

UNIVERSIDADE DO MINHO  
MESTRADO INTEGRADO EM ENGENHARIA INFORMÁTICA

**Gestão e Virtualização de Redes**  
**Virtualização de Redes**

TRABALHO PRÁTICO 2

---

**Docker Microservices**

---



JOANA ISABEL AFONSO GOMES **a84912**

29 DE JUNHO DE 2021

# CONTEÚDO

---

<b>1</b>	<b>Introdução e Contextualização</b>	<b>2</b>
<b>2</b>	<b>Implementação e Desenvolvimento</b>	<b>3</b>
2.1	Serviço de Autenticação . . . . .	4
2.2	Servidor HTTP . . . . .	8
<b>3</b>	<b>Docker</b>	<b>10</b>
3.1	Dockerfile Autenticação . . . . .	10
3.2	Dockerfile HTTP . . . . .	11
3.3	dockercompose.yml . . . . .	11
<b>4</b>	<b>Demo</b>	<b>13</b>
<b>5</b>	<b>Conclusão</b>	<b>16</b>

## INTRODUÇÃO E CONTEXTUALIZAÇÃO

---

No âmbito da Unidade Curricular de **Virtualização de Redes** foi-me proposto este segundo trabalho prático, que consiste na procura de aquisição de conhecimentos no que toca à virtualização com *Docker*.

O proposto neste trabalho traduz-se na implementação de *microservices* em *docker containers*, seguindo algumas regras na forma como interagem umas com as outras.

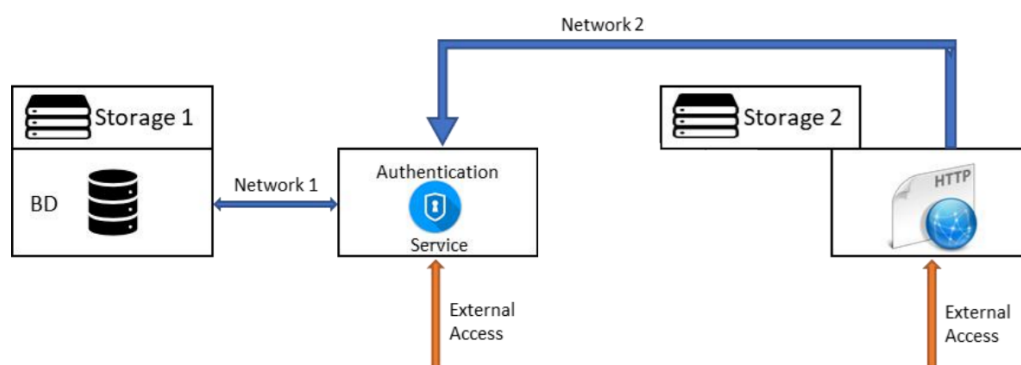


Figura 1.1: Arquitetura proposta.

## IMPLEMENTAÇÃO E DESENVOLVIMENTO

---

Para implementar com sucesso a arquitetura proposta, foi necessário, primeiramente, criar um **serviço de autenticação** que comunique com uma **base de dados**.

Importante referir que desenvolvi a globalidade do projeto (tanto a implementação do serviço de autenticação como do *server* HTTP) com a *framework* **Node.js**.

No que toca à **base de dados**, optei por utilizar o *software* **MongoDB**, visto que, aquando das minhas pesquisas iniciais e preparação para o desenvolvimento do projeto, me deparei com o módulo *mongoose* do *Node.js* e concluí que seria uma mais valia perante uma análise inicial do projeto.

Para além das componentes referidas, o projeto envolve ainda o desenvolvimento de um **servidor HTTP** que deve comunicar diretamente com o serviço de autenticação e que deverá estar associado a um volume para permanência de dados.

O objetivo final é todos os *containers* a criar serem implementados em **docker files**, para estes serem transformados em *docker builds* e assim, com um ficheiro **docker-compose.yml**, seja possível executar e reproduzir a arquitetura em qualquer computador.

