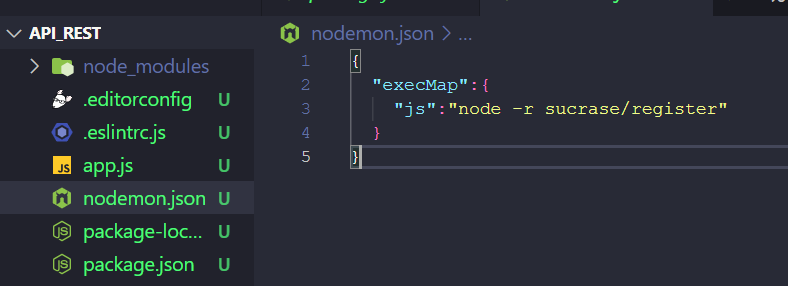
- npm i nodemon sucrase -–save-dev.

Abrindo o arquivo packege.json vemos as dependências instaladas:



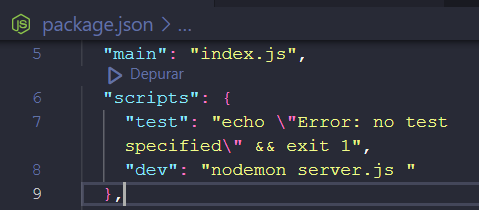
Temos que fazer o ‘nodemon’ registrar o ‘sucrase’ quando agente utilizar ele para executar alguma cois.

Para isso vamos criar um arquivo na raiz do projeto chamado ‘nodemon.json’ e nesse arquivo iremos criar o parâmetro de registro do sucrase.



Antes de testar devemos criar um novo script no nosso arquivo ‘package.json’ para rodar a aplicação.

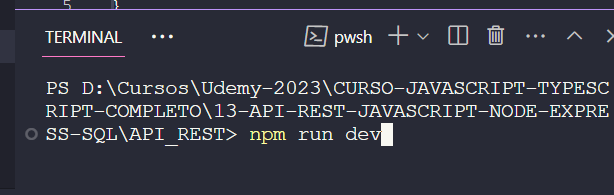
Abra o arquivo ‘package.json’ e dentro do atributo ‘scripts’ adicione a nova regra:



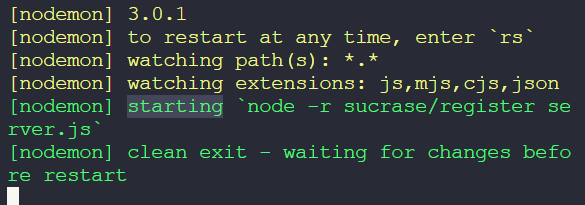
Para testar a regra criada no arquivo ‘nodemon.josn’vamos criar um novo arquivo chamado ‘server.js’.

Após criado o arquivo vamos rodar a aplicação digite:

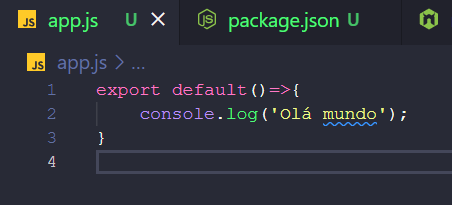
- npm run dev



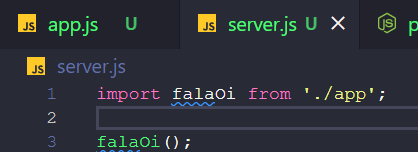
Vemos que o sucrase está sendo executado dentro do server.js



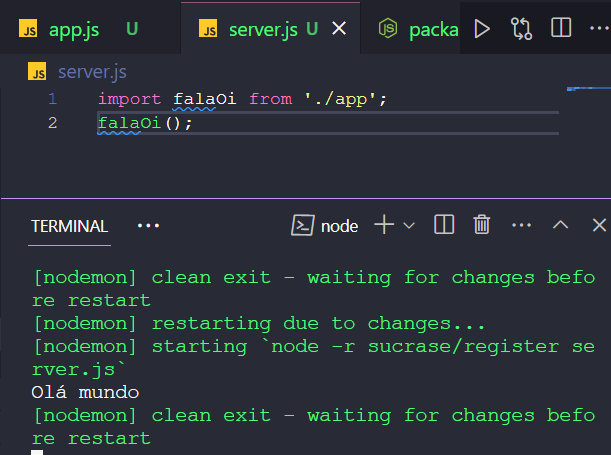
Vamos agora realizar um teste com o ‘import’ e ‘export’ para isso vamos abrir o arquivo ‘app.js’ e exportar uma função qualquer.



Par ver o resultado vamos abrir o arquivo ‘server.js’ e importar a função criada no ‘app.js’.



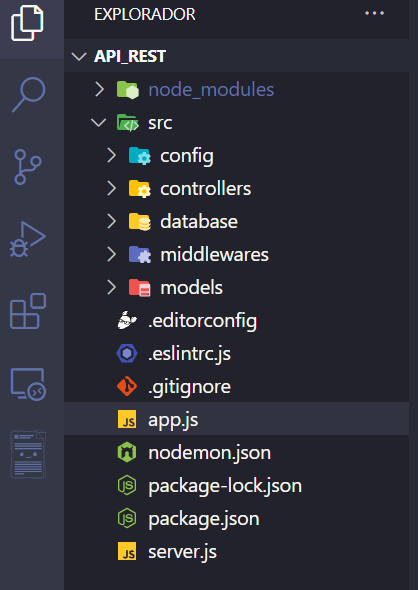
Vemos que está funcionando normalmente pelo terminal.



173 – Estruturando o projeto

Nessa aula iremos configurar a estrutura do nosso projeto, vamos criar a pasta ‘src’ onde ficará todo nosso condigo fonte, a não ser os arquivos anteriores.

Como nosso projeto será uma API não teremos a pasta ‘views’ do modelo MVC. Teremos também a pasta ‘config’ onde iremos salvar arquivos de configuração como o ‘.env’, a pasta ‘database’ que terá nossa base de dados, ‘middlewares’ que terão nossos middlewares que serão utilizados na nossa aplicação.



Vamos agora configurar o ‘express’, para isso vamos realizar a instalação, digite:

- npm i express

Após realizar a instalação vamos abrir o arquivo ‘app.js’ e iremos importar o ‘express’ para ser usado pelo ‘app’.

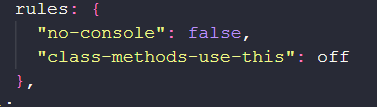
Vamos criar uma classe chamada App que terá um construtor com um atributo chamado ‘app’ que irá receber ‘express’ importado.

Teremos alguns métodos nessa classe, como o ‘middlewares’ e ‘routes’.

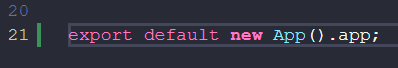
Vamos adicionar esses métodos no construtor para que toda vez que a classe ‘App’ for instanciada esses métodos serão executados.



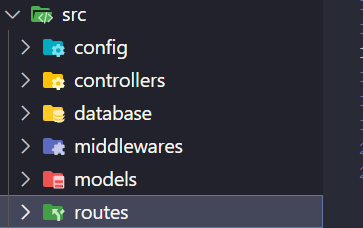
Vamos realizar uma alteração nas regras do ‘.eslinrc’.



No final da classe ‘App’ iremos realizar a exportação somente do atributo ‘app’ da classe. Esse atributo na verdade é somente o express.



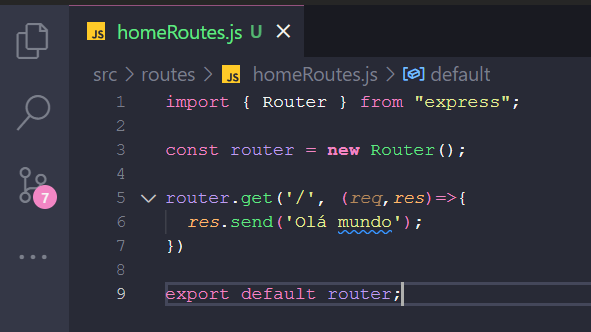
Vamos criar outra pasta dentro de ‘src’ que será ‘routes’ que terão as rotas da aplicação.



Dentro da pasta ‘routes’ vamos criar o arquivo de rotas ‘homeRoutes.js’.

Na pasta ‘controllers’ vamos criar o arquivo ‘homeController.js’.

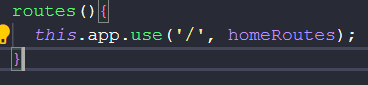
No arquivo ‘homeRoutes.js’ vamos importar o ‘Router’ do express, criar uma nova instancia para o ‘Router’ e na sequencia criar uma rota de teste, exportando no final o router.



Agora vamos no arquivo ‘app.js’ e vamos importar o arquivo ‘homeRoutes.js’ que criamos para que seja executada a rota de teste.

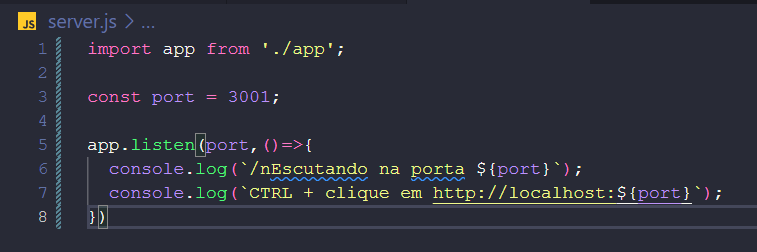


Na sequencia teremos que ir no método ‘routes’ e pedir para o ‘app’ usar o ‘homeRoutes’que importamos na rota ‘/’.



Temos que configurar o express para que ele fique escutando em alguma porta as requisições que serão realizadas, para isso vamos abrir o arquivo ‘server.js’ e importar o ‘app’.

Vamos também criar uma variável que terá o número da porta, essa variável será configurada posteriormente no arquivo ‘.env’.



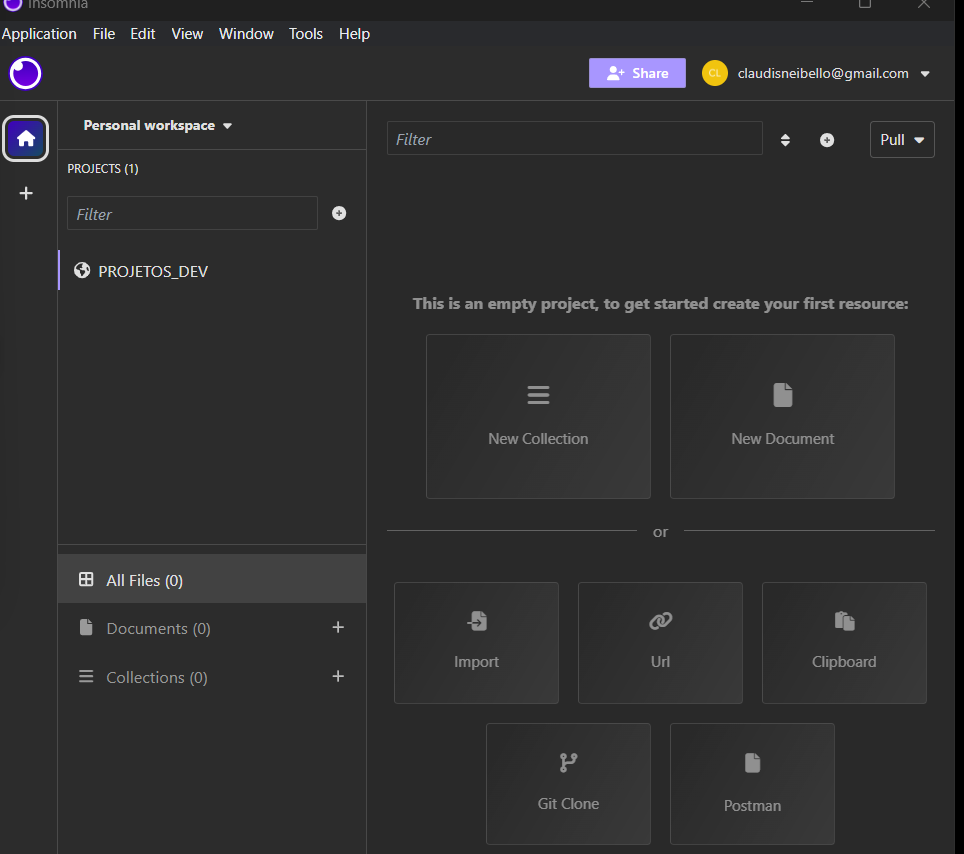
Vamos testar nossa aplicação, digite :

- npm run dev

Vemos que a palavra ‘olá mundo’ foi escrita na página do browser.

Porém não queremos utilizar o browser nessa aplicação e sim o ‘insomnia’.

Por isso vamos abrir o ‘insomnia’ e criar uma collection chamada ESCOLA



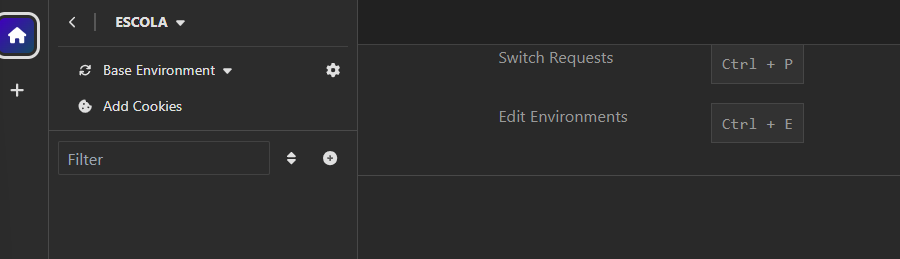


Dentro da collection ‘ESCOLA’ será onde iremos armazenar todas as informações de alunos.

Temos que criar a nossa ‘url’ base que será ‘http://localhost:3001.

Para isso teremos que criar dentro de ‘Base Enviroment’ uma chave chamada “base\_url” que terá como valor <http://localhost:3001>.

Para isso clique no icone de configuração.

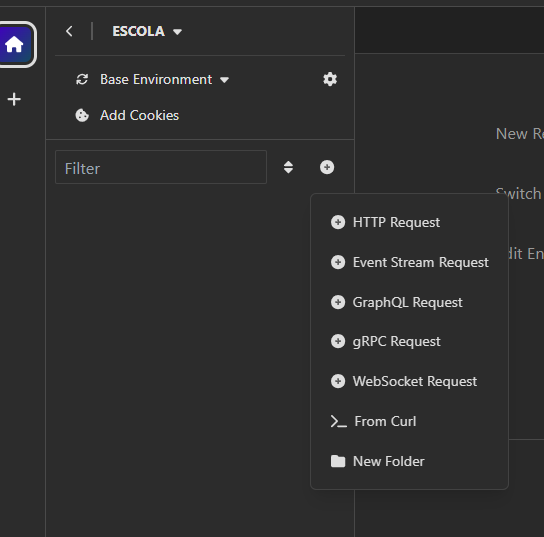


Dentro de ‘Base Enviroment’ adicione o atributo.



