

GUIA DEFINITIVO

COMO CRIAR UMA API REST COM NODE.JS EM EXPRESS E PRISMA ORM

ex



Ebook desenvolvido por:

Kleber Coelho

SUMÁRIO

INTRODUÇÃO.....	2
ESTRUTURA DO PROJETO.....	3
CONFIGURANDO O PACKAGE.JSON.....	5
CAPÍTULO 1: INICIANDO O PROJETO COM EXPRESS E PRISMA.....	6
1. INICIALIZANDO O PROJETO NODE.JS.....	6
2. INSTALANDO DEPENDÊNCIAS ESSENCIAIS.....	6
3. CONFIGURANDO O PRISMA.....	7
CAPÍTULO 2: INICIANDO O SERVIDOR.....	10
CAPÍTULO 3: CONFIGURANDO O EXPRESS E AS ROTAS.....	11
CAPÍTULO 4: MIDDLEWARE DE AUTENTICAÇÃO COM JWT.....	13
CAPÍTULO 5: FUNÇÃO DE HASH DE SENHA COM BCrypt.....	14
CAPÍTULO 6: CONFIGURANDO AS ROTAS DE AUTENTICAÇÃO.....	15
CAPÍTULO 7: CONFIGURANDO AS ROTAS DE CLIENTES.....	16
CAPÍTULO 8: IMPLEMENTANDO O CONTROLADOR DE AUTENTICAÇÃO.....	18
CAPÍTULO 9: IMPLEMENTANDO O CONTROLADOR DE CLIENTES.....	22
CAPÍTULO 10: FAZENDO REQUISIÇÕES PELO INSOMNIA.....	27
CAPÍTULO 11: BONUS! CONFIGURANDO O BANCO DE DADOS.....	35
RESUMÃO 1: AuthController.js.....	41
RESUMÃO 2: ClientsController.js.....	45
RESUMÃO 3: authenticateToken.js.....	49
RESUMÃO 4: passwordUtils.js.....	50
RESUMÃO 5: authRouter.js.....	50
RESUMÃO 6: clientsRouter.js.....	51
RESUMÃO 7: app.js.....	52
RESUMÃO 8: server.js.....	52

INTRODUÇÃO

Bem-vindo(a) a este e-book, onde você aprenderá a criar uma API REST moderna utilizando **Node.js, Express e Prisma**. Com a crescente demanda por aplicações web robustas, escaláveis e seguras, as APIs REST tornaram-se essenciais no desenvolvimento backend. Este guia foi criado para fornecer uma compreensão clara dos conceitos essenciais, juntamente com instruções detalhadas para que você possa construir uma API do zero, seguindo as melhores práticas do mercado.

Nosso objetivo é mostrar como estruturar, desenvolver e implementar uma API de forma organizada e eficiente. Vamos abordar desde a configuração inicial do projeto até a criação de controladores, rotas e middlewares, além da integração com bancos de dados utilizando o **Prisma ORM**. Cada etapa será explicada de maneira simples e prática, para que você possa acompanhar e aplicar os conhecimentos no seu próprio ritmo.

Se você está começando no desenvolvimento de APIs ou busca aprimorar suas habilidades utilizando ferramentas modernas, como o Prisma, este e-book é ideal para você. A cada capítulo, vamos focar em um aspecto crucial da construção de uma API, proporcionando uma experiência de aprendizado prática e acessível.

O que você vai aprender

- Como configurar um ambiente de desenvolvimento Node.js com Express.
- Estruturar uma API REST de forma organizada e escalável.
- Gerenciar a autenticação de usuários com segurança.
- Integrar e utilizar o Prisma para interagir com o banco de dados de forma simples e eficiente.
- Melhorar o desempenho e a segurança da sua API com middlewares e boas práticas.

Ao final deste e-book, você terá desenvolvido uma API completa e funcional, que poderá servir como base para futuros projetos ou como ponto de partida para implementar funcionalidades mais avançadas. Com o conhecimento adquirido, você estará pronto para criar soluções mais complexas e robustas com confiança.

Deixei para facilitar o entendimento os resumos comentados em código com explicação de cada linha de código feita.

Então, vamos começar!

ESTRUTURA DO PROJETO

1. prisma/

- **schema.prisma:** Contém a definição do banco de dados, incluindo modelos de dados, campos e relacionamentos. O Prisma utiliza este arquivo para gerar consultas SQL e interagir com o banco de dados.

2. src/

- Contém todo o código-fonte da aplicação, incluindo controladores, middlewares, repositórios e rotas.

2.1 app/

- Diretório principal que agrupa todas as funcionalidades da aplicação.

2.1.1 controllers/

- **auth/AuthController.js:** Controlador responsável por gerenciar a autenticação e operações relacionadas a login, registro e tokens de usuário.
- **client/ClientsController.js:** Controlador responsável por gerenciar as operações relacionadas a clientes, como criação, leitura, atualização e exclusão de informações.

2.1.2 middleware/

- **authUser/authenticateToken.js:** Middleware que verifica se o token de autenticação enviado pelo usuário é válido. Este arquivo é responsável por proteger rotas que necessitam de autenticação.

2.1.3 repositories/

- **passwordUtils.js:** Funções utilitárias para lidar com hashes de senha e comparar senhas no processo de login. Facilita o gerenciamento seguro de senhas dos usuários.

2.1.4 routes/

- Contém as definições de rotas da API, cada uma delas direcionada para seu respectivo controlador.
- **https/:** Relacionado a configurações de rotas seguras (HTTPS);
- **authUser/authRouter.js:** Define as rotas relacionadas à autenticação de usuários, como login e registro.
- **client/ClientsRouter.js:** Define as rotas relacionadas às operações com clientes, como criação, atualização e listagem de clientes.

2.2 app.js

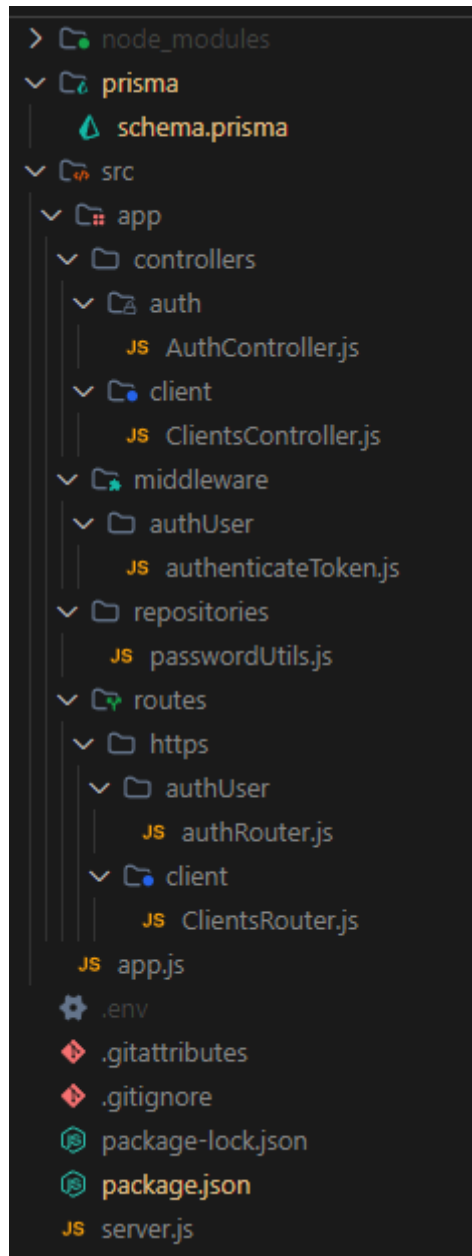
- Arquivo principal da aplicação, onde é feita a configuração do servidor, middlewares globais e a integração das rotas.

2.3 server.js

- Ponto de entrada da aplicação. Inicia o servidor Express e escuta as requisições HTTP.

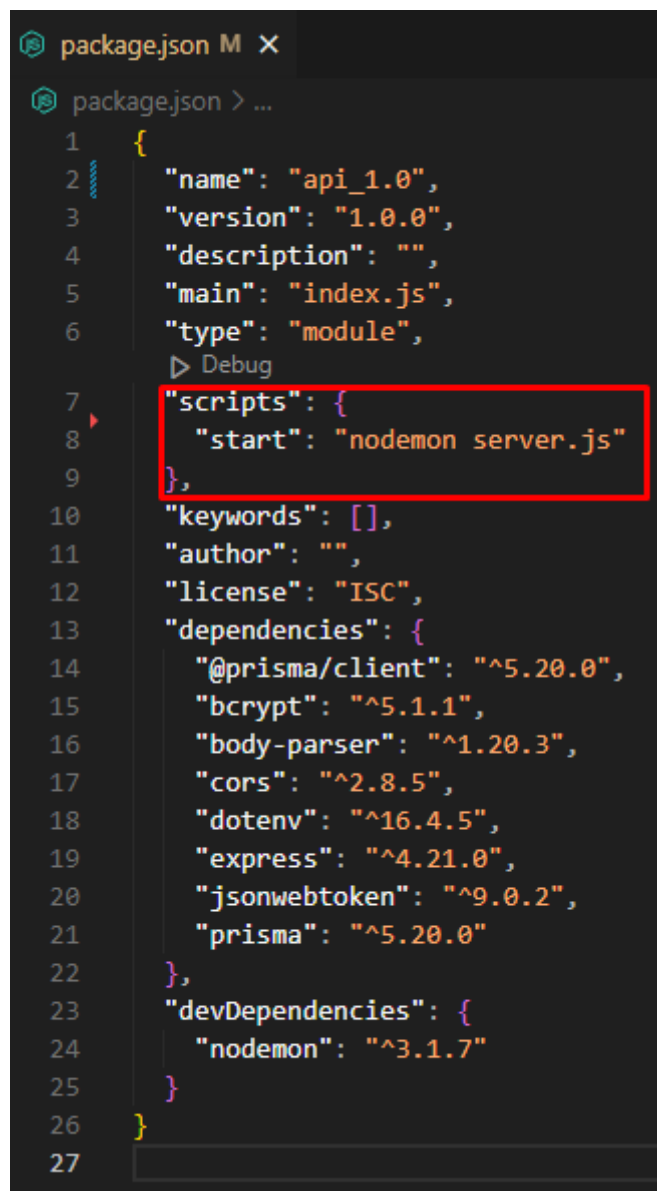
Arquivos de Configuração

- **.env**: Contém variáveis de ambiente, como informações sensíveis do banco de dados, chaves de API, etc.
- **.gitignore**: Lista de arquivos e diretórios que não devem ser versionados no Git.
- **package.json**: Lista as dependências do projeto, scripts e informações básicas como o nome do projeto e versão.



CONFIGURANDO O PACKAGE.JSON:

Crie um script “start” para poder iniciar o servidor passando o “nodemon server.js”



```
package.json M X
package.json > ...
1  {
2    "name": "api_1.0",
3    "version": "1.0.0",
4    "description": "",
5    "main": "index.js",
6    "type": "module",
7    "scripts": {
8      "start": "nodemon server.js"
9    },
10   "keywords": [],
11   "author": "",
12   "license": "ISC",
13   "dependencies": {
14     "@prisma/client": "^5.20.0",
15     "bcrypt": "^5.1.1",
16     "body-parser": "^1.20.3",
17     "cors": "^2.8.5",
18     "dotenv": "^16.4.5",
19     "express": "^4.21.0",
20     "jsonwebtoken": "^9.0.2",
21     "prisma": "^5.20.0"
22   },
23   "devDependencies": {
24     "nodemon": "^3.1.7"
25   }
26 }
27
```

LEMBRANDO DE INICIAR O PROJETO NO TERMINAL COM O COMANDO “NPM START”

CAPÍTULO 1: INICIANDO O PROJETO COM EXPRESS E PRISMA

Neste capítulo, você aprenderá como iniciar o seu projeto criando uma API REST usando o **Express** e integrando o **Prisma** como ORM (Object-Relational Mapping). **O Express é um framework minimalista para Node.js, amplamente utilizado para criar APIs e servidores web de forma rápida e eficiente. Já o Prisma é um ORM (Object-Relational Mapping) moderno, que facilita o gerenciamento e interação com o banco de dados de maneira tipada, garantindo segurança e produtividade.**

Vamos passar pelos seguintes passos:

1. Inicializando o projeto Node.js.
2. Instalando as dependências essenciais.
3. Configurando o Prisma.
4. Configurando o Express.
5. Iniciando o servidor.

1. INICIALIZANDO O PROJETO NODE.JS.

Para começar, crie uma nova pasta para o seu projeto e navegue até ela no terminal:

```
mkdir minha-api
```

```
cd minha-api
```

Inicialize o projeto Node.js:

```
npm init -y
```

Esse comando criará um arquivo package.json com a configuração básica do seu projeto.

2. INSTALANDO DEPENDÊNCIAS ESSENCIAIS

Agora, instale as dependências necessárias para construir a API:

```
npm install express bcrypt body-parser cors dotenv jsonwebtoken @prisma/client
```

E as dependências de desenvolvimento:

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

Aqui está o que cada pacote faz:

- **express:** Framework para criar o servidor e gerenciar as rotas.
- **bcrypt:** Biblioteca para criptografar senhas.
- **body-parser:** Middleware que processa o corpo das requisições HTTP.

- **cors:** Middleware que habilita o CORS (Cross-Origin Resource Sharing).
- **dotenv:** Carrega variáveis de ambiente do arquivo `.env`.
- **jsonwebtoken:** Para gerenciar tokens JWT para autenticação.
- **@prisma/client:** Cliente do Prisma que interage com o banco de dados.
- **nodemon:** Utilitário para reiniciar o servidor automaticamente durante o desenvolvimento.
- **prisma:** CLI do Prisma para gerar o cliente e gerenciar o esquema do banco de dados.

3. CONFIGURANDO O PRISMA

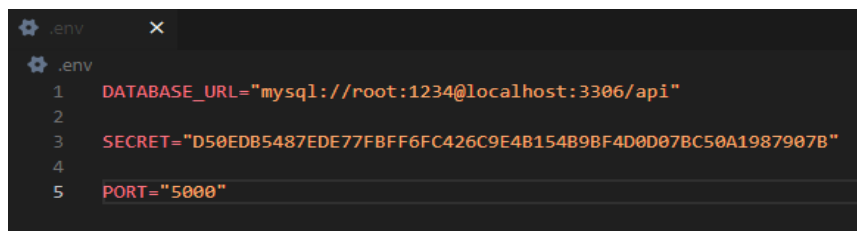
Agora vamos configurar o Prisma para se conectar ao banco de dados e gerar as tabelas necessárias.

1. Inicie o Prisma na pasta do projeto:

```
npx prisma init
```

Este comando cria a pasta `prisma/` e o arquivo `schema.prisma`, onde você define os modelos de dados que serão mapeados para tabelas no banco de dados. Além disso, o arquivo `.env` será criado para configurar a URL de conexão ao banco de dados.

2. Abra o arquivo `.env` e configure a URL de conexão com seu banco de dados. Por exemplo, se você estiver usando o PostgreSQL ou MySQL:
3. Configure a `SECRET` que será utilizado para verificar o token, indico criar um SHA224 Hash para ter mais segurança, lembrando o `SECRET` pode ser qualquer informação.
4. Crie a `PORT` para caso não achar a porta que definimos inicialmente



```
.env
1 DATABASE_URL="mysql://root:1234@localhost:3306/api"
2
3 SECRET="D50EDB5487EDE77FBFF6FC426C9E4B154B9BF4D0D07BC50A1987907B"
4
5 PORT="5000"
```

Defina os modelos de dados no arquivo `prisma/schema.prisma`. Aqui está um exemplo simples para usuários e clientes:


```

prisma > schema.prisma > ...
Generate
1 generator client {
2   provider = "prisma-client-js"
3 }
4
5 datasource db {
6   provider = "mysql"
7   url      = env("DATABASE_URL")
8 }
9
10 model user {
11   id      Int      @id @default(autoincrement())
12   uuid    String   @unique(map: "User_uuid_key") @default(uuid())
13   email   String   @unique(map: "User_email_key")
14   login   String   @unique(map: "User_login_key")
15   password String
16   name    String
17   cpf_cnpj String @unique(map: "User_cpf_cnpj_key")
18   createdAt DateTime @default(now())
19   updatedAt DateTime @updatedAt
20   celular String
21   client client[]
22 }
23
24 model client {
25   id      Int      @id @default(autoincrement())
26   uuid    String   @unique(map: "Client_uuid_key") @default(uuid())
27   name    String
28   cpf     String
29   celular String
30   userUuid String
31   user    user     @relation(fields: [userUuid], references: [uuid], map: "Client_userUuid_fkey")
32
33   @@index([userUuid], map: "Client_userUuid_fkey")
34 }
35

```

Resumo do schema.prisma

Este arquivo define dois modelos principais para o banco de dados: **user** e **client**, além de configurar o uso do MySQL como o banco de dados.