## 2.1 Serviço de Autenticação

Comecei por desenvolver um serviço de autenticação, criando um projeto em *Node.js*, com o objetivo de criar um serviço que, aquando de uma autenticação correta, gerasse um *token* que permitirá o acesso ao *server* HTTP.

Inicialmente criei uma conexão do serviço de autenticação à base de dados MongoDB onde serão guardados os utilizadores (base de dados que estará depois num *container* diretamente ligado ao *container* com a aplicação do serviço de autenticação por uma *network* (*network1*).

```
var mongoDB = "mongodb://mongodb/vr-tp2";

mongoose.connect(mongoDB, {
  useNewUrlParser: true,
  useUnifiedTopology: true,
  useFindAndModify: false
});

var db = mongoose.connection;

db.on("error", () => {
  console.log("MongoDB connection failed...");
});

db.once("open", () => {
  console.log("MongoDB connection successful...");
});
```

A *WebApp* de serviço de autenticação consiste numa área para *login* e uma área de registo. Aquando do primeiro acesso, existe já um utilizador criado na base de dados com o *role* de administrador (**admin**).

O registo de um utilizador passa pela atribuição dos parâmetros que podemos observar na parte dos *models* da aplicação:

```
var UserSchema = new mongoose.Schema({  
  _id: String,  
  name: String,  
  password: String,  
  level: Number  
});
```

Sendo que o parâmetro *level* corresponde ao **role** de um utilizador (*admin* ou *user*).

Assim, na área de *Sign Up* podem ser criados novos utilizadores, aos quais será atribuído o *role* de **users**, ou seja, terão uma distinção de conteúdo a que podem aceder quando comparados com o administrador.

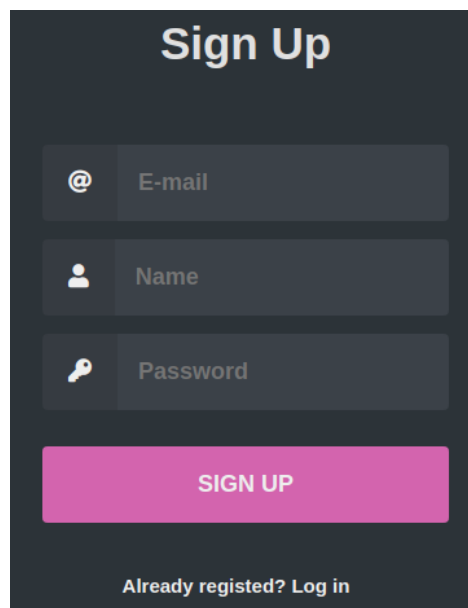
A image showing a 'Sign Up' form on a dark background. The form has three input fields: 'E-mail' with an '@' icon, 'Name' with a person icon, and 'Password' with a key icon. Below these fields is a large pink button labeled 'SIGN UP'. At the bottom, there is a link that says 'Already registered? Log in'.

Figura 2.1: Área de *Sign Up* da WebApp.

Com a inserção de novos utilizadores (e, claro, o administrador já existente), foi desenvolvida uma função para permitir a geração de um *token* e que redirecionará o utilizador para a parte do HTTP.

Para permitir a autenticação dos utilizadores, é feito um POST com uma chave previamente gerada (***privateKey***). Recebendo o pedido, será verificada a chave recebida e, caso seja

uma chave válida, é atribuído um *token* assinado com a *privateKey*. O uso de *tokens* visa segurança e uma boa utilização num contexto de login único.

Posteriormente, é enviado através de *JSON* o *token*, que foi gerado com o módulo *jsonwebtoken* (jwt). Utilizei este módulo com fim de transmitir ao *server* HTTP os dados do utilizador e saber qual o seu *role*.

Um utilizador registado preenche então o formulário com as credenciais (E-mail e *Password*). Após isso, é verificada a existência desse *user* na base de dados e se a *password* é a correspondente. Caso tal se verifique, é gerado o *token*, que é guardado nas *cookies* do browser.