Isso facilita para não termos que digitar a url toda hora e quando formos trocar a url basta trocar uma vez.

Agora vamos criar uma pasta dentro da collection ‘ESCOLA’, chamada ‘Home’ que terá as rotas de ‘home’.



Na pasta ‘Home’ vamos criar uma nova requisição do tipo ‘get’, clique em ‘http request’.

Daremos o nome de index para essa rota.

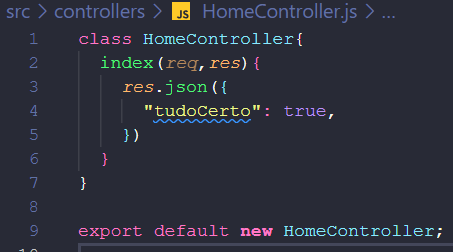
No campo da URL vamos digitar base\_url e clicar em ‘Send’ veremos o resultado da requisição.



Mas para funcionar temos que rodar a aplicação no vscode.

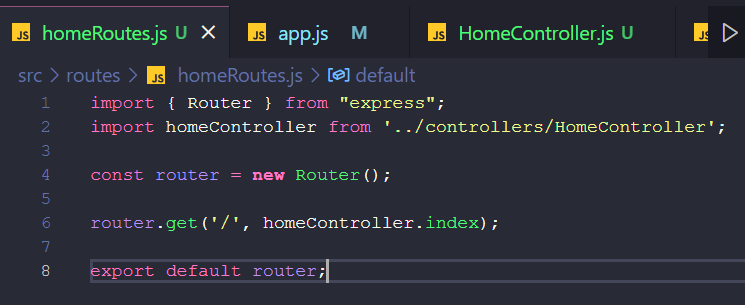
Agora vamos configurar nosso arquivo ‘HomeController’ que será uma classe e terá os métodos que serão nossas requisições.

Vamos então criar primeiramente o método ‘index’ que irá retornar um ‘json’ com uma chave “tudoCerto” com o valor “true”.

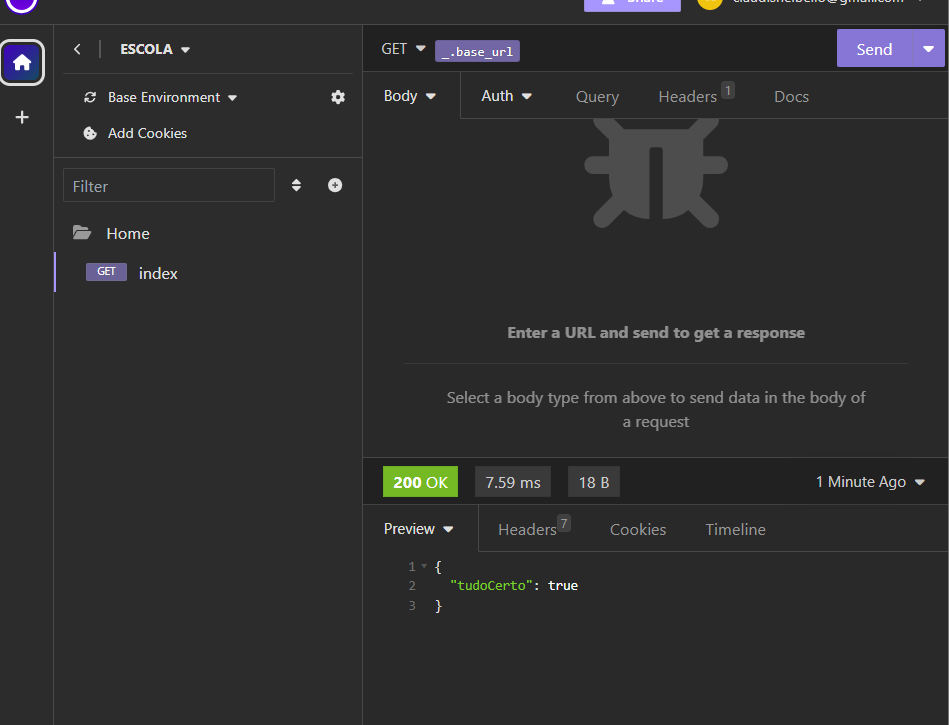


Vamos exportar a classe já instanciada.

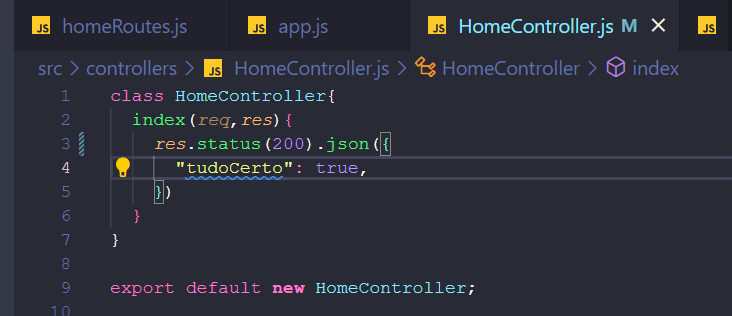
Agora vamos voltar no nosso ‘homeRoutes’ vamos importar o HomeController e substituir a função da rota ‘route.get’ pela função ‘index’ do ‘homeController’.



Realizando o teste da rota ‘index’ no ‘insomnia’ vamos ver como resultado o ‘json’ que foi configurado no ‘HomeController’.



Vemos no teste que foi retornado o código ‘200 OK’, na API\_REST vamos trabalhar com diversos códigos e iremos passar esses códigos em nossos controllers.



Vamos ainda criar uma pasta chamada ‘uploads’ que irá armazenar os arquivos que iremos entregar em nossas requisições.

**174 – Configurando e testando o Sequelize.**

Vamos configurar o sequelize que fará o controle de nossa base de dados, para criar, editar, excluir campos em nossas tabelas da base de dados.

Primeiramente vamos configurar o ‘.env’ para colocarmos senhas e outras informações confidenciais somente nele.

Então vamos intalar o ‘dotEnv’, digite:

- npm i dotenv

Vamos criar o arquivo ‘.env’ na raiz e realizar algumas configurações.

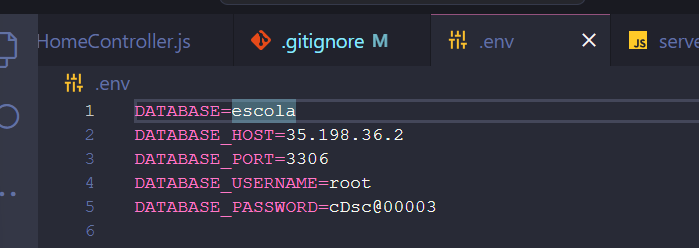
‘DATABASE’ será ‘escola’.

‘DATABASE\_HOST’ devemos iniciar o servidor no google cloud e copiar o IP externo.

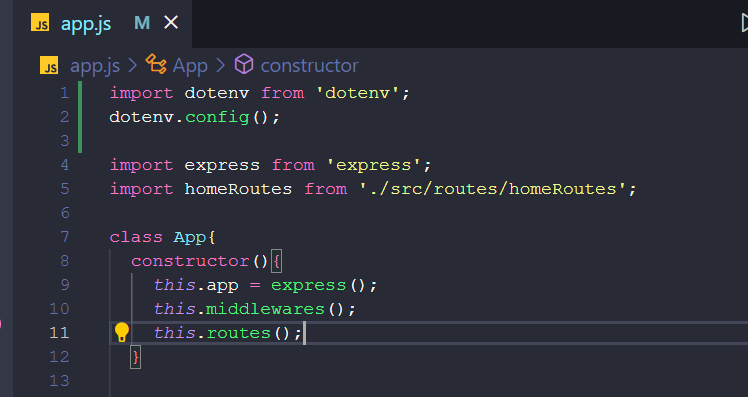
‘DATABASE\_PORT’ será 3306 já que vamos trabalhar com o container que criamos do MySql.

‘DATABASE\_USERNAME’ será root.

‘DATABASE\_PASSWORD’ será a senha que definimos na criação do container do MySql.



Agora vamos configurar o arquivo ‘app’ importando o ‘dotenv’ nele.

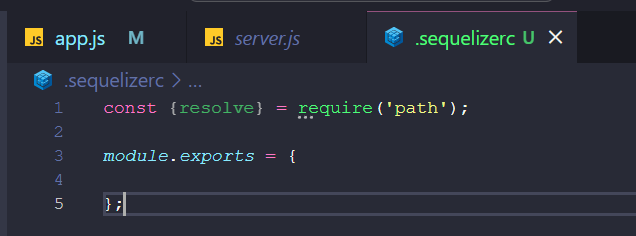


Agora iremos configurar o sequelize.

Vamos criar um arquivo chamado ‘.sequelizerc’ na raiz do projeto, abrindo o arquivo vamos digitar ‘CTRL + SHIFT + P’ e selecionar a opção ‘change language mode’ onde iremos selecionar ‘javascript’. Desta forma o arquivo irá executar o eslint do javascritp com correções e autocompletes e síntese hilight.

Aqui não consigueremos utilizar o import/export e sim o require.

A primeira configuração que faremos será fazer um destruct de ‘path’ pegando somente o ‘resolve’ para resolvermos os caminhos no sequelize.



Iremos criar um parâmetro chamado ‘config’que será o nosso parâmetro de configuração e precisaremos passar para ele o caminho onde está o arquivo de configuração, por isso iremos utilizar o ‘\_\_dirname’ que é nossa raiz do projeto seguido das pastas ‘src’ e ‘config’ e o nome do arquivo que será ‘database.js’ que será criado posteriormente.

 config: resolve(\_\_dirname,'src','config','database.js'),

teremos ainda outro parâmetro que será o caminho onde estará nossos models.

'models-path': resolve(\_\_dirname,'src','models'),

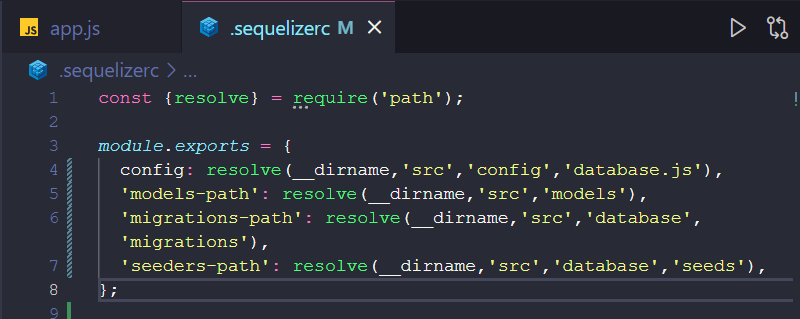
Outro parâmetro será o caminho das ‘migrations’ que são os arquivos criados no banco de dados pelo sequelize e o histórico de alterações na base de dados.

O nome do parâmetro será ‘migrations-path’ onde iremos utilizar o resolve passando a raiz do projeto ‘\_\_dirname’, ‘src’, ‘database’ e ‘migrations’ que será a pasta criada automaticamente.

'migrations-path': resolve(\_\_dirname,'src','database','migrations'),

Iremos criar outro parâmetro chamado ‘seeders-path’ e irá utilizar o mesmo caminho do parâmetro ‘migrations-path’ e a pasta ‘seeds’ que será criada automaticamente também.

'seeders-path': resolve(\_\_dirname,'src','database','seeds')



Agora podemos criar o arquivo ‘database.js’ dentro da pasta ‘config’ que terá as configurações da nossa base de dados.

Nesse arquivo iremos utilizar o require para importar o ‘dotenv’.

*module*.*exports* = {

  dialect: 'mariadb',

  host: process.env.DATABASE\_HOST,

  port:process.env.DATABASE\_PORT,

  username:process.env.DATABASE\_USERNAME,

  password: process.env.DATABASE\_PASSWORD,

  database:process.env.DATABASE,

  define:{

    timestamps: true,

    underscored: true,

    underscoredAll: true,

    createdAt:'created\_at',

    updatedAt:'updated\_at'

  },

  dialectOptions:{

    timezone: 'America/Sao\_Paulo'

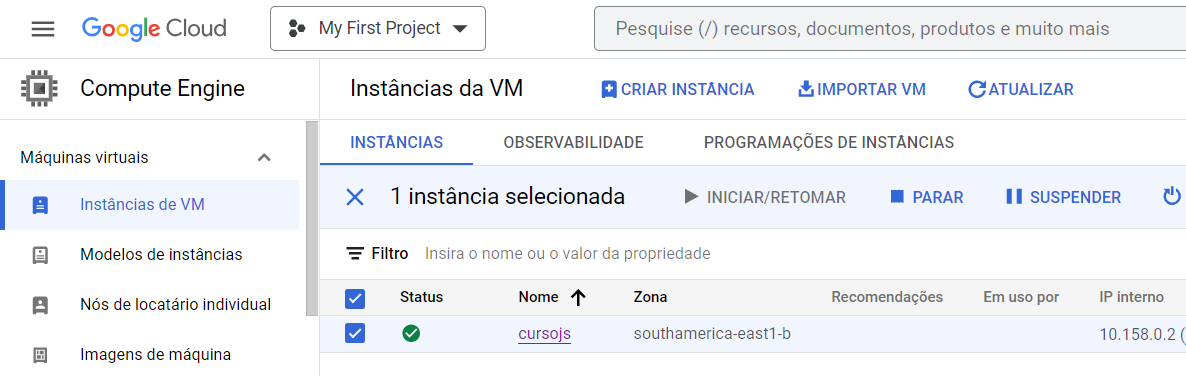
  },

  timezone: 'America/Sao\_Paulo'

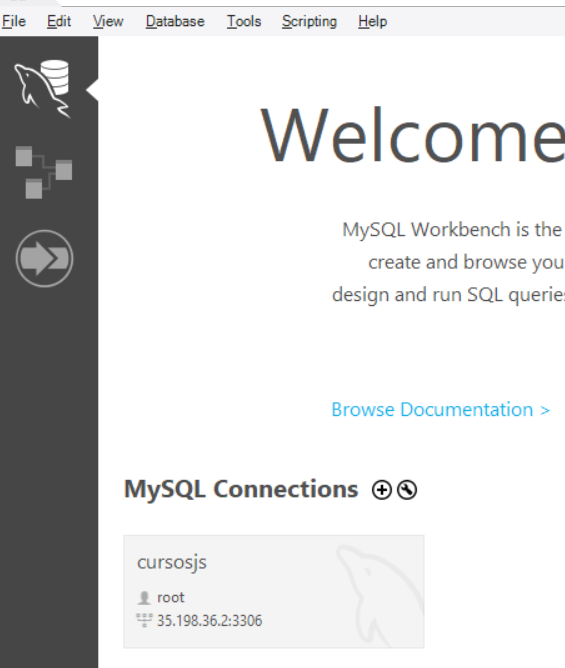
}

Essa configuração pode ser utilizada em outros projetos, sendo utilizado como modelo.