1. Configurações Gerais

- O **generator** é configurado para usar o prisma-client-js, que gera automaticamente o cliente Prisma para interagir com o banco de dados.
- A **datasource** está configurada para usar o MySQL, com a URL de conexão definida pela variável de ambiente DATABASE_URL.

2. Modelo user

- **id**: Chave primária que incrementa automaticamente.
- **uuid**: Campo exclusivo com valor gerado automaticamente usando UUID.
- **email, login, cpf_cnpj**: Todos esses campos são exclusivos (únicos) no banco de dados.
- **password**: Armazena a senha do usuário.
- **name**: Nome do usuário.

- **celular**: Número de celular do usuário.
- **createAt**: Armazena a data de criação do registro, com valor padrão como o momento atual.
- **updateAt**: Atualiza automaticamente a data toda vez que o registro for modificado.
- **client[]**: Relacionamento de um para muitos (um usuário pode ter vários clientes).

3. Modelo client

- **id**: Chave primária que incrementa automaticamente.
- **uuid**: Campo exclusivo com valor gerado automaticamente usando UUID.
- **name**: Nome do cliente.
- **cpf**: CPF do cliente.
- **celular**: Número de celular do cliente.
- **userUuid**: Chave estrangeira que referencia o uuid do usuário, conectando o cliente ao usuário.
- **user**: Relacionamento com o modelo user, definindo o vínculo entre cliente e usuário.
- **@@index([userUuid])**: Índice criado para otimizar consultas que usam a coluna userUuid

Após configurar o schema, podemos utilizar o comando para criar a Migration e enviar os dados ao banco de dados

```
npx prisma migrate dev --name init
```

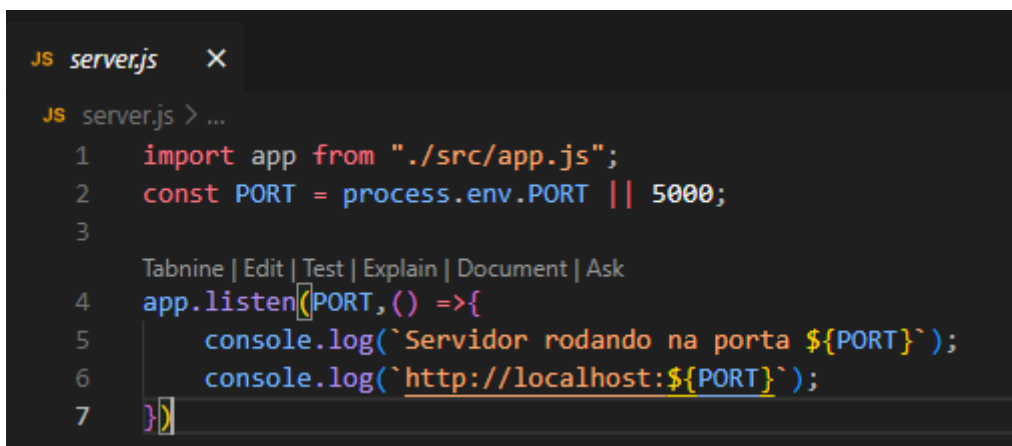
Onde está o “--name init” deve ser utilizado para definir o nome da migration

```
npx prisma migrate dev --first_model
```

CAPÍTULO 2: INICIANDO O SERVIDOR

Neste capítulo, vamos configurar o servidor para rodar a aplicação Express. Vamos utilizar o código para iniciar o servidor e fazer com que ele escute as requisições na porta definida.

O arquivo **server.js** serve como ponto de entrada da nossa aplicação. Ele inicializa o servidor Express e define a porta na qual a aplicação vai rodar, enquanto o **app.js** lida com a configuração dos middlewares e rotas.



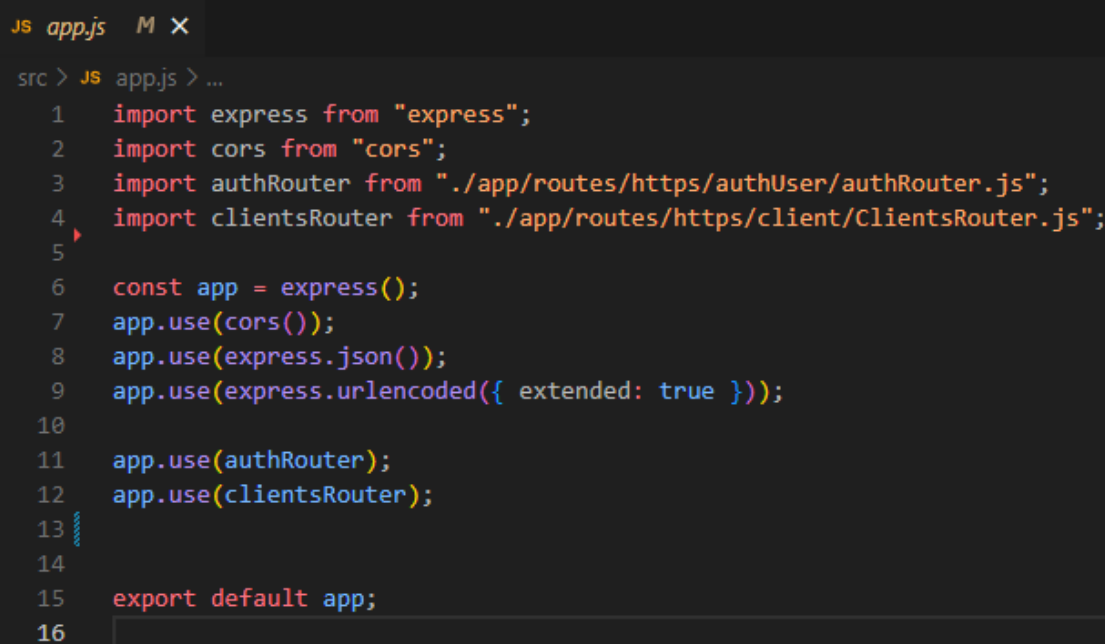
```
JS server.js X
JS server.js > ...
1 import app from "../src/app.js";
2 const PORT = process.env.PORT || 5000;
3
  Tabnine | Edit | Test | Explain | Document | Ask
4 app.listen(PORT, () => {
5   console.log(`Servidor rodando na porta ${PORT}`);
6   console.log(`http://localhost:${PORT}`);
7 })
```

Explicação:

- **import app from "../src/app.js";**
Aqui, estamos importando a instância do Express configurada no arquivo app.js. Esse arquivo contém toda a lógica e os middlewares que foram configurados para lidar com requisições HTTP.
- **const PORT = process.env.PORT || 5000;**
A constante PORT define a porta onde o servidor será iniciado. Ele tenta pegar o valor da porta definido em uma variável de ambiente (process.env.PORT). Caso não haja nenhuma porta definida, o servidor será executado na porta **5000**.
- **app.listen(PORT, () => {...});**
O método app.listen inicia o servidor e faz com que ele escute as requisições HTTP na porta definida. O callback é executado quando o servidor começa a rodar, imprimindo no console a mensagem informando que o servidor está ativo, junto com a URL para acessar localmente: <http://localhost:PORT>.

CAPÍTULO 3: CONFIGURANDO O EXPRESS E AS ROTAS

Agora que temos o servidor rodando, vamos configurar o Express para lidar com requisições HTTP, e configurar as rotas da nossa API. Vamos adicionar middlewares essenciais como **CORS** e **body-parser**, além de configurar as rotas de autenticação e clientes.



```
JS app.js M X
src > JS app.js > ...
1 import express from "express";
2 import cors from "cors";
3 import authRouter from "../app/routes/https/authUser/authRouter.js";
4 import clientsRouter from "../app/routes/https/client/ClientsRouter.js";
5
6 const app = express();
7 app.use(cors());
8 app.use(express.json());
9 app.use(express.urlencoded({ extended: true }));
10
11 app.use(authRouter);
12 app.use(clientsRouter);
13
14
15 export default app;
16
```

Explicação:

- **import express from "express";**
Importamos o framework **Express**, que será utilizado para lidar com requisições HTTP e gerenciar rotas.
- **import cors from "cors";**
CORS (Cross-Origin Resource Sharing) é uma política de segurança que impede que determinados recursos da API sejam acessados por páginas web em domínios diferentes. O middleware `cors()` habilita o CORS na nossa API, permitindo que ela seja acessada por outros domínios.
- **import authRouter from "../app/routes/https/authUser/authRouter.js";**
Aqui, estamos importando as rotas de autenticação de usuários, que estão definidas no arquivo `authRouter.js`. Essas rotas podem incluir operações como login, registro, etc.
- **import clientsRouter from "../app/routes/https/client/ClientsRouter.js";**
Assim como o arquivo anterior, aqui importamos as rotas relacionadas aos **clientes**. Isso permite que nossa API trate de operações CRUD (criação, leitura, atualização e exclusão) para os dados dos clientes.

Middlewares:

- **app.use(cors());**
O middleware `cors()` é aplicado a todas as rotas, permitindo que o servidor responda a requisições de diferentes origens (domínios).
- **app.use(express.json());**
O middleware `express.json()` é usado para garantir que a API seja capaz de interpretar dados no formato JSON enviados nas requisições. Como muitas APIs REST trabalham com JSON, esse middleware é essencial para processar as requisições.
- **app.use(express.urlencoded({ extended: true }));**
Este middleware permite que o Express interprete dados enviados via formulários **URL-encoded**. Ele torna possível processar dados de formulários HTML e incluí-los no objeto `req.body`.

Rotas:

- **app.use(authRouter);**
Com essa linha, integramos as rotas relacionadas à autenticação do usuário na nossa API. Todas as requisições enviadas para essas rotas serão processadas pelo `authRouter`, que contém a lógica de autenticação.
- **app.use(clientsRouter);**
Da mesma forma, adicionamos as rotas dos **clientes** à aplicação. As rotas definidas no `clientsRouter` serão responsáveis por gerenciar os dados dos clientes.

Resumo:

Neste capítulo, configuramos o Express para lidar com requisições JSON e formulários, habilitamos o CORS para que nossa API possa ser acessada por outros domínios, e finalmente, configuramos as rotas para autenticação de usuários e gerenciamento de clientes. Agora, nossa API pode responder a essas requisições corretamente.

CAPÍTULO 4: MIDDLEWARE DE AUTENTICAÇÃO COM JWT

Neste capítulo, vamos adicionar uma camada de segurança à nossa API, garantindo que apenas usuários autenticados possam acessar determinadas rotas. Usaremos **JSON Web Tokens (JWT)** para realizar essa autenticação.

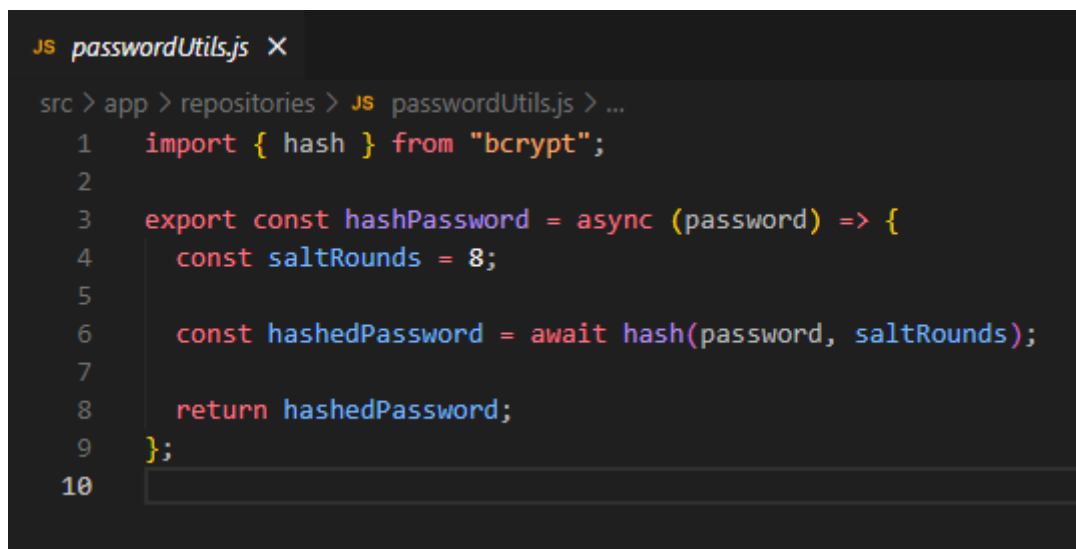
```
JS authenticateToken.js X
src > app > middleware > authUser > JS authenticateToken.js > ...
1  import jwt from "jsonwebtoken";
2
3  const authenticateToken = (req, res, next) => {
4    const authHeader = req.headers.authorization;
5    const token = authHeader && authHeader.split(" ")[1];
6
7    if (!token) {
8      return res.status(401).json({ message: "Nenhum token fornecido" });
9    }
10
11    jwt.verify(token, process.env.SECRET, (err, user) => {
12      if (err) {
13        return res.status(403).json({ message: "Falha ao autenticar" });
14      }
15
16      req.user = user;
17      next();
18    });
19  };
20
21  export default authenticateToken;
22
```

Explicação:

- **import jwt from "jsonwebtoken";**
Aqui, importamos a biblioteca jsonwebtoken, que será usada para verificar a validade do token JWT.
- **authenticateToken:**
Este middleware verifica se o token JWT está presente no cabeçalho de autorização (Authorization). O token é extraído e verificado usando o método `jwt.verify()`. Se o token for válido, o usuário autenticado é anexado à requisição (`req.user`), e a execução da requisição continua com `next()`. Caso contrário, o middleware retorna um erro 401 (não autorizado) ou 403 (proibido), dependendo da situação.
- **Verificação de Erros:**
Se o token estiver ausente ou inválido, uma resposta de erro é enviada ao cliente, garantindo que apenas usuários autenticados possam acessar as rotas protegidas.

CAPÍTULO 5: FUNÇÃO DE HASH DE SENHA COM BCRIPT

Neste capítulo, vamos usar a biblioteca **bcrypt** para criptografar senhas de forma segura antes de armazená-las no banco de dados.



```
JS passwordUtils.js X
src > app > repositories > JS passwordUtils.js > ...
1  import { hash } from "bcrypt";
2
3  export const hashPassword = async (password) => {
4    const saltRounds = 8;
5
6    const hashedPassword = await hash(password, saltRounds);
7
8    return hashedPassword;
9  };
10
```

Explicação:

- **import { hash } from "bcrypt";**
Aqui, importamos a função hash do bcrypt, que será usada para gerar o hash das senhas.
- **hashPassword:**
Esta função recebe uma senha como entrada e usa o bcrypt para gerar um hash seguro. O número de **salt rounds** (número de iterações) é definido como 8, o que garante um bom equilíbrio entre segurança e performance. A senha criptografada é retornada e pode ser armazenada no banco de dados.

Por que usar Bcrypt?

O uso de bcrypt para criptografar senhas é essencial para garantir que as senhas dos usuários não sejam armazenadas em texto simples no banco de dados. Mesmo que o banco seja comprometido, as senhas continuarão protegidas.

