Sistemas de Computação na Cloud

Luís Ventuzelos, Ricardo Faria, João Cunha
Grupo 6
SCC, EST, Instituto Politécnico do Cávado e Ave,4750-810 Barcelos, Portugal

Abstract

In this report we will present an API service oriented to the cloud. This API only contains the relevant endpoints for a client registry and authentication of an application called Viter.
We split this report in 3 distinct parts.
In the first part of the report, we present the structure of the Dockerfile for our service, as well as the subsequent Docker-Compose file that accompanies it.
In the second part we will explain the main endpoints of our Authentication API and we will also explain the JWT Token implementation.
In the end we will expose a small introductory guide for a Kubernets cluster deployment using our API service.

Keywords: Docker, Docker-Compose, API, JWT, Token, Kubernets.

Resumo

Neste relatório apresentamos um serviço de *API* orientada à *cloud*. Esta *API* apenas contém *endpoints* relevantes ao registo e autenticação de utilizadores de uma aplicação chamada *Viter*.
Dividimos o relatório em 3 partes distintas.
Na primeira parte iremos apresentar a estrutura do *Dockerfile* do nosso serviço, assim como do ficheiro *Docker-Compose* que o acompanha.
Na segunda parte iremos explicar os principais *endpoints* da nossa *API* de Autenticação assim como a implementação do *JWT Token*.
Por fim iremos expor um pequeno guia de introdução a um *deployment* num cluster de *Kubernets*.

Palavras-chave: *Docker*, *Docker-Compose, API*, *JWT*, *Token*, *Kubernets*.

1. Introdução

Aplicações modernas requerem (tipicamente) soluções modernas.

Nos últimos anos a camada aplicacional da indústria tecnológica começou a transitar para aplicações *cloud native*. [1] A adoção de *containers* e de ferramentas como o *Docker* e *Kubernets* estão na linha da frente da migração das aplicações monólitos tradicionais para novas implementações mais orientadas a micro-serviços e *deployments* na *cloud*. [2] [3]

Atendendo ao contexto anteriormente mencionado é necessário dissecar e compreender como é possível, nas camadas tecnológicas atuais, criar e implementar uma aplicação orientada a um serviço nativo na *cloud*.

Através da análise deste trabalho é expectável que se obtenha uma visão de alto nível de como é possível criar um serviço de *Node.js* e, através da sua imagem usando o *Docker,* fazer o seu *deployment* com o auxílio de uma camada de base de dados usando o *Docker-Compose*.

Também iremos demonstrar como é possível transitar de uma simples aplicação local para um *deployment* num *cluster* de *Kubernets* num vendedor de *cloud* moderno.

2. Docker

2.1. Dockerfile

Um *Dockerfile* é um arquivo de texto simples que contém os comandos necessários para montar uma imagem de *Docker*.

Este ficheiro pode incluir a instalação de pacotes, criação de pastas e definição de variáveis de ambiente entre outras coisas.

2.1.1. FROM node:16

O *Dockerfile* usa como base a imagem do *Docker-Hub node:16*

2.1.2. WORKDIR /usr/src/app

Através do comando *WORKDIR* consideramos a pasta /usr/src/app como a nossa pasta principal da nossa imagem.

2.1.3. COPY package*.json./

Através do comando *COPY*, copiamos todas as referências de package*.json de maneira a conseguirmos instalar todas as dependências necessárias para correr a nossa *API*.

2.1.4. RUN npm install

O comando *RUN* vai correr o comando "*npm install*" que vai permitir instalar as dependências do *node* que estão referenciadas nos ficheiros *package*.json*.

2.1.5. COPY . /usr/src/app

De seguida corremos o comando *COPY* para copiar os restantes ficheiros da aplicação. A decisão de fazer os comandos *COPY* faseados foi tomada por uma questão de eficiência. Se alguma dependência do *node* der erro ao correr o comando "*npm install*" a *build* da imagem vai falhar antes de copiarmos os restantes ficheiros.

2.1.6. EXPOSE 3000

A instrução *EXPOSE*, expõe uma porta específica com um protocolo específico dentro do *Docker Container*. A instrução diz ao *Docker* para obter todas as informações necessárias, durante o tempo de execução, de uma porta específica. No caso da nossa imagem usaremos a porta 3000.

2.1.7. CMD ["npm", "start"]

Preparamos o comando *CMD* para que assim que o nosso *container* for criado execute o comando "*npm start*" de maneira a correr a nossa *API*.

2.2. Docker-Compose

2.2.1. version: "3.9"

Indica que estamos a utilizar a versão 3.9 do *Docker Compose*, desta forma o *Docker* fornece os recursos apropriados.

2.2.2. services

Esta seção define todos os diferentes *containers* que iremos criar.

2.2.3. api

Este será o nome do nosso 1º serviço. O *Docker Compose* criará um *container* com esse nome.

```
build: .
ports:
   - "8080:3000"
depends_on:
   - db
networks:
-       viternetwork
```

No parâmetro *build* definimos a localização do ficheiro *docker-compose.yml* em relação ao *Dockerfile*, de seguida em *ports* mapeamos as portas do *container* para a máquina *host* e por fim adicionamos o parâmetro *depends_on* para que este serviço inicie após o serviço definido for iniciado. Controlando

assim a ordem pela qual os serviços serão inicializados.

No serviço api mapeamos a porta 3000 do container para a porta 8080 da máquina *host*.

2.2.4. db

O 2º serviço será a nossa base de dados.

```
image: "mongo"
restart: always
ports:
   - "5432:27017"
env_file:
   - .env
volumes:
   - ./scripts/mongo/init/:/docker-entrypoint-initdb.d
   - ./scripts/mongo/init:/home/mongodb
   - ./scripts/mongo/seed/:/home/mongodb/seed
   - ./data/db:/data/db
networks:
-       viternetwork
```

No parâmetro *image* definimos uma imagem pré-construída do *mongo* que será utilizada como um serviço de base de dados, no nosso caso *Mongo* (esta imagem está armazenada no *Docker-Hub*). O *Docker Compose* irá criar um container com uma cópia dessa imagem.