Vamos ligar nosso servidor no google cloud



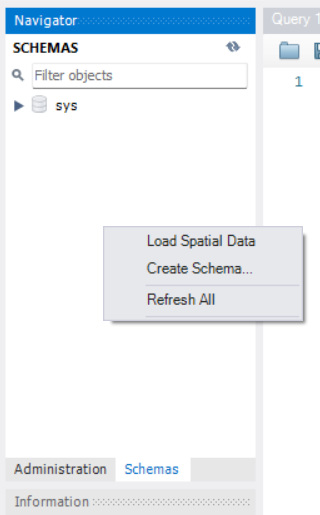
Depois vamos abrir o MySQL workbench e clicar na conexão que criamos anteriormente.



Clique na aba ‘Schemas’



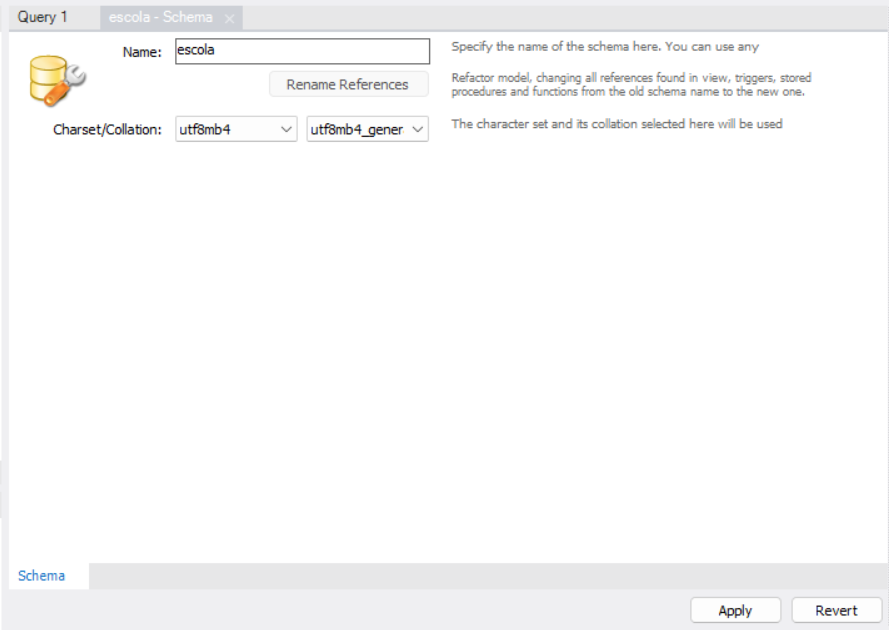
Agora dê um clique com o botão esquerdo e clique em create Schema



De o nome ‘escola’ que é o mesmo que utilizamos na configuração da nossa base de dados.

E no campo Charset/Collation: vamos selecionar ‘utf8mb4’ e no campo collation: vamos selecionar ‘utf8mb4\_general\_ci.

Depois clique em Apply.



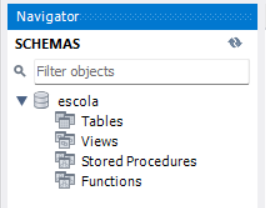
Clique em Apply novamente.



Clique em Finish.



Agora já temos criado nosso esquema ‘escola’.



O resto o sequelize irá fazer.

Vamos agora instalar o sequelize e o mariadb

**- npm i sequelize mariadb**

Temos que instalar também o sequelize-cli como Dev dependency.

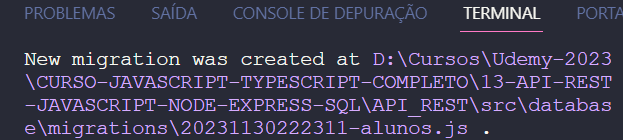
**-npm i -D sequelize-cli**

Nesse ponto iremos criar uma ‘migration’ para o sequelize criar uma tabela chamada ‘alunos’ dentro do banco de dados.

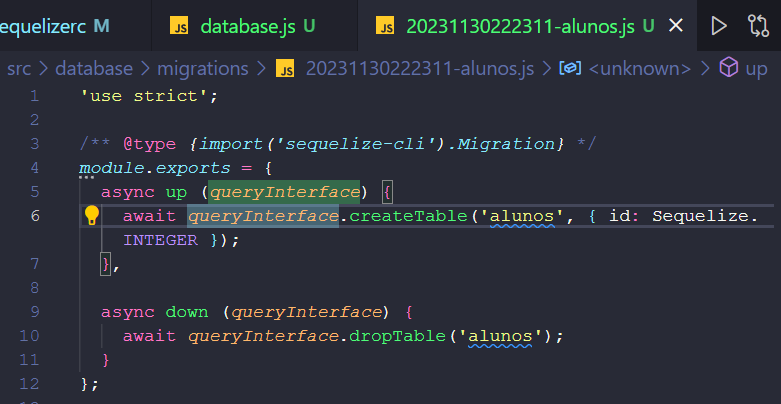
Vamos digitar o comando:

**- npx sequelize migration:create –name=alunos**

Foi retornado a mensagem



Vamos abrir o arquivo ‘migration’ que foi criado, retirando dele os comentários e deixando conforme abaixo:



Devemos lembrar sempre que para a tabela que estamos criando ‘alunos’ deveremos ter um model que será chamado ‘aluno’.

Vamos agora configurar o campo ‘id’ da tabela e criar outros campos.

  async up (*queryInterface, Sequelize*) {

    await *queryInterface*.createTable('alunos', {

      id: {

        type: Sequelize.INTEGER,

        allowNull: false,

        autoIncrement: true,

        primaryKey: true

      },

      nome:{

        type: Sequelize.STRING,

        allowNull: false,

      },

      sobrenome:{

        type: Sequelize.STRING,

        allowNull: false,

      },

      email:{

        type: Sequelize.STRING,

        allowNull: false,

      },

      idade:{

        type: Sequelize.INTEGER,

        allowNull: false,

      },

      peso:{

        type: Sequelize.FLOAT,

        allowNull: false,

      },

      altura:{

        type: Sequelize.FLOAT,

        allowNull: false,

      },

      created\_at:{

        type: Sequelize.DATE,

        allowNull: false,

      },

      updated\_at:{

        type: Sequelize.DATE,

        allowNull: false,

      }

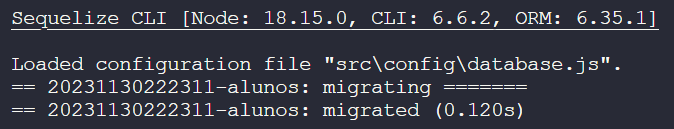
    });

  },

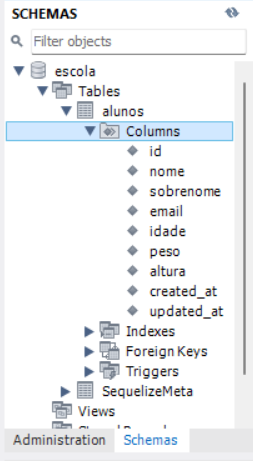
Agora vamos rodar a migration, para isso digite:

**-npx sequelize db:migrate**

Será retornado a mensagem



Se abrirmos nossa base de dados veremos a tabela ‘alunos’ que foi criada.



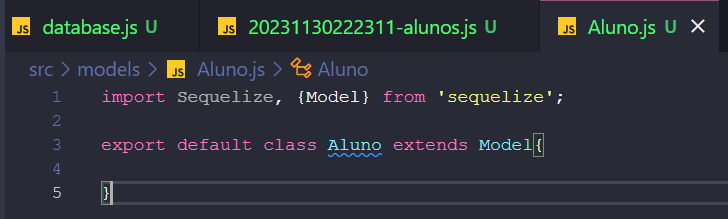
Vemos também a tabela ‘SequelizeMeta’ que é criada automaticamente para manter um histórico.

Agora vamos criar um model e vamos fazer esse model conectar na base de dados e efetuar um registro para testarmos.

**Criando o model Aluno.**

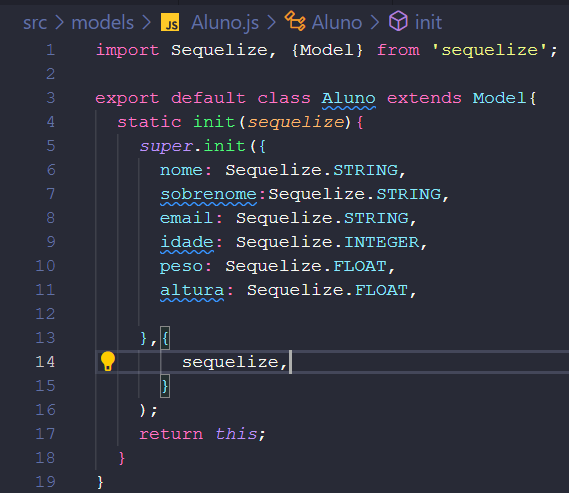
Na pasta ‘models’ vamos criar o arquivo **Aluno.js**.

Nesse arquivo vamos importar o model e o ‘Sequelize’ faremos uma classe de configuração conforme a documentação do ‘sequelize’ essa classe extende ‘Model’.



Na classe vamos criar um método estático ‘init()’ que irá receber como argumento ‘sequelize’ que será a conexão que esse model irá receber.

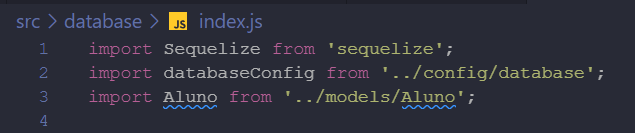
Nesse método iremos chamar o método ‘init’ da classe pai, através de ‘super.init()’ esse método da classe pai irá receber dois objetos um com os nossos campos, os mesmos criados na configuração que realizamos no migration aluno, com exceção que não teremos os campos id, created\_at e updated\_at pois ele serão criados automaticamente e outro campo será o próprio sequelize.



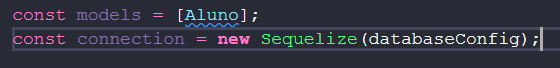
no final iremos retornar o próprio método.

Feito isso agora vamos criar o arquivo de conexão que será utilizado nesse model. Para isso vamos criar na pasta ‘database’ o arquivo ‘index.js’.

Nesse ‘index.js’ iremos importar o ‘Sequelize’ o ‘databaseConfig’ e os models.



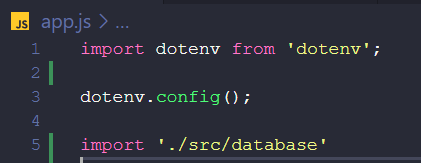
Vamos agora criar um array que terá todos os model que foram ou serão importados nesse arquivo. Também temos que criar uma constante ‘connection’ que irá receber as configurações da nossa base de dados.



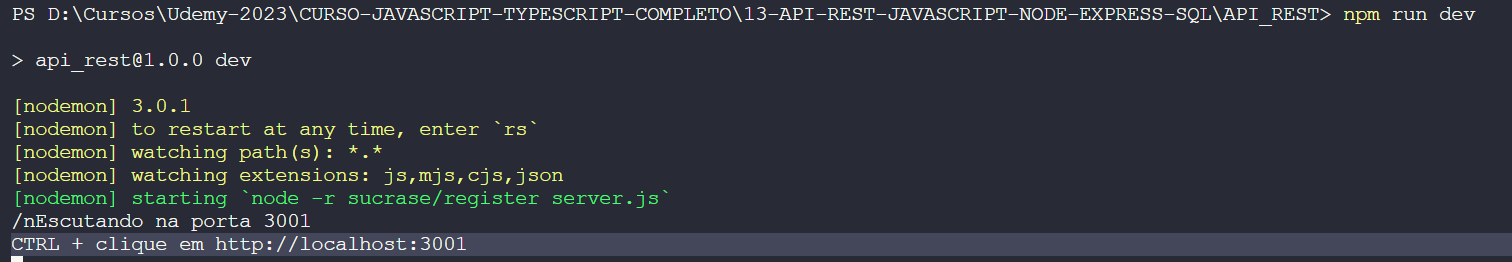
Agora iremos percorrer todos os models e enviar para cada um deles a conexão com o banco de dados.



Temos ainda que chamar esse arquivo ‘index.js’ no momento que inicializamos o nosso servidor. Para isso vamos abrir o arquivo ‘app.js’ e importar para ele o ‘database/index.js’, porém não há necessidade de informar ‘index.js’ pois é subentendido.



Agora vamos rodar a aplicação para verificar se está ocorrendo algum erro nessa configuração.



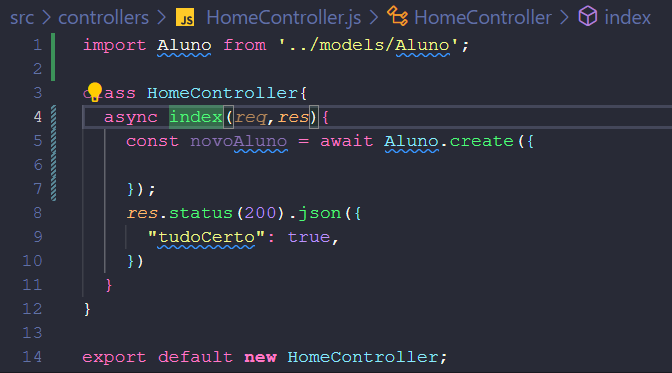
A aplicação rodou sem erros.

**Testando a criação de dados na base de dados**

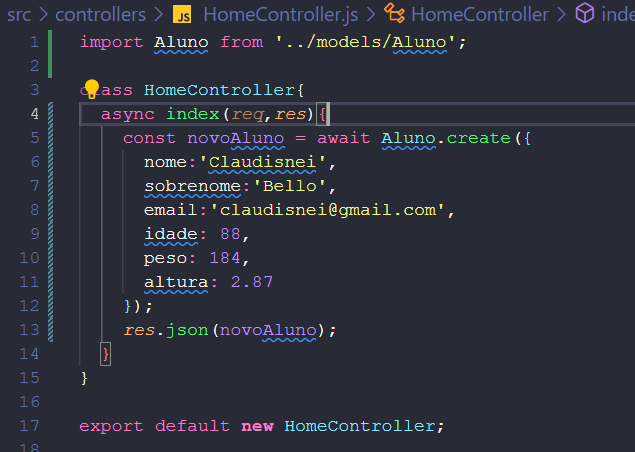
Vamos abrir nosso arquivo ‘HomeController.js’ somente para testar, pois deveríamos utilizar o ‘model de Aluno’ no ‘controller de Aluno’.

Vamos começar importando o model ‘Aluno’.

