

CRIANDO PROJETO NODE.JS

Crie a pasta do Projeto e depois o comando abaixo, para gerar o package.json.

```
npm init -y
```

Instalar o Express.js

```
npm install express --save
```

Instalar o nodemon – save-dev

```
npm install nodemon --save-dev
```

```
npm install express-async-errors --save
```

Crie uma pasta chamada **src** e um arquivo chamado **server.js**.

Endereço: src/server.js

```
const express = require('express')

const app = express()

const PORT = 3333
app.listen(PORT, () => console.log(`Server is running on Port
${PORT}`))
```

No arquivo package.json, crie os scripts de inicialização abaixo:

```
"scripts": {
  "start": "node ./src/server.js",
  "dev": "nodemon ./src/server.js"
},
```

Para executar o servidor, execute um dos comandos:

- node src/server.js
- npm start
- npm run dev

Link github:

https://github.com/brunobandeiraf/API_Express_Prisma_Node_JS/commit/236d9e579ec0448f18de0ae4b66af783203fd31d

1. ORM PRISMA

O ORM [Prisma](#) é uma ferramenta de mapeamento objeto-relacional (ORM) moderna e poderosa para bancos de dados. Ele facilita a interação entre a aplicação e o banco de dados, permitindo que os desenvolvedores escrevam consultas de banco de dados usando uma sintaxe familiar de linguagem de programação, em vez de consultas SQL diretamente.

O Prisma é uma ferramenta de código aberto que oferece suporte a vários bancos de dados populares, como PostgreSQL, MySQL e SQLite. Ele permite que os desenvolvedores definam modelos de dados usando uma linguagem de definição de modelo declarativa e, em seguida, fornece métodos para realizar operações de leitura, gravação, atualização e exclusão (CRUD) nesses modelos de dados.

O Prisma também oferece suporte a recursos avançados, como transações, relacionamentos entre tabelas, migrações de banco de dados e consultas complexas. Além disso, ele é compatível com muitos frameworks e tecnologias de back-end populares, o que o torna uma escolha atraente para muitos desenvolvedores e equipes de desenvolvimento.

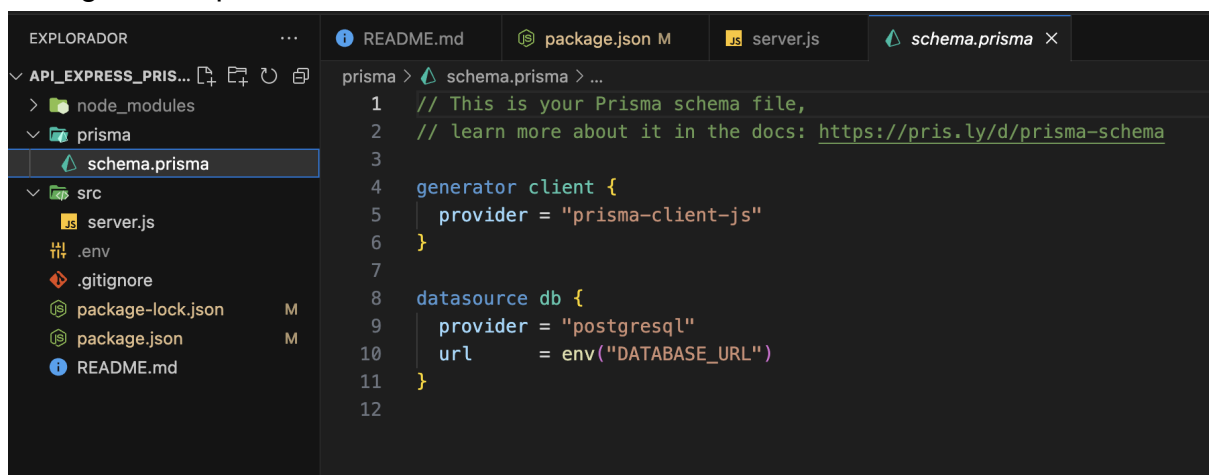
Instale o Prisma como dependência de desenvolvimento

```
npm i prisma -D
```

Inicializar o prisma.

```
npx prisma init
```

Será gerada a pasta abaixo:



The screenshot shows a code editor with a file explorer on the left and a code editor on the right. The file explorer shows a project structure with a 'prisma' folder containing a 'schema.prisma' file. The code editor shows the content of 'schema.prisma' with the following text:

```
prisma > schema.prisma > ...
1 // This is your Prisma schema file,
2 // learn more about it in the docs: https://pris.ly/d/prisma-schema
3
4 generator client {
5   provider = "prisma-client-js"
6 }
7
8 datasource db {
9   provider = "postgresql"
10  url      = env("DATABASE_URL")
11 }
12
```

2.1 CRIANDO TABELA USERS

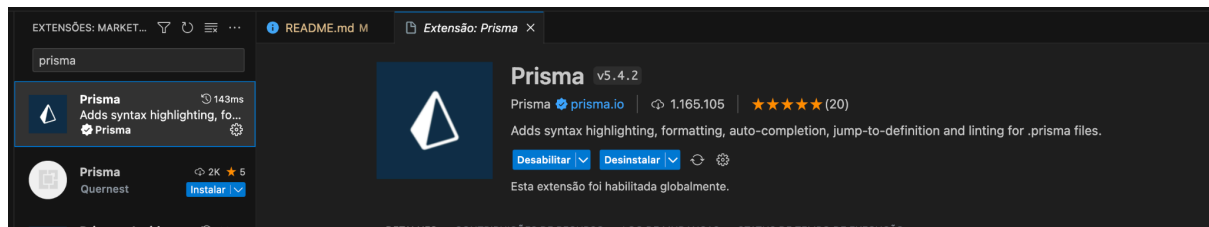
Vamos trabalhar com uma tabela simples de usuário porque o objetivo desta aula é utilizar o Express junto com o ORM Prisma. Mais detalhes sobre o Prisma, consulte o material da aula anterior.

Crie o código para criar a tabela User, logo abaixo do código gerado no schema.prisma.

Endereço: prisma/schema.prisma

```
model User {  
  id String @id @default(uuid())  
  name String  
  email String  
  @@map("users")  
}
```

Faça a instalação da extensão [Prisma](#) no VSCode



Para criar as tipagem dos dados criados, utilizando o comando abaixo.

`npx prisma generate`

Comando para buscar as tabelas do bd já criado e criar os modelos no JS/TS (Prisma Client).

`npm prisma db pull`

`npm i @prisma/client`

Link do github:

https://github.com/brunobandeiraf/API_Express_Prisma_Node_JS/commit/7f7ee3de3ff518df088065e395663b0a9e6e6bf5

2. 'POSTGRESQL COM DOCKER

[Docker](#) é uma plataforma de software que permite que você crie, teste e implemente aplicativos rapidamente. Ele permite empacotar e distribuir aplicativos em contêineres, que são ambientes leves e portáteis que incluem tudo o que é necessário para executar um software, incluindo o código, as bibliotecas, as dependências e as variáveis de ambiente.

Os contêineres Docker são executados em cima do sistema operacional do host e compartilham o núcleo do sistema operacional, o que os torna mais eficientes em termos de recursos do que as máquinas virtuais tradicionais. Eles garantem que os aplicativos sejam executados da mesma maneira em diferentes ambientes, o que ajuda a eliminar problemas de inconsistência entre desenvolvimento, teste e produção.

Com o Docker, os desenvolvedores podem empacotar seus aplicativos juntamente com todas as dependências em um contêiner Docker, o que facilita a criação, a implantação e o gerenciamento de aplicativos em vários ambientes, como data centers locais, nuvens públicas e privadas, bem como em máquinas individuais.

Vamos começar acessando o hub com as imagens disponíveis no Docker:
<https://hub.docker.com/search?q=postgres>

A imagem do bitnami possui mais segurança do que a imagem oficial da postgresql.
<https://hub.docker.com/r/bitnami/postgresql>

Após a instalação do Docker, verifique a instalação com o comando:

docker -v

Algumas considerações. Para criar a imagem do docker:

```
docker run - -name NOME_DO_DOCKER -e POSTGRESQL_USERNAME=docker -e POSTGRESQL_PASSWORD=docker -e POSTGRESQL_DATABASE=api_node -p 5432:5432 bitnami/postgresql
```

Listar todos os containers rodando

docker ps

Listar todos os container criados em algum momento

docker ps -a

Inicializar um container docker, basta digitar o id do container ou o nome, como segue

docker start NOME_DO_DOCKER

Para parar, basta digitar o comando

docker stop NOME_DO_DOCKER

Para excluir, basta digitar o comando

docker rm NOME_DO_DOCKER

3.1 Docker Compose

O Docker Compose é usado para definir os serviços de uma aplicação em um arquivo YAML e, em seguida, pode ser usado para criar e iniciar todos os serviços de uma só vez. Com o Docker Compose, você pode executar várias partes de um aplicativo como contêineres separados, em vez de colocá-los todos em um único contêiner.