Adicionamos a instrução de "*restart: always"* de maneira que sempre que o nosso *container* parar seja, logo de seguida, inicializado fazendo com que o *container* que contém a base de dados se mantenha sempre ativo.

Em *ports* mapeamos as portas do *container* expondo a porta 5432 na máquina *host*.

Em *env_file* definimos a localização do ficheiro com as variáveis de ambiente que queremos passar para o serviço (*usernames*, *passwords*, conexão a base de dados, etc…).

Por fim, definimos *volumes*, de maneira a termos os dados principais persistidos na nossa máquina *host*. Desta forma, não teremos que reconstruir a base de dados sempre que for necessário realizar alterações. Os primeiros 3 volumes foram criados de modo a facilitar a criação de *mock data* na base de dados, sendo o último volume usado para armazenar todos os dados do *MongoDB*.

2.2.5. network

Definimos também no nosso *Dockerfile* uma *network* de maneira a interligar os nossos *containers*.

```
networks:
  viternetwork:
    driver : bridge
```

A *network* usada neste trabalho é do tipo *bridge,* o que permite que *containers* conectados à mesma network *bridge* comuniquem entre si, permitindo assim isolamento relativamente aos *containers* que não se encontram ligados aquela *bridge*.

O driver *bridge* instala automaticamente regras na máquina *host* para que os *containers* em diferentes *bridge networks* não possam comunicar diretamente entre si.

2.3. Mock Data

De maneira a facilitar a criação de *mock data* utilizamos um pequeno *Shell script* que irá correr sempre que o utilizador desejar.

```
#!/bin/bash
mongo import -- host $MONGO_HOSTNAME
--authenticationDataBase %DB_NAME
--username $APP_USER
--password $APP_PWD
--db $DB_NAME
--collection $DB_COLLECTION_NAME
--file init.json --jsonArray
```

Através deste script é possível importar um ficheiro *json*, usando o comando *mongoimport,* que por sua vez irá utilizar o ficheiro *init.json*, facilitando assim a introdução de dados na nossa base de dados.

```
  }
    "name": "Joe Smith",
    "email": "smith@gmail.com",
    "password": "$2a$10$FLlbUkQc5vhSnfCKq4KhheYCljy/TalnfSs3CJC.axTAdLNZo2Fdu",
    "role": "user"
  },
  {
    "name": "Jen Ford",
    "email": "jford@gmail.com",
    "password": "$2a$10$FLlbUkQc5vhSnfCKq4KhheYCljy/TalnfSs3CJC.axTAdLNZo2Fdu",
    "role": "user"
  }
  {
    "name": "John Doe",
    "email": "jDoe@hotmail.com",
    "password": "$2a$10$mvxl9Aa6FvNowIjRHahLguw8LpxuGk/wDh5bRNSo8OiltJWRV1Z62",
    "role": "admin"
  }
```

3. API

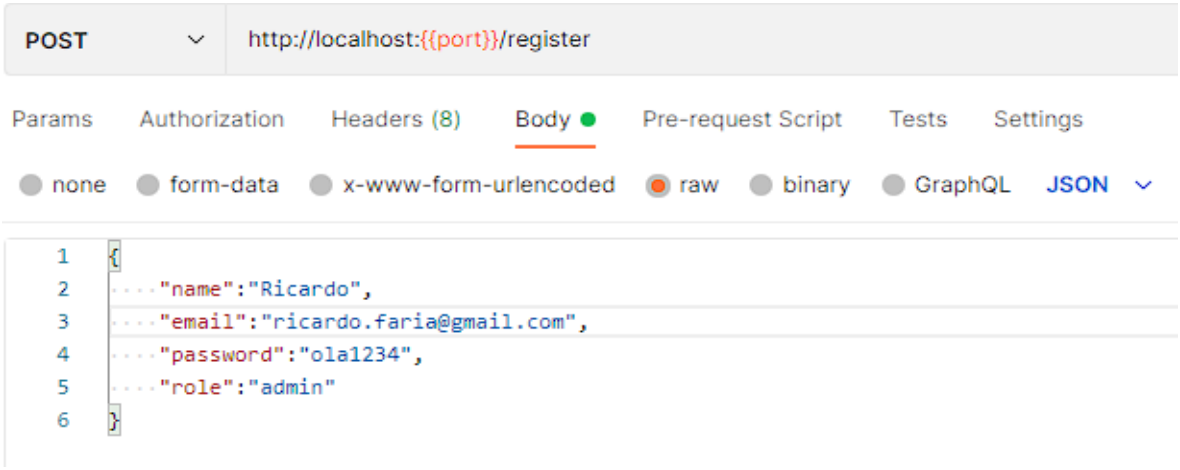
Para desenvolver a nossa aplicação utilizamos o *Node.js*, que é uma *"runtime framework"* em *Javascript*. Utilizamos também a *framework Express* para *Node.js.* O *Express* é utilizado para facilitar a criação de aplicações *web* e *APIs REST* de forma rápida e eficiente.