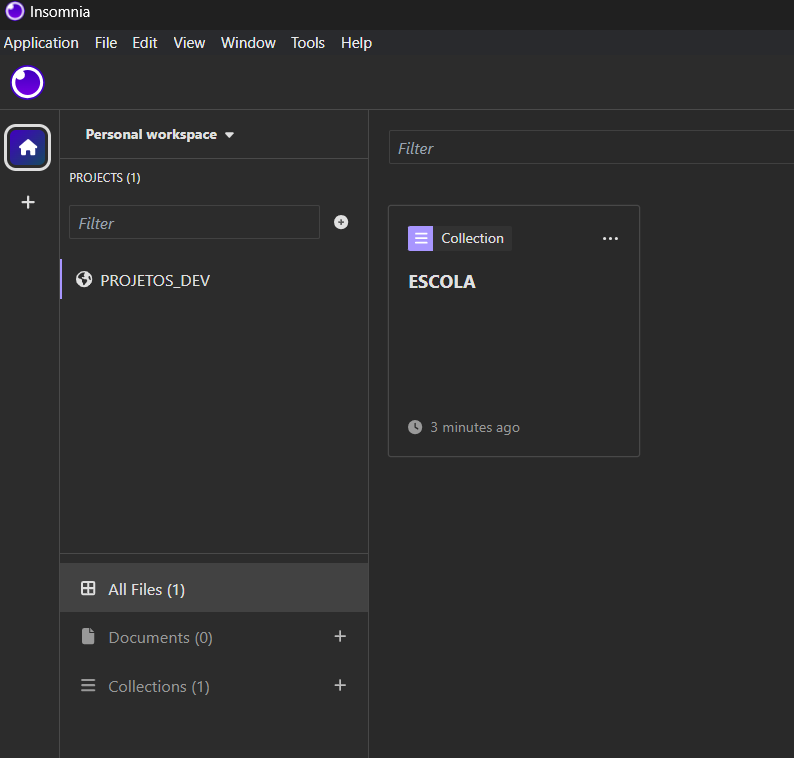
Depois dentro do classe iremos passar o método ‘index’ para ‘async’e dentro desse método iremos criar uma constante chamada ‘novoAluno’ que irá receber o aluno criado através do método ‘create’no model ‘Aluno’, lembrando que deveremos utilizar o await.



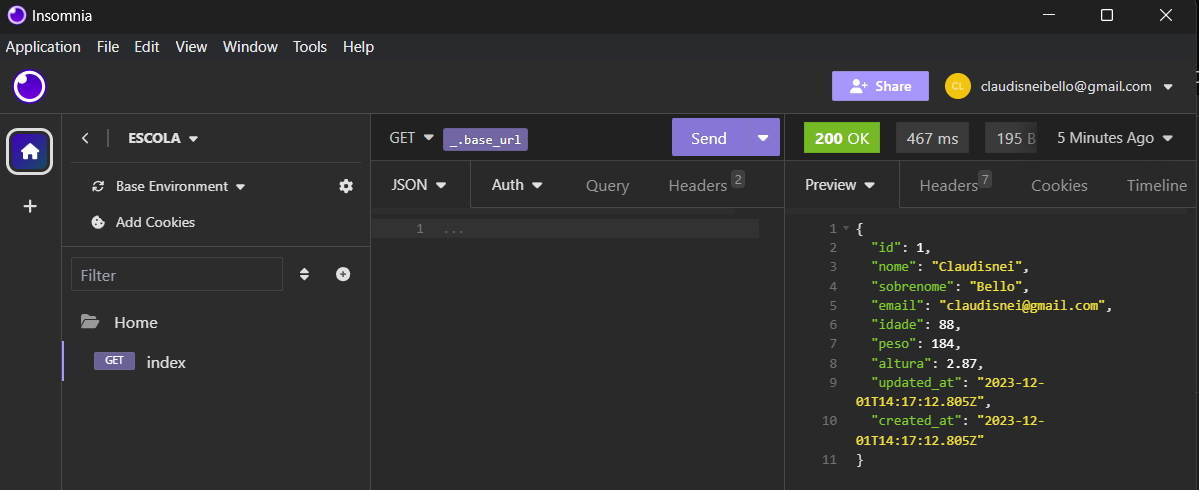
Para criarmos o aluno temos que passar seus dados dentro do objeto em ‘create’. Os dados devem respeitar os mesmos campos que foram definidos na ‘migration’ e também no model ‘Aluno’. Vamos passar a constante ‘novoAluno’ para a resposta, esse é um padrão de uma API\_REST onde quando criamos um recurso, já obtemos ele devolta.



Agora para verificarmos se está funcionando vamos abrir no ‘Insomnia’

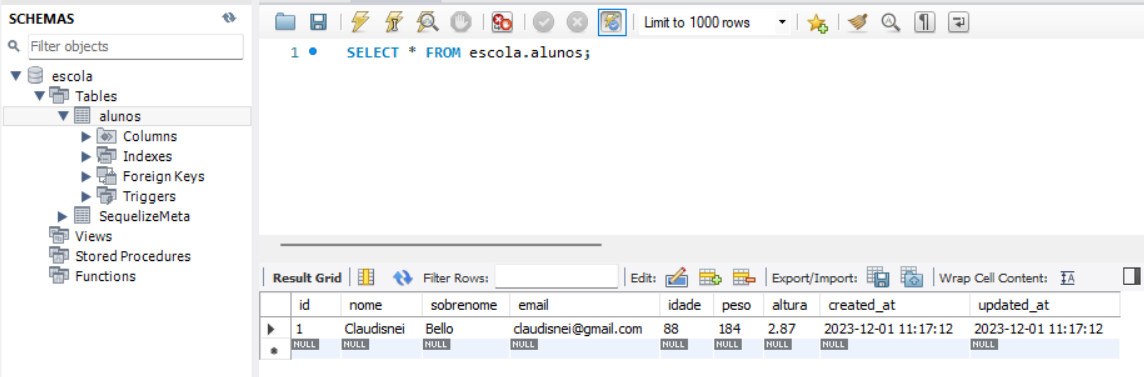


Clique na collection ESCOLA.

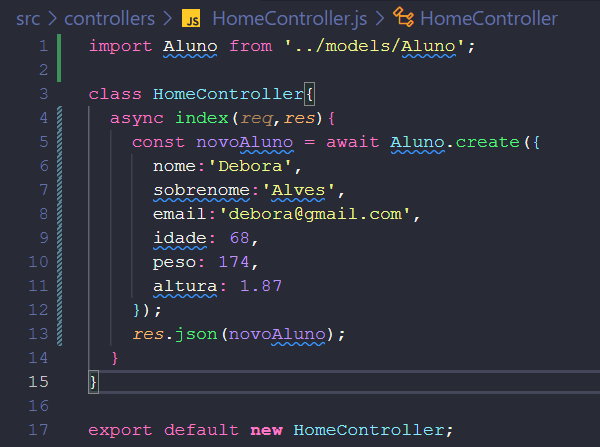


Clique em ‘Sende’ e já vemos o retorno no formato JSON .

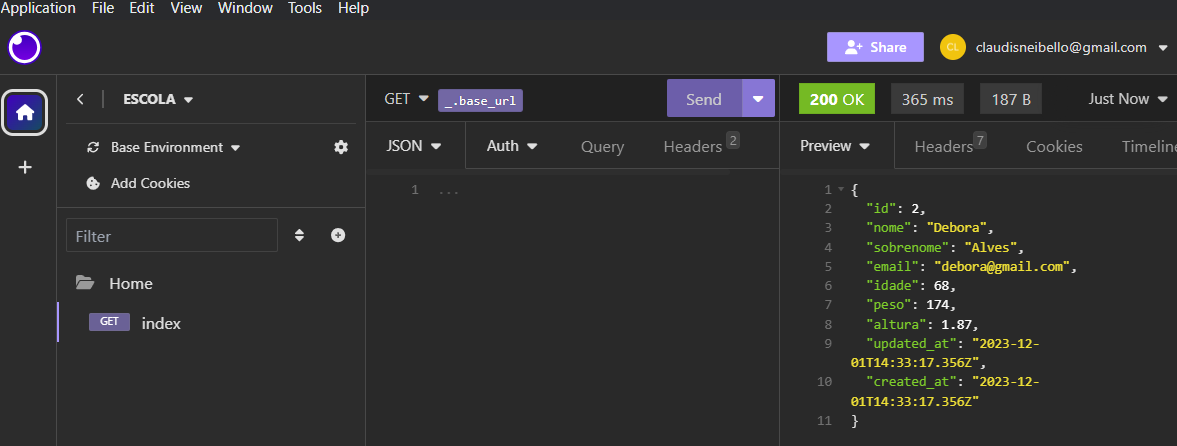
Se abrirmos o ‘MySQL workbench’ veremos o primeiro aluno criado.



Se alteramos os dados no controller ‘HomeController’ iremos criar um novo aluno.



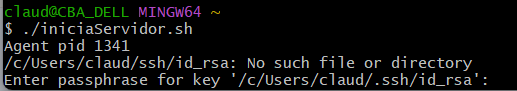
Agora voltamos no ‘Insomnia’ e clicamos em ‘Send’ e temos o novo aluno criado na base de dados.



**175 – Criando Usuários**

Primeiramente inicie o servidor no google cloud e também no git bash.





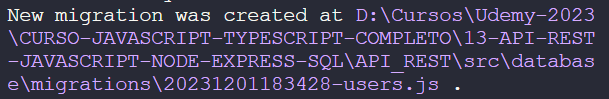
Digite a Senha: debora31

Agora Iremos criar uma feature para usuários de maneira que somente pessoas cadastradas possam acessar o sistema.

Primeiramente teremos que criar uma tabela em nossa base de dados e para fazer isso teremos que fazer uma migration.

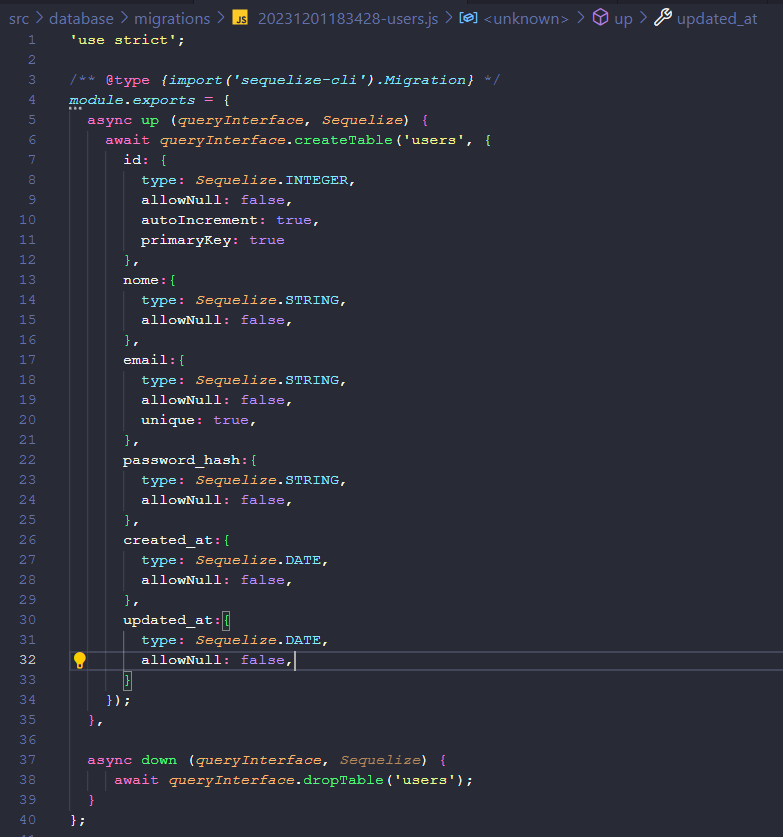
Digite no terminal :

**- npx sequelize migration:create –-name=users**

****

Após criação da migration, vamos copiar os campos da migration-alunos e manter somente os campos ID, NOME, EMAIL, CREATED\_AT, UPDATED\_AT. Sendo que no campo email iremos utilizar também o atributo ‘unique’ pois iremos utilizar o email para logar o usuário e por isso deve ser único no banco de dados.

Teremos que ter outro campo que será o ‘password\_hash’ onde iremos substituir a senha do usuário por uma string.

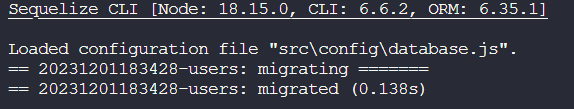
****

Após conferir se todos os campos estão corretos, vamos realizar a migração ou seja iremos criar essa tabela no banco de dados.

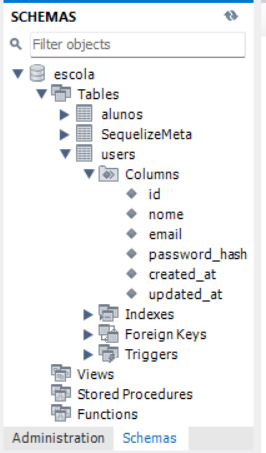
Digite :

**- npx sequelize db:migrate**

Irá aparecer a mensagem:



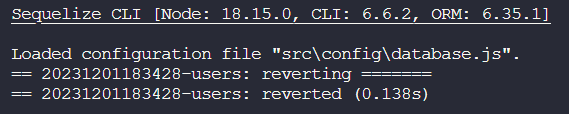
No MySQL Workbench podemos ver as tabelas criadas.



Se quisermos desfazer as alterações da migration, devemos utilizar o comando:

**- npx sequelize db:migrate:undo**

Será exibido a mensagem de confirmação.

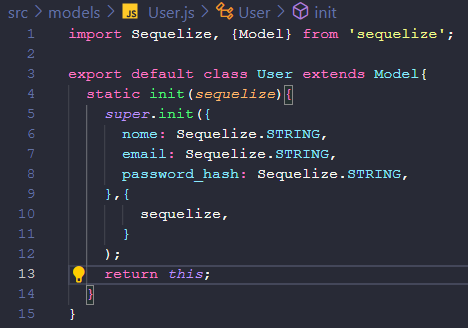


E no banco de dados veremos que a tabela não existe mais.

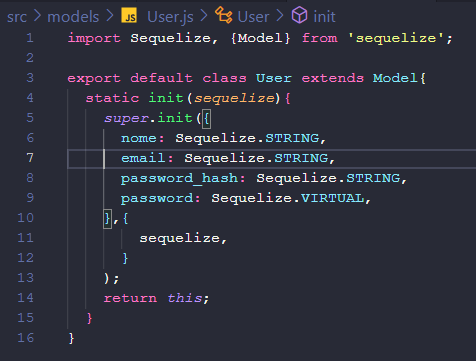


**Criando o model Users.**

Vamos copiar o model ‘Aluno’ e renomear para ‘User’, ajustando os campos.



Nesse model iremos fazer a validação e iremos também ter outro campo chamado ‘password’ que será a senha digitada pelo usuário sem ser transformada em hash, esse campo será do tipo Sequelize.VIRTUAL e desta forma ele não irá existir na base de dados.



Para fazer a validação iremos utilizar os parâmetros como objetos onde iremos passar atributos de validação. Os paramentros do ‘sequelize’ são similares aos utilizados no ‘validator’.