Ao usar o Docker Compose, é possível definir os serviços, redes e volumes necessários para executar um aplicativo Docker completo. Isso inclui especificar variáveis de ambiente, comandos de inicialização, mapeamentos de porta e muito mais.

Na raiz do projeto, no arquivo **.env**, faça a seguinte alteração:

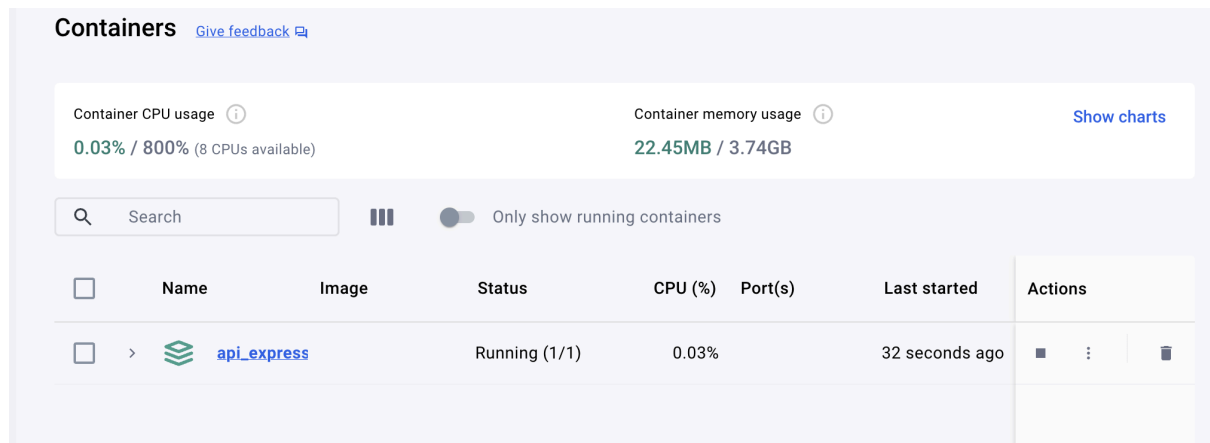
```
DATABASE_URL="postgresql://docker:docker@localhost:5432/api_express_prisma?schema=public"
```

Agora, ainda na raiz do projeto, crie um arquivo chamado **docker-compose.yml**

```
version: '3'

services:
  api_prisma_node:
    image: bitnami/postgresql
    ports:
      - 5432:5432
    environment:
      - POSTGRESQL_USERNAME=docker
      - POSTGRESQL_PASSWORD=docker
      - POSTGRESQL_DATABASE=api_express_prisma
```

docker compose up -d



Utilize o comando acima para inicializar o container configurado. Também é possível parar o container inicializado com o comando stop.

docker compose stop

Com o container do docker rodando, vamos criar a tabela Users no banco de dados criado. Para isso, vamos executar o comando de criação de migration do Prisma. Em seguida será questionado o nome da migration, para controle de versão. Adicionei o nome migrate_users