3.1. *EndPoints*

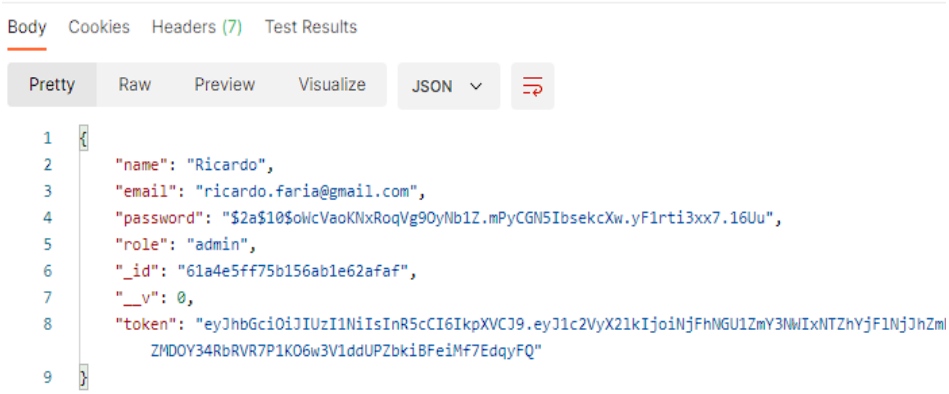
Como não implementamos *front-end*, utilizamos, para testar a *API* criada, a ferramenta *Postman* e será a partir dela que vamos retirar as capturas de ecrã que provam o funcionamento do nosso serviço.

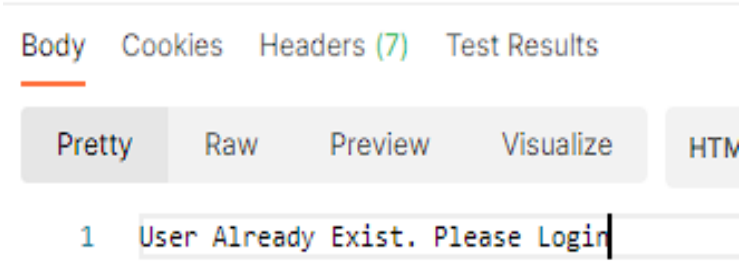
3.1.1. *Register*

Para registar um utilizador é feito um *POST Request* para o *endpoint* ***register*** com o *body* contendo o *name*, *email*, *password* e *role.* Este último campo servirá para conceder privilégios ao utilizador, pois existem *endpoints* que apenas os utilizadores "*admin*" podem executar.



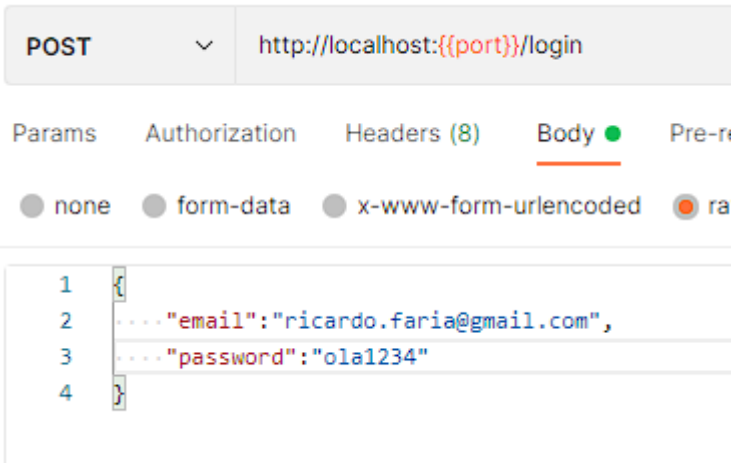
Como resultado poderemos ter 2 respostas. A primeira é confirmação que o utilizador foi registado, retornando no pedido os dados que foram introduzidos na base de dados assim como o *token (jwt),* que será utilizado em futuras chamadas à *API.* A segunda é quando o utilizador já existe e, portanto, recebemos essa indicação.





3.1.2. *Login*

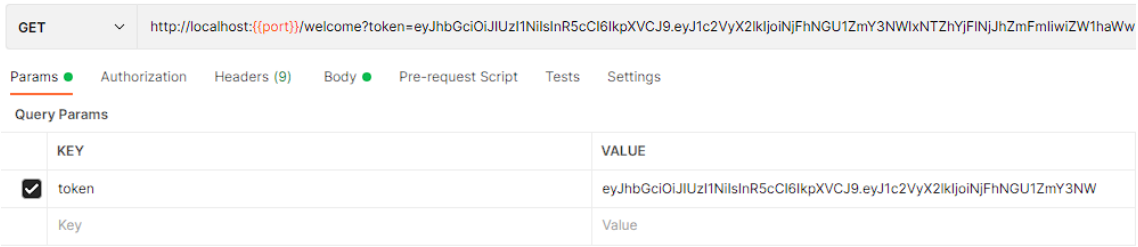
Para efetuar o login é feito um *POST Request* para o *endpoint* ***login*** com o *body* contendo o *email* e *password.*



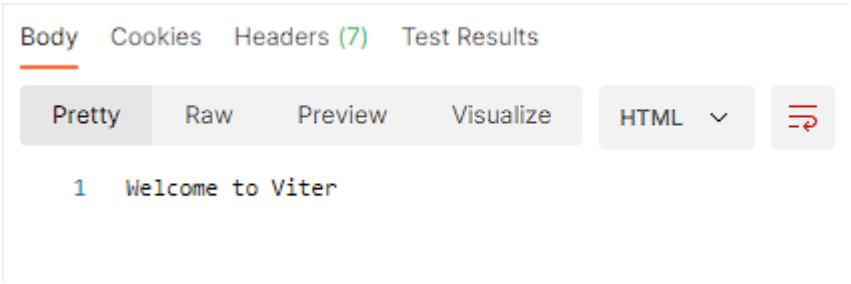
Como resultado obtemos a informação referente ao utilizador, o *token (jwt)* e também um outro token (*refreshToken*) que é guardado pelo serviço e será utilizado para obter novos *tokens(jwt).* Isto faz com que tenhamos outra forma para obter novos *tokens,* sem o utilizador precisar de fazer *login* novamente.