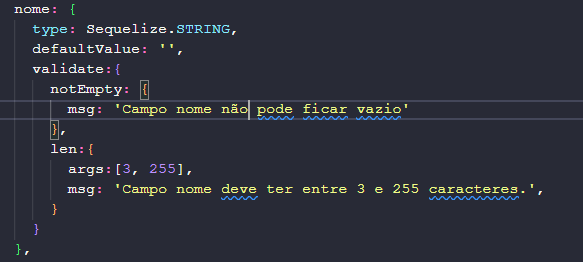
Nesse objeto iremos utilizar um atributo chamado ‘defaultValue’ onde iremos passar um valor padrão que será atribuído caso não seja fornecido nenhum valor.

Teremos outro atributo também que será ‘validate’ onde teremos objetos com as regras de validação, argumentos e mensagem.

O tipo ‘notEmpty’ irá exibir uma mensagem caso o campo esteja vazio.

Já o tipo ‘len’ terá um atributo chamado args onde iremos passar o tamanho mínimo e máximo e também um atributo ‘msg’ onde passaremos a mensagem de erro de validação.

Tem a validação ‘isEmail’ que irá verificar se o valor passado é do tipo email.



Vamos copiar essa configuração para os outros campos.

import Sequelize, {Model} from 'sequelize';

export default class User extends Model{

  static init(*sequelize*){

*super*.init({

      nome: {

        type: Sequelize.STRING,

        defaultValue: '',

        validate:{

          notEmpty: {

            msg: 'Campo nome não pode ficar vazio'

          },

          len:{

            args:[3, 255],

            msg: 'Campo nome deve ter entre 3 e 255 caracteres.',

          }

        }

      },

      email: {

        type: Sequelize.STRING,

        defaultValue: '',

        validate:{

          isEmail: {

            msg: 'Email inválido!'

          },

          len:{

            args:[3, 255],

            msg: 'Campo nome deve ter entre 3 e 255 caracteres.',

          }

        }

      },

      password\_hash: {

        type: Sequelize.STRING,

        defaultValue: '',

      },

      password: {

        type: Sequelize.VIRTUAL,

        defaultValue: '',

        validate:{

          len:{

            args:[6, 50],

            msg: 'Campo senha deve ter entre 6 e 50 caracteres.',

          }

        }

      },,

    },{

        sequelize,

      }

    );

    return *this*;

  }

}

Para criar o hash vamos utilizar o bcrypt e fazer um Hook para que antes de salvar os dados no banco de dados, o valor calculado do hash seja passado para o campo.

Primeiro vamos instalar o bcrypt.

**- npm i bcryptjs**

E depois importa-lo no model User.

import bcryptjs from 'bcryptjs';

Agora temos que criar o ‘hook’ antes do ‘return this’.

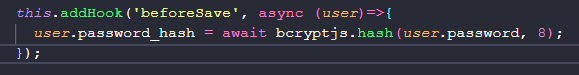


Podemos ver que existe vários tipos de ‘hooks’ vamos utilizar o ‘beforeSave’.

*this*.addHook('beforeSave');

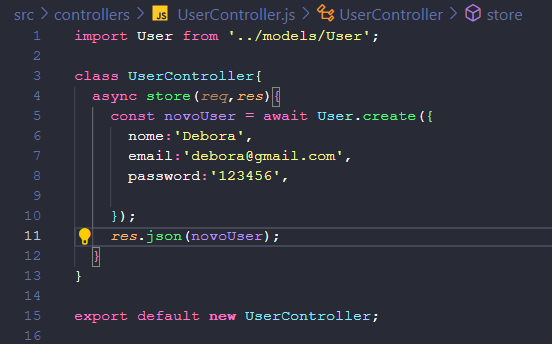
esse Hook terá ainda como segundo parâmetro uma função que terá o próprio usuário que está sendo criado.

e desta forma podermos alterar o valor do campo ‘password\_hash’ utilizand o ‘bcryptjs’ que importamos. Deveremos utilizar ‘async/await’ pois deveremos esperar o resultado do ‘bcryptjs’.



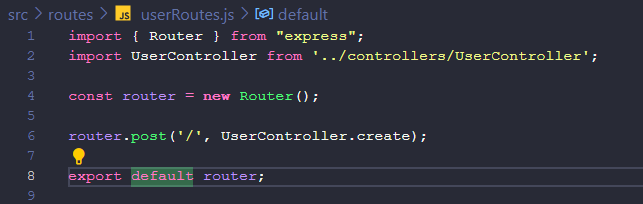
**Criando o controller de User.**

Vamos copiar o arquivo de HomeController.js, renomeando e alterando os campos.



Agora vamos criar uma rota de usuário, para isso vamos na pasta ‘routes’, copie o arquivo ‘homeRoutes.js’ e renomear para ‘userRoutes’, vamos utilizar o método ‘create’ n

o UserController.



Regras de um controller.

/\*\*

 \* podemos ter em cada controller até cinco métodos

 \* - Linstando todos os usuário : index (no insomnia utiliza o método GET)

 \* - Cria um novo usuário: store ou create (no insomnia utiliza o método POST)

 \* - Apaga um usuário: delete (no insomnia utiliza o método DELETE)

 \* - Mostra um usuário: show (no insomnia utiliza o método GET)

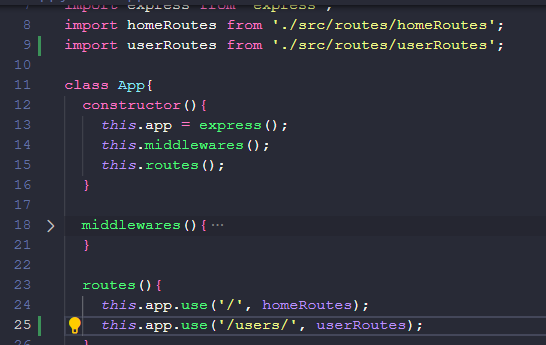
 \* - Atualiza um usuário: update. (no insomnia utiliza o método PATCH OU PUT)

 \*

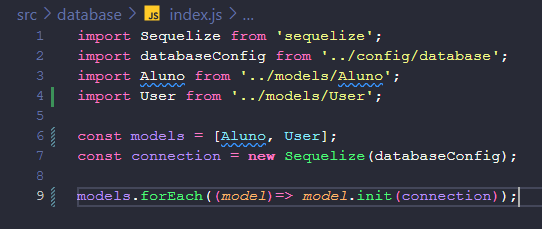
 \* Se o controller tiver mais métodos isso tá errado.

 \*/

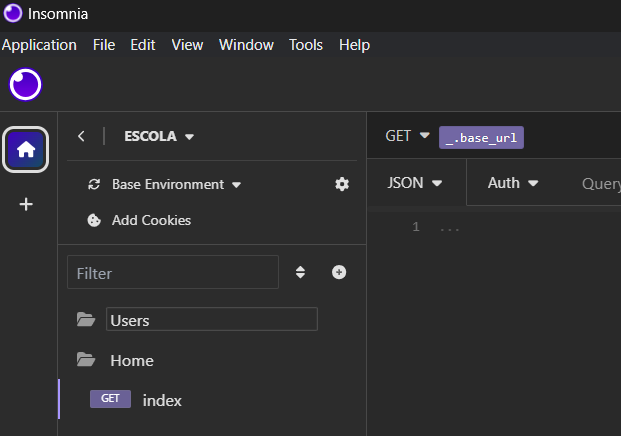
Agora temos que importar essa rota dentro nosso ‘app.js’ e criar a rota ‘/users/’.



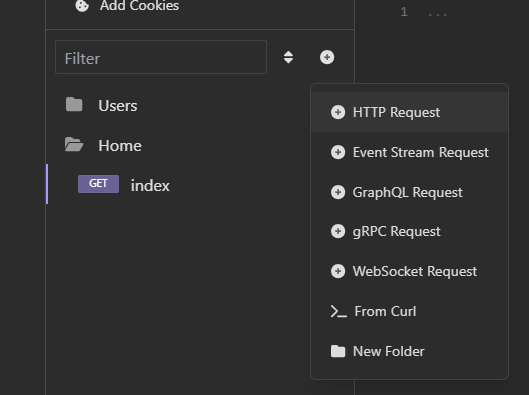
Temos também que ir no arquivo ‘index.js’ da pasta ‘database’ e temos que adicionar o model criado ao array e importa-lo para o arquivo.



Feito isso vamos abrir o Insomnia e criar a pasta ‘Users’ que terá as requisições do usuário.



Então dentro da pasta ‘Users’ vamos criar novo HTTP Request e dar o nome de ‘Store’

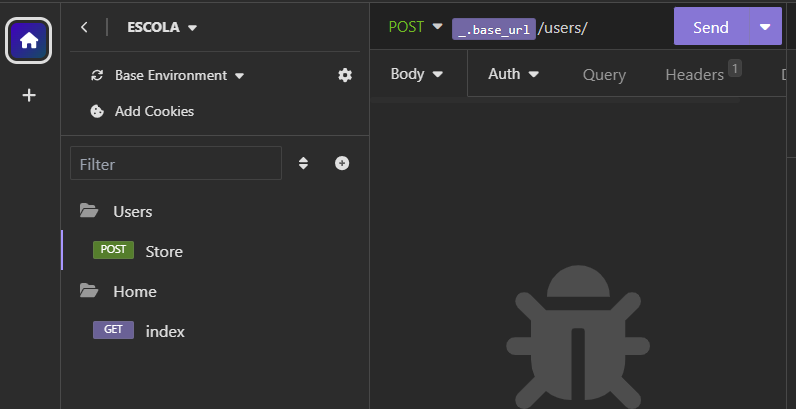


E selecionar o método POST.

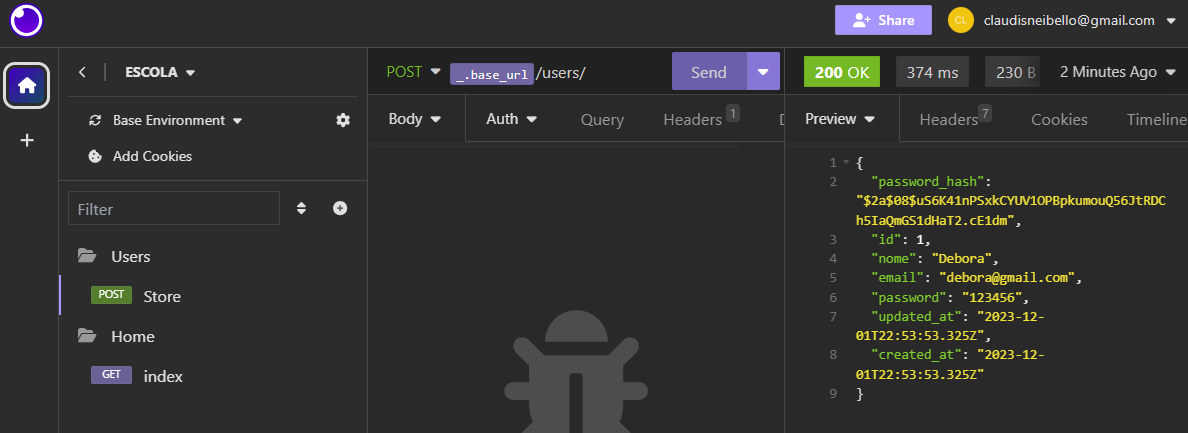


Vamos passar nossa ‘base\_url’ para o endereço, mais ‘/users/’.

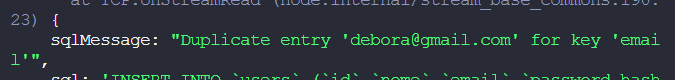
Agora temos que iniciar o servidor para podermos executar no insomnia a rota que criamos.

****

Agora quando clicar no botão ‘Send’ teremos o usuário criado.



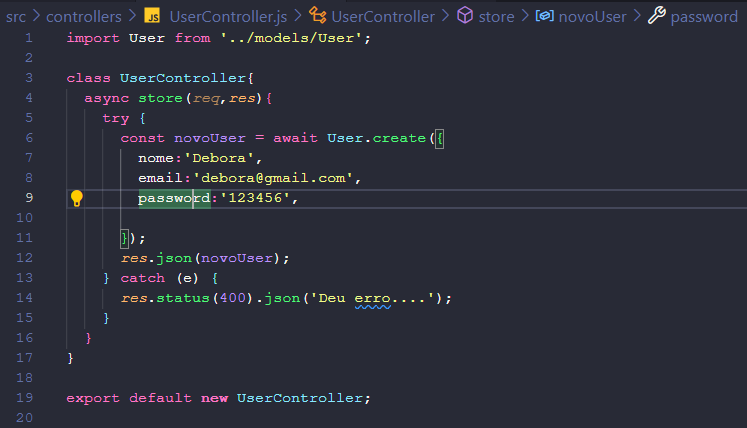
Se tentarmos adicionar novamente o mesmo usuário, teremos um erro.



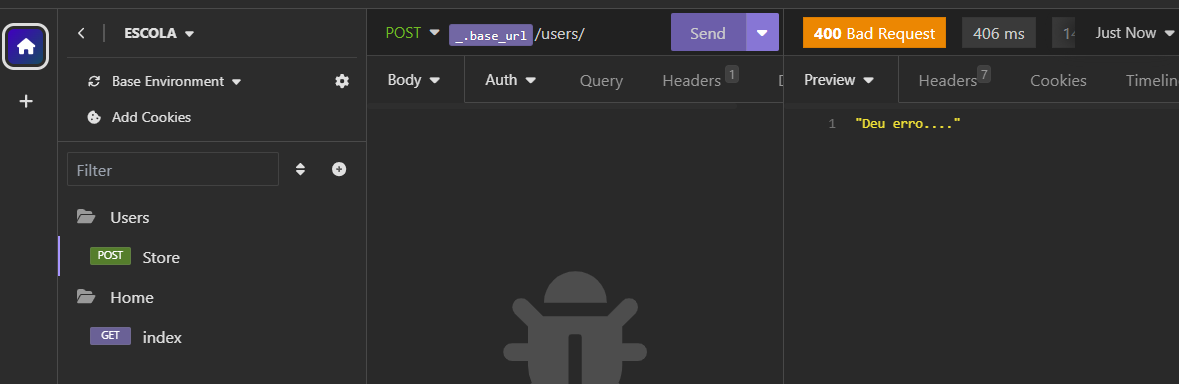
Teremos que tratar esse erro no UserController.

Vamos colocar a chamada do método User.create(), dentro de um try e se der um erro iremos dar uma resposta.

Para isso vamos alterar o status da mensagem para ‘bad request’ esse erro é 400 e depois passar uma mensagem.

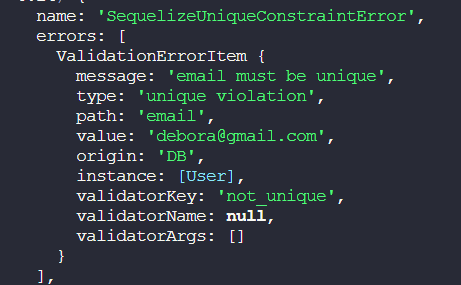


No insomnia teremos a mensagem.