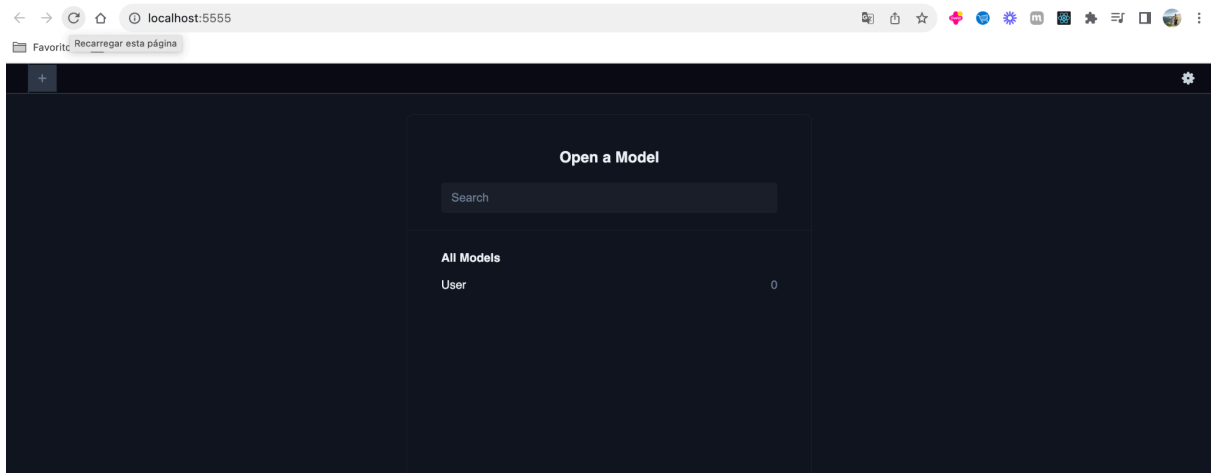
npx prisma migrate dev

The screenshot shows a VS Code editor with a file explorer on the left and a code editor on the right. The file explorer shows a project structure with folders like 'node_modules', 'prisma', 'migrations', and '20231120162609_migrate_users'. The file 'migration.sql' is selected. The code editor shows the content of 'migration.sql', which is a Prisma migration file. It starts with 'prisma > migrations > 20231120162609_migrate_users > migration.sql > -- CreateTable'. The main content is a SQL 'CREATE TABLE' statement for a table named 'users' with columns 'id', 'name', and 'email', all of type 'TEXT NOT NULL'. The 'id' column is also the primary key, indicated by 'CONSTRAINT "users_pkey" PRIMARY KEY ("id")'.

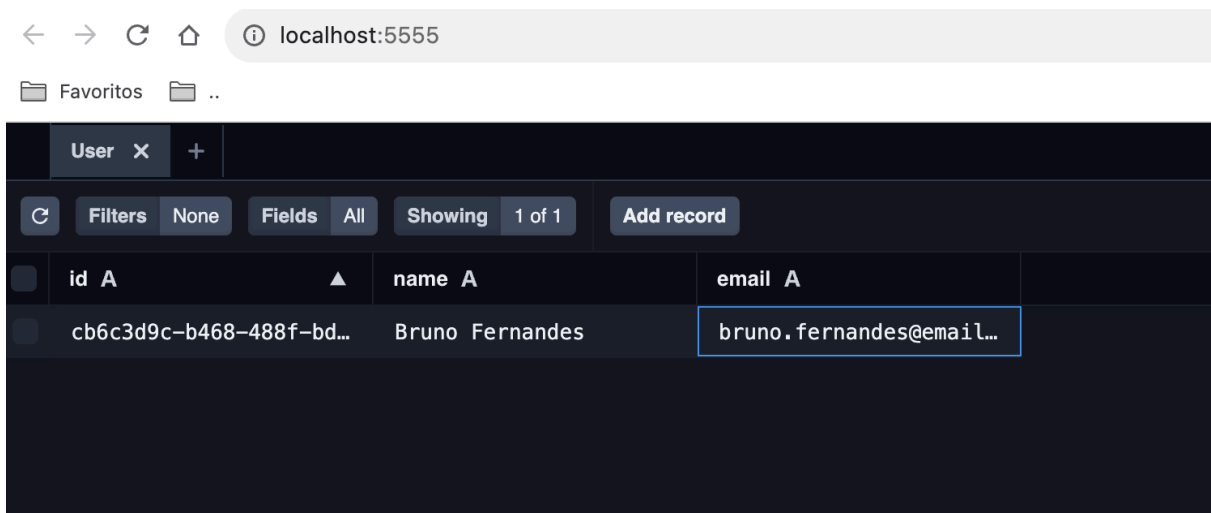
```
prisma > migrations > 20231120162609_migrate_users > migration.sql > -- CreateTable
1  -- CreateTable
2  CREATE TABLE "users" (
3    "id" TEXT NOT NULL,
4    "name" TEXT NOT NULL,
5    "email" TEXT NOT NULL,
6
7    CONSTRAINT "users_pkey" PRIMARY KEY ("id")
8  );
9
```

Para visualizar a tabela criada, execute o comando para inicializar o Prisma Studio

npx prisma studio



O Prisma Studio será automaticamente inicializado no navegador, sendo possível visualizar as tabelas criadas e ainda editar os registros das tabelas.