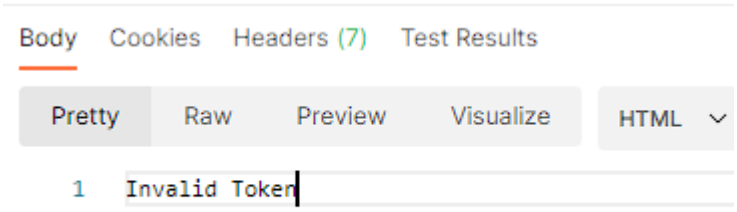
3.1.3. *Welcome*

Para testar se o *token(jwt)* obtido está ativo é feito um *GET Request* para o *endpoint* ***welcome***.



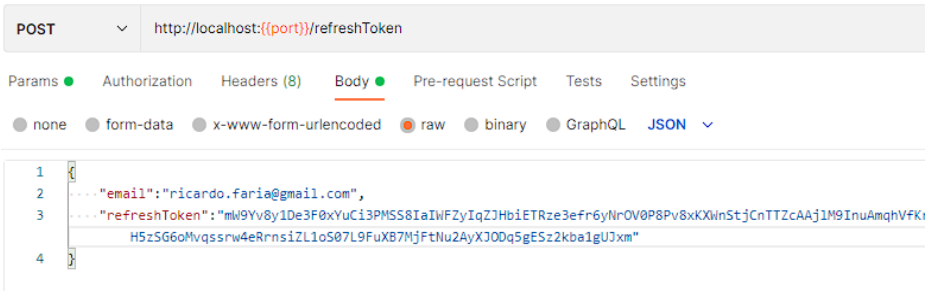
Como resultado obtemos uma pequena mensagem de boas-vindas caso o *token(jwt)* estiver válido, ou então uma mensagem a informar que não está válido.





3.1.4. *Refresh Token*

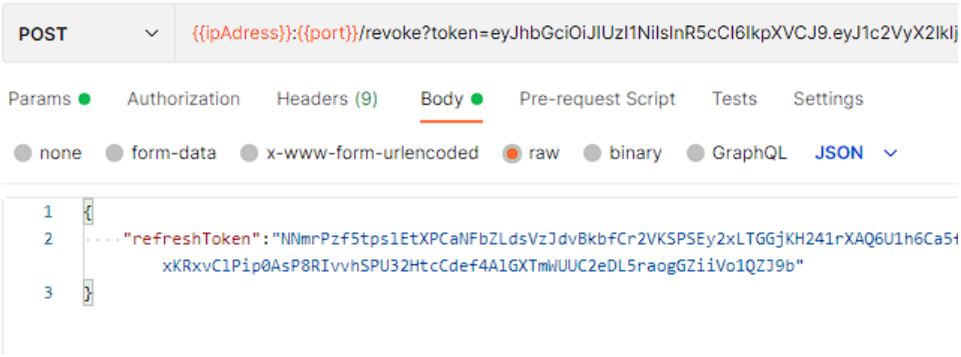
Para efetuar o pedido de novo *token(jwt)* é feito um *POST Request* para o *endpoint* ***refreshToken*** com o *body* contendo o *email* e *refreshToken* (obtido no login). Este *endpoint* serve para termos uma outra forma de obter novos *tokens(jwt)* para além do *login.*

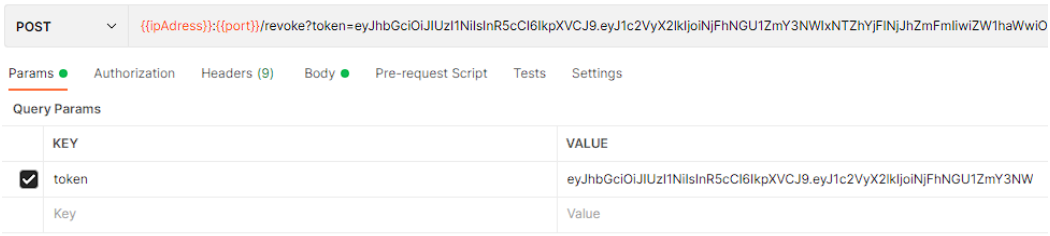


Como resultado obtemos a informação do utilizador e o novo *token(jwt).*

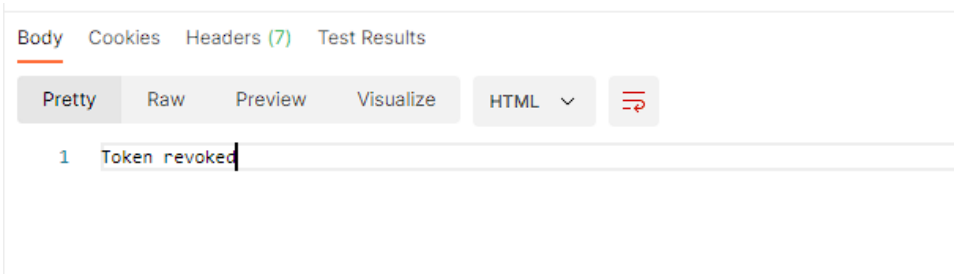
3.1.5. *Revoke*

Para invalidar o *refreshToken* é feito um *POST Request* para o *endpoint* **revoke** com o *body* contendo o *refreshtoken*. Juntamente com esse body é enviado um *token(jwt)* válido utilizado também para verificar se o utilizador tem permissões para executar o pedido.