Caso não se verifique uma autenticação correta, estando o utilizador não registado ou sendo a *password* não correspondente, são emitidos os respetivos erros na *WebApp* (fig. 2.1).

```
router.post('/login', function(req, res, next) {
  User.lookup(req.body._id).then((dados) => {
    const user = dados;
    if (! user) {
      res.render('loginError', { title: 'Login', error: 'User not registered' });
    } else {
      if (req.body.password == user.password) {
        var privateKey = fs.readFileSync('./private.key', 'utf8');
        jwt.sign({
          id: user._id,
          name: user.name,
          level: user.level
        }, privateKey, {
          expiresIn: "30s",
          algorithm: 'RS256'
        }, function (err, token) {
          if (err) {
            res.render('loginError', { title: 'Login', error: 'Could not
              ↪ login' });
          } else {
            res.cookie('token', token)
            if (user.level == 1) {
              res.redirect('http://0.0.0.0:4004/admin')
            }
          }
        })
      }
    }
  })
})
```

```

        }
        else if(user.level==0){
            res.redirect('http://0.0.0.0:4004/user')
        }
    }
});
} else {
    res.render('loginError', { title: 'Login',error:'Wrong password' });
}
}
});
});

```

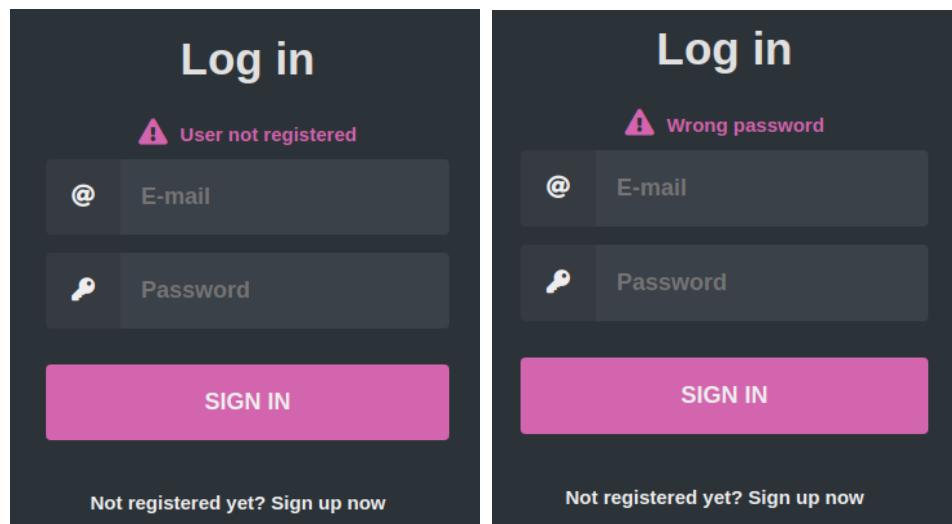


Figura 2.2: Erros no Login.

No caso de uma autenticação correta, a função reencaminha o utilizador *loggado* para a página correspondente do *server* HTTP, que irá verificar a legitimidade do *token*, como explicarei na secção 2.2.



## 2.2 Servidor HTTP

O servidor HTTP é desenvolvido com vista a estar diretamente ligado ao serviço de autenticação. Quando o serviço de autenticação redireciona para o servidor HTTP, este verifica a legitimidade do *token* recorrendo à chave pública correspondente.

Após a autenticação e direcionamento para as páginas respetivas, o sistema não permite obviamente, por exemplo, que alguém que esteja *logado* como *user* aceda ao *link* da página do administrador.