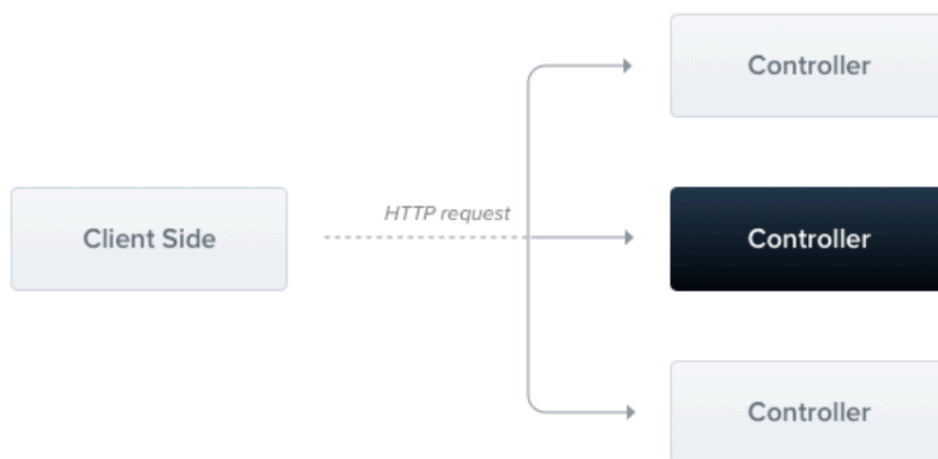
Link github:

https://github.com/brunobandeiraf/API_Express_Prisma_Node_JS/commit/c856640f2dc4fd5d8fbe5a57c33c26cbaf335556

3. CONTROLLER DE USERS

Agora iremos criar um controller para receber as requisições HTTP e tratar as informações, podendo realizar ações como validar dados, buscar informações do banco de dados e enviar uma resposta ao cliente. O controller é uma parte importante da arquitetura de um servidor web, e ajuda a manter as regras de negócio separadas do restante da aplicação.

Segundo a própria documentação do [Nest.JS](#), um importante Framework JS, os controladores são responsáveis por lidar com as solicitações recebidas e retornar as respostas ao cliente.



O objetivo de um controlador é receber solicitações específicas para a aplicação. O mecanismo de roteamento controla qual controlador recebe as solicitações. Frequentemente, cada controlador possui mais de uma rota, e rotas diferentes podem executar ações diferentes.

Os métodos HTTP (Hypertext Transfer Protocol) são utilizados para indicar a ação que está sendo solicitada ou realizada em um recurso específico. Os métodos mais comuns são GET, POST e PATCH. Abaixo, explico cada um deles:

GET:

- **Descrição:** o método GET é utilizado para solicitar dados de um recurso específico. É uma operação "segura" e "idempotente", o que significa que não deve causar alterações no estado do servidor e pode ser repetida sem efeitos colaterais.

- **Exemplo de Uso:** uma requisição GET é comumente usada ao acessar uma página da web. Por exemplo, ao digitar uma URL no navegador, o navegador envia uma solicitação GET para obter a página associada.

POST:

- **Descrição:** o método POST é utilizado para enviar dados ao servidor para criar um novo recurso. Ao contrário do GET, as requisições POST não são consideradas idempotentes, pois podem causar alterações no estado do servidor a cada chamada.
- **Exemplo de Uso:** Formulários em páginas da web geralmente usam requisições POST para enviar dados do usuário ao servidor.

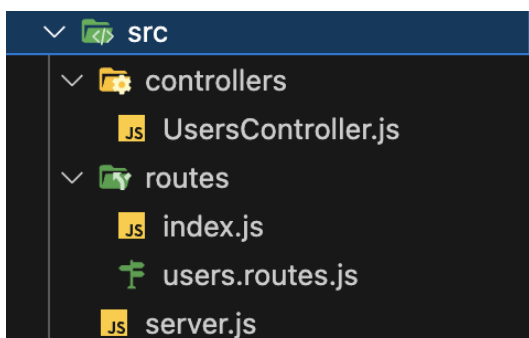
PATCH:

- **Descrição:** o método PATCH é utilizado para aplicar modificações parciais a um recurso. Ele é frequentemente utilizado para atualizar parte dos dados de um recurso sem substituir o recurso como um todo.
- **Exemplo de Uso:** Pode ser usado em situações em que apenas alguns campos de um recurso precisam ser atualizados, sem a necessidade de enviar todas as informações novamente.