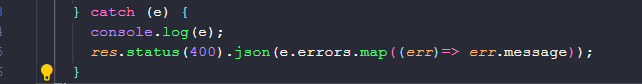
Para melhorar a mensagem podemos capturar a mensagem que está sendo gerada e exibila.

Vamos então utilizar um console.log(), mostrando os erros que estão sendo gerados.

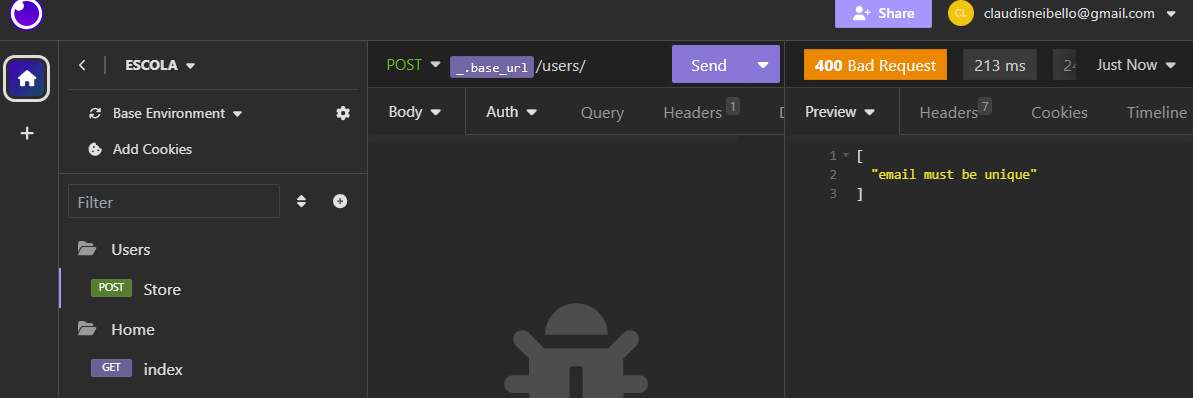


Vemos que está sendo gerado um array chamado errors que possui vários parâmetros.

Nesse array iremos fazer um ‘map’ para retornar somente as mensagens de erro que ocorreu durante a execução do Model.



Agora se clicarmos em ‘Send’ no insomnia teremos a mensagem que foi gerada.



A mensagem criada está em inglês pois não configuramos isso no model.

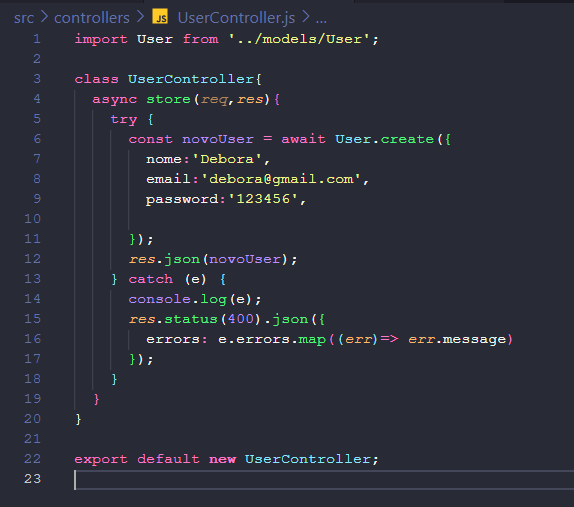
Vamos configurar o campo email do model ‘User’ e vamos adicionar o parâmetro ‘unique’ com o atributo ‘msg’ e a mensagem ‘Email já existe’.



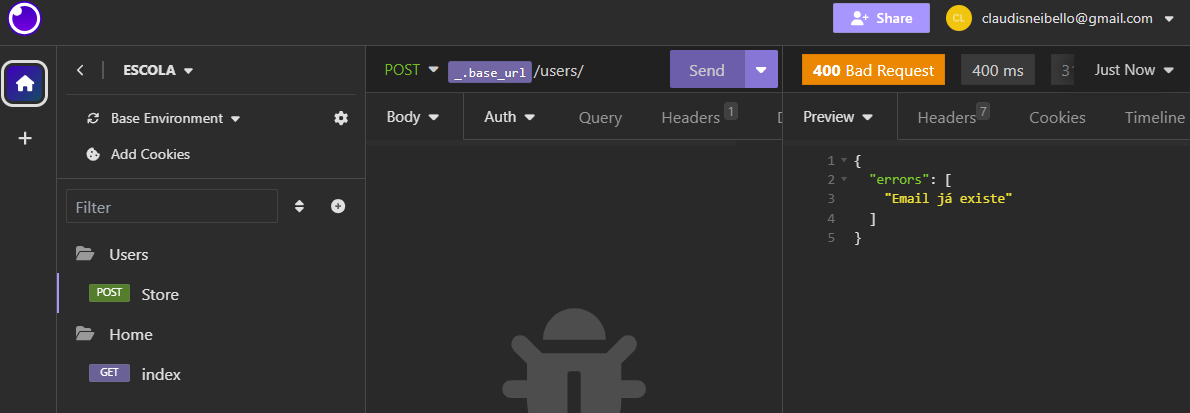
Agora teremos a mensagem que configuramos no model User ao executar no Insomnia.



Para melhorar a captura do erro, vamos criar um objeto na mensagem com uma chave chamada ‘errors’ onde será passado o erro original.

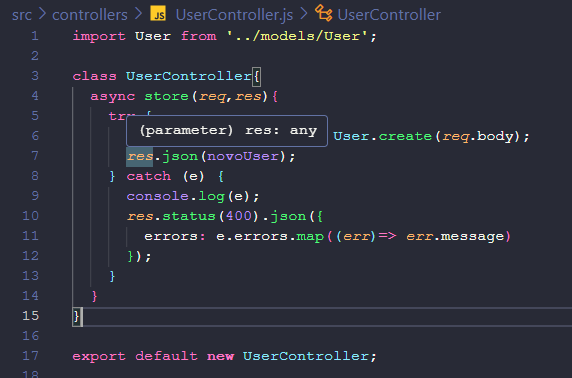


No insomnia teremos o erro.

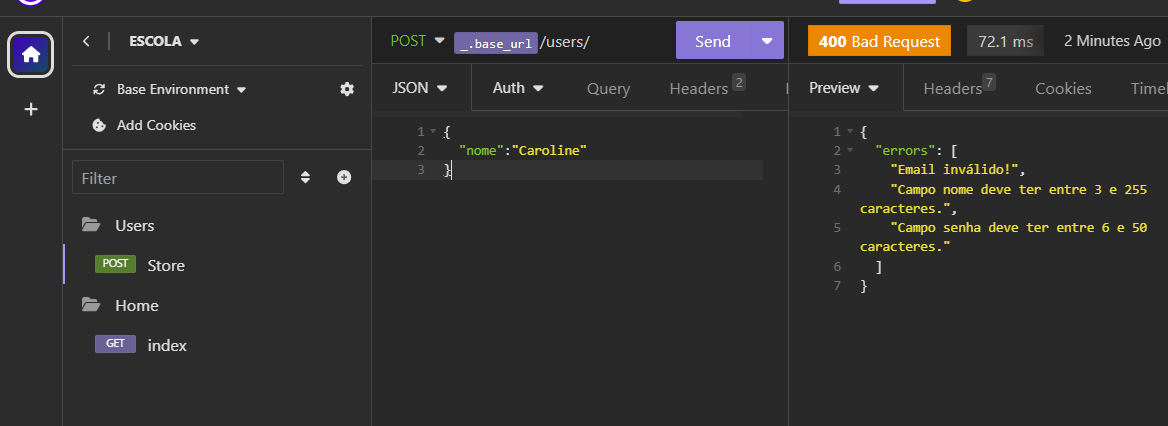


Agora vamos realizar uma alteração no UserController para podermos enviar os dados através do insomnia.

Vamos utilizar os dados da requisição que estão no body.

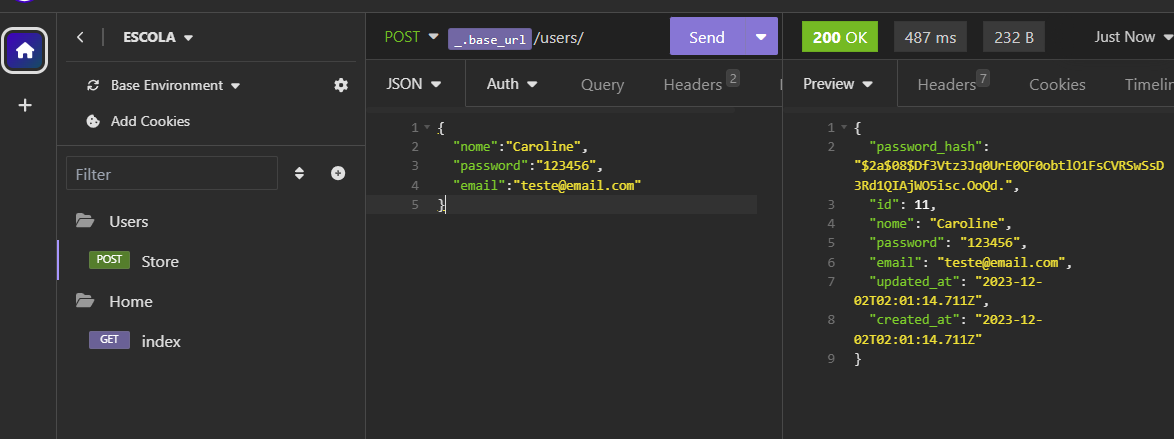


No insomnia vamos alterar de ‘Body’ para ‘JSON’ e vamos passar os dados no formato de objeto.



Vemos os erros causados pela falta de dados do objeto.

Se passarmos os dados corretos teremos o novo usuário salvo com sucesso.



**176 – CRUD de Usuários**

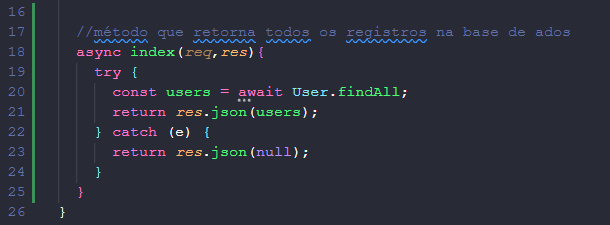
Vamos criar os métodos que faltam no UserController, são eles: **store, index, show, update, delete.**

O método store já foi implementado, vamos agora começar com o index esse método irá retornar todos os usuários que estão cadastrados no banco de dados.

**Método index.**

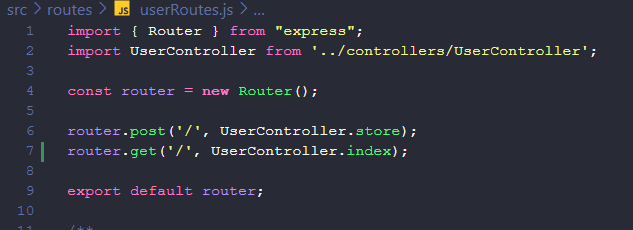
Vamos utilizar um try/catch onde iremos buscar todos os usuários do banco de dados para isso utilizamos o método **findAll** do Model **User,** iremos retornar a constante **users** que terá todos os usuários cadastrados.

Caso ocorra algum erro nessa busca iremos retornar um **json** com valor **null.**



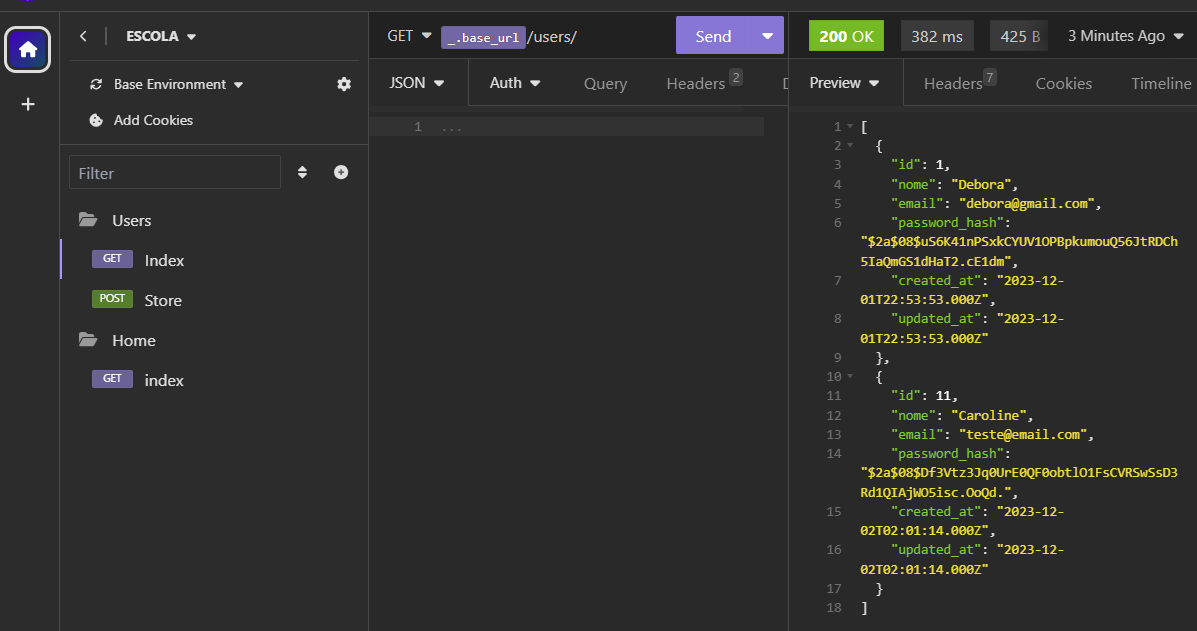
**Cadastro da rota para executar a busca de todos usuários.**

Vamos abrir o arquivo **userRoutes.js** e nele vamos inserir mais uma rota do tipo **get** que irá chamar o **userController.index.**



**Cadastrando a rota no Insomnia**

No Insomnia vamos abrir o banco de dados Escola e na pasta Users vamos criar mais um **HTTP Request** que será do tipo **GET** e daremos o nome de **Index**. No endereço vamos utilizar nossa Url base **\_.base\_url** mais a rota de usuários que é **/users/**. Ao clicarmos em **Send** veremos os dados exibidos no **Preview.**



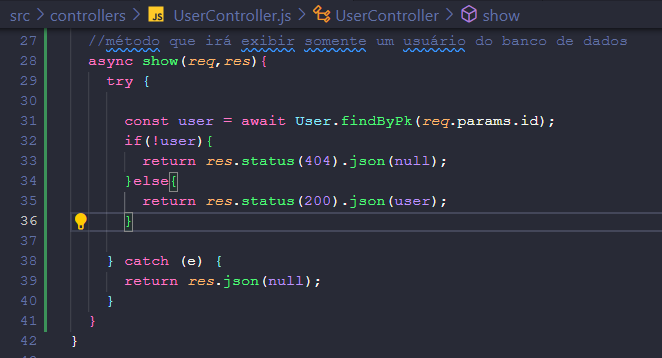
**Criando o método Show no UserController**

Vamos criar o método **Show** que irá exibir somente um usuário, esse método irá utilizar o método **findByPk** do model **User** porém iremos passar um parâmetro que será o **id** do usuário.