CAPÍTULO 6: CONFIGURANDO AS ROTAS DE AUTENTICAÇÃO

Neste capítulo, vamos configurar as rotas de autenticação da API. Aqui, o usuário pode se registrar, fazer login, e obter informações sobre si mesmo.

```
JS authRouter.js X
src > app > routes > https > authUser > JS authRouter.js > ...
1  import { Router } from "express";
2
3  import authenticateToken from "../../middleware/authUser/authenticateToken.js";
4  import authController from "../../controllers/auth/authController.js";
5  const authRouter = Router();
6
7  authRouter.post("/registrar", authController.registrar);
8  authRouter.post("/login", authController.login);
9  authRouter.get("/usuarios/:userId", authController.umUsuario);
10
11 authRouter.get("/me", authenticateToken, authController.me);
12
13 export default authRouter;
14
```

Explicação:

- **import { Router } from "express";**
Importamos o Router do Express para definir e gerenciar as rotas.
- **authRouter:**
Este objeto gerencia todas as rotas relacionadas à autenticação.
- **Rotas principais:**
 - **POST /registrar:** Rota para registrar um novo usuário, onde o controlador `authController.registrar` processa a lógica de registro.
 - **POST /login:** Rota para o login do usuário, onde o controlador `authController.login` realiza a autenticação e gera um token JWT.
 - **GET /usuarios/ :** Rota para buscar informações de um usuário específico pelo seu UUID.
 - **GET /me:** Rota protegida pela autenticação JWT. Para acessar essa rota, o token do usuário precisa ser válido. O controlador `authController.me` retorna as informações do usuário autenticado.

Essa configuração permite que o sistema de autenticação seja seguro e organizado, protegendo dados confidenciais e fornecendo operações de login e registro.

CAPÍTULO 7: CONFIGURANDO AS ROTAS DE CLIENTES

Neste capítulo, vamos configurar as rotas relacionadas ao gerenciamento de **clientes**. As rotas permitirão operações CRUD (Criar, Ler, Atualizar e Deletar) nos dados dos clientes.

```
src > app > routes > https > client > JS ClientsRouter.js > ...
1  import { Router } from "express";
2  import authenticateToken from "../../middleware/authUser/authenticateToken.js";
3  import ClientsController from "../../controllers/client/ClientsController.js";
4
5  const clientsRouter = Router();
6
7  clientsRouter.post(
8    "/clientes",
9    authenticateToken,
10   ClientsController.registerClient
11 );
12 clientsRouter.put(
13   "/clientes/:uuid",
14   authenticateToken,
15   ClientsController.updateClient
16 );
17 clientsRouter.get(
18   "/clientes",
19   authenticateToken,
20   ClientsController.findManyClient
21 );
22 clientsRouter.delete(
23   "/clientes/:uuid",
24   authenticateToken,
25   ClientsController.deleteClient
26 );
27
28 export default clientsRouter;
```

Explicação:

- **import { Router } from "express";**
Novamente, usamos o Router do Express para gerenciar as rotas específicas para clientes.
- **clientsRouter:**
Este objeto define todas as rotas relacionadas ao CRUD de clientes.
- **Rotas principais:**
 - **POST /clientes:** Rota protegida por JWT para registrar um novo cliente. A lógica de registro está no `ClientsController.registerClient`.
 - **PUT /clientes/:** Rota protegida para atualizar as informações de um cliente específico, identificado pelo UUID. A lógica está no `ClientsController.updateClient`.

- **GET /clientes:** Rota para buscar todos os clientes registrados. A lógica está no `ClientsController.findManyClient`.
- **DELETE /clientes/:** Rota protegida para deletar um cliente específico pelo UUID, com a lógica no `ClientsController.deleteClient`.

Autenticação nas Rotas

As rotas protegidas por **authenticateToken** garantem que apenas usuários autenticados com um token válido possam criar, atualizar ou deletar clientes. Isso mantém a integridade e segurança dos dados dos clientes.

CAPÍTULO 8: IMPLEMENTANDO O CONTROLADOR DE AUTENTICAÇÃO

Neste capítulo, vamos implementar o controlador responsável pelas funcionalidades de **registro**, **login** e **consulta de usuários** na nossa API. Este controlador usa o **Prisma** para interagir com o banco de dados e **JWT** para gerar tokens de autenticação.

Vamos criar um async function para registrar um usuario novo:

```
js AuthController.js M X
src > app > controllers > auth > js AuthController.js > registrar > user > data
1  import { PrismaClient } from "@prisma/client";
2  import jwt from "jsonwebtoken";
3  import bcrypt from "bcrypt";
4  import { hashPassword } from "../../repositories/passwordUtils.js";
5
6  const prisma = new PrismaClient();
7
8  Tabnine | Edit | Test | Explain | Document | Ask
9  async function registrar(req, res) {
10     const { email, login, password, name, cpf_cnpj, celular } = req.body;
11     const hashedPassword = await hashPassword(password);
12
13     try {
14         const user = await prisma.user.create({
15             data: {
16                 email,
17                 login,
18                 password: hashedPassword,
19                 name,
20                 cpf_cnpj,
21                 celular,
22             },
23         });
24         res.status(201).json(user);
25     } catch (error) {
26         res.status(400).json({ message: error.message });
27     }
28 }
```

Vamos criar outra async function para realizar o login:

```
async function login(req, res) {
  try {
    const { email, password } = req.body;
    const user = await prisma.user.findUnique({
      where: {
        email,
      },
    });

    if (!user) {
      return res.status(401).json({ message: "E-mail ou senha incorretos!" });
    }

    const isPasswordValid = await bcrypt.compare(password, user.password);

    if (!isPasswordValid) {
      return res.status(401).json({ message: "E-mail ou senha incorretos!" });
    }

    const token = jwt.sign({ uuid: user.uuid }, process.env.SECRET, {
      expiresIn: "1h",
    });

    res.json({ token, user_uuid: user.uuid });
  } catch (error) {
    res.status(400).json({ message: error.message });
  }
}
```

Vamos criar um async function para trazer apenas 1 usuario pelo UUID:

```
async function umUsuario(req, res) {
  try {
    const user = await prisma.user.findUnique({
      where: {
        uuid: req.params.userUuid,
      },
      include: {
        clients: true,
      },
    });

    if (!user) {
      return res.status(404).json({ message: "Usuário não encontrado" });
    }

    res.json(user);
  } catch (error) {
    res.status(500).json({ message: error.message });
  }
}
```

E por ultimo vamos criar outra async function para controller de autenticação, vamos chamar de ME

```
async function me(req, res) {
  try {
    const user_uuid = req.user.uuid;

    if (!user_uuid) {
      return res
        .status(400)
        .json({ message: "O ID do usuário está faltando no token" });
    }

    const user = await prisma.user.findUnique({
      where: {
        uuid: user_uuid,
      },
      include: {
        client: true,
      },
    });

    if (!user) {
      return res.status(404).json({ message: "Usuário não encontrado" });
    }

    res.json(user);
  } catch (error) {
    res.status(500).json({ message: error.message });
  }
}
```

Agora para finalizar. Vamos exportar as funções para poder utilizar nas Routes

```
export default {
  registrar,
  login,
  umUsuario,
  me,
};
```

Explicação:

1. Importações:

- PrismaClient é usado para conectar e realizar operações no banco de dados.
- jsonwebtoken (JWT) é utilizado para gerar tokens de autenticação.
- bcrypt é utilizado para comparar senhas criptografadas e para o processo de login.

- hashPassword é uma função utilitária que criptografa a senha do usuário antes de salvá-la.

2. registrar(req, res):

- Esta função é responsável por registrar um novo usuário. Ela recebe os dados do formulário (e-mail, login, senha, nome, CPF/CNPJ e celular), criptografa a senha com bcrypt e salva o usuário no banco de dados usando o Prisma. Se a operação for bem-sucedida, retorna o usuário recém-criado com um código de status **201** (Criado). Caso haja algum erro, retorna uma mensagem de erro com um status **400** (Requisição inválida).

3. login(req, res):

- Esta função autentica um usuário. Ela verifica se o e-mail existe no banco de dados e, se existir, compara a senha inserida com a senha criptografada armazenada. Se a senha for válida, é gerado um **token JWT** com validade de 1 hora, que é retornado ao usuário. Se a autenticação falhar, uma mensagem de erro com status **401** (Não autorizado) é enviada.

4. umUsuario(req, res):

- Esta função busca os detalhes de um usuário específico pelo seu UUID. Além dos dados do usuário, os clientes associados ao usuário também são incluídos na resposta através da associação definida no Prisma. Se o usuário não for encontrado, é retornado um status **404** (Não encontrado).

5. me(req, res):

- Esta rota é protegida por autenticação JWT e retorna os dados do usuário autenticado, usando o UUID presente no token JWT. Se o token estiver ausente ou o UUID não for encontrado, uma mensagem de erro é retornada. Se o usuário não existir, um status **404** é enviado.

Considerações Importantes:

- **Segurança:**
Todas as funções de autenticação lidam com a segurança das senhas usando bcrypt para garantir que senhas nunca sejam armazenadas em texto plano no banco de dados. O uso de JWT para autenticação oferece uma camada de segurança extra, permitindo que apenas usuários com tokens válidos acessem certas rotas.
- **Tratamento de Erros:**
Em todos os métodos, foram implementados blocos try...catch para capturar e lidar com erros que possam ocorrer durante as operações de banco de dados ou validações.

CAPÍTULO 9: IMPLEMENTANDO O CONTROLADOR DE CLIENTES

Neste capítulo, vamos implementar o controlador responsável pelas operações relacionadas aos **clientes**. Este controlador permite registrar, atualizar, deletar e consultar clientes associados a um usuário autenticado.

Função para criar um cliente:

```
JS ClientsController.js M X
src > app > controllers > client > JS ClientsController.js > ...
1  import { PrismaClient } from "@prisma/client";
2
3  const prisma = new PrismaClient();
4
5  // registra o cliente
6  Tabnine | Edit | Test | Explain | Document | Ask
7  async function registerClient(req, res) {
8    try {
9      const { name, cpf, celular } = req.body;
10
11      if (!req.user) {
12        return res.status(401).json({ error: "Usuário não autenticado" });
13      }
14
15      const userUuid = req.user.uuid;
16
17      if (!name || !cpf || !celular) {
18        return res.status(400).json({ error: "Todos os dados são obrigatórios" });
19      }
20
21      const client = await prisma.client.create({
22        data: {
23          name,
24          cpf,
25          celular,
26          userUuid,
27        },
28      });
29
30      res.status(201).json({
31        message: "Cliente criado com sucesso",
32        status: 201,
33        success: true,
34        data: client,
35      });
36    } catch (error) {
37      res.status(400).json({
38        message: "Erro ao criar cliente",
39        status: 400,
40        success: false,
41        error: error.message,
42      });
43    }
44  }
```

Função para deletar um cliente:

```
async function deleteClient(req, res) {
  try {
    const uuid = req.params.uuid;

    if (!req.user) {
      return res.status(401).json({ error: "Usuário não autenticado" });
    }

    const client = await prisma.client.findUnique({
      where: {
        uuid: uuid,
      },
    });

    if (!client) {
      return res.status(404).json({ error: "Cliente não encontrado" });
    }

    if (client.userUuid !== req.user.uuid) {
      return res
        .status(403)
        .json({ error: "Você não tem permissão para excluir este cliente" });
    }

    await prisma.client.delete({
      where: {
        uuid: uuid,
      },
    });

    return res.status(200).json({
      message: "Cliente deletado com sucesso",
      status: 200,
      success: true,
      client: client,
    });
  } catch (error) {
    res.status(400).json({
      message: "Erro ao deletar cliente",
      status: 400,
      success: false,
      error: error.message,
    });
  }
}
```


Vamos criar outra função para editar o cliente, essa função é a mais complexa:

```
async function updateClient(req, res) {  
  try {  
    const uuid = req.params.uuid;  
    const { name, cpf, celular } = req.body;  
  
    if (!req.user) {  
      return res.status(401).json({ error: "Usuário não autenticado" });  
    }  
    const client = await prisma.client.findUnique({  
      where: {  
        uuid: uuid,  
      },  
    });  
    if (!client) {  
      return res.status(404).json({ error: "Cliente não encontrado" });  
    }  
    if (client.userUuid !== req.user.uuid) {  
      return res  
        .status(403)  
        .json({ error: "Você não tem permissão para atualizar este cliente" });  
    }  
    const updatedClient = await prisma.client.update({  
      where: {  
        uuid: uuid,  
      },  
      data: {  
        name,  
        cpf,  
        celular,  
      },  
    });  
    res.status(200).json({  
      message: "Cliente atualizado com sucesso",  
      status: 200,  
      success: true,  
      data: updatedClient,  
    });  
  } catch (error) {  
    res.status(400).json({  
      message: "Erro ao atualizar cliente",  
      status: 400,  
      success: false,  
      error: error.message,  
    });  
  }  
}
```

E por ultimo a função de buscar o cliente:

```

async function findManyClient(req, res) {
  try {
    if (!req.user) {
      return res.status(401).json({ error: "Usuário não autenticado" });
    }

    const userUuid = req.user.uuid; // Obtenha o UUID do usuário autenticado
    const clients = await prisma.client.findMany({
      where: {
        userUuid: userUuid, // Filtra clientes apenas do usuário autenticado
      },
    });

    return res.status(200).json(clients);
  } catch (error) {
    return res.status(500).json({ error: error.message });
  }
}

export default {
  registerClient,
  findManyClient,
  deleteClient,
  updateClient,
};

```

Explicação:

1. Importação do Prisma:

O PrismaClient é importado para interagir com o banco de dados. Ele permite realizar operações de CRUD nas tabelas definidas no schema.

2. registerClient(req, res):

- Esta função registra um novo cliente associado ao usuário autenticado. Ela verifica se o usuário está autenticado e se todos os campos necessários (nome, CPF e celular) estão presentes.
- Um novo cliente é criado no banco de dados e, se bem-sucedido, uma resposta JSON com os detalhes do cliente e um status **201** é retornada. Caso ocorra um erro, um status **400** é enviado com a mensagem de erro.

3. deleteClient(req, res):

- Esta função permite que um usuário autenticado exclua um cliente. Ela verifica se o cliente existe e se o UUID do cliente corresponde ao UUID do usuário autenticado.
- Se a validação passar, o cliente é deletado do banco de dados. Se a operação for bem-sucedida, um status **200** é retornado. Em caso de erro, um status **400** é enviado.

4. **updateClient(req, res):**

- Similar à função de deleção, esta função atualiza os dados de um cliente existente. Ela verifica se o usuário está autenticado e se o cliente existe, além de confirmar que o usuário tem permissão para atualizar aquele cliente específico.
- Se tudo estiver correto, os dados do cliente são atualizados no banco de dados e um status **200** é retornado com os dados atualizados. Em caso de erro, um status **400** é enviado.

5. **findManyClient(req, res):**

- Esta função retorna todos os clientes associados ao usuário autenticado. Ela garante que o usuário esteja autenticado antes de buscar os clientes.
- O método findMany do Prisma é utilizado para buscar todos os clientes relacionados ao UUID do usuário. Um status **200** é retornado com os dados dos clientes. Se houver um erro, um status **500** é enviado.

Considerações Importantes:

- **Autenticação:**

Todas as funções (exceto findManyClient) verificam se o usuário está autenticado antes de permitir operações que alterem o banco de dados.