Além desses, existem outros métodos HTTP como **PUT** (substitui um recurso ou cria um se não existir) e **DELETE** (remove um recurso). A escolha do método adequado depende da semântica da operação que você está realizando.

41. Create

Primeiramente dentro da pasta src iremos criar uma pasta chamada '**controllers**', responsáveis por possuir todos os controllers criados na aplicação. Dentro do pasta, crie outra pasta chamada '**users**'. Em seguida, iremos criar um arquivo chamado '**routes**', responsável por possuir/conhecer todas as rotas existentes.



src/controllers/UsersController.js

```
const { PrismaClient } = require('@prisma/client');
const prisma = new PrismaClient();

class UsersController{

  async create(request, response){
    try{
      const { name, email } = request.body

      const user = await prisma.user.create({
        data: {
          name,
          email,
        },
      })

      response.json(user)
    }catch (err) {
      return response.status(409).send()
    }
  }
}

module.exports = UsersController
```

src/routes/users.router.js

```
const { Router } = require('express')

const UsersController = require('../controllers/UsersController')

const usersRoutes = Router()

// Controller
const usersController = new UsersController()

// Rotas
usersRoutes.post('/create', usersController.create)
```

```
// Exporta
module.exports = usersRoutes
```

src/routes/index.js

```
const { Router } = require('express')

const usersRoutes = require('./users.routes')

const routes = Router()

// Rotas dos controllers
routes.use('/users', usersRoutes)

module.exports = routes
```

Atualize o arquivo server.js

```
const express = require('express')

// Index das routes
const routes = require('./routes')

const app = express()
app.use(express.json())

app.use(routes)

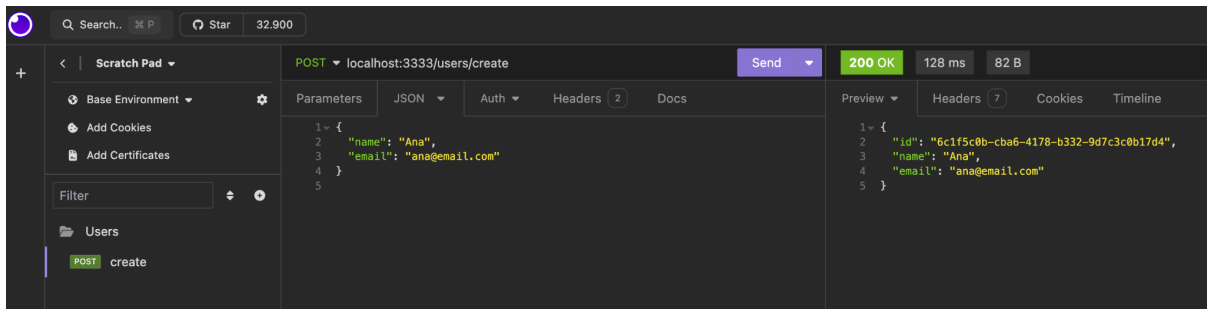
const PORT = 3333
app.listen(PORT, () => console.log(`Server is running on Port ${PORT}`))
```