Então iremos passar para esse método **findByPk** o conteúdo que foi passado via parâmetro no corpo da requisição, fazemos isso utilizando o **req.paramns.id** .

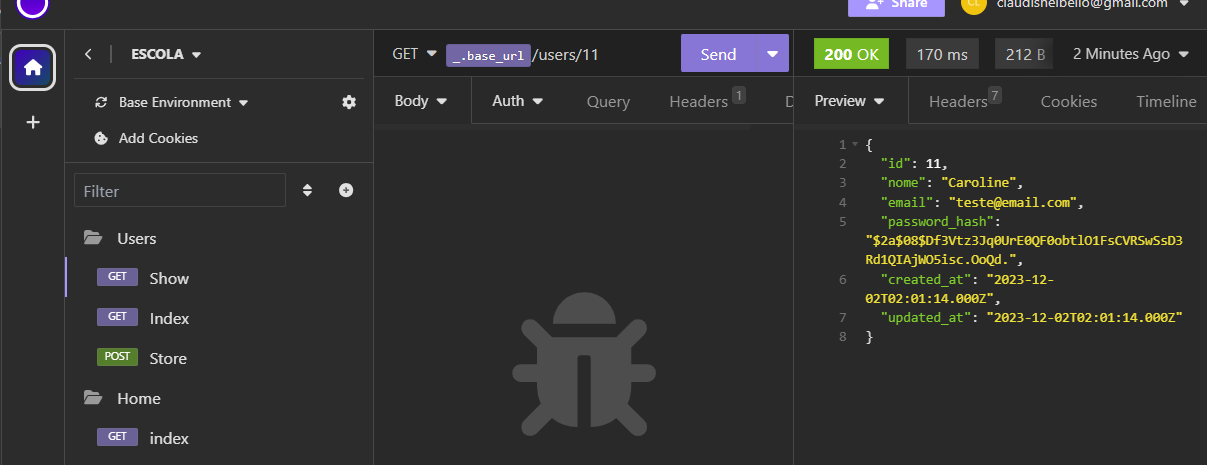
Iremos fazer algumas verificações para não ocorrer erro ao buscar o usuário,

Iremos verificar se o usuário foi localizado, se for localizado será retornado o usuário caso contrario será retornado **null**.

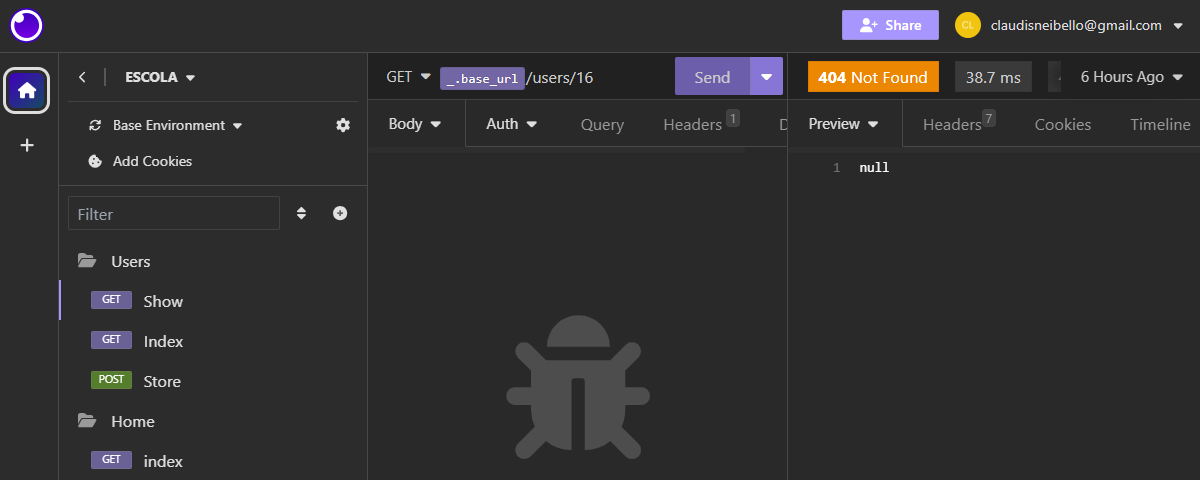


**Criando a rota Show no Insomnia.**

Na pasta **Users** vamos criar mais um **HTTP Request** do tipo **GET**, vamos passar a nossa **base\_url/users/** mais o id que queremos buscar.Ex: **base\_url/users/11** irá retornar o usuário de id 11.



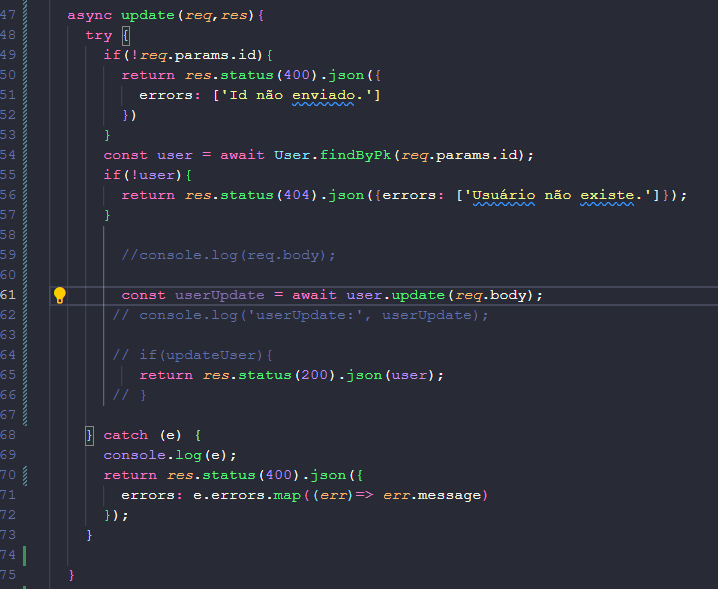
Se passarmos um id que não existe será retornado um erro.



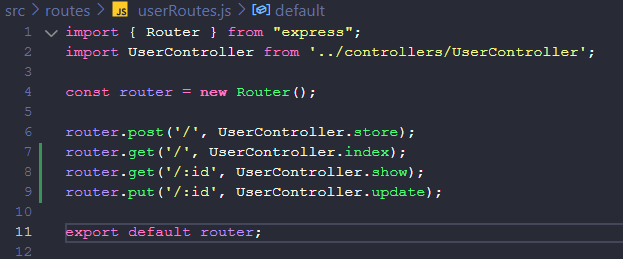
**Criando o método update no UserController**

Iremos verificar primeiramente se o id está sendo passado no corpo da requisição e caso contrário iremos retornar um erro.

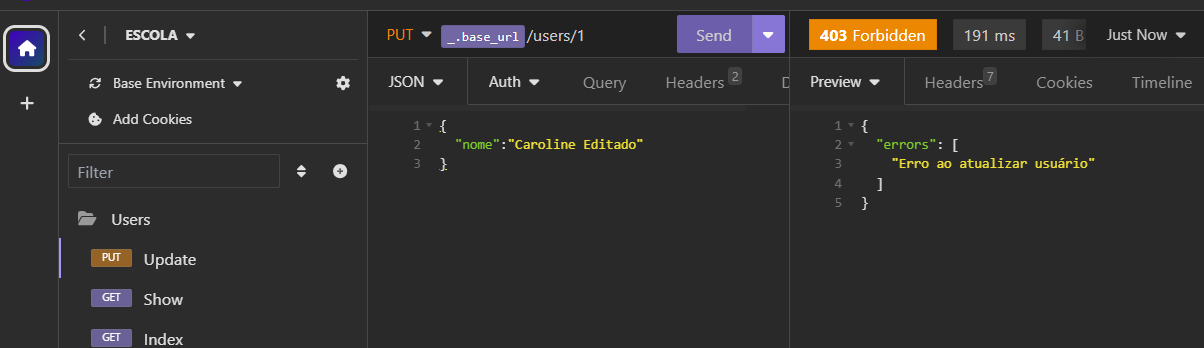
Se o **id** for passado na requisição, iremos realizar uma busca do usuário, se ele não existir iremos retornar novamente um erro. E caso contrário iremos realizar o update do usuário com os dados passados no corpo da requisição.



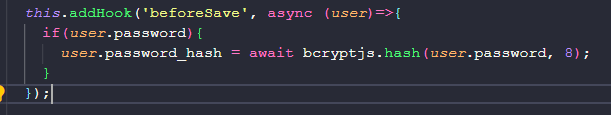
Temos ainda que criar a rota para update do usuário no arquivo **userRoutes** essa rota irá utilizar o método **put** do **router.**

****

Vamos acessar o Insomnia e na pasta Users vamos criar outro **HTTP Request** com o método **PUT** vamos utilizar \_.base/url/users/ onde iremos passar o id do usuário.



Porém está ocorrendo um erro pois não estamos fornecendo o password do usuário para resolver esse problema iremos alterar o model User onde iremos criar a hash somente se a password do usuário for fornecido.

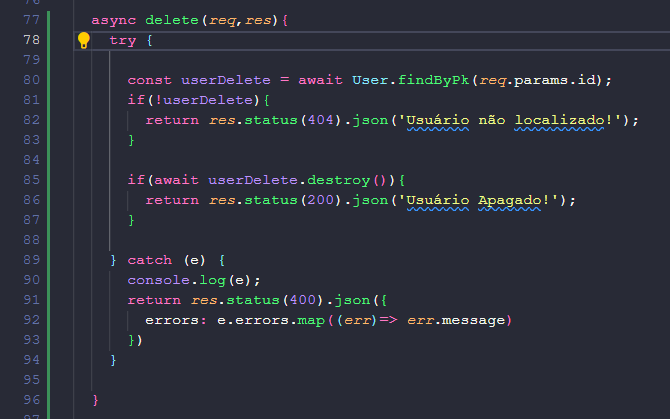


**Criando o método apagar no userController.**

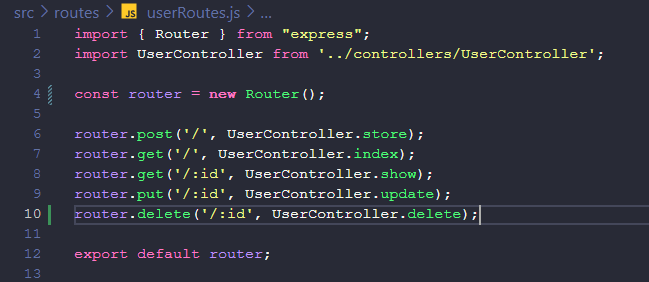
Esse método será muito parecido com o método **update** porém não vamos enviar um usuário e sim apaga-lo.

Para isso vamos primeiramente localizar o usuário que será apagado através do **id** fornecido na url.

Depois iremos utilizar o método **destroy()** do Model para apagar o usuário que foi localizado anteriormente.

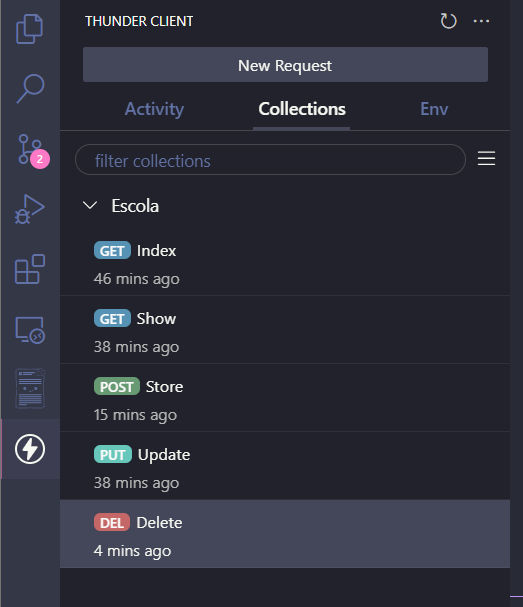


Temos também que cria a nova rota no arquivo **userRoutes** onde iremos criar a rota para apagar um usuário, nesse caso utilizamos o método **delete** e iremos enviar um parâmetro na requisição **id.**

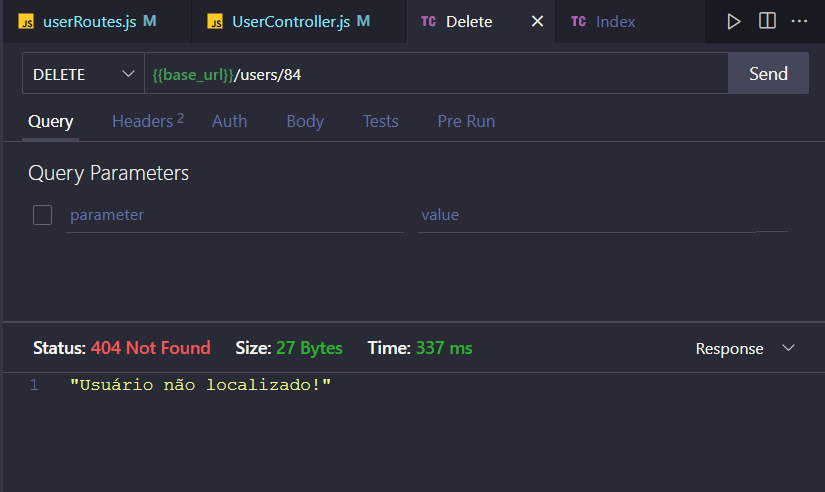


Foi instalado uma nova extenção no vscode que é o Thunder Client que será utilizado no lugar do Insomnia.

A extensão é bem intuitiva e muito parecida com o Insomnia, e já foram criadas as requisições anteriores, agora vamos criar somente a requisição **delete.**

****

A requisição para o **Delete** ficou da seguinte forma.



**174 – Gerando o JWT do usuário**

Na sessão anterior utilizávamos sessão para logar o usuário, porém vamos agora utilizar um token que será gerado para o usuário.

Dessa forma o servidor não irá precisar controlar o acesso do usuário através de seção pois será utilizado um token que iremos fornecer para o usuário e toda vez que ele for logar na aplicação, iremos conferir se a chave enviada pelo usuário bate com a chave que criamos e dessa forma deixamos ele logar ou não.

**Criando variáveis no .env**

Teremos que criar mais duas chaves no arquivo **.env** que será o **TOKEN\_SECRET** que será utilizado para gerar o **Token** para o usuário e também o **TOKEN\_EXPIRATION** que será o prazo de validade do token podendo ser 1 dia ou vários.



**Instalando o JWT**

Digite :

**- npm i jsonwebtoken**

**Criando TokenController.js**

Temos que criar um controller para o token. Vamos criar o arquivo **TokenController.js** na pasta controllers.

