



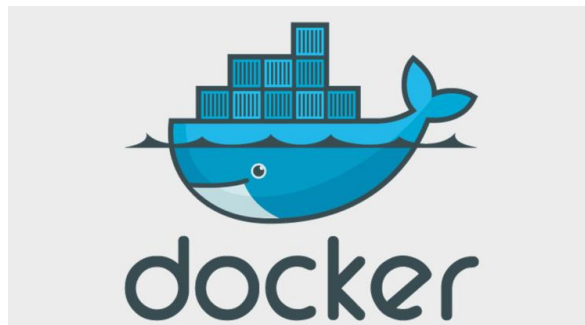
Universidade do Minho
Escola de Engenharia

UNIVERSIDADE DO MINHO

Virtualização de Redes

Trabalho prático 2

Docker Microservices



Autores:

Bruno Rodrigues pg41066

Carlos Alves pg41840

Paulo Bento a81139

1 ÍNDICE

INTRODUÇÃO	3
ESTRUTURAÇÃO	4
Serviço de base de dados	5
Serviço de Autenticação	6
Aplicação Flask	6
Gestão da base dados	7
Templates	7
Serviço HTTP	8
Serviço FTP	9
DockerFile	10
Autenticação && FTP Server && HTTP Server	10
DOCKER-COMPOSE	10
Volumes	10
Networks	11
Implementação Reverse proxy	11
ANEXO A	13

INTRODUÇÃO

Este trabalho enunciado na unidade curricular Virtualização de Redes tem como objetivo de desenvolver uma serie de micro serviços utilizando o Docker.

Estes serviços consistem em dois file servers e um serviço de autenticação, este último deverá gerar *tokens* apos uma correta autenticação. Além disto, o *token* deverá permitir acesso aos dois outros file servers. E os dois file servers deverão verificar o *token* recebido através de uma rede interna que ligará os dois serviços *fileservice* com o serviço de autenticação, e este por si deverá ligar-se á base de dados, sendo que só será possível ligar á base de dados através do serviço de autenticação.

Todos os serviços criados no desenrolar do projeto, serão gerados usando *dockerfiles* e por fim, de modo a obter o sistema como um todo e com um simples comando, é então gerado um *Docker-compose* que permitirá utilizar estes serviços em qualquer máquina, com recurso ao Docker-hub, onde as imagens serão guardadas.

ESTRUTURAÇÃO

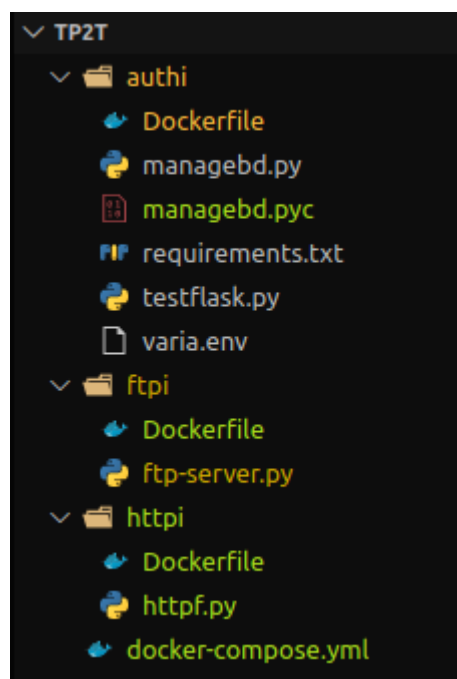
Bem inicialmente começamos por pensar da arquitetura, como ela está desenhada e de que forma a iríamos abordar. Em que linguagem iríamos codificar e só depois iríamos procurar as ferramentas para desenvolver o trabalho.

Primeiramente, a escolha da linguagem de programação não foi escolhida de forma assertiva, acabando assim por ocorrer uma duplicação do trabalho em diferentes linguagens (Python e Javascript). Talvez esta tenha sido o fator que nos mais atrasou, além dos problemas provenientes da utilização do **Docker**.