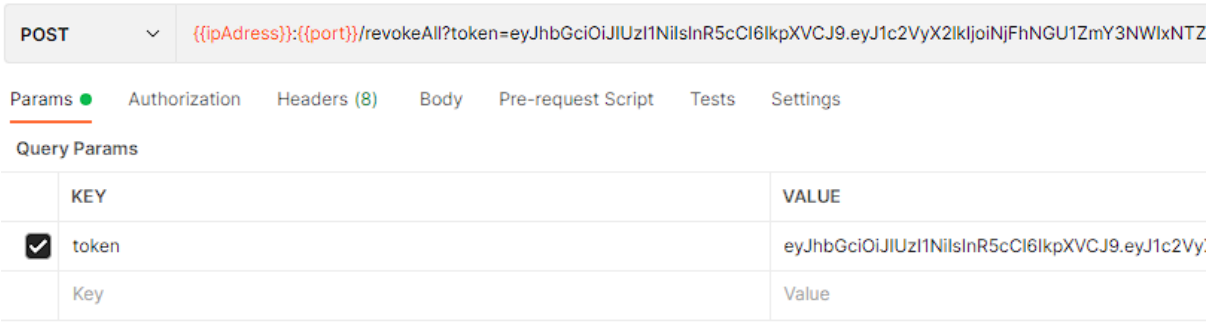


Como resultado obtemos a mensagem de confirmação da invalidação do *refreshToken.*

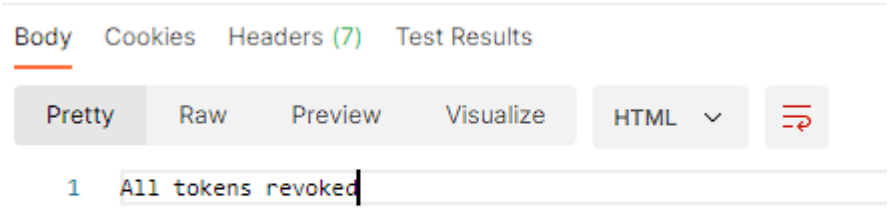


3.1.6. *Revoke All*

Para invalidar todos os *refreshTokens* é feito um *POST* para o *endpoint* **revokeAll**. Juntamente é enviado um *token(jwt) que* efectua as mesmas verificações como no pedido anterior.

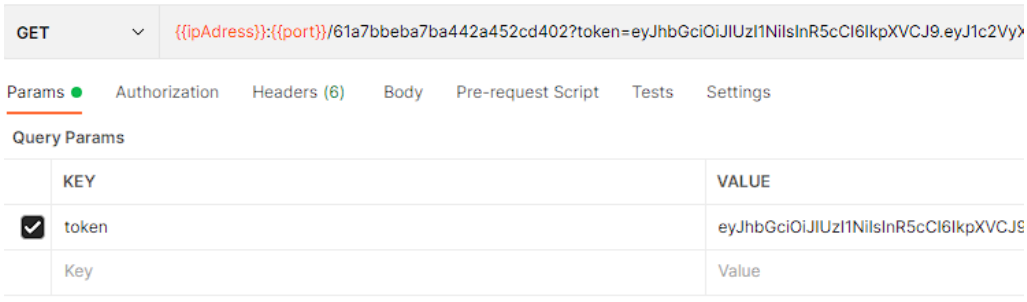


Como resultado obtemos a mensagem de confirmação da invalidação de todos os *refreshTokens*.

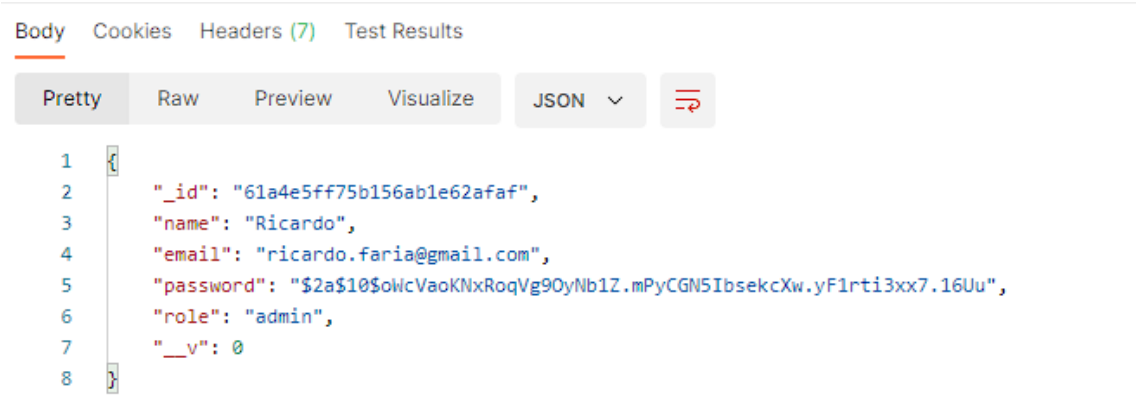


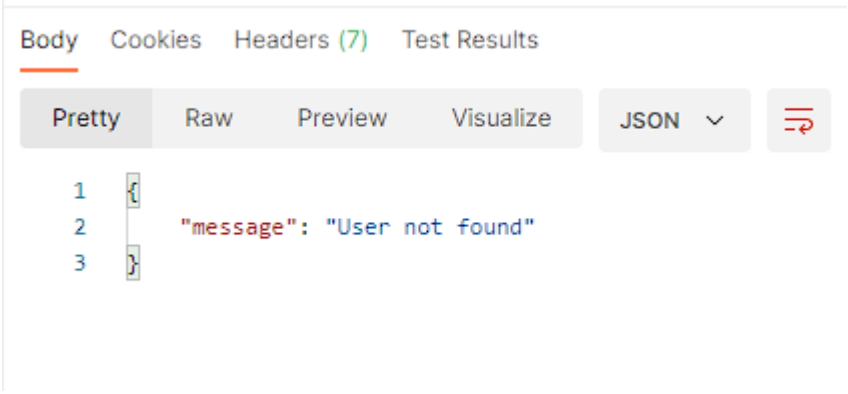
3.1.7. *Find user*

Para obtermos a informação de um utilizador é feito um *GET Request* para o *endpoint* que está à espera de parâmetro ***id*** (/:id). Este *endpoint* é um pouco diferente dos anteriores pois utiliza como parâmetro a informação enviada no *URL*, neste caso utiliza o valor após "/" como *id.* Juntamente é enviado um *token(jwt)* válido utilizado também para verificar se o utilizador tem permissões para executar o pedido.