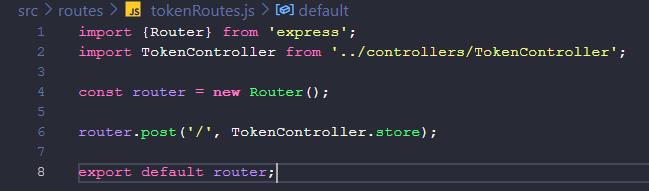
Nesse arquivo vamos importar o model User e criar a classe TokenController, vamos criar um método **store** que será utilizado para gerar o token.



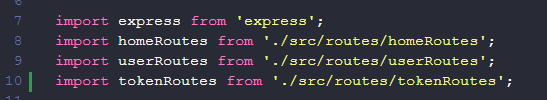
**Criando tokenRoutes.js**

Agora vamos criar uma rota para gerar os tokens então na pasta **routes** vamos criar o arquivo **tokenRoutes.js** nele iremos importar também o **Router** e o **TokenController.**

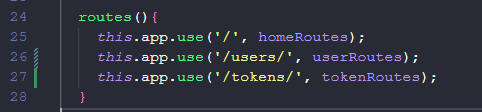
Iremos criar uma rota do tipo **post** que irá chamar o método **TokenController.store**.



Agora temos que ira no arquivo **app.js** e registrar essa rota lá, para isso vamos importar o arquivo **tokenRoutes.**

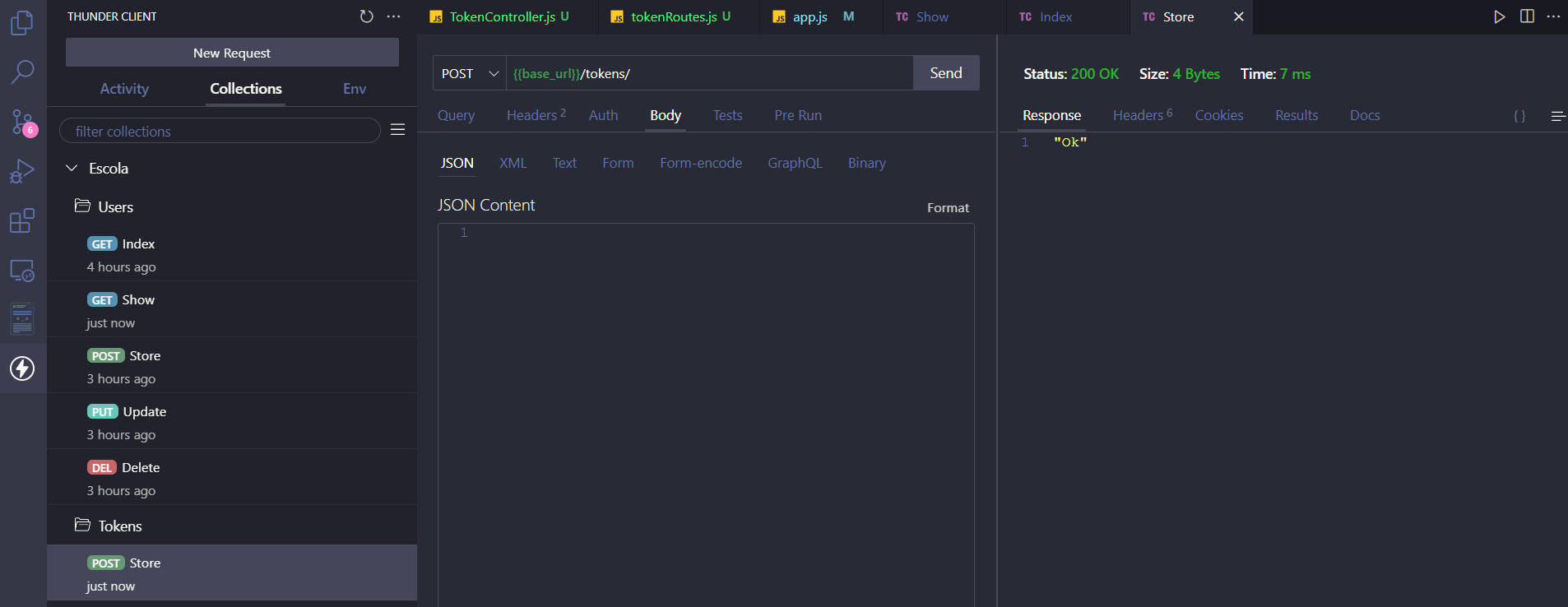


e no método **routes()** iremos configurar o uso.



**Gerando o Token no Thunder Client**

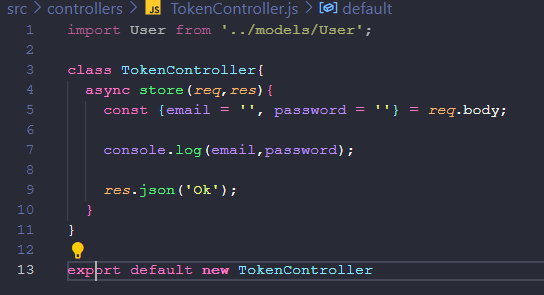
Vamos criar a pasta Tokens e nela vamos criar um **HTTP Request** do tipo **post** que será chamado de **store,** iremos utilizar a url **http://localhost:3001/tokens/**

****

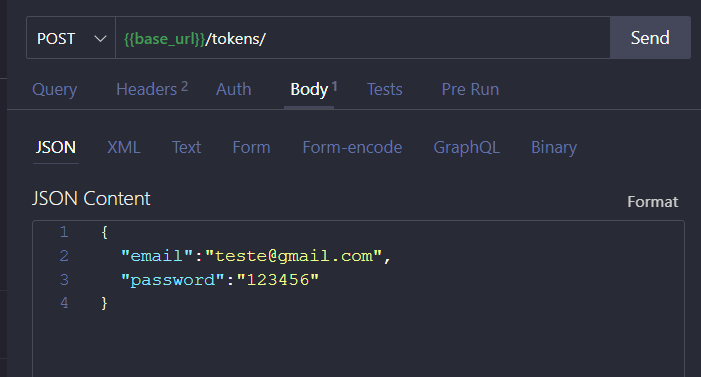
Vemos que está funcionando a rota pois foi retornado a mensagem “OK”.

**Configurando o método store do TokenController.**

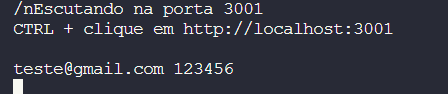
Nesse ponto podemos configurar o controller para que seja gerado o token, para isso vamos fazer um destruct do req.body pegando o **email** e **password** que foi enviado pelo usuário. Vamos executar um console.log para verificar se a requisição esta ocorrendo.



Passando os dados na requisição veremos o resultado no conole.



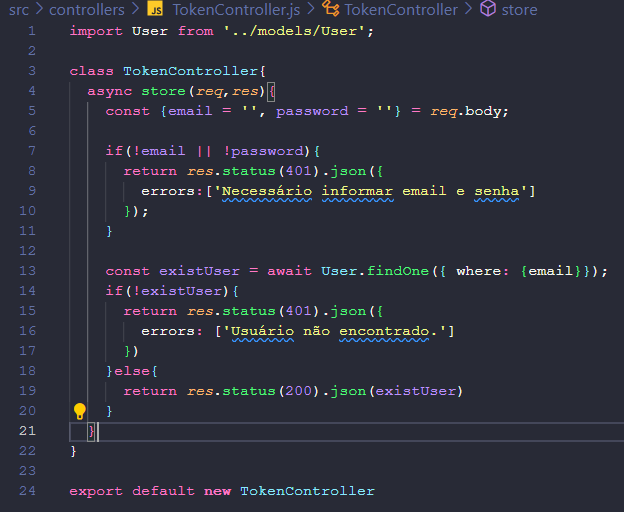
Resultado no console.



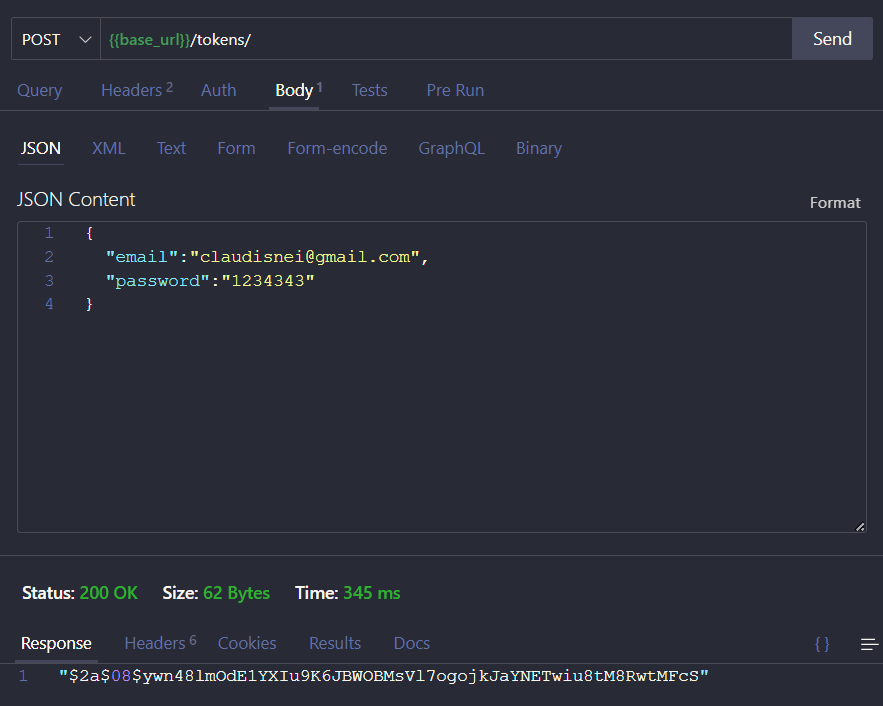
**Verificando o email e senha do usuário.**

Vamos criar uma constante que irá utilizar o método **findOne** do model **User** teremos que passar uma cláusula **Where** onde iremos passar o email que está na requisição.

Se o usuário existir será retornado “Ok”caso contrário será exibido a mensagem “Usuário não encontrado.”.

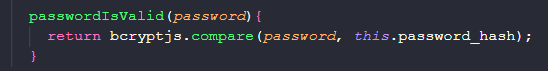


Realizando a requisição no Thunder Client veremos o resultado.



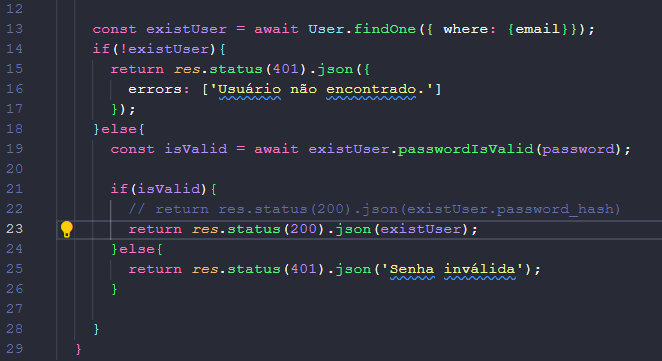
**Criando método passwordIsValid no model User**

Vamos criar esse método para verificar se a senha passada pelo usuário no controller é igual o hash cadastrado no sistema.



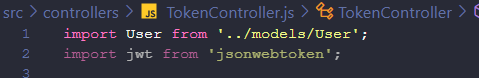
No arquivo **TokenController** podemos agora verificar se a senha passada pelo usuário é mesma cadastrada no banco de dados, para isso vamos criar uma constante que irá receber o resultado da chamada do método **passwordIsvalid** do model.

Nesse caso para testar estamos enviando mensagem de ‘Senha inválida’ ou o usuário.



Vamos realizar algumas alterações, pois queremos retornar para o usuário um token e não seus dados.

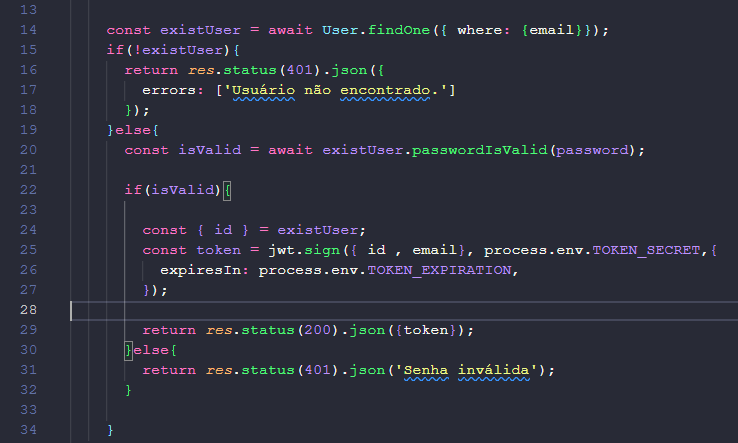
Para isso vamos importar o jwt do ‘jsonwebtoken’.



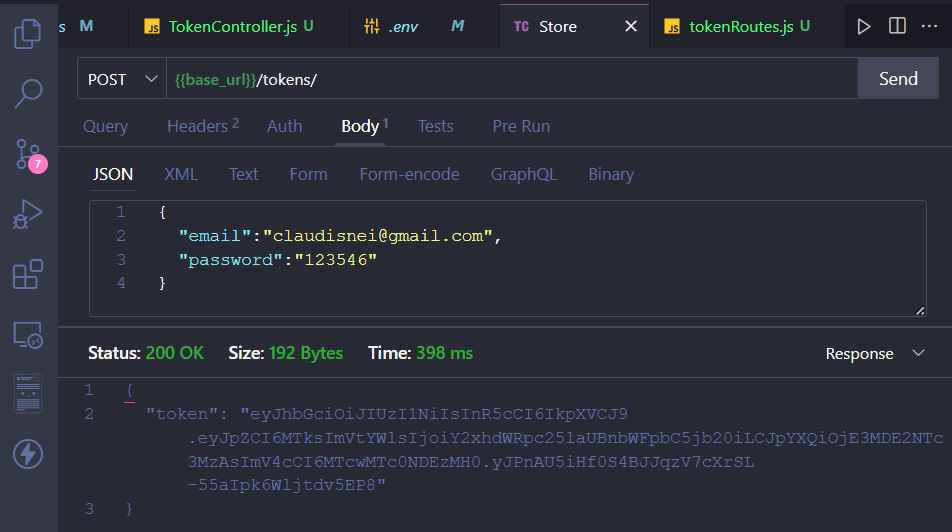
Vamos criar uma constante chamada **token**  que irá receber o token criado pelo método **sign()** do jwt. Teremos que passar alguns parâmetros para esse método.

Que será o **id, email** do usuário, o **token\_secret** que está no arquivo **.env** e o tempo para expirar o token, esse tempo está no **.env**.Para passar o **id** teremos que fazer um destruct no usuário.

Esses dados correspondem ao **payload** do jwt, ou seja quando um usuário mandar um token, o jwt irá verificar se no token os dados de **id** e **email** são válidos e irá retornar o token criado.



Será retornado o token quando executar a requisição, toda vez que realizarmos a requisição, o token será gerado novamente.



**175 -**