Depois de escolhida a linguagem, começamos por percorrer a web em busca de qual/quais seriam as melhores ferramentas para determinado serviço a implementar. No serviço de base de dados, experimentamos alguns sistemas já existentes para este fim, como o **Django**, o **MongoDB** e por fim o **Postgres**. Após uma experimentação intensiva nestes sistemas gerenciadores de base de dados, decidimos utilizar no projeto o **Postgres**, com alguma relutância pois o **MongoDB** também seria uma opção viável para realizar este projeto. Inclusive, neste documento iremos apresentar um pouco do trabalho efetuado usando o **MongoDB**, apenas para quesitos de documentação (**Anexo-A**).

Como *framework* web, decidimos usar o **Flask** – um micro *framework* Web construído em Python. A escolha deveu-se ao fato deste não precisar de ferramentas específicas ou mesmo plugins para funcionar e claro por ser leve e flexível. A vantagem de implementar o programa **Flask** com o Docker é que se torna possível replicar a aplicação em outros servidores com uma fácil reconfiguração.

Após termos as ferramentas todas instaladas e escolhidas, foi necessário desenvolver realmente o projeto. Primeiramente, criamos a estrutura como se pode verificar ao lado.



Ao longo deste documento, iremos explicar cada um dos ficheiros e as suas funcionalidades.

Nota: A estrutura em cima não é a estrutura do projeto final, mas sim durante o desenvolvimento.

Serviço de base de dados

Como já referido, foi utilizado o **postgres** como serviço de base de dados. Visto que já se encontrava a imagem no docker Hub, apenas a adicionamos ao nosso *docker-compose.yml*. Mas posteriormente vimos que para realizar um *autobuild* completo do projeto todo, foi necessário criar um Dockerfile do postgres, pois para conseguir a independência do Docker-compose, mas ao mesmo tempo criar as tabelas sem ser necessário o input do utilizador, criar um Dockerfile foi mais simples. Mais abaixo iremos abordar o *docker-compose*.

O servidor postgres tem como objetivo armazenar os dados de registo do utilizador, e mais tarde recuperá-los, isto conforme o solicitado pelos outros serviços a serem desenvolvidos. O bom do postgres é que também lida com consultas SQL complexas usando muitos métodos de indexação que não estão disponíveis em outras bases de dados. Mas como o intuito do trabalho não é propriamente aprofundar em base de dados, apenas “raspamos” a superfície do postgres. Ainda assim, testamos uma ferramenta para gerenciar o conteúdo na base de dados do postgres, o Adminer – uma alternativa ao phpMyAdmin e o PGAdmin. Este serviço estaria apenas ligado à base de dados, necessitando de outras credenciais para adentrar no serviço.

A base de dados é iniciada pelo Dockerfile, que cria a base de dados, como bem o utilizador associado

A tabela é iniciada automaticamente, devido ao facto que a imagem do postgres no Docker hub descobre automaticamente ficheiros sql e roda-os.

Neste caso o script usado está referenciado em baixo.

```
CREATE TABLE public.clients
(
  "Id" integer NOT NULL GENERATED ALWAYS AS IDENTITY ( INCREMENT 1 START 1 MINVALUE
  "username" VARCHAR (128) UNIQUE NOT NULL,
  "password" VARCHAR (256) NOT NULL,
  "Token" VARCHAR (256) ,
  "email" VARCHAR (355) UNIQUE NOT NULL,
  "isAdmin" boolean NOT NULL,
  PRIMARY KEY ("Id")
)
```

Serviço de Autenticação

Visto que estávamos a codificar em Python e trabalhar com o **Postgres**, tivemos de instalar um adaptador que fornecesse apoio entre a linguagem de programação Python e o **Postgres**, assim fornecendo o acesso a muitos outros recursos oferecidos pelo **Postgres**, como por exemplo extensíveis com novos adaptadores para converter objetos tipos do **Postgres** novamente em objetos Python – Psycopg2.