- **Tratamento de Erros:**

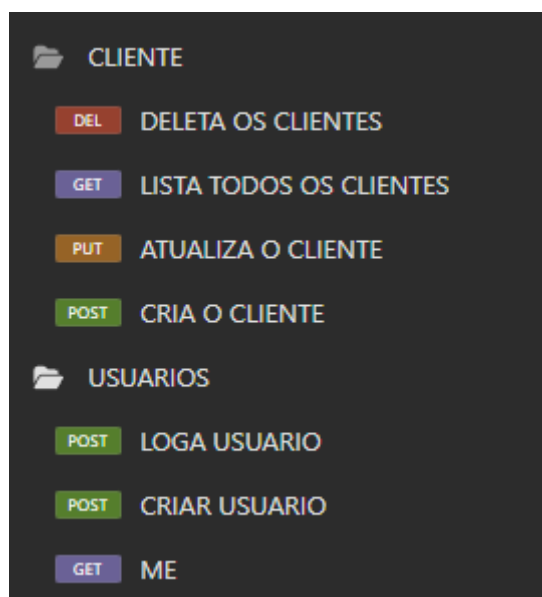
Cada função inclui tratamento de erros robusto, enviando respostas apropriadas para o cliente em caso de falhas ou dados ausentes.

Esse controlador fornece uma interface robusta para gerenciar clientes, garantindo que apenas usuários autenticados possam modificar os dados associados a eles.

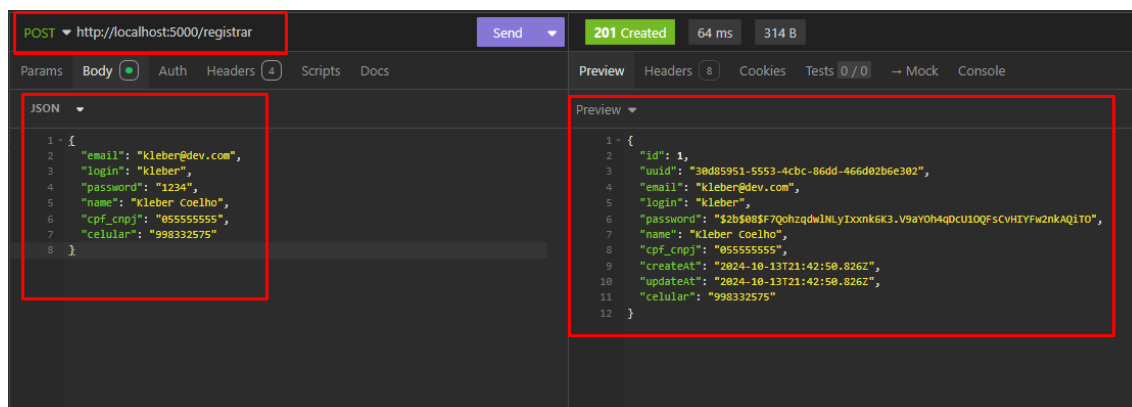
CAPÍTULO 10: FAZENDO REQUISIÇÕES PELO INSOMNIA

O Insomnia é uma ferramenta popular para testar e desenvolver APIs. Ele oferece uma interface simples e organizada para criar requisições HTTP, facilitando o processo de teste de endpoints da API.

Gosto de criar pastas para separar as informações, nesse caso, crie uma pasta referente ao USUARIO e outra referente aos CLIENTES

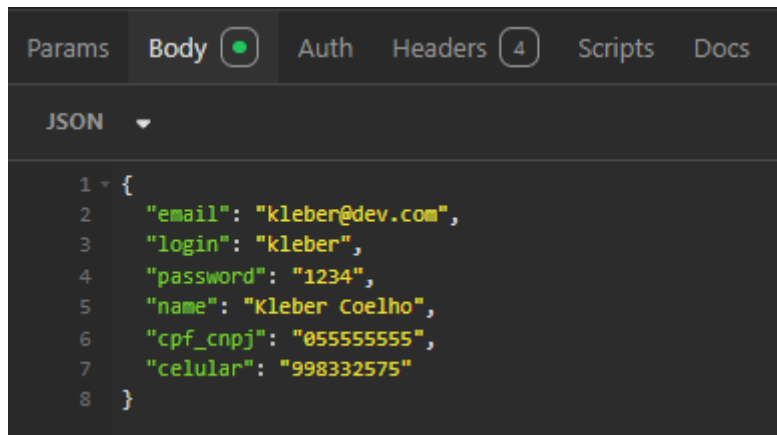


Vamos começar pelo registro do Usuario em CRIAR USUARIO:

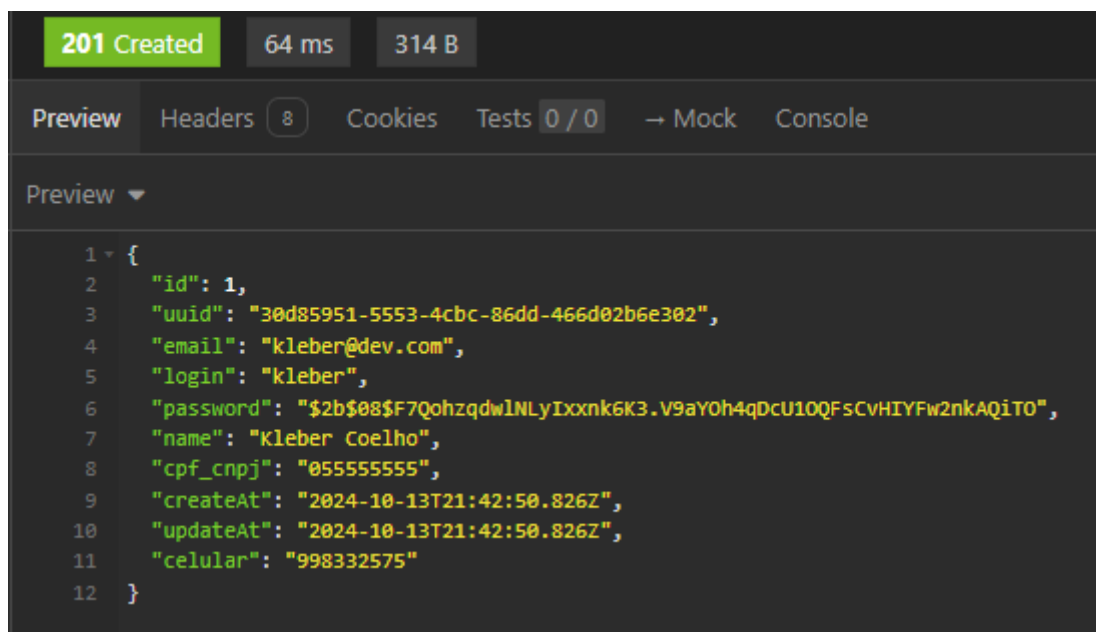


Definindo método POST com o link da API e o endpoint <http://localhost:5000/regar>

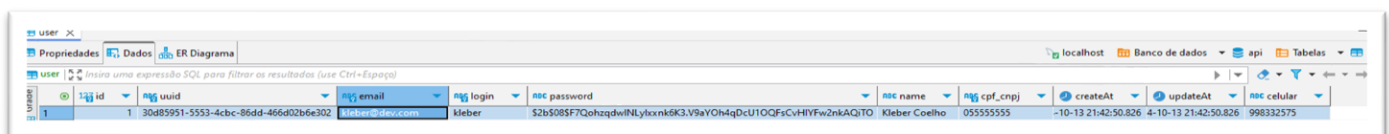
Passando no body da requisição o json com as informações que utilizamos para criar o controller



Ao clicar em SEND para enviar para o banco, será criado o usuario

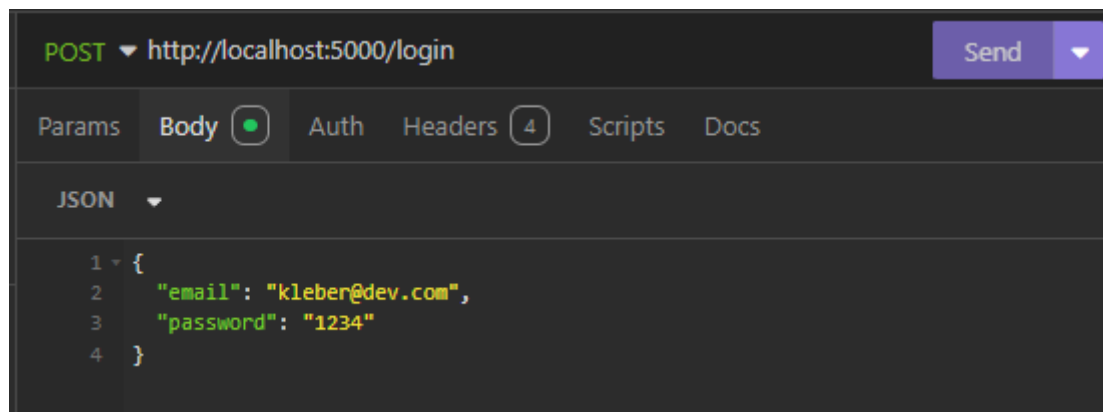


Ao visualizar o banco de dados teremos as mesmas informações:

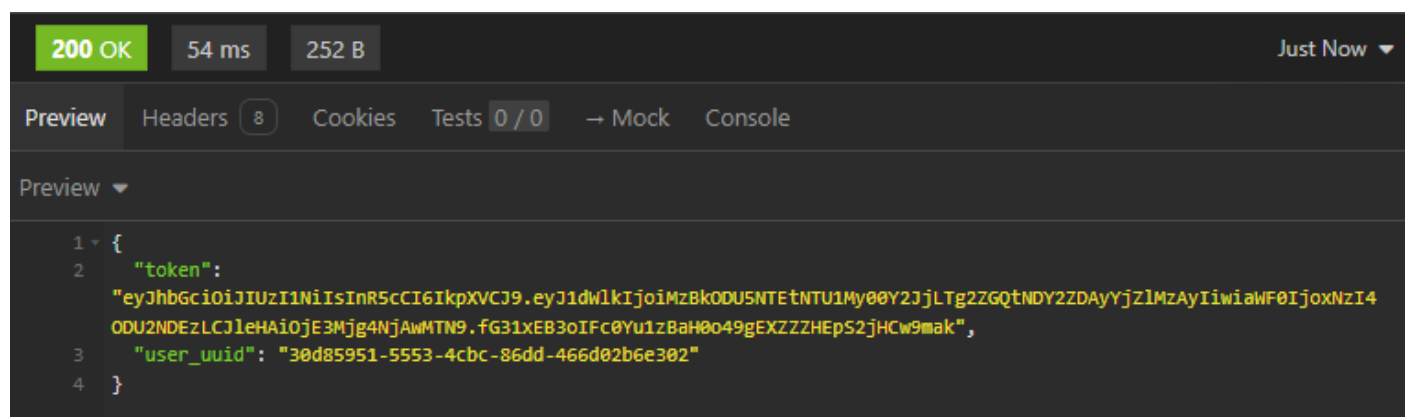


Agora vamos realizar o LOGIN em LOGA USUARIO:

Vamos definir o link da API: <http://localhost:5000/login> com o metodo POST e passar as seguintes informações no corpo da requisição

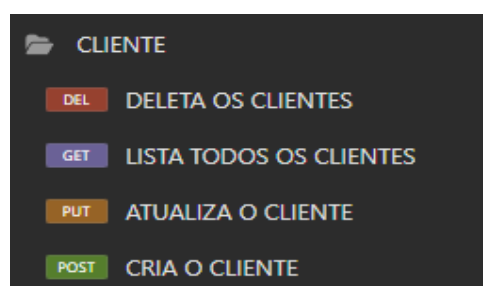


A resposta vai ser o token de autenticação gerado com o retorno do UUID do usuario autenticado

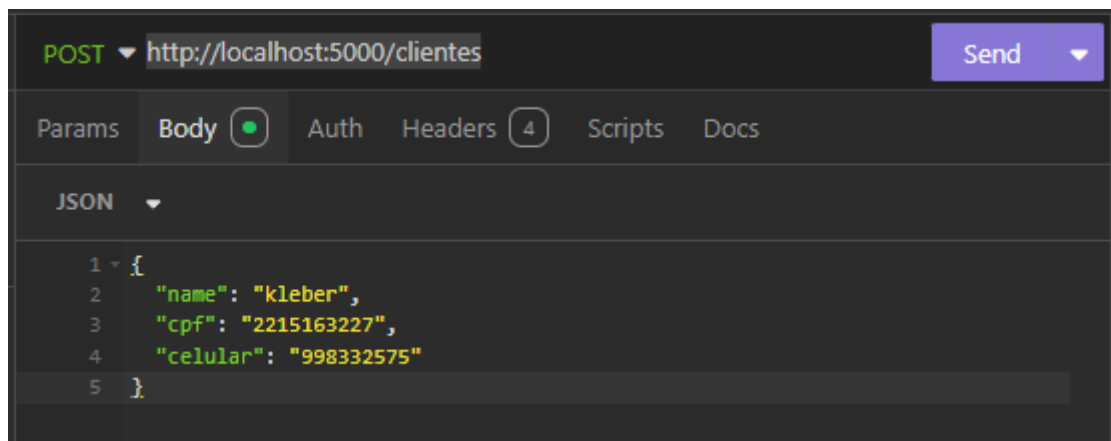


Lembrando que agora para criar os CLIENTES vamos precisar passar o TOKEN no Header da requisição, então ao fazer o login, copie o token.

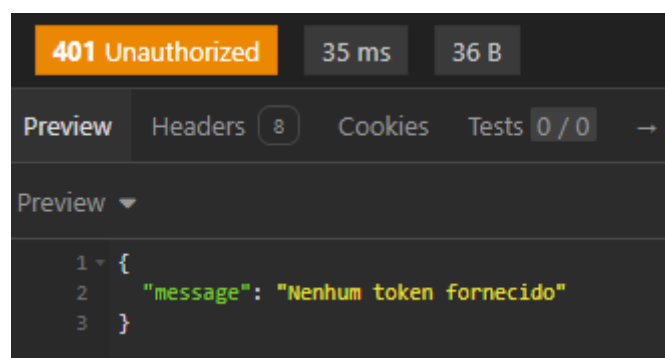
Vamos criar o cliente:



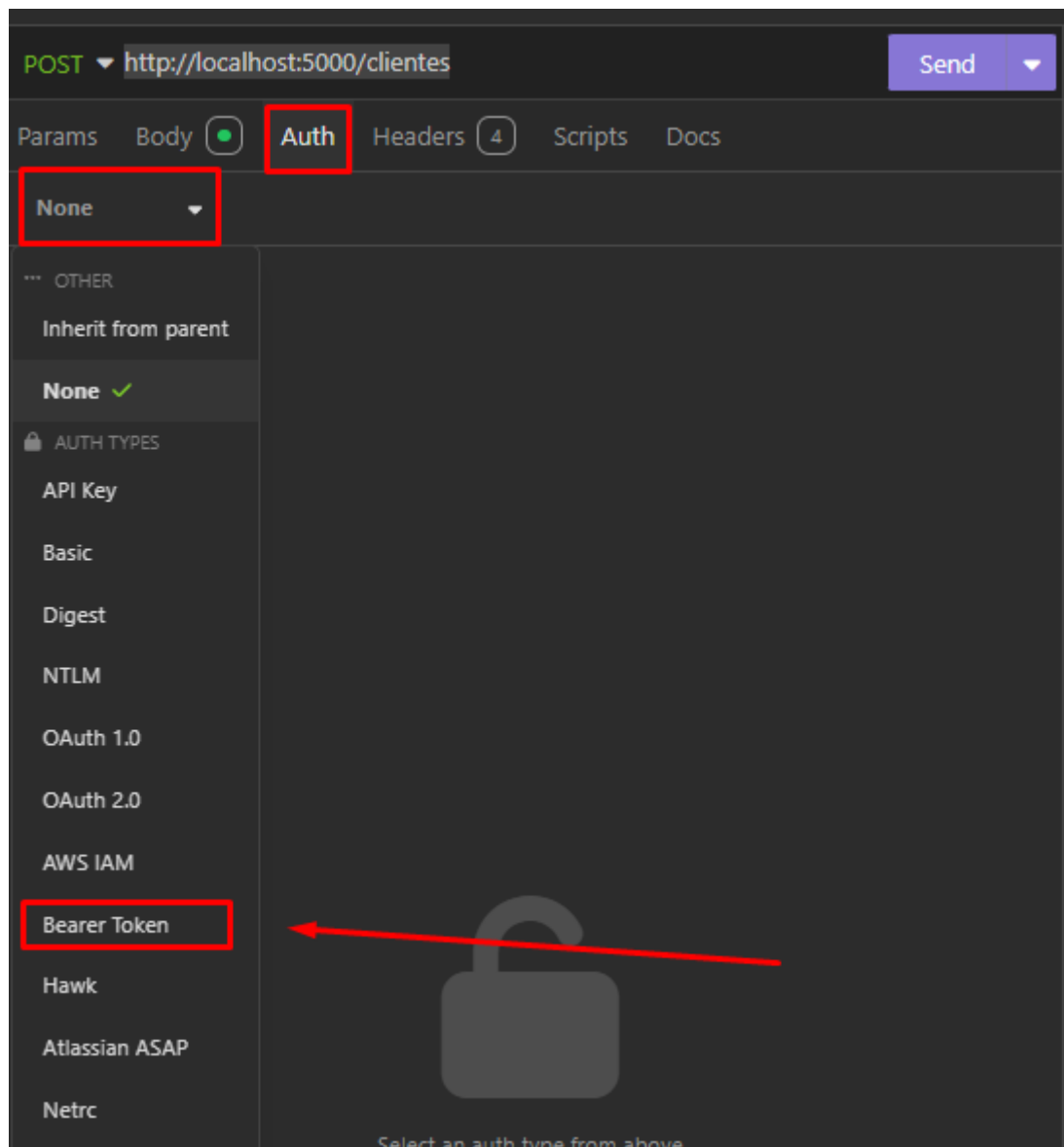
Em CRIA O CLIENTE vamos passar no body um json com as informações e definindo o método POST com o URL: <http://localhost:5000/clientes>