Como resultado poderemos ter 2 respostas. A primeira é informação do utilizador. A segunda é quando o utilizador não existe.





3.1.8. *Roles*

Nesta fase estamos a usar o parâmetro *role* dos utilizadores, para verificar se esse utilizador pode executar determinadas chamadas à *API*. Apenas os utilizadores com o *role* de administrador (‘*admin’*) podem realizar esses pedidos (p.e.: *revoke* and *refresh tokens*).

*4. **Kubernets***

O *deployment* orientado a um cluster de *Kubernets* não era obrigatório para a conclusão com sucesso deste trabalho, mas consideramos que é um dos pontos mais interessantes do mesmo. As arquiteturas modernas orientadas a aplicações *cloud native* pedem cada vez mais um *deployment* rápido, eficaz e seguindo as melhores práticas de uma aplicação *cloud native* e o *Kubernets* está na frente desses princípios. [4]

4.1. Ferramenta *Kompose*

De maneira a facilitar o uso do *Kubernets* decidimos usar a ferramenta ***Kompose*** [5]. Através desta ferramenta é possível converter o nosso ficheiro ***docker-compose-yaml*** em ficheiros auxiliares (todos com a extensão *YAML*) que permitem facilitar o *deployment* num cluster de *Kubernets*. Não iremos proceder a uma explicação muito detalhada do processo de converter o Docker-Compose em ficheiros compatíveis com o *Kubernets*, preferindo antes mostrar, de uma maneira mais abstrata, o que cada um desses ficheiros permite fazer.

4.2. Ficheiros de Auxílio a um *deployment* de *Kubernets*

Após executar o comando *Kompose* procedemos à reestruturação dos ficheiros gerados por essa ferramenta (alguma *metadata* desnecessária teve de ser limpa).

A nossa estrutura de ficheiros será a seguinte:

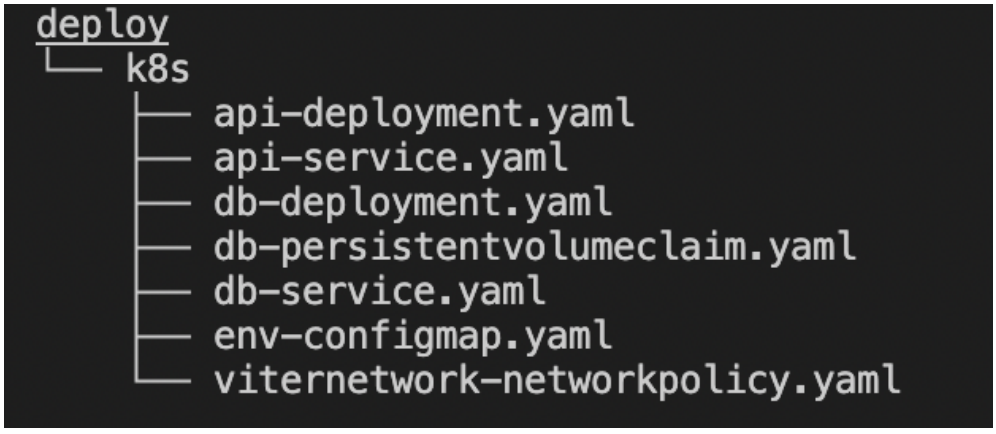


Figura 1 - Estrutura dos ficheiros de deployment

4.2.1. *api-deployment*

Este ficheiro permite declarar o estado desejado do nosso serviço *API* [6]. É neste ficheiro que declaramos que imagem os nossos *pods* vai utilizar e quantas *replicaSets* pretendemos para o nossos deployment.

No caso do nosso serviço de *API* definimos que iremos ter a correr, a qualquer momento, 3 *pods* em simultâneo.

Sempre que quisermos fazer uma atualização no nosso serviço, por exemplo aumentar o número de *pods*, só precisamos de alterar este ficheiro e o usar o comando "*kubectl apply"*.

4.2.2. *api-service*

Este ficheiro permite abstrair a nossa aplicação para correr nos *pods*, anteriormente definidos, para o nosso serviço de *API*. Aqui definimos o mapeamento de portas dos *Pods* para o nosso Cluster de *Kubernets*. [7]

4.2.3. *db-deployment*

Este ficheiro é similar ao ficheiro de *api-deployment* com a única diferença é que no caso da base de dados apenas estamos a criar um *pod*.

Não queremos ter mais que um *pod* da base de dados pois pode gerar inconsistência de dados.

4.2.4. *db-persistentvolumeclaim*

Um *Persistence Volume* [8] é um fragmento de armazenamento que pode ser dinâmico ou provisionado pelo administrador do *Cluster*.

O *Persitence Volume Claim* é um pedido de armazenamento efetuado pelo utilizador.

No caso da nossa aplicação, são efetuados 4 pedidos de *Pertence Volume* que depois vão ser "consumidos" pela nossa base de dados.

Podemos ver isso no ficheiro *YAML* de deployment da Base de Dados:

```
volumes:
  - name: db-claim0
    persistentVolumeClaim:
      claimName: db-claim0
  - name: db-claim1
    persistentVolumeClaim:
      claimName: db-claim1
  - name: db-claim2
    persistentVolumeClaim:
      claimName: db-claim2
  - name: db-claim3
    persistentVolumeClaim:
      claimName: db-claim3
```

Figura 2 - Ficheiro db-deployment

4.2.5. *db-service*

Abstração do nosso serviço de base de dados. O seu funcionamento é similar ao *api-service*.