É criado um novo ficheiro chamado *varia.env* onde será guardada as variáveis de ambiente para desenvolvimento. Neste ficheiro, colocamos as variáveis ambiente do **Postgres** que iremos colocar no *docker-compose*, que será explicado no final do documento.

De modo a simplificar o processo de adicionar as ferramentas usadas ao ficheiro *requirements.txt*, utilizamos uma lib – *pipreqs* - que nos permite gerar esse mesmo ficheiro *requirements.txt* com base nos *imports* do projeto.

Aplicação Flask

Contém a logica do serviço de autenticação e faz uso do *managedb* para realizar as tarefas de registo, login e geração de *token*.

Em suma, o login segue o seguinte algoritmo:

- Em caso de login, é recebido os parâmetros nome do utilizador e a password; (é utilizado uma função hash na password, sha256)
- É verificado se se encontra na base de dados;
- Verifica se é um utilizador já está registado;
 - Se não for é lhe emitido uma mensagem de erro.
- Caso este esteja registado, é emitido um token
- A premir o botão Registrar, é redirecionado para a aba de registo /register.

No método de verificar, é então verificado o token se este realmente é válido.

No método de Registrar, são realizadas uma série de pedidos **gets** dos dados a introduzir pelo utilizador (*username*, password e email) e por fim regista esse utilizador na base de dados e redireciona-o para a página de login.

A segurança aqui é implementada apenas na forma como é digerida a password, usando uma função *hash(256)* e uma chave secreta aleatória necessária.

Gestão da base dados

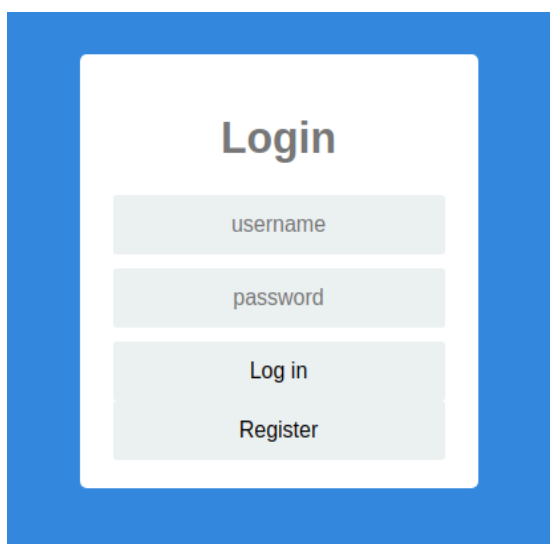
No script `managedb.py`, que tem como objetivo fornecer um suporte e facilidade na gestão da base de dados. Usado pela aplicação **Flask** ou pelo serviço de autenticação. Neste foram desenvolvidos métodos para *Registo* e *Verificação* de utilizadores, como também era pedido a implementação de um sistema de *tokens* (**JSON Web Tokens - JWT**) foi então acrescentado métodos responsáveis pela geração(`encode`) e configuração desses mesmos *tokens*, como por exemplo a duração de vida do *token*. E por fim um método de verificação de *tokens*, onde irá tentar verificar o *token* recebido é válido, se não irá fornecer uma das mensagens de erro (*token* inválido ou expirado).

Voltando ao método de Registo, aqui é recebido o nome do utilizador, a palavra-chave, o e-mail e ainda booleano para verificar de se tratar de um registo de *admin*. Com o adaptador `psycopg2` conectamos essas entradas à base de dados já criada, abrimos um cursor para realizar as diversas operações da base de dados e só depois executamos o comando. Neste caso para preencher os espaços reservados da *query*. E por fim faz-se *commit* para fazer as alterações no banco de dados(*persistent*). Em caso de não conseguir prosseguir com tal tarefa, devolvemos um erro e é então fechado o cursor e a ligação com a base de dados.