No postman ou Insonmia

Rota: <http://localhost:3333/users/create>

JSON:

```
{
  "name": "user",
  "email": "user@email.com"
}
```

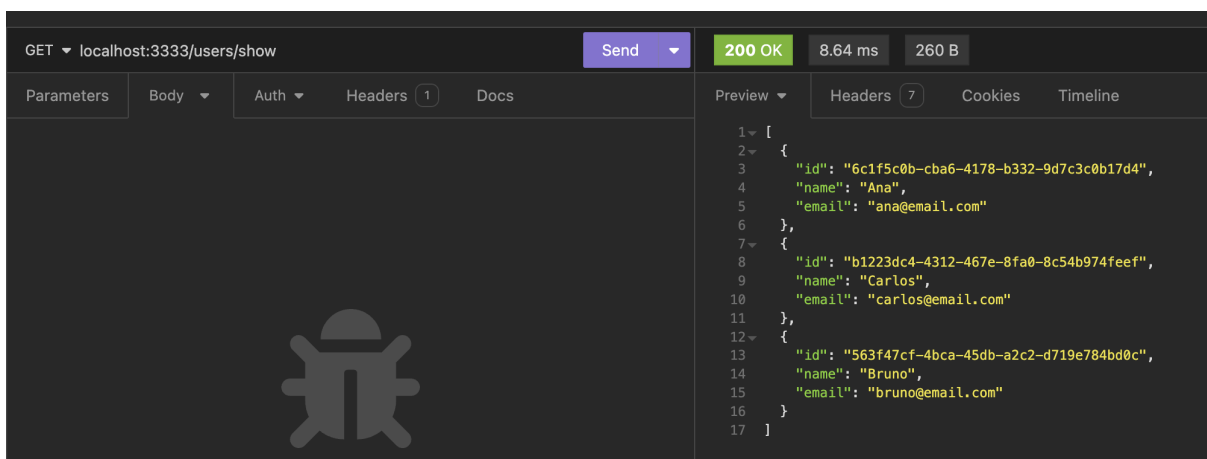


Link github:

https://github.com/brunobandeiraf/API_Express_Prisma_Node_JS/commit/354b2b3977bc188c514bcb882203678053fdf7e5

4.2. Show Users

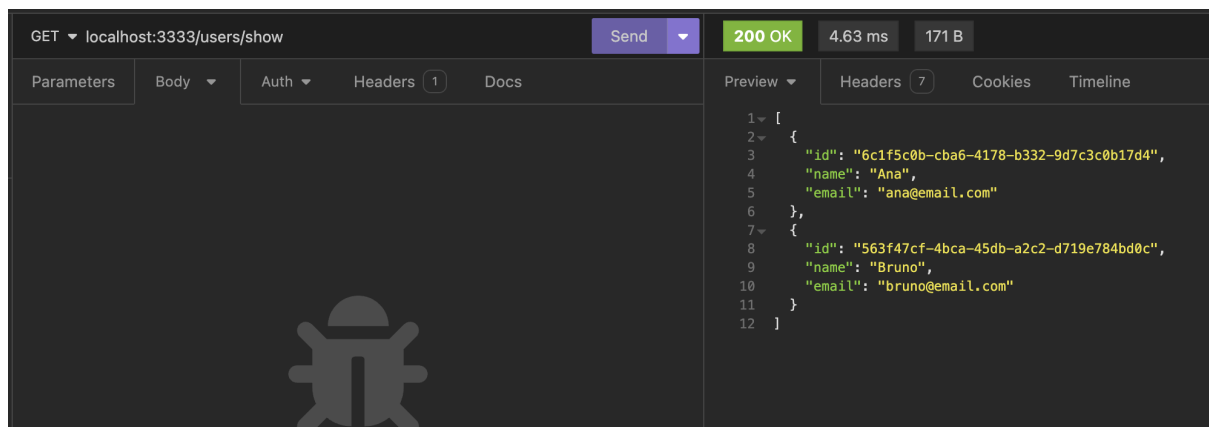
localhost:3333/users/show



```
async show(request, response) {  
  try{  
    const users = await prisma.user.findMany();  
  
    response.json(users)  
  
  }catch (err) {  
    return response.status(409).send()  
  }  
}
```

```
}
```

```
// Rotas  
usersRoutes.post('/create', usersController.create)  
usersRoutes.get('/show', usersController.show)
```



GET localhost:3333/users/show

Send

200 OK 4.63 ms 171 B

Parameters Body Auth Headers 1 Docs

Preview Headers 7 Cookies Timeline

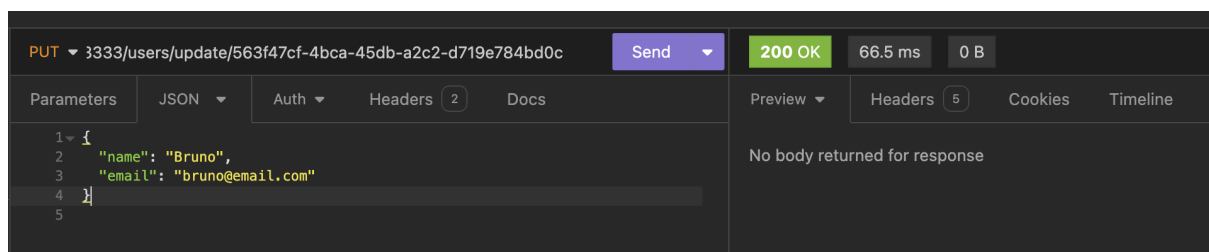
```
1 [
2   {
3     "id": "6c1f5c0b-cba6-4178-b332-9d7c3c0b17d4",
4     "name": "Ana",
5     "email": "ana@email.com"
6   },
7   {
8     "id": "563f47cf-4bca-45db-a2c2-d719e784bd0c",
9     "name": "Bruno",
10    "email": "bruno@email.com"
11  }
12 ]
```