Para reproduzir este acesso seguro, foram criadas duas funções importantes nas *routes* do HTTP (*verifyIdAdmin* e *verifyIdUser*).

```
router.get("/admin", verifyIdAdmin, function (req, res) {
  var content = JSON.parse(fs.readFileSync('./httpvol/httpLog.json','utf8'));
  res.render('admin', { title: 'admin' , content});
})

router.get("/user", verifyIdUser, function (req, res) {
  res.render('user', { title: 'user' });
})

router.all('*',verifyIdUser, function (req, res, next) {
  res.redirect('/user');
});

function verifyIdAdmin(req, res, next) {
  var token = req.cookies.token;
  if (token == null) res.redirect('http://0.0.0.0:4000');
  var secretKey = fs.readFileSync('./public.key','utf8');
  jwt.verify(token, secretKey, { algorithm: ["RS256"] }, function (err, decoded) {
    if (err) res.redirect('http://0.0.0.0:4000');
    if (decoded === undefined) res.redirect('http://0.0.0.0:4000');
    if (decoded.level !== 1) res.redirect('/user');
    next();
  });
}
```

```

function verifyIdUser(req, res, next) {
  var token = req.cookies.token;
  if (token == null) res.redirect('http://0.0.0.0:4000');
  var secretKey = fs.readFileSync('./public.key', 'utf8');
  jwt.verify(token, secretKey, { algorithm: ["RS256"] }, function (err, decoded) {
    if (err) res.redirect('http://0.0.0.0:4000');
    if (decoded === undefined) res.redirect('http://0.0.0.0:4000');
    if (decoded.level !== 0) res.redirect('/admin');
    var jsonLog = [];
    jsonLog = JSON.parse(fs.readFileSync('./httpvol/httpLog.json', 'utf8'));
    if(decoded.level!=1){
      var loggar = decoded.id;
      jsonLog.push(loggar);
      fs.writeFileSync('./httpvol/httpLog.json', JSON.stringify(jsonLog));
    }
    next();
  });
}

```

Nesta *routes* consta ainda uma função que permite o *logout* de um utilizador.

Como se pode verificar na função *verifyIdUser*, cada vez que um utilizador entra na sua área, esse acesso é registado num ficheiro *log*. Este ficheiro *log* (*httpLog.log*) está associado a um volume de permanência de dados do *container* que terá a aplicação web.

O administrador, na sua área, tem acesso ao registo de *login* dos utilizadores do site, através da consulta deste *log*.

# 3

## DOCKER

---

A implementação deste projeto foi totalmente direcionada e teve como grande objetivo o desenvolvimento da arquitetura explicitada no início deste relatório, com recurso a serviços *Docker*.

No meu projeto criei então dois *docker files*, um para o serviço de autenticação e outro para o HTTP. Para ser possível recriar a arquitetura a partir de qualquer computador, dei *build* a estes *docker files* originando duas imagens no docker hub (*joanafonsogomes/auth* e *joanafonsogomes/http*).

### 3.1 Dockerfile Autenticação

```
FROM node:latest
WORKDIR .
COPY /package.json ./
COPY . .
RUN npm install --production
EXPOSE 4000
CMD [ "npm", "start" ]
```

## 3.2 Dockerfile HTTP

```
FROM node:latest
WORKDIR .
COPY /package.json ./
COPY . .
RUN npm install --production
EXPOSE 4004
CMD [ "npm", "start" ]
```

## 3.3 dockercompose.yml

Para a criação do *docker compose*, foi preciso ter em vista quais as componentes da arquitetura que pretendia desenvolver.

Assim, existe primeiramente um *container* com a imagem pré-existente da base de dados do MongoDB. Este *container* está ligado à *network1*, que por sua vez efetuará a comunicação com o *container* da aplicação web do serviço de autenticação.

Há, portanto, outro *container* que contém precisamente a imagem desse serviço, associado à porta 4000 e que está ligado à *network1* (que interliga à base de dados) e à *network2*, que comunica com o *container* que tem o serviço HTTP que validar o funcionamento do serviço de autenticação.