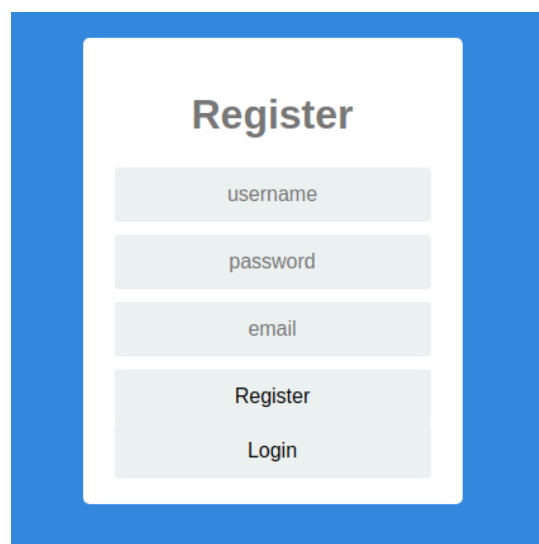
No método verificar, o processo é semelhante apenas acrescentamos a inserção do token e executamos apenas a *query* para determinar se as entradas inseridas pelo utilizador estão corretas ou presentes na base de dados.

Templates

Foram criados os *templates* necessários para a realização de um **login** e **register**. Para isso foi desenvolvido dois ficheiros *html*, uma para cada função (login e register) e um *css* para a interface ficar com um rosto mais bonito. Abaixo é possível observar o token gerado após um login efetuado com sucesso.



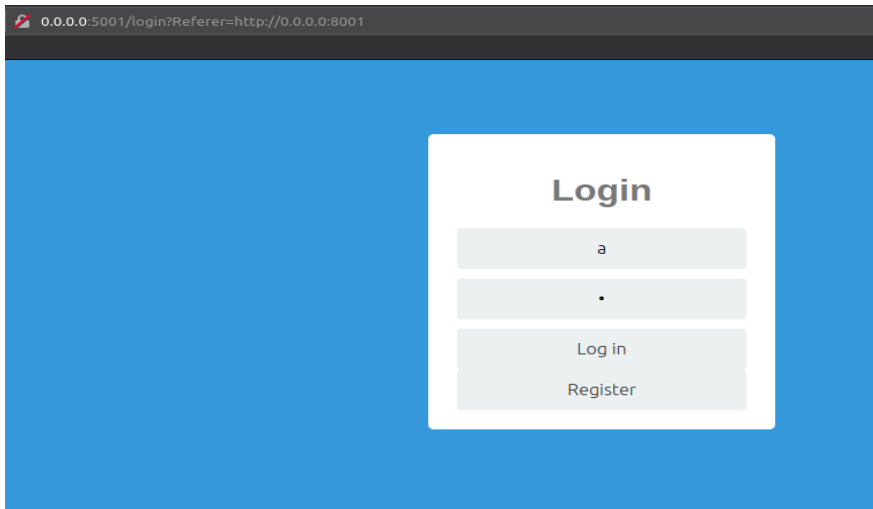
The image shows a login form template with a blue border. Inside, there is a white box with the title "Login" in bold. Below the title, there are two input fields: "username" and "password". At the bottom, there are two buttons: "Log in" and "Register".



The image shows a register form template with a blue border. Inside, there is a white box with the title "Register" in bold. Below the title, there are three input fields: "username", "password", and "email". At the bottom, there are two buttons: "Register" and "Login".

Serviço HTTP

No serviço de HTTP, tivemos por base o server *http* fornecido pelo docente, onde desenvolvemos o mecanismo que invés de ter algum tipo de autenticação, fosse redirecionado para o serviço de autenticação para ser autenticado.



Redirecionamento do http para o serviço de login

Quando este realiza o login com sucesso, o token é incorporado no url e redirecionado de volta para o servidor http, que em si recolhe o token e verifica através de um get para o /verify do serviço de autenticação para verificar a validade deste.