4.2.6. *env-configmap*

Este ficheiro é de simples compreensão. Ele vai permitir armazenar no cluster variáveis de ambiente que irão ser consumidas pelos nossos *pods*.

É aqui que iremos definir as nossas *passwords* da base de dados e a *connection string* para a nossa *API* se conectar a essa mesma.

Numa implementação real a maioria destes valores deveriam ser guardados em *secrets*, e o *Kubernets* permite isso, mas atendendo que esta parte do trabalho é só um *proof of concept* achamos que não seria necessário explorar esse caminho. [9]

4.2.7. *viternetwork-networkpolicy*

Este ficheiro permite abstrair a maneira como as nossas *pods* podem comunicar com outras entidades na rede. Essas permissões são dadas explicitamente aos nossos *pods* através deste ficheiro. [10]

4.3. *Minikube Deployment*

De maneira a testarmos que os nossos ficheiros de *deployment* funcionam corretamente usamos o *minikube* [11] para criar um cluster local.

É necessário instalar na máquina host as seguintes ferramentas:

- Minikube
- Kubectl
- Docker

O deployment no *Minikube* é simples e pode ser efetuado em 4 simples passos.

4.3.1. Inicialização do *Cluster* local

Iniciamos o nosso cluster Local com o comando "*minikube start*".

```
luisventuzelos@Luiss-MBP SCC % minikube start
😄  minikube v1.24.0 on Darwin 12.0.1 (arm64)
✨  Automatically selected the docker driver
👍  Starting control plane node minikube in cluster minikube
🚜  Pulling base image ...
🔥  Creating docker container (CPUs=2, Memory=1988MB) ...
🐳  Preparing Kubernetes v1.22.3 on Docker 20.10.8 ...
    ▪ Generating certificates and keys ...
    ▪ Booting up control plane ...
    ▪ Configuring RBAC rules ...
🔎  Verifying Kubernetes components...
    ▪ Using image gcr.io/k8s-minikube/storage-provisioner:v5
🌟  Enabled addons: storage-provisioner, default-storageclass
🏄  Done! kubectl is now configured to use "minikube" cluster and "default" namespace by default
```

Figura 3 - Inicialização do nosso cluster local

Executando o comando *"kubectl get nodes"* podemos verificar que o nosso cluster está pronto.

```
luisventuzelos@Luiss-MBP SCC % kubectl get nodes
NAME       STATUS   ROLES                  AGE    VERSION
minikube   Ready    control-plane,master   112s   v1.22.3
```

Figura 4 - Estado do nosso node

4.3.2. Aplicar os ficheiros de *deployment*

Para fazer *deploy* do nosso serviço temos de executar o comando *"kubectl apply -f deploy/k8s/"* em que *deploy/k8s/* é o caminho dos nossos ficheiros *YAML* anteriormente explicados.

Se tudo correr como esperado a seguinte mensagem tem de aparecer no terminal:

```
luisventuzelos@Luiss-MBP SCC % kubectl apply -f deploy/k8s/
deployment.apps/api created
service/api created
deployment.apps/db created
persistentvolumeclaim/db-claim0 created
persistentvolumeclaim/db-claim1 created
persistentvolumeclaim/db-claim2 created
persistentvolumeclaim/db-claim3 created
service/db created
configmap/env created
networkpolicy.networking.k8s.io/apinetwork created
```

Figura 5 - Deployment de Kubernets local

4.3.3. Verificar estado do serviço

De maneira a verificar que o nosso serviço está a correr devemos correr o comando *"kubectl get pods"*. Este comando irá listar todos os *pods* que estão a correr no nosso cluster.

```
luisventuzelos@Luiss-MBP SCC % kubectl get pods
NAME                   READY   STATUS    RESTARTS   AGE
api-688bd697d6-4v8nz   1/1     Running   0          70s
api-688bd697d6-dv5zl   1/1     Running   0          70s
api-688bd697d6-kkwmt   1/1     Running   0          70s
db-5b8965c5bf-b8mdt    1/1     Running   0          70s
```

Figura 6 - Estado dos nossos pods

4.3.4. Expor e testar o nosso serviço

Para testar o nosso serviço temos de primeiro expor a nossa API usando o comando *"minikube service --url api"*

```
luisventuzelos@Luiss-MBP SCC % minikube service --url api
😿  service default/api has no node port
🏃  Starting tunnel for service api.
|-----------|------|-------------|------------------------|
| NAMESPACE | NAME | TARGET PORT |          URL           |
|-----------|------|-------------|------------------------|
| default   | api  |             | http://127.0.0.1:49958 |
|-----------|------|-------------|------------------------|
http://127.0.0.1:49958
❗  Because you are using a Docker driver on darwin, the terminal needs to be open to run it.
```

Figura 7 - Serviço da nossa API exposto na porta 49958

Depois de expor o nosso serviço de *API*, fazemos um *POST Request* usando a porta definida pelo *minikube*.

No caso do nosso exemplo a porta é a 49958.