No ficheiro *dockercompose.yml* encontra-se ainda a criação do *container* com o server HTTP, associado à porta 4004 e ligado, como havia referido, à *network2* (comunicando com o serviço de autenticação). Este *container* tem ainda associado o volume para permanência de dados do ficheiro *log* que tem a informação de *login* de *users* para o administrador consultar, como referido na secção 2.2.

```
version: '3'

services:
  db:
    image: mongo
    container_name: mongodb
    volumes:
      - database:/home
    restart: always
    networks:
      - network1
  auth:
    image: joanafonsogomes/auth
    container_name: auth
    restart: always
    ports:
      - '4000:4000'
    networks:
      - network1
      - network2
  http:
    image: joanafonsogomes/http
    container_name: http
    volumes:
      - httpvolume:/httpvol
    ports:
      - '4004:4004'
    networks:
      - network2

volumes:
  database:
  httpvolume:

networks:
  network1:
  network2:
```

## DEMO

---

De seguida exibe-se uma pequena *demo* (substantialmente gráfica) dos dois serviços interligados após a criação com o *dockercompose*.

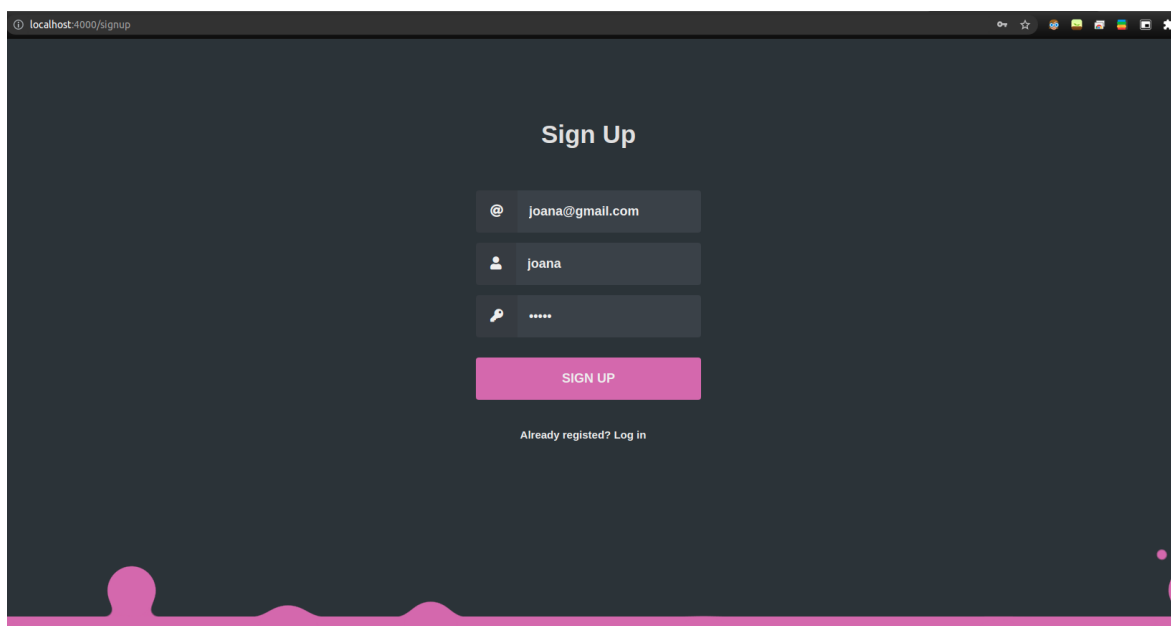


Figura 4.1: Registo de um utilizador (role *user*).