Caso seja valido, este token é guardado localmente para ser acedido mais tarde, caso seja invalido, o usuário é novamente redirecionado para o serviço de autenticação

Directory listing for /?

token=eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJleHAiOiJlODgzNDIzMzUsImhhdCI6IjZ8YE3OuQ1XE

-
- [docker-entrypoint.sh](#)
 - [Dockerfile](#)
 - [http.py](#)
 - [requirements.txt](#)
 - [testfile.txt](#)
-

Acesso com sucesso

Serviço FTP

No serviço FTP utilizamos o script fornecido pelo docente, onde apenas tivemos de realizar algumas alterações para que este em vez de usar o *username* e a *password* do serviço de autenticação, recebesse um *token* de autenticação e o verificasse com o serviço de autenticação. Para tal o utilizador apenas tem de aceder ao serviço de login, copiar o token que resulta deste e colá-lo quando este é pedido.

A biblioteca de servidores FTP do Python fornece uma interface portátil de alto nível para gravar facilmente servidores FTP muito eficientes, escaláveis e assíncronos com o Python.

DockerFile

Autenticação && FTP Server && HTTP Server

Na construção do *dockerfile* para correr o server FTP, server HTTP e o serviço de autenticação, utilizamos uma imagem do Ubuntu, copiamos os ficheiros para dentro do container criado e executamos os comandos essenciais para correr uma aplicação python (*python, pip, requests, pyftplib*)

DOCKER-COMPOSE

Ao utilizar *docker compose* conseguimos executar todos as imagens ou serviços apenas com um simples comando. Ainda podemos criar as redes e volumes nesse mesmo ficheiro, configurando-o conforme as nossas necessidades. Este serviço fornecido pelo próprio Docker, facilita o *deployment* dum projeto em diferentes ambientes.

Volumes

Para persistir os dados além do tempo de vida do container, criamos um volume. Assim conseguimos vincular o **postgres_data** ao diretório que especificamos no container. Também adicionamos uma chave **Environment** para definir um nome para a base de dados e um utilizador e uma password.

```
calvares@calvares:~$ docker volume ls
DRIVER          VOLUME NAME
local           3c8676afad508cd3ab1debcfe3
local           74e6f11a85992f570e68e531ff
local           537bd1e1c1bfb2c3c4fedfb54e
local           new-test-vol
local           pgdata
local           teste2_postgres
local           tp2t2_data-volume
local           tp2t2_postgres-vol
local           tp2t_data-volume
local           tp2t_postgres-vol
local           tp2vr_postgres
```

Como é possível observar na imagem acima, o volume foi criado (último volume).

Com o comando **docker volume inspect <nomedoContainer>**, verificamos o seu conteúdo.

```
calvares@calvares:~$ docker volume inspect tp2vr_postgres
[
  {
    "CreatedAt": "2020-05-01T14:42:38+01:00",
    "Driver": "local",
    "Labels": {
      "com.docker.compose.project": "tp2vr",
      "com.docker.compose.volume": "postgres"
    },
    "Mountpoint": "/var/snap/docker/common/var-lib-docker/volumes/tp2vr_postgres/_data",
    "Name": "tp2vr_postgres",
    "Options": null,
    "Scope": "local"
  }
]
```

Networks

As networks foram adicionadas ao documento *docker-compose.yml* assim como os volumes e containers serão criados automaticamente após a execução do *docker-compose*. As redes criadas neste projeto foram as de ligação entre:

- Serviço da base de dados e o serviço de autenticação;
- Serviço de Autenticação e o Serviço **Http**;
- Serviço de Autenticação e o Serviço **FTP**.

```
0475381a6c4f      tp2t_httpnetwork  bridge          local
a18b55df8adf      tp2vr_ftpnetwork  bridge          local
1b2230533ec9      tp2vr_httpnetwork bridge          local
123f37b7ebd9      tp2vr_postgres    bridge          local
calvares@calvares:~$
```

Implementação Reverse proxy (Não implementado no projeto final)