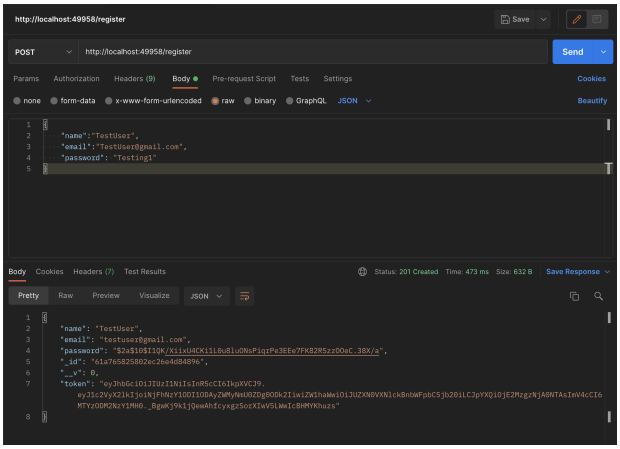


Figura 8 - Exemplo de pedido com sucesso ao nosso cluster

4.4. *Azure Deployment*

O *deployment* num *cluster* moderno traz consigo algumas dificuldades. A primeira é que tipicamente os *clusters* de *Kubernets* na *cloud* têm custos associados. Outra dificuldade é a de construir um cluster na *cloud*. Os *cloud providers* mais usados como o *Azure*, a *AWS* e o *GCP* já abstraem muita da configuração, mas mesmo assim pode ser complicado a sua configuração.

4.4.1. Criar o *Cluster* de *Kubernets*

Para criar um *cluster* de *Kubernets* usamos o *website* do *Azure* e configuramos tudo a partir de lá. Não iremos especificar como configuramos o cluster pois este passo pode ser diferente consoante o *cloud provider*.

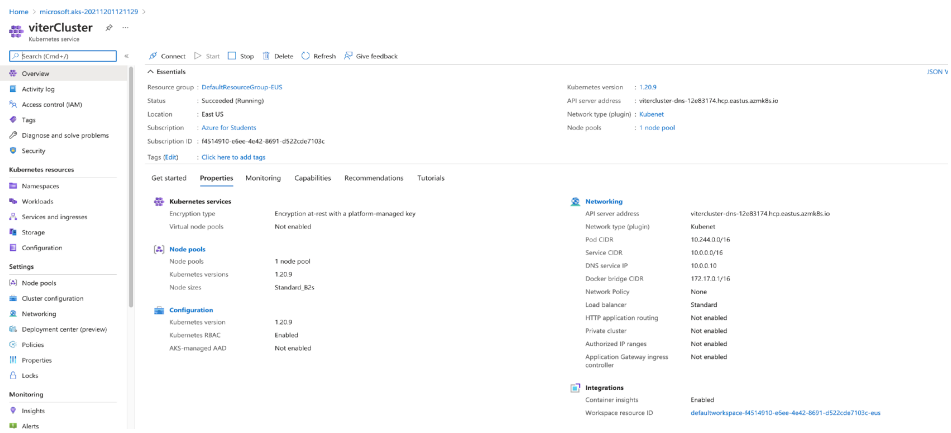


Figura 9 - Cluster de Kubernets no Azure

4.4.2. Conectar com o nosso Cluster

Para conectar com o cluster de *Kubernets* no *Azure* temos de instalar a ferramenta *Azure CLI* e conectar usando o comando *"az aks get-credentials --resource-group OUR_RESOURCE_GROUP --name OUR_CLUSTER"*.

```
luisventuzelos@Luiss-MBP SCC % az aks get-credentials --resource-group DefaultResourceGroup-EUS --name vi
terCluster
A different object named viterCluster already exists in your kubeconfig file.
Overwrite? (y/n): y
A different object named clusterUser_DefaultResourceGroup-EUS_viterCluster already exists in your kubecon
fig file.
Overwrite? (y/n): y
Merged "viterCluster" as current context in /Users/luisventuzelos/.kube/config
luisventuzelos@Luiss-MBP SCC %
```

Figura 10 - Conexão com sucesso ao nosso viterCluster

4.4.3. Fazer o *Deployment*

Depois de conectar com sucesso ao nosso cluster o *deployment* é igual ao *minikube*, é só executar o comando *"kubectl apply -f deploy/k8s/"*. Este passo é um dos grandes benefícios de abstrair o nosso deployment para ficheiros de configuração, facilmente podemos mudar de *cloud provider* e fazer um *deployment* com sucesso num *cluster*.

```
luisventuzelos@Luiss-MBP SCC % kubectl apply -f deploy/k8s/
deployment.apps/api created
service/api created
deployment.apps/db created
persistentvolumeclaim/db-claim0 created
persistentvolumeclaim/db-claim1 created
persistentvolumeclaim/db-claim2 created
persistentvolumeclaim/db-claim3 created
service/db created
configmap/env created
networkpolicy.networking.k8s.io/apinetwork created
```

Figura 11 - Ficheiros de deployment executados com sucesso

4.4.4. Expor o nosso serviço usando um *load balancer*

Para expor o nosso serviço vamos usar um *load balancer*. O comando é muito simples pois o *Kubernets* já fornece o sue próprio *load balancer out-of-the-box*, só precisamos de correr o comando *"kubectl expose deployment api --type=LoadBalancer --name=viter-load-balancer"*.

```
luisventuzelos@Luiss-MBP SCC % kubectl expose deployment api --type=LoadBalancer --name=viter-load-balancer
service/viter-load-balancer exposed
```

Figura 12 - Servieço de Load Balancer

4.4.5. Testar o nosso serviço

Correndo o comando *"kubectl get services"* já podemos ver o nosso serviço de *load balacing* a correr e nesse serviço é nos atribuído um IP para testar a nossa API.

```
[luisventuzelos@Luiss-MBP SCC % kubectl get services
NAME                TYPE           CLUSTER-IP     EXTERNAL-IP   PORT(S)          AGE
api                 ClusterIP      10.0.120.128   <none>        8080/TCP         69s
db                  ClusterIP      10.0.133.135   <none>        5432/TCP         68s
kubernetes          ClusterIP      10.0.0.1       <none>        443/TCP          6m50s
viter-load-balancer LoadBalancer   10.0.112.202   20.84.1.151   3000:32054/TCP   61s
```

Figura 13 - Cluster Services

Fazendo o *POST Request* de registo usando o IP e porta que nos foi atribuída podemos verificar que a nossa *API* está a correr com sucesso.