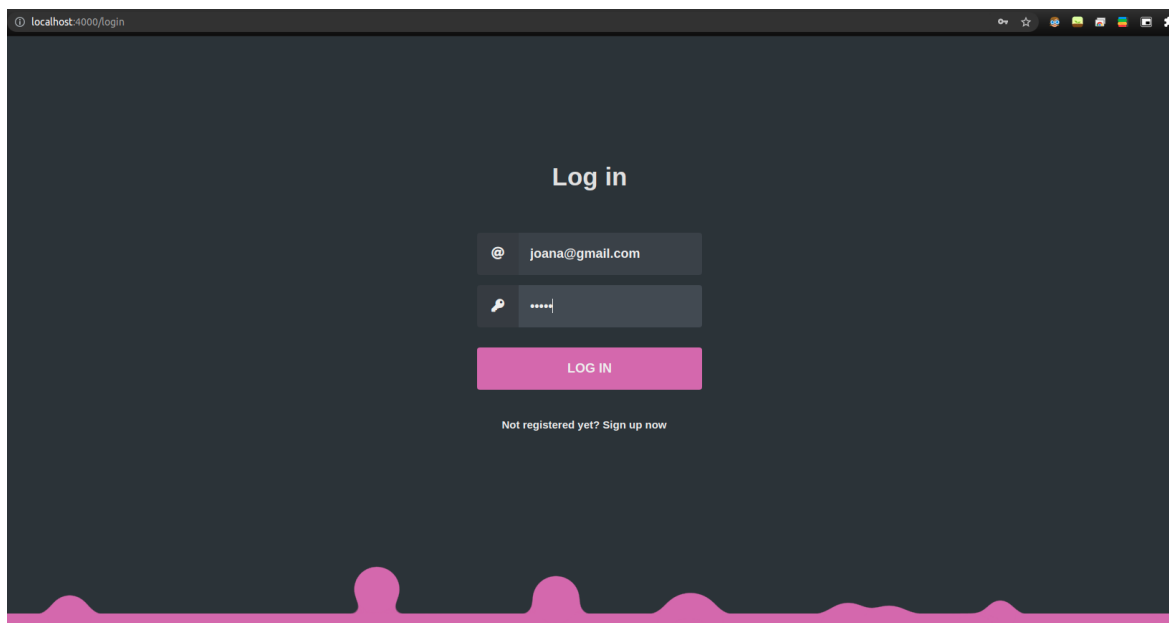


Figura 4.2: Utilizador efetua *login*.

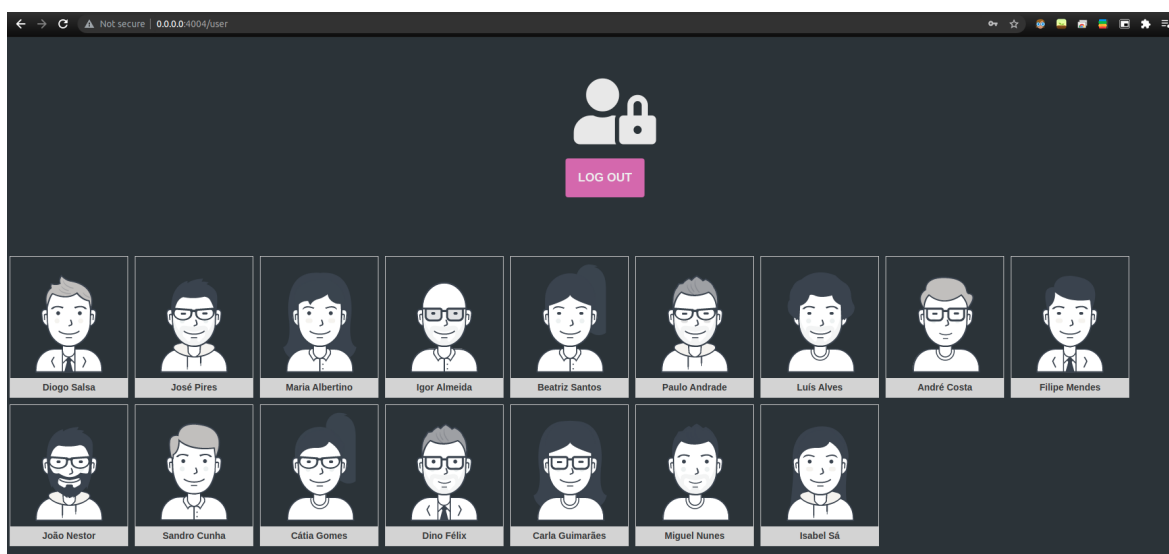


Figura 4.3: Página a que acede um utilizador com role *user*.

Se houver uma tentativa de redirecionar o *link* para, por exemplo, o *link* da página do administrador, é automaticamente redirecionado de volta para esta página.