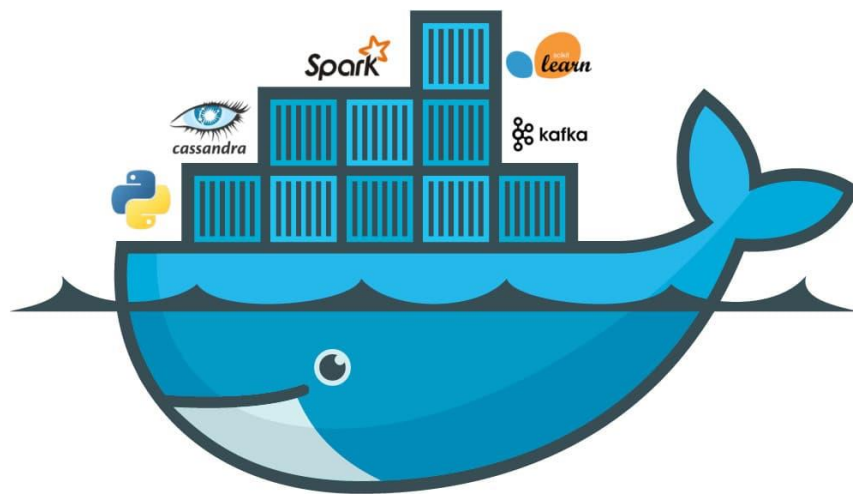
Devido a algumas imprevisibilidades como o Nginx não conseguir correr em containers da mesma network e também ao tempo restante para conclusão do projeto, não conseguimos implementar na totalidade esta componente. Mas fica documentado o progresso e o reverse proxy utilizado.

Ainda no *docker-compose*, adicionamos o **Nginx** junto com os restantes serviços que irá agir como um reverse proxy. Foi criada uma nova diretoria dentro da pasta do projeto, onde são criados um *dockerfile* e um *nginx.conf*.

O objetivo deste serviço seria ser um servidor intermediário que encaminha.se as solicitações de conteúdo de clientes para diferentes servidores. Neste caso seria um container que permitiria que todos os serviços descritos neste documento, fossem acedidos na mesma porta, mas com diferentes caminhos.