Link github:

https://github.com/brunobandeiraf/API_Express_Prisma_Node_JS/commit/8f2a32b313e271f8b7d99c5e5f0201c9021f631b

4.3. Update User

localhost:3333/users/update/563f47cf-4bca-45db-a2c2-d719e784bd0c



PUT 3333/users/update/563f47cf-4bca-45db-a2c2-d719e784bd0c

Send

200 OK 66.5 ms 0 B

Parameters JSON Auth Headers 2 Docs

Preview Headers 5 Cookies Timeline

```
1 {
2   "name": "Bruno",
3   "email": "bruno@email.com"
4 }
5
```

No body returned for response

```
async update(request, response) {  
  try{
```

```
    const { name, email } = request.body
    const { id } = request.params

    const result = await prisma.user.update({
      where: {
        id: id,
      },
      data: {
        name: name,
        email: email,
      },
    });

    return response.status(200).send()

  } catch (err) {
    return response.status(409).send()
  }
}
```

```
// Rotas
usersRoutes.post('/create', usersController.create)
usersRoutes.get('/show', usersController.show)
usersRoutes.put('/update/:id', usersController.update)
```

Link github:

https://github.com/brunobandeiraf/API_Express_Prisma_Node_JS/commit/4ce2fecad6bdc7f0aef8f6c2cdf993827db88320

4.4. Delete User

```
// Rotas
usersRoutes.post('/create', usersController.create)
usersRoutes.get('/show', usersController.show)
usersRoutes.put('/update/:id', usersController.update)
usersRoutes.delete('/delete/:id', usersController.delete)
```

```
async delete(request, response) {

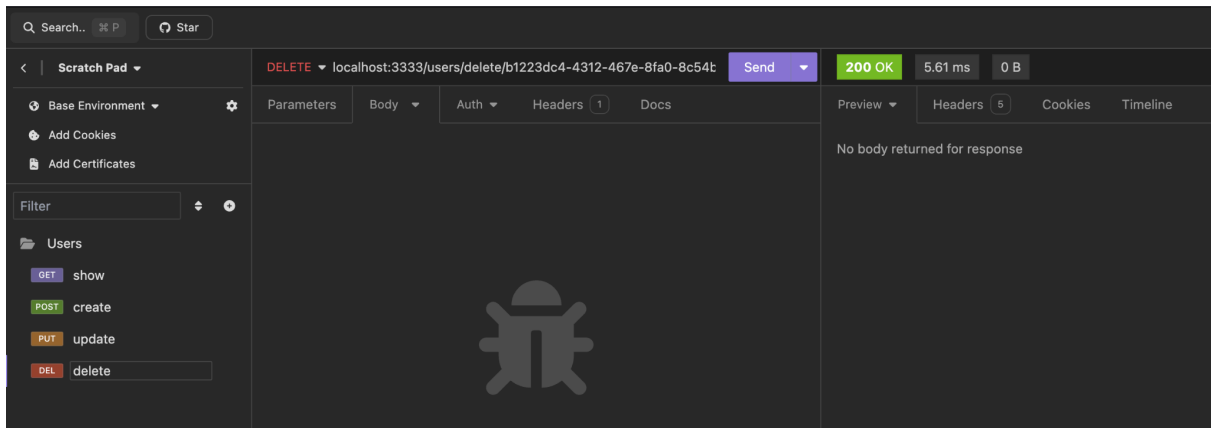
  try{
    const { id } = request.params
    //const { id } = request.body
    console.log(`id: ${id}`)

    const deleteUser = prisma.user.delete({
      where: {
        id: id,
      }
    })

    return response.status(200)

  }catch (err) {
    return response.status(409).send()
  }

}
```



Link github:

https://github.com/brunobandeiraf/API_Express_Prisma_Node_JS/commit/3849aa5635f42063b930e818c542b9b5b26c9b62