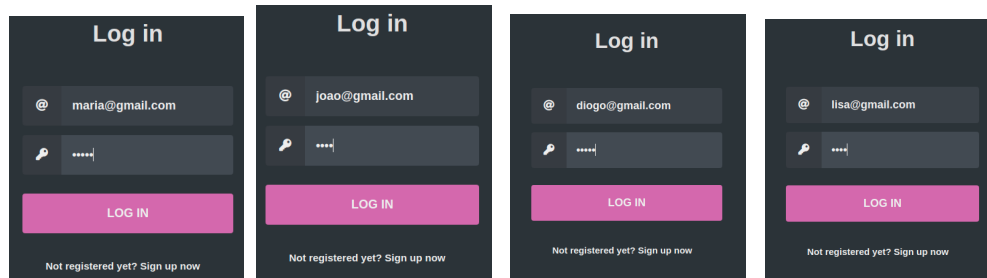


Figura 4.4: Se vários utilizador efetuarem *login*...

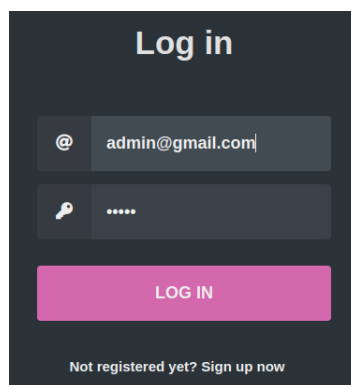


Figura 4.5: .. e após esses logins, o administrador acede à sua área...

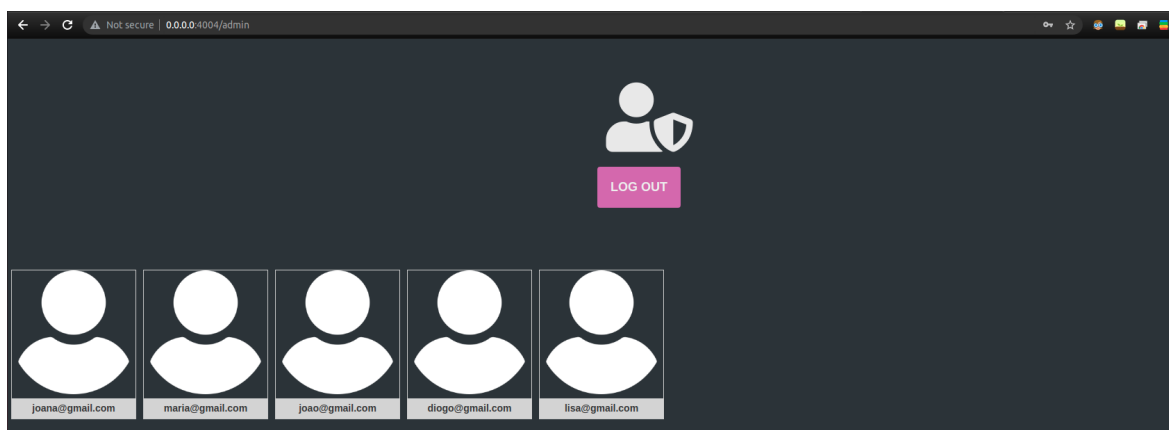


Figura 4.6: Área do administrador com o registo de todos os users que efetuaram *login*.



## CONCLUSÃO

---

Com este projeto foi possível aprofundar os meus conhecimentos no desenvolvimento de serviços com Docker.

Achei o contexto deste segundo trabalho bastante desafiante, tendo várias vertentes interessantes e interligadas ao tema principal de criar uma arquitetura de *Docker containers* com *microservices*.

O desenvolvimento de serviço de autenticação que redirecciona o utilizador, e também do serviço para o qual é redirecionado, ajudou imenso a prespetivar o contexto de um *login* eficaz e seguro.

Um projeto prático como este foi extremamente útil para entender o contexto de virtualização com Docker e tudo aquilo que lhe é adjacente, tendo me dado uma visão muito alargada e ideal de como funciona, concluindo portanto um desfecho muito positivo.