Conclusão

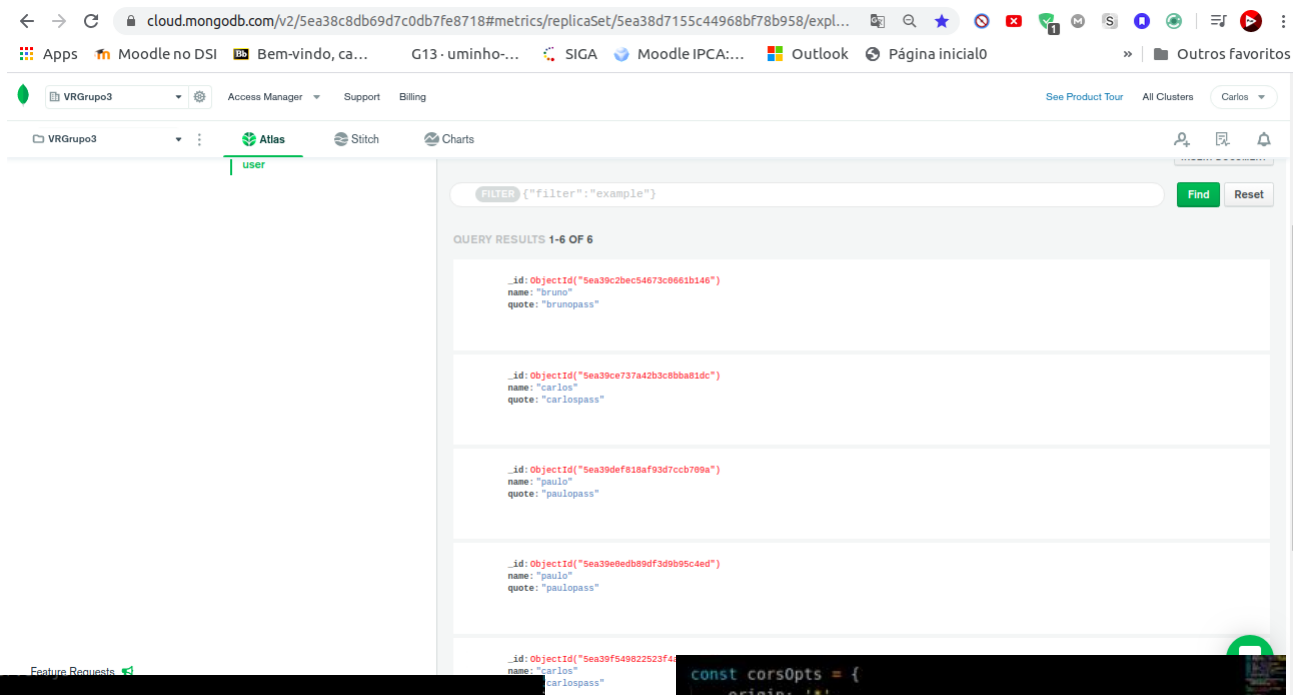
Com este trabalho podemos concluir que o uso de Docker consegue ser bastante completo e resolver muitos problemas que desenvolvedores possam ter. O fato dos ambientes serem semelhantes, isto é, a partir do momento que conseguimos transformar o nosso programa numa imagem Docker, ela poderá ser facilmente replicada e utilizada em qualquer máquina. Além disso, vimos que com Docker é possível empacotar toda aplicação e as seus dependências, isto facilita a sua distribuição pois basta a execução da imagem. Como se pode verificar no nosso trabalho, acabamos por replicar e reutilizar vários mecanismos e isto graças ao Docker.



ANEXO A

Implementação do Serviço de Autenticação, usando o mongodb (NÃO UTILIZADO NO PRODUTO FINAL – Apenas para documentar).

LOGIN



```

so do Express-
st express = require('express');
st bodyParser = require('body-parser')
st MongoClient = require('mongodb').MongoClient
st app = express()
st jwt = require('jsonwebtoken')
st users = require()

adeia de conexao
connectionString = "mongodb+srv://dbCarlos:carlao@cluster1

conectar ao MongoDB
goClient.connect(connectionString, { useUnifiedTopology: t
then(client => {
  console.log('Connected to Database')
  const db = client.db('user-credencial')
  const userCollection = db.collection('user')

  //Indica ao express q usamos ejs como mecanismo de modelo
  //app.set('view engine', 'ejs')
  //middlewares and other routes here
  //Renderizacao do HTML
  //view- nome do arquivo a processar, locals- sao os dados
  //res.render(view, locals)

  //body-parser antes do CRUD handlers
  //urlencoded diz ao analisador do corpo para extrair dado
  //adicionais a propriedade body no objeto request.
  app.use(bodyParser.urlencoded({extended: true })))

```

```

const corsOpts = {
  origin: '*',
  credentials: true,
  methods: ['GET', 'PUT', 'POST', 'DELETE', 'OPTIONS'],
  allowedHeaders: ['Accept', 'Authorization', 'Cache-Con
}

const app = express();
app.use(bodyParser.urlencoded({ extended: false })))
app.use(cors(corsOpts))

//Handlers
//metodo post- o path deve ser aquele q um gajo mete la no
app.post('/registro', (req, res) => {
  //mock-user é aAPI de autenticação -https://www.npmjs.
  const user = req.body
  users.newUser( (parameter) res: any newUser
    .then(user => res.json(user))
  })

app.post('/autenticar', (req, res) =>{
  users.getUser(req.body.username)
    .then(user => {
      if(user && user.password === req.body.password) {
        jwt.sign({ user: req.body }, 'secretkey', { ex
          res.json({
            token
          });
        }
      } else {

```