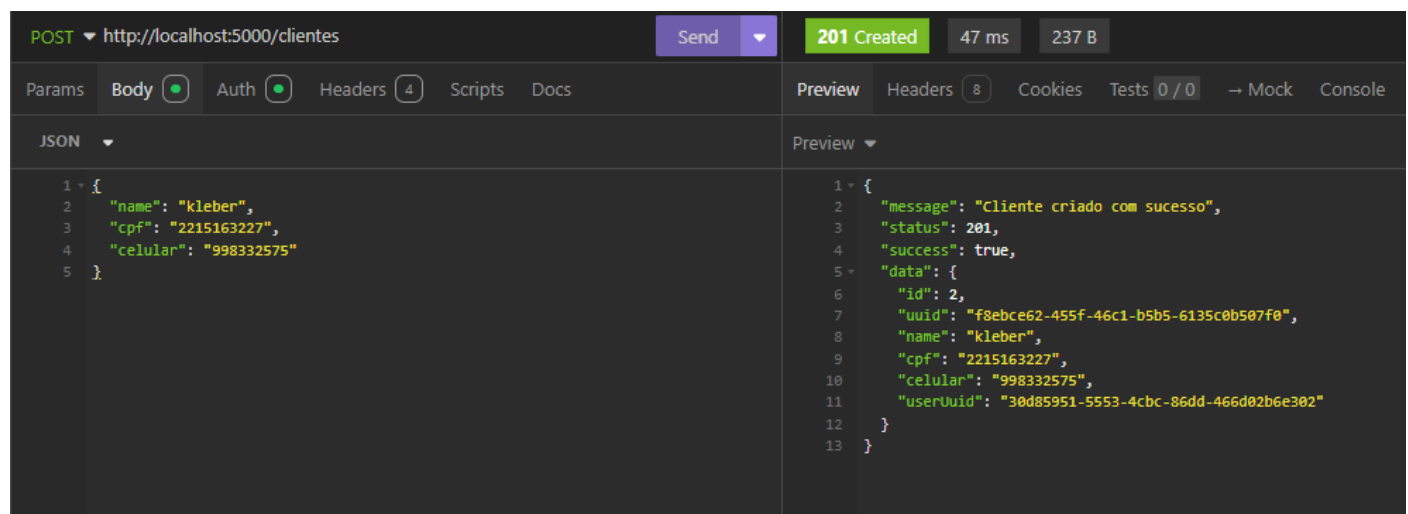
Ao clicar em SEND vamos ter esse retorno:



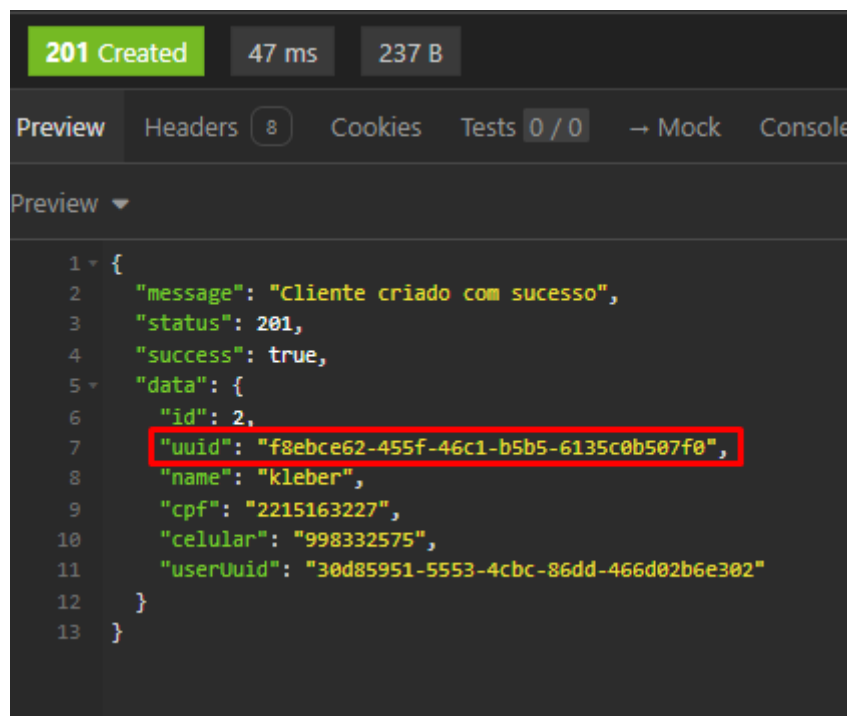
Pois não definimos o TOKEN no HEADER, então vamos clicar em AUTH > NONE e definir o Bearer Token e colar o Token gerado no login.



Colando o Token no local correto e clicando em SEND vamos ter essa resposta:



Vamos utilizar o método PUT para realizar a edição do cliente, passando o link <http://localhost:5000/clientes/UUID DO CLIENTE>, conseguimos o UUID na hora de criar ele



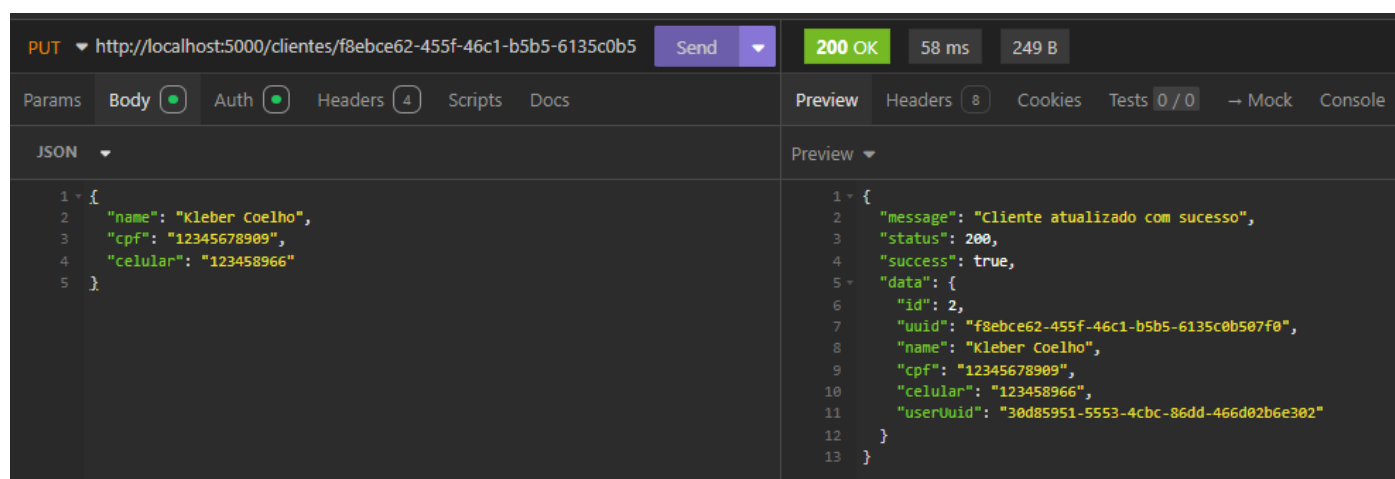
```

201 Created 47 ms 237 B
Preview Headers 8 Cookies Tests 0 / 0 → Mock Console
Preview
1 {
2   "message": "Cliente criado com sucesso",
3   "status": 201,
4   "success": true,
5   "data": {
6     "id": 2,
7     "uuid": "f8ebce62-455f-46c1-b5b5-6135c0b507f0",
8     "name": "kleber",
9     "cpf": "2215163227",
10    "celular": "998332575",
11    "userUuid": "30d85951-5553-4cbc-86dd-466d02b6e302"
12  }
13 }
  
```

No meu caso ficaria assim:

<http://localhost:5000/clientes/f8ebce62-455f-46c1-b5b5-6135c0b507f0>

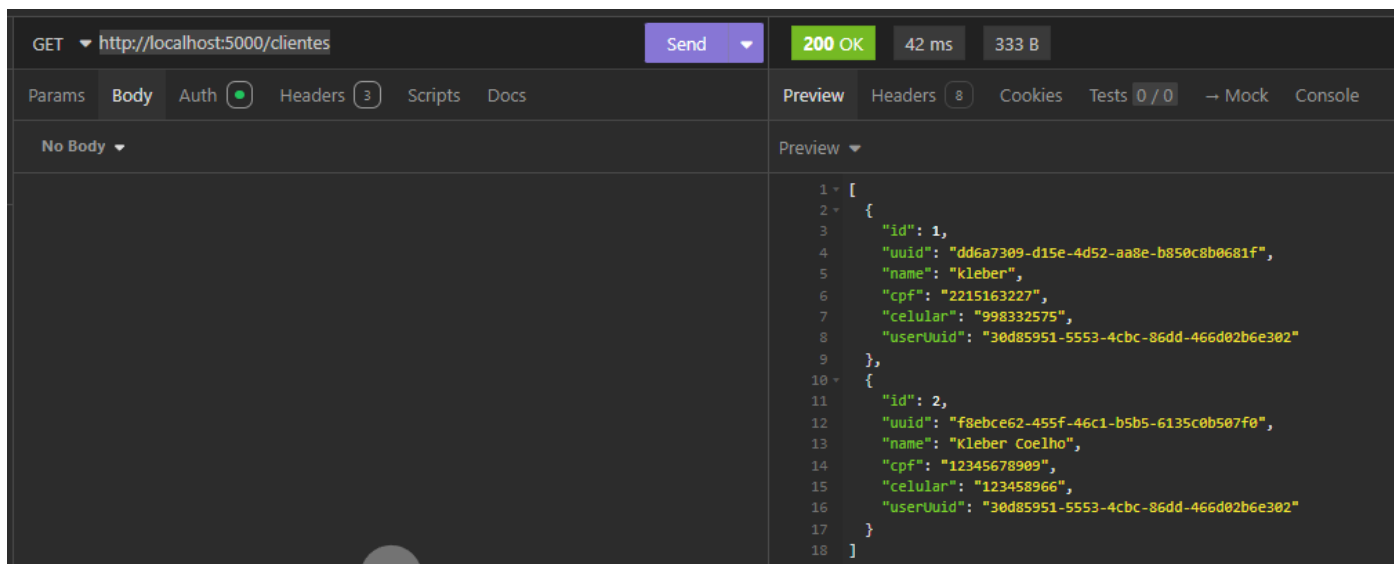
E no corpo da requisição vamos passar as informações que vamos editar juntamente com o TOKEN gerado ao realizar o login



```

PUT http://localhost:5000/clientes/f8ebce62-455f-46c1-b5b5-6135c0b5 Send 200 OK 58 ms 249 B
Params Body Auth Headers 4 Scripts Docs Preview Headers 8 Cookies Tests 0 / 0 → Mock Console
JSON Preview
1 {
2   "name": "Kleber Coelho",
3   "cpf": "12345678909",
4   "celular": "123458966"
5 }
1 {
2   "message": "Cliente atualizado com sucesso",
3   "status": 200,
4   "success": true,
5   "data": {
6     "id": 2,
7     "uuid": "f8ebce62-455f-46c1-b5b5-6135c0b507f0",
8     "name": "Kleber Coelho",
9     "cpf": "12345678909",
10    "celular": "123458966",
11    "userUuid": "30d85951-5553-4cbc-86dd-466d02b6e302"
12  }
13 }
  
```

Vamos verificar como que funciona o LISTA TODOS OS CLIENTES, passando o metodo GET com o link: <http://localhost:5000/clientes> , nesse caso vamos passar apenas o TOKEN no Header e vamos verificar o retorno



Retornando 2 clientes criado para esse usuario.

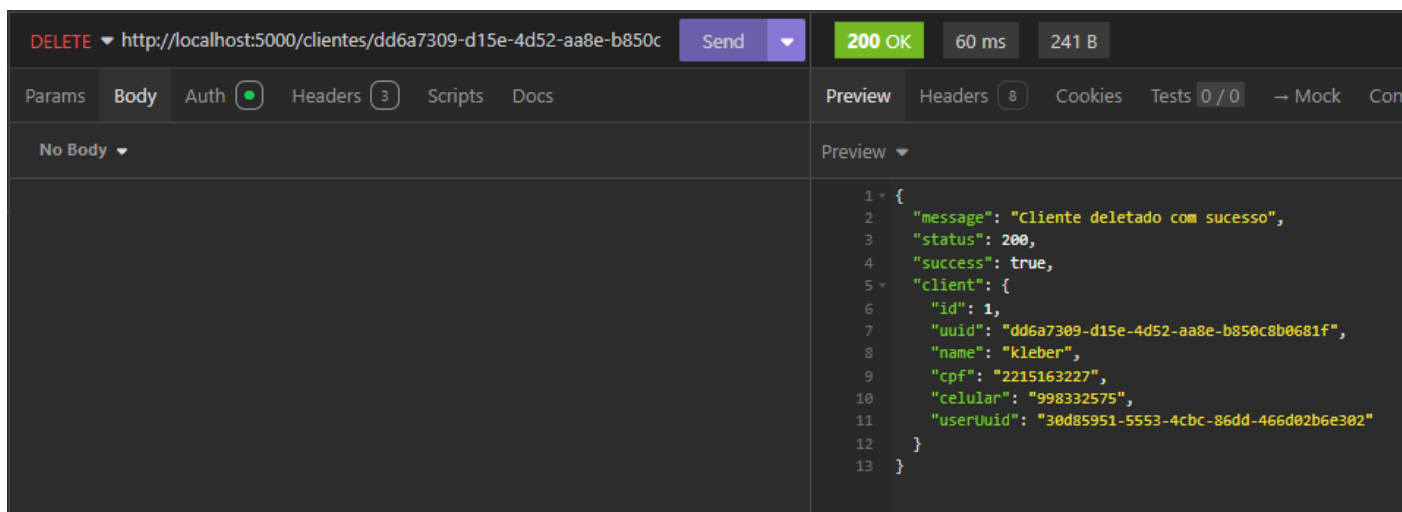
Vamos verificar como que funciona o DELETA OS CLIENTES, passando o método DELETE, no seguinte link:

<http://localhost:5000/clientes/>

Lembrando de atualizar depois do clientes/ o UUID do cliente, no meu caso ficaria assim:

<http://localhost:5000/clientes/dd6a7309-d15e-4d52-aa8e-b850c8b0681f>

Ao clicar em SEND vamos ter essa resposta:



Por último vamos utilizar a rota autenticada, em rota ME, passando apenas o Token no Header com isso vamos ter essa resposta:

GET ▼ http://localhost:5000/me

Send

200 OK

57 ms

819 B

Params

Body

Auth ●

Headers 3

Scripts

Docs

Preview

Headers 8

Cookies

Tests 0 / 0

→ Mock

Console

Bearer Token ▼

ENABLED ☒

TOKEN eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1dWlkjoiMzBkODU5NTetNTU1My00

PREFIX

Preview ▼

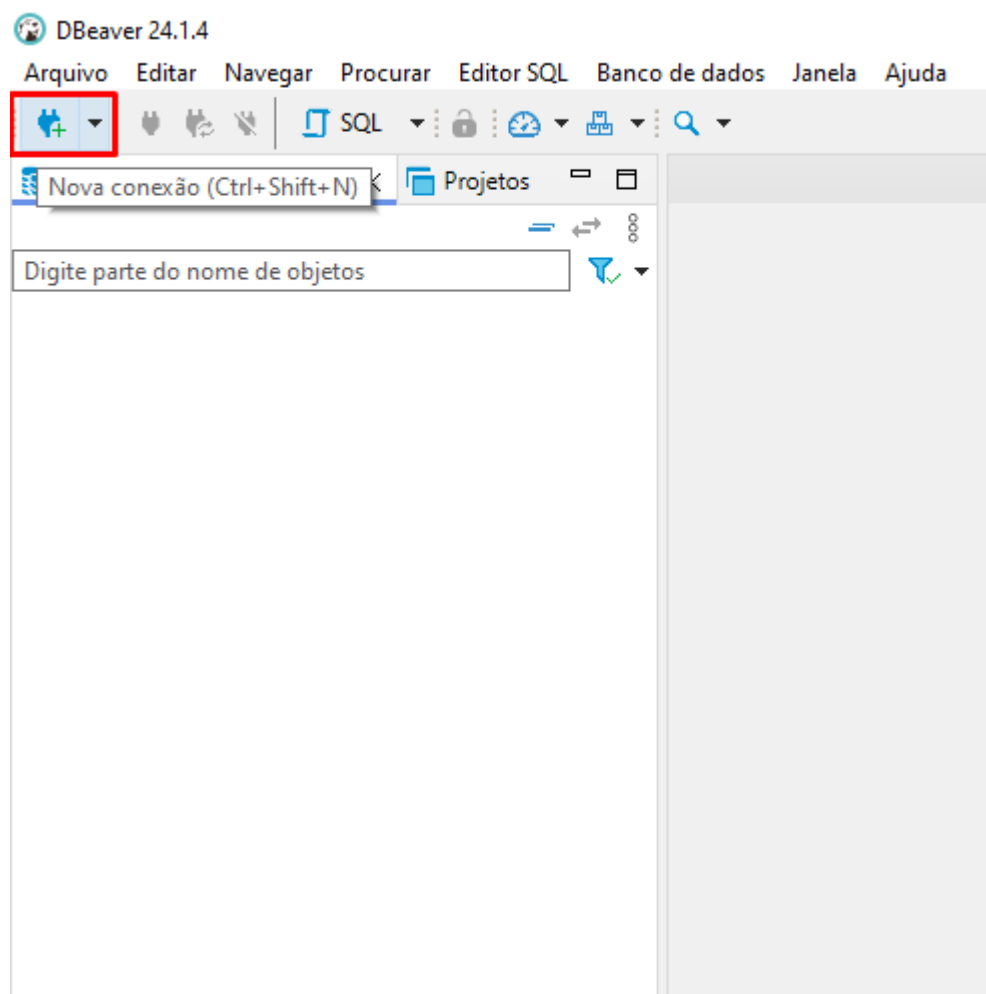
```

1 {
2   "id": 1,
3   "uuid": "30d85951-5553-4cbc-86dd-466d02b6e302",
4   "email": "kleber@dev.com",
5   "login": "kleber",
6   "password": "$2b$08$F7Qohzqdw1NLyIxxnk6K3.V9aY0h4qDcU10QFsCvHIYFw2nkAQiT0",
7   "name": "Kleber Coelho",
8   "cpf_cnpj": "055555555",
9   "createAt": "2024-10-13T21:42:50.826Z",
10  "updateAt": "2024-10-13T21:42:50.826Z",
11  "celular": "998332575",
12  "client": [
13    {
14      "id": 2,
15      "uuid": "f8ebce62-455f-46c1-b5b5-6135c0b507f0",
16      "name": "Kleber Coelho",
17      "cpf": "12345678909",
18      "celular": "123458966",
19      "userUuid": "30d85951-5553-4cbc-86dd-466d02b6e302"
20    },
21    {
22      "id": 3,
23      "uuid": "43b4cded-d2a5-4b75-9356-61bbb59c5a74",
24      "name": "kleber",
25      "cpf": "2215163227",
26      "celular": "998332575",
27      "userUuid": "30d85951-5553-4cbc-86dd-466d02b6e302"
28    },
29    {
30      "id": 4,
31      "uuid": "1501ef9a-20ec-4031-8eef-529fc10385a3",
32      "name": "kleber",
33      "cpf": "2215163227",
34      "celular": "998332575",
35      "userUuid": "30d85951-5553-4cbc-86dd-466d02b6e302"
36    }
37  ]
38 }
```

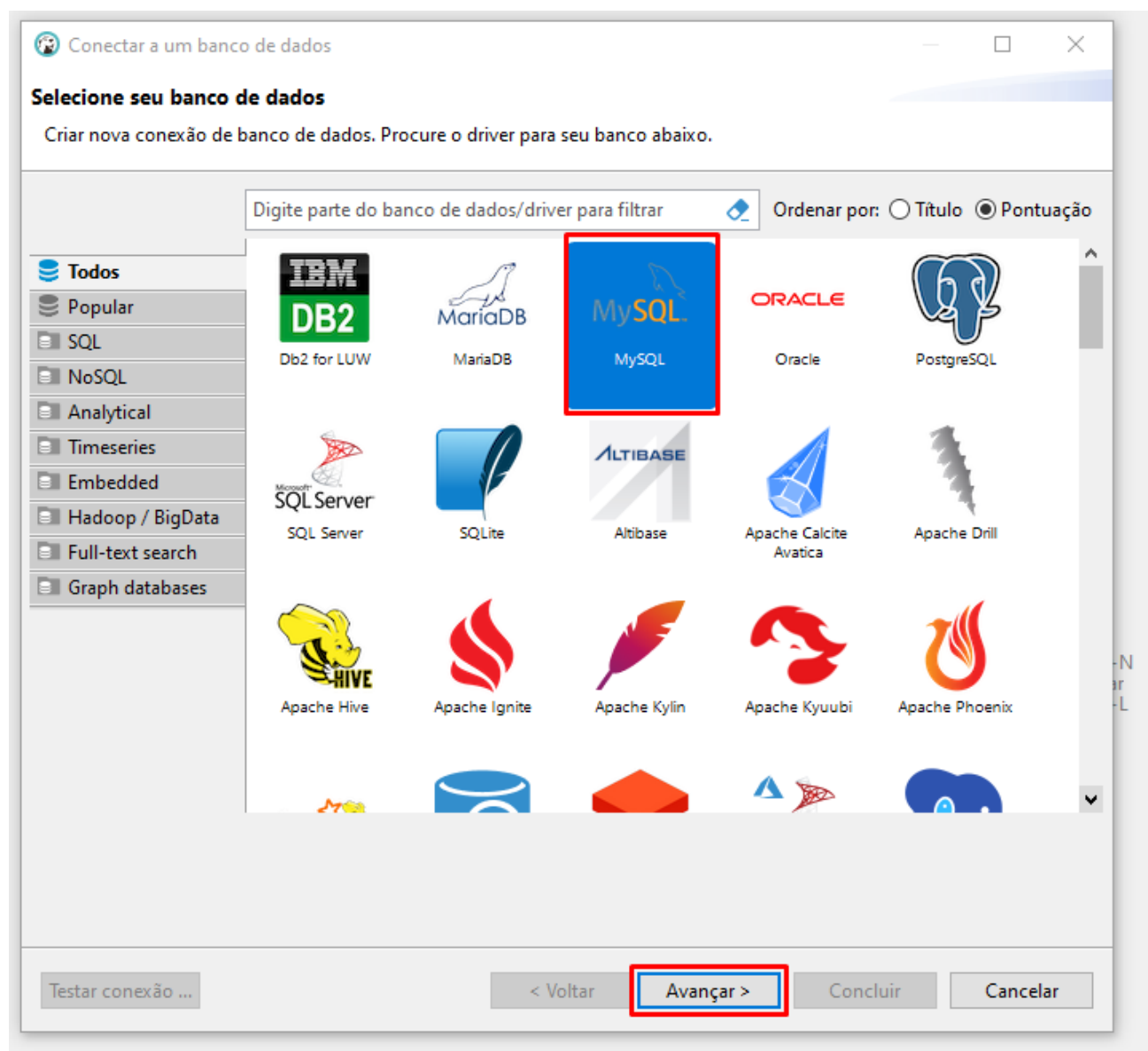
CAPÍTULO 11: BONUS! CONFIGURANDO O BANCO DE DADOS

Para usar o MySQL no nosso projeto vamos precisar instalar o MySQL no pc juntamente com o DBeaver para poder criar o banco em Local Host, não irei entrar em detalhes de como instalar o MySQL e DBeaver pois tem muito material explicando.

Ao iniciar o DBeaver vamos clicar em Nova Conexão:



Clicar em MySQL e Avançar:



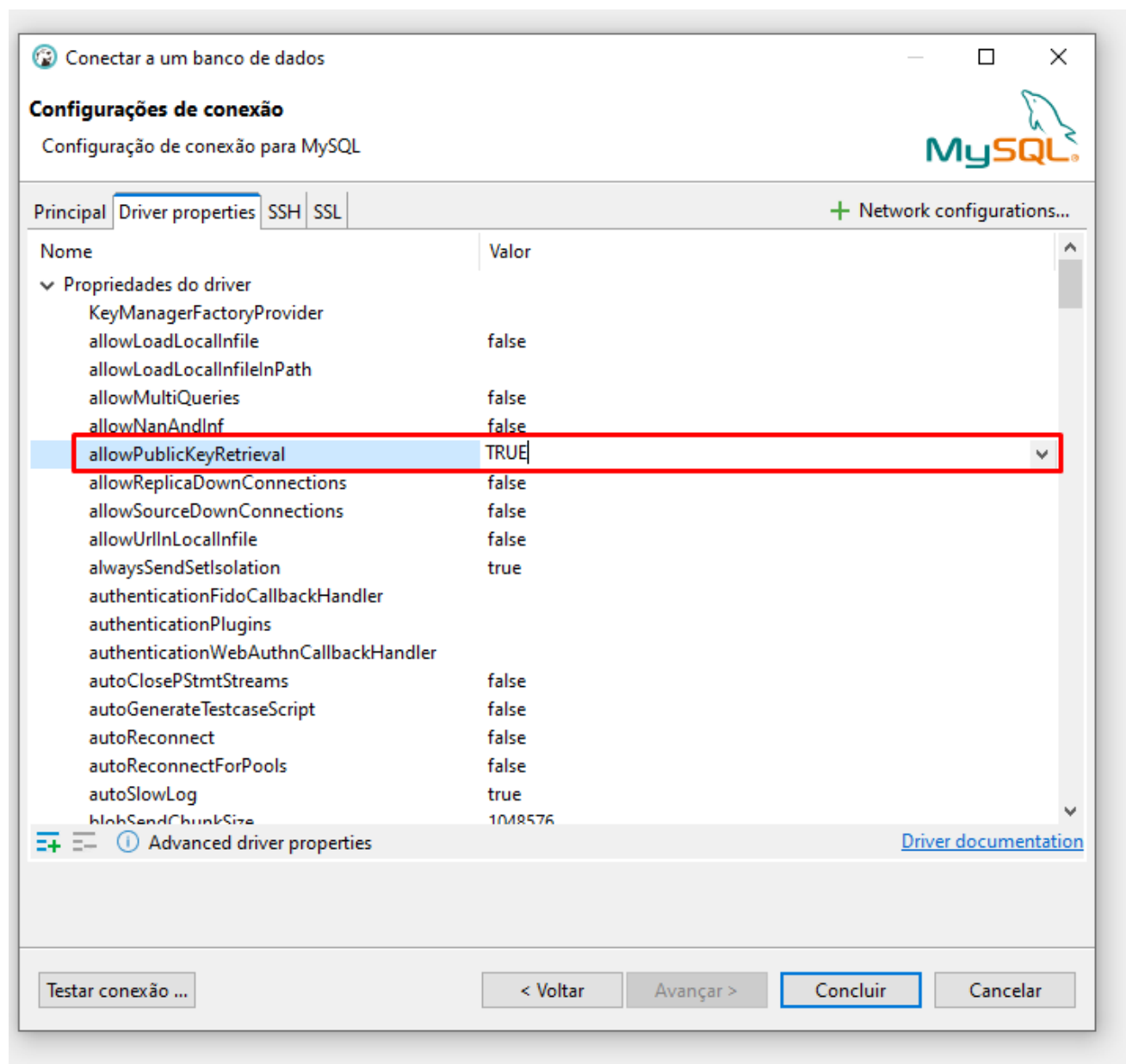
Informar o nome do banco de dados, no caso escolhi API e informar o Nome de Usuário e Senha definido quando foi instalado o MySql no seu pc.

The image shows the 'Conectar a um banco de dados' (Connect to a database) window in the MySQL Connector/ODBC driver configuration. The window is titled 'Configurações de conexão' (Connection settings) and is for a MySQL connection. The 'Principal' (Main) tab is selected, showing the following fields:

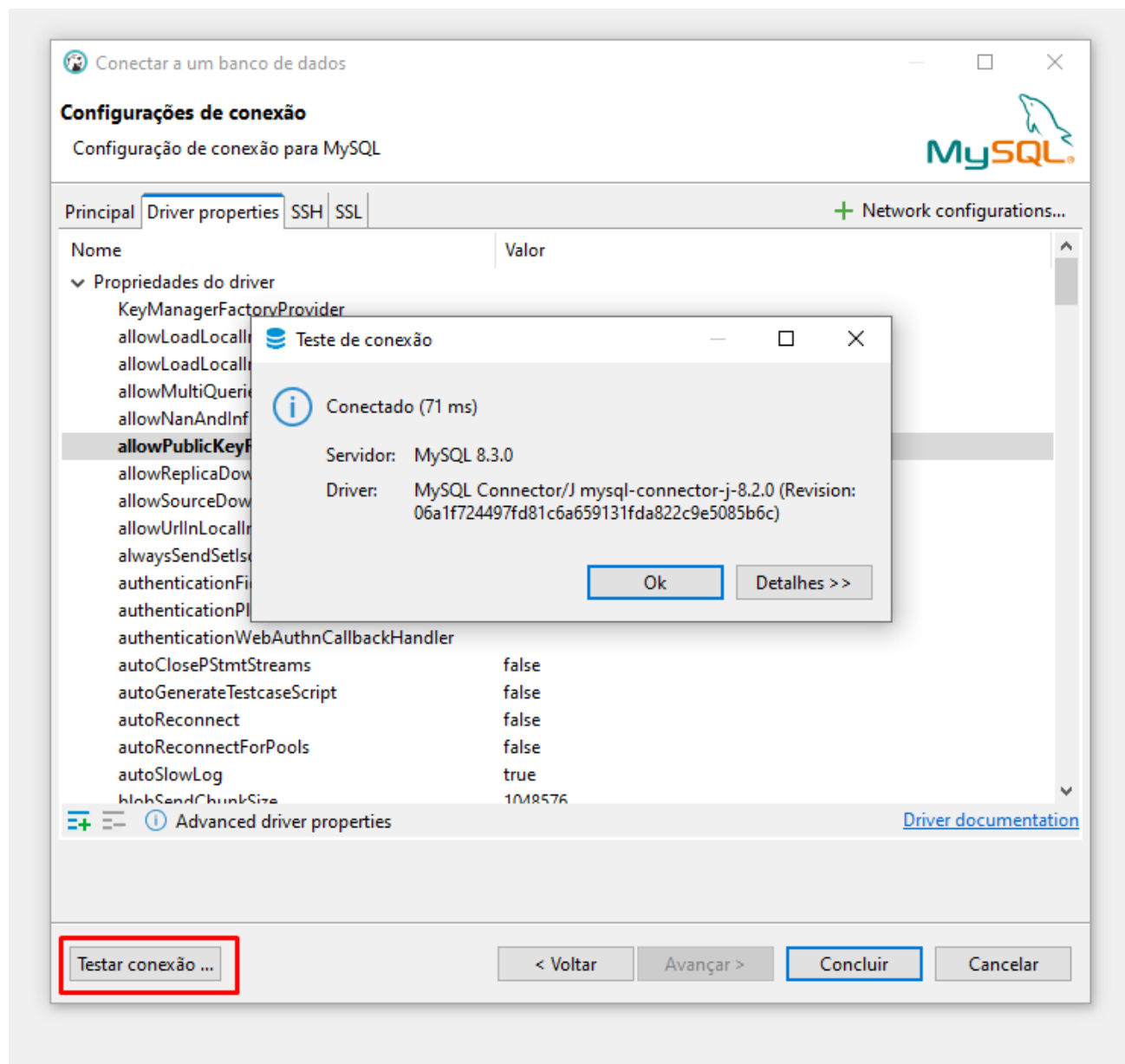
- Servidor** (Server):
 - Conecte usando: ☒ Host ☐ URL
 - URL:
 - Server Host: Port:
 - Database:** (This field is highlighted with a red box)
- Autenticação (Database Native)** (Authentication):
 - Nome de usuário:** (This field is highlighted with a red box)
 - Senha:** (This field is highlighted with a red box)
 - ☒ Salvar senha (Save password)
- Advanced**:
 - Server Time Zone:
 - Local Client:

At the bottom of the window, there are links for 'Connection variables information', 'Database documentation', and 'Detalhes da conexão (nome, tipo, ...)' (Connection details (name, type, ...)). The driver name is set to 'MySQL'. At the bottom right, there are buttons for 'Configurações de driver' (Driver settings) and 'Driver license'. At the bottom left, there is a 'Testar conexão ...' (Test connection ...) button. At the bottom center, there are navigation buttons: '< Voltar' (Back), 'Avançar >' (Next), 'Concluir' (Finish), and 'Cancelar' (Cancel). The 'Concluir' button is highlighted with a blue border.

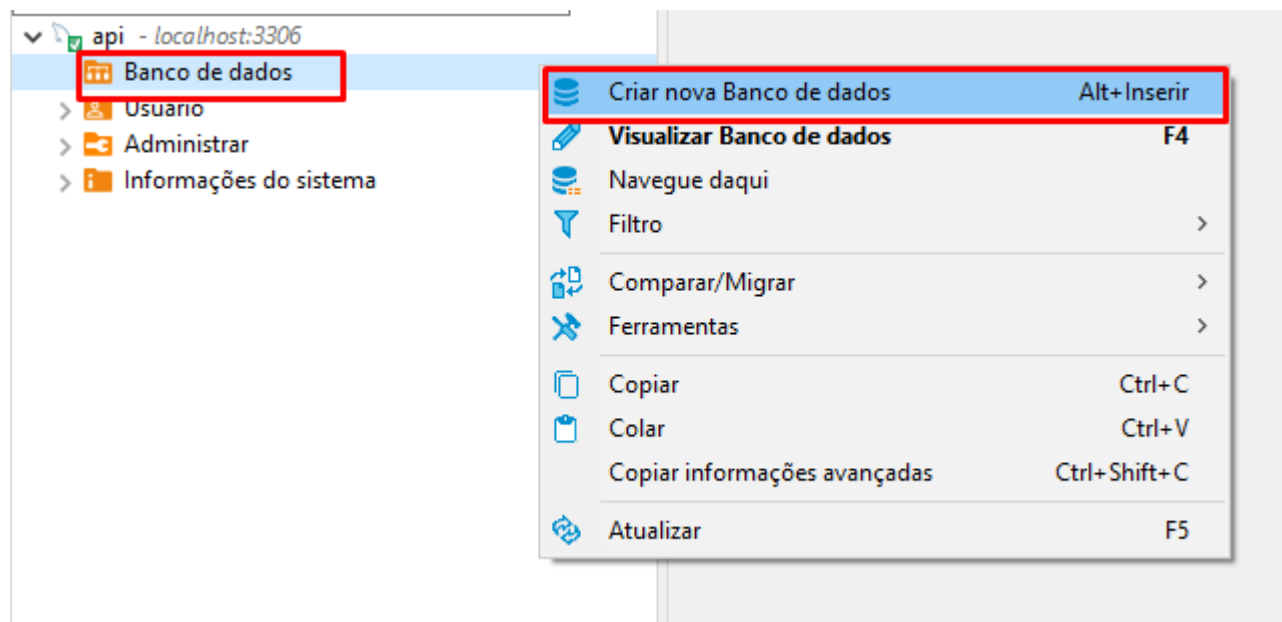
Antes de clicar em Testar Conexão vamos habilitar um Driver em Driver Properties:



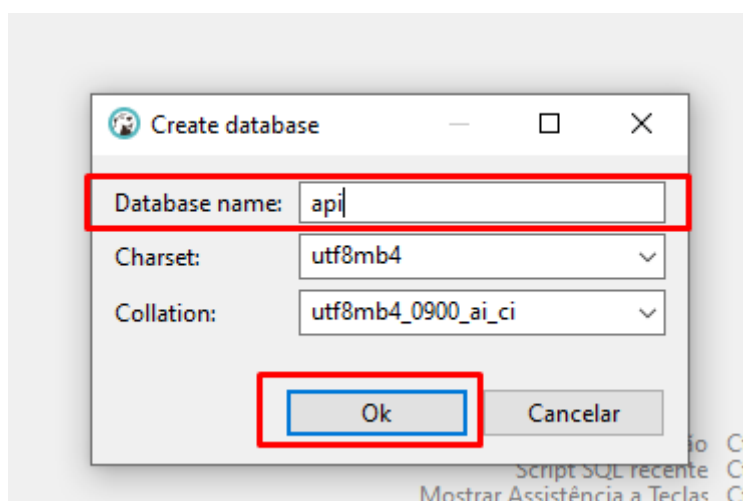
Marcando como TRUE logo em seguida clicando em Testar Conexão, caso der tudo certo uma mensagem de Conectado irá aparecer:



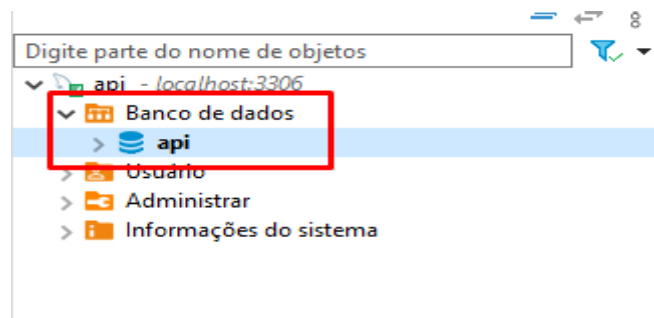
Clicando em Ok e Concluir o DBeaver irá criar um banco em Local Host, logo em seguida podemos clicar com o botão direito do mouse em cima de Banco de Dados e criar uma nova base para o nosso projeto:



E em Database name: Informar o nome do banco, vamos utilizar: API, como demonstração e clicando em Ok para confirmar:



Será criado a base local, lembrando que para levar as informações para esse banco precisa configurar o Prisma conforme o passo a passo passado no começo do e-book.



RESUMÃO 1

AUTHCONTROLLER.JS

```

JS AuthController.js M X
src > app > controllers > auth > JS AuthController.js > registrar > user
1 // Importa a classe PrismaClient do pacote @prisma/client para interagir com o banco de dados
2 import { PrismaClient } from "@prisma/client";
3
4 // Importa o módulo jsonwebtoken para criar e verificar tokens JWT
5 import jwt from "jsonwebtoken";
6
7 // Importa o módulo bcrypt para realizar hashing de senhas e comparações seguras
8 import bcrypt from "bcrypt";
9
10 // Importa a função hashPassword do arquivo passwordUtils.js localizado no diretório ../../repositories
11 import { hashPassword } from "../../repositories/passwordUtils.js";
12
13 // Cria uma instância do PrismaClient para realizar operações no banco de dados
14 const prisma = new PrismaClient();
15
16 // Função assíncrona para registrar um novo usuário
17 async function registrar(req, res) {
18   // Desestrutura os campos email, login, password, name, cpf_cnpj e celular do corpo da requisição
19   const { email, login, password, name, cpf_cnpj, celular } = req.body;
20
21   // Chama a função hashPassword para obter a senha criptografada
22   const hashedPassword = await hashPassword(password);
23
24   try {
25     // Usa o Prisma para criar um novo registro de usuário no banco de dados com os dados fornecidos
26     const user = await prisma.user.create({
27       data: {
28         email, // Define o email do usuário
29         login, // Define o login do usuário
30         password: hashedPassword, // Define a senha criptografada
31         name, // Define o nome do usuário
32         cpf_cnpj, // Define o CPF ou CNPJ do usuário
33         celular, // Define o número de celular do usuário
34       },
35     });
36
37     // Retorna uma resposta com status 201 (Criado) e os dados do usuário criado em formato JSON
38     res.status(201).json(user);
39   } catch (error) {
40     // Em caso de erro, retorna uma resposta com status 400 (Bad Request) e a mensagem de erro
41     res.status(400).json({ message: error.message });
42   }
43 }

```

```

JS AuthController.js M X
src > app > controllers > auth > JS AuthController.js > registrar > user
Tabnine | Edit | Test | Explain | Document | Ask
46  async function login(req, res) {
47    try {
48      // Desestrutura os campos email e password do corpo da requisição
49      const { email, password } = req.body;
50
51      // Usa o Prisma para encontrar um usuário único com o email fornecido
52      const user = await prisma.user.findUnique({
53        where: {
54          email, // Condição de busca pelo email
55        },
56      });
57
58      // Se o usuário não for encontrado, retorna status 401 (Não Autorizado) com mensagem de erro
59      if (!user) {
60        return res.status(401).json({ message: "E-mail ou senha incorretos!" });
61      }
62
63      // Compara a senha fornecida com a senha armazenada (criptografada) usando bcrypt
64      const isValidPassword = await bcrypt.compare(password, user.password);
65
66      // Se a senha não for válida, retorna status 401 com mensagem de erro
67      if (!isValidPassword) {
68        return res.status(401).json({ message: "E-mail ou senha incorretos!" });
69      }
70
71      // Gera um token JWT contendo o uuid do usuário, utilizando a chave secreta e com validade de 1 hora
72      const token = jwt.sign({ uuid: user.uuid }, process.env.SECRET, {
73        expiresIn: "1h",
74      });
75
76      // Retorna uma resposta com o token gerado e o uuid do usuário em formato JSON
77      res.json({ token, user_uuid: user.uuid });
78    } catch (error) {
79      // Em caso de erro, retorna uma resposta com status 400 e a mensagem de erro
80      res.status(400).json({ message: error.message });
81    }
82  }

```

JS AuthController.js M X

src > app > controllers > auth > JS AuthController.js > registrar > user

```
83
84 // Função assíncrona para obter informações de um usuário específico
85 Tabnine | Edit | Test | Explain | Document | Ask
86 async function umUsuario(req, res) {
87   try {
88     // Usa o Prisma para encontrar um usuário único com o uuid fornecido nos parâmetros da requisição
89     const user = await prisma.user.findUnique({
90       where: {
91         uuid: req.params.userUuid, // Condição de busca pelo uuid do usuário
92       },
93       include: {
94         clientes: true, // Inclui relacionamentos com clientes associados ao usuário
95       },
96     });
97
98     // Se o usuário não for encontrado, retorna status 404 (Não Encontrado) com mensagem de erro
99     if (!user) {
100       return res.status(404).json({ message: "Usuário não encontrado" });
101     }
102
103     // Retorna os dados do usuário encontrado em formato JSON
104     res.json(user);
105   } catch (error) {
106     // Em caso de erro interno, retorna status 500 (Erro Interno do Servidor) com a mensagem de erro
107     res.status(500).json({ message: error.message });
108   }
109 }
```

JS AuthController.js M X

```

src > app > controllers > auth > JS AuthController.js > registrar > user
109
110 // Função assíncrona para obter informações do próprio usuário autenticado
    Tabnine | Edit | Test | Explain | Document | Ask
111 async function me(req, res) {
112   try {
113     // Obtém o uuid do usuário a partir do objeto de usuário anexado à requisição (presumivelmente pelo middleware de autenticação)
114     const user_uuid = req.user.uuid;
115
116     // Se o uuid do usuário não estiver presente, retorna status 400 com mensagem de erro
117     if (!user_uuid) {
118       return res
119         .status(400)
120         .json({ message: "O ID do usuário está faltando no token" });
121     }
122
123     // Usa o Prisma para encontrar um usuário único com o uuid obtido
124     const user = await prisma.user.findUnique({
125       where: {
126         uuid: user_uuid, // Condição de busca pelo uuid do usuário
127       },
128       include: {
129         client: true, // Inclui relacionamentos com clientes associados ao usuário
130       },
131     });
132
133     // Se o usuário não for encontrado, retorna status 404 com mensagem de erro
134     if (!user) {
135       return res.status(404).json({ message: "Usuário não encontrado" });
136     }
137
138     // Retorna os dados do usuário encontrado em formato JSON
139     res.json(user);
140   } catch (error) {
141     // Em caso de erro interno, retorna status 500 com a mensagem de erro
142     res.status(500).json({ message: error.message });
143   }
144 }
145
146 // Exporta as funções registrar, login, umUsuario e me como parte do objeto padrão para serem usadas em outros módulos
147 export default {
148   registrar,
149   login,
150   umUsuario,
151   me,
152 };
153

```

RESUMÃO 2

CLIENTSCONTROLLER.JS

```
Js ClientsController.js M X
src > app > controllers > client > Js ClientsController.js > registerClient
1 // Importa a classe PrismaClient do pacote @prisma/client para interagir com o banco de dados
2 import { PrismaClient } from "@prisma/client";
3 // Cria uma instância do PrismaClient para realizar operações no banco de dados
4 const prisma = new PrismaClient();
5 // Função assíncrona para registrar um novo cliente
6 async function registerClient(req, res) {
7   try {
8     // Destruturiza os campos name, cpf e celular do corpo da requisição
9     const { name, cpf, celular } = req.body;
10    // Verifica se o usuário está autenticado; req.user deve estar definido pelo middleware de autenticação
11    if (!req.user) {
12      // Retorna uma resposta com status 401 (Não Autorizado) e uma mensagem de erro
13      return res.status(401).json({ error: "Usuário não autenticado" });
14    }
15    // Obtém o UUID do usuário autenticado a partir do objeto req.user
16    const userUuid = req.user.uuid;
17    // Verifica se todos os campos obrigatórios (name, cpf, celular) estão presentes
18    if (!name || !cpf || !celular) {
19      // Retorna uma resposta com status 400 (Bad Request) e uma mensagem de erro
20      return res.status(400).json({ error: "Todos os dados são obrigatórios" });
21    }
22    // Usa o Prisma para criar um novo registro de cliente no banco de dados com os dados fornecidos
23    const client = await prisma.client.create({
24      data: {
25        name, // Define o nome do cliente
26        cpf, // Define o CPF do cliente
27        celular, // Define o número de celular do cliente
28        userUuid, // Associa o cliente ao UUID do usuário autenticado
29      },
30    });
31    // Retorna uma resposta com status 201 (Criado) e um objeto JSON contendo informações sobre o sucesso e os dados do cliente criado
32    res.status(201).json({
33      message: "Cliente criado com sucesso",
34      status: 201,
35      success: true,
36      data: client,
37    });
38  } catch (error) {
39    // Em caso de erro, retorna uma resposta com status 400 (Bad Request) e um objeto JSON contendo informações sobre o erro
40    res.status(400).json({
41      message: "Erro ao criar cliente",
42      status: 400,
43      success: false,
44      error: error.message,
45    });
46  }
47 }
48
```

```

.js ClientsController.js M X
src > app > controllers > client > .js ClientsController.js > deleteClient
49 // Função assíncrona para deletar um cliente
    Tabnine | Edit | Test | Explain | Document | Ask
50 async function deleteClient(req, res) {
51   try {
52     // Obtém o UUID do cliente a partir dos parâmetros da requisição
53     const uuid = req.params.uuid;
54     // Verifica se o usuário está autenticado
55     if (!req.user) {
56       // Retorna uma resposta com status 401 (Não Autorizado) e uma mensagem de erro
57       return res.status(401).json({ error: "Usuário não autenticado" });
58     }
59     // Usa o Prisma para encontrar o cliente com o UUID fornecido
60     const client = await prisma.client.findUnique({
61       where: {
62         uuid: uuid, // Condição de busca pelo UUID do cliente
63       },
64     });
65     // Se o cliente não for encontrado, retorna status 404 (Não Encontrado) com uma mensagem de erro
66     if (!client) {
67       return res.status(404).json({ error: "Cliente não encontrado" });
68     }
69     // Verifica se o UUID do usuário autenticado corresponde ao UUID associado ao cliente
70     if (client.userUuid !== req.user.uuid) {
71       // Retorna status 403 (Proibido) com uma mensagem de erro indicando falta de permissão
72       return res
73         .status(403)
74         .json({ error: "Você não tem permissão para excluir este cliente" });
75     }
76     // Usa o Prisma para deletar o cliente do banco de dados
77     await prisma.client.delete({
78       where: {
79         uuid: uuid, // Condição de deleção pelo UUID do cliente
80       },
81     });
82     // Retorna uma resposta com status 200 (OK) e um objeto JSON indicando sucesso na deleção
83     return res.status(200).json({
84       message: "Cliente deletado com sucesso",
85       status: 200,
86       success: true,
87       client: client, // Inclui os dados do cliente deletado
88     });
89   } catch (error) {
90     // Em caso de erro, retorna uma resposta com status 400 (Bad Request) e um objeto JSON contendo informações sobre o erro
91     res.status(400).json({
92       message: "Erro ao deletar cliente",
93       status: 400,
94       success: false,
95       error: error.message,
96     });
97   }
98 }

```

```

JS ClientsController.js M X
src > app > controllers > client > JS ClientsController.js > updateClient
100 // Função assíncrona para atualizar um cliente existente
101 Tabnine | Edit | Test | Explain | Document | Ask
101 async function updateClient(req, res) {
102   try {
103     // Obtém o UUID do cliente a partir dos parâmetros da requisição
104     const uuid = req.params.uuid;
105     // Desestrutura os campos name, cpf e celular do corpo da requisição
106     const { name, cpf, celular } = req.body;
107     // Verifica se o usuário está autenticado
108     if (!req.user) {
109       // Retorna uma resposta com status 401 (Não Autorizado) e uma mensagem de erro
110       return res.status(401).json({ error: "Usuário não autenticado" });
111     }
112     // Usa o Prisma para encontrar o cliente com o UUID fornecido
113     const client = await prisma.client.findUnique({
114       where: {
115         uuid: uuid, // Condição de busca pelo UUID do cliente
116       },
117     });
118     // Se o cliente não for encontrado, retorna status 404 (Não Encontrado) com uma mensagem de erro
119     if (!client) {
120       return res.status(404).json({ error: "Cliente não encontrado" });
121     }
122     // Verifica se o UUID do usuário autenticado corresponde ao UUID associado ao cliente
123     if (client.userUuid !== req.user.uuid) {
124       // Retorna status 403 (Proibido) com uma mensagem de erro indicando falta de permissão
125       return res
126         .status(403)
127         .json({ error: "Você não tem permissão para atualizar este cliente" });
128     }
129     // Usa o Prisma para atualizar os dados do cliente no banco de dados
130     const updatedClient = await prisma.client.update({
131       where: {
132         uuid: uuid, // Condição de atualização pelo UUID do cliente
133       },
134       data: {
135         name, // Atualiza o nome do cliente
136         cpf, // Atualiza o CPF do cliente
137         celular, // Atualiza o número de celular do cliente
138       },
139     });
140     // Retorna uma resposta com status 200 (OK) e um objeto JSON indicando sucesso na atualização, incluindo os dados atualizados
141     res.status(200).json({
142       message: "Cliente atualizado com sucesso",
143       status: 200,
144       success: true,
145       data: updatedClient,
146     });
147   } catch (error) {
148     // Em caso de erro, retorna uma resposta com status 400 (Bad Request) e um objeto JSON contendo informações sobre o erro
149     res.status(400).json({
150       message: "Erro ao atualizar cliente",
151       status: 400,
152       success: false,
153       error: error.message,
154     });
155   }
156 }
157

```



```

158 // Função assíncrona para buscar múltiplos clientes associados ao usuário autenticado
159 async function findManyClient(req, res) {
160   try {
161     // Verifica se o usuário está autenticado
162     if (!req.user) {
163       // Retorna uma resposta com status 401 (Não Autorizado) e uma mensagem de erro
164       return res.status(401).json({ error: "Usuário não autenticado" });
165     }
166
167     // Obtém o UUID do usuário autenticado a partir do objeto req.user
168     const userUuid = req.user.uuid; // Obtenha o UUID do usuário autenticado
169
170     // Usa o Prisma para encontrar todos os clientes associados ao UUID do usuário
171     const clients = await prisma.client.findMany({
172       where: {
173         userUuid: userUuid, // Filtra clientes apenas do usuário autenticado
174       },
175     });
176
177     // Retorna uma resposta com status 200 (OK) e um array de clientes em formato JSON
178     return res.status(200).json(clients);
179   } catch (error) {
180     // Em caso de erro, retorna uma resposta com status 500 (Erro Interno do Servidor) e um objeto JSON contendo informações sobre o erro
181     return res.status(500).json({ error: error.message });
182   }
183 }
184
185 // Exporta as funções registerClient, findManyClient, deleteClient e updateClient como parte do objeto padrão para serem usadas em outros módulos
186 export default {
187   registerClient,
188   findManyClient,
189   deleteClient,
190   updateClient,
191 };
192

```

RESUMÃO 3

AUTHENTICATETOKEN.JS

```
js authenticateToken.js M X
src > app > middleware > authUser > js authenticateToken.js > ...
1 | // Importa o módulo jwt (JSON Web Token) para manipular e verificar tokens JWT
2 | import jwt from "jsonwebtoken";
3 |
4 | // Função middleware para autenticar o token JWT em requisições
5 | const authenticateToken = (req, res, next) => {
6 |   // Obtém o cabeçalho de autorização (Authorization header) da requisição
7 |   const authHeader = req.headers.authorization;
8 |
9 |   // Se o cabeçalho estiver presente, divide o valor para pegar o token (esperando que esteja no formato "Bearer <token>")
10 |   const token = authHeader && authHeader.split(" ")[1];
11 |
12 |   // Verifica se o token foi fornecido
13 |   if (!token) {
14 |     // Se não houver token, retorna status 401 (Não Autorizado) e uma mensagem de erro
15 |     return res.status(401).json({ message: "Nenhum token fornecido" });
16 |   }
17 |
18 |   // Verifica o token usando a chave secreta armazenada nas variáveis de ambiente (process.env.SECRET)
19 |   jwt.verify(token, process.env.SECRET, (err, user) => {
20 |     // Se houver um erro na verificação (exemplo: token inválido ou expirado), retorna status 403 (Proibido)
21 |     if (err) {
22 |       return res.status(403).json({ message: "Falha ao autenticar" });
23 |     }
24 |
25 |     // Se o token for válido, anexa o objeto user decodificado à requisição (req.user) para uso posterior nas rotas
26 |     req.user = user;
27 |
28 |     // Chama a próxima função/middleware na fila (permite que a requisição continue)
29 |     next();
30 |   });
31 | };
32 |
33 | // Exporta a função middleware para ser usada em outros módulos
34 | export default authenticateToken;
35 |
```

RESUMÃO 4

PASSWORDUTILS.JS

```

JS passwordUtils.js M X
src > app > repositories > JS passwordUtils.js > ...
1 // Importa a função `hash` do pacote `bcrypt`, que é usada para gerar um hash de uma senha
2 import { hash } from "bcrypt";
3
4 // Exporta uma função assíncrona chamada `hashPassword`, que será usada para criptografar uma senha
5 export const hashPassword = async (password) => {
6 // Define o número de "rounds" de salt para o algoritmo de hash. Um número maior significa mais segurança, mas aumenta o tempo de processamento
7 const saltRounds = 8;
8
9 // Usa a função `hash` do bcrypt para gerar um hash da senha. O algoritmo aplica o salt para dificultar ataques de força bruta
10 const hashedPassword = await hash(password, saltRounds);
11
12 // Retorna a senha criptografada (hash)
13 return hashedPassword;
14 };
15

```

RESUMÃO 5

AUTHROUTER.JS

```

JS authRouter.js M X
src > app > routes > https > authUser > JS authRouter.js > ...
1 // Importa o Router do pacote `express` para criar rotas no servidor
2 import { Router } from "express";
3
4 // Importa o middleware de autenticação de token JWT
5 import authenticateToken from "../../middleware/authUser/authenticateToken.js";
6
7 // Importa o controlador de autenticação, que contém as funções para registro, login e manipulação de usuários
8 import authController from "../../controllers/auth/authController.js";
9
10 // Cria uma nova instância de Router, que será usada para definir rotas de autenticação
11 const authRouter = Router();
12
13 // Define a rota POST para registrar um novo usuário, que usa o método `registrar` do controlador de autenticação
14 authRouter.post("/registrar", authController.registrar);
15
16 // Define a rota POST para fazer login, que usa o método `login` do controlador de autenticação
17 authRouter.post("/login", authController.login);
18
19 // Define a rota GET para buscar informações de um usuário específico, usando o UUID do usuário. Usa o método `umUsuario` do controlador
20 authRouter.get("/usuarios/:userId", authController.umUsuario);
21
22 // Define a rota GET para obter as informações do usuário autenticado (usando o token JWT). A rota usa o middleware `authenticateToken`,
23 // para garantir que o usuário esteja autenticado, antes de chamar o método `me` do controlador
24 authRouter.get("/me", authenticateToken, authController.me);
25
26 // Exporta o roteador de autenticação para ser utilizado em outras partes da aplicação
27 export default authRouter;
28

```

RESUMÃO 6

CLIENTSROUTER.JS

```
JS clientsRouter.js M X
src > app > routes > https > client > JS clientsRouter.js > ...
1 | // Importa o Router do pacote `express` para criar rotas no servidor
2 | import { Router } from "express";
3 |
4 | // Importa o middleware de autenticação de token JWT
5 | import authenticateToken from "../../middleware/authUser/authenticateToken.js";
6 |
7 | // Importa o controlador de clientes, que contém funções para registrar, atualizar, buscar e deletar clientes
8 | import ClientsController from "../../controllers/client/ClientsController.js";
9 |
10 | // Cria uma nova instância de Router para definir as rotas relacionadas a clientes
11 | const clientsRouter = Router();
12 |
13 | // Rota POST para registrar um novo cliente
14 | // Esta rota usa o middleware `authenticateToken` para garantir que o usuário esteja autenticado antes de permitir o registro
15 | clientsRouter.post(
16 |   "/clientes",
17 |   authenticateToken,
18 |   ClientsController.registerClient
19 | );
20 |
21 | // Rota PUT para atualizar os dados de um cliente existente
22 | // A rota requer autenticação e atualiza o cliente identificado pelo `uuid` passado como parâmetro
23 | clientsRouter.put(
24 |   "/clientes/:uuid",
25 |   authenticateToken,
26 |   ClientsController.updateClient
27 | );
28 |
29 | // Rota GET para buscar todos os clientes do usuário autenticado
30 | // O middleware `authenticateToken` é usado para garantir que apenas clientes do usuário autenticado sejam retornados
31 | clientsRouter.get(
32 |   "/clientes/",
33 |   authenticateToken,
34 |   ClientsController.findManyClient
35 | );
36 |
37 | // Rota DELETE para excluir um cliente específico identificado pelo `uuid`
38 | // Requer autenticação e permite que apenas o proprietário do cliente exclua esse registro
39 | clientsRouter.delete(
40 |   "/clientes/:uuid",
41 |   authenticateToken,
42 |   ClientsController.deleteClient
43 | );
44 |
45 | // Exporta o roteador de clientes para ser usado em outras partes da aplicação
46 | export default clientsRouter;
```

RESUMÃO 7

APP.JS

```

JS app.js M X
src > JS app.js > ...
1 | // Importa o pacote `express`, que é um framework para construir aplicações web em Node.js
2 | import express from "express";
3 |
4 | // Importa o pacote `cors`, que é um middleware para habilitar o Cross-Origin Resource Sharing (CORS),
5 | // permitindo que a aplicação seja acessada de diferentes origens
6 | import cors from "cors";
7 |
8 | // Importa o roteador de autenticação, que contém rotas para registro, login e manipulação de usuários
9 | import authRouter from "../app/routes/https/authUser/authRouter.js";
10 |
11 | // Importa o roteador de clientes, que contém rotas para manipulação de clientes (criação, atualização, listagem e exclusão)
12 | import clientsRouter from "../app/routes/https/client/ClientsRouter.js";
13 |
14 | // Cria uma instância da aplicação `express`
15 | const app = express();
16 |
17 | // Habilita o middleware `cors`, que permite que a aplicação seja acessada por domínios diferentes do servidor de origem
18 | app.use(cors());
19 |
20 | // Middleware do `express` que permite a aplicação entender dados no formato JSON enviados no corpo das requisições
21 | app.use(express.json());
22 |
23 | // Middleware do `express` que permite a aplicação entender dados enviados via URL encoded,
24 | // (normalmente usados em formulários). A opção `extended: true` permite suportar objetos e arrays complexos
25 | app.use(express.urlencoded({ extended: true }));
26 |
27 | // Define o uso das rotas de autenticação importadas do `authRouter`. Isso adiciona as rotas,
28 | // relacionadas a autenticação (registro, login, etc.) na aplicação
29 | app.use(authRouter);
30 |
31 | // Define o uso das rotas de clientes importadas do `clientsRouter`. Isso adiciona as rotas relacionadas,
32 | // a operações de clientes (criação, atualização, exclusão, etc.) na aplicação
33 | app.use(clientsRouter);
34 |
35 | // Exporta a instância da aplicação `app` para ser utilizada em outros arquivos, como por exemplo no arquivo que inicializa o servidor
36 | export default app;

```

RESUMÃO 8

SERVER.JS

```

JS server.js M X
JS server.js > ...
1 | // Importa a instância da aplicação `app` do arquivo "src/app.js".
2 | // Esse arquivo contém a configuração do servidor, rotas e middlewares.
3 | import app from "../src/app.js";
4 |
5 | // Define a porta do servidor. Primeiro tenta usar a variável de ambiente `PORT`,
6 | // se não estiver definida, usa a porta 5000 como padrão.
7 | const PORT = process.env.PORT || 5000;
8 |
9 | // Inicia o servidor, ouvindo na porta definida. O método `listen` do `express`
10 | // começa a escutar requisições HTTP.
11 | app.listen(PORT, () => {
12 |   // Quando o servidor começa a rodar, imprime no console uma mensagem indicando,
13 |   // que o servidor está rodando na porta definida.
14 |   console.log(`Servidor rodando na porta ${PORT}`);
15 |
16 |   // Exibe no console o link para acessar o servidor localmente, útil para desenvolvimento.
17 |   console.log(`http://localhost:${PORT}`);
18 | });
19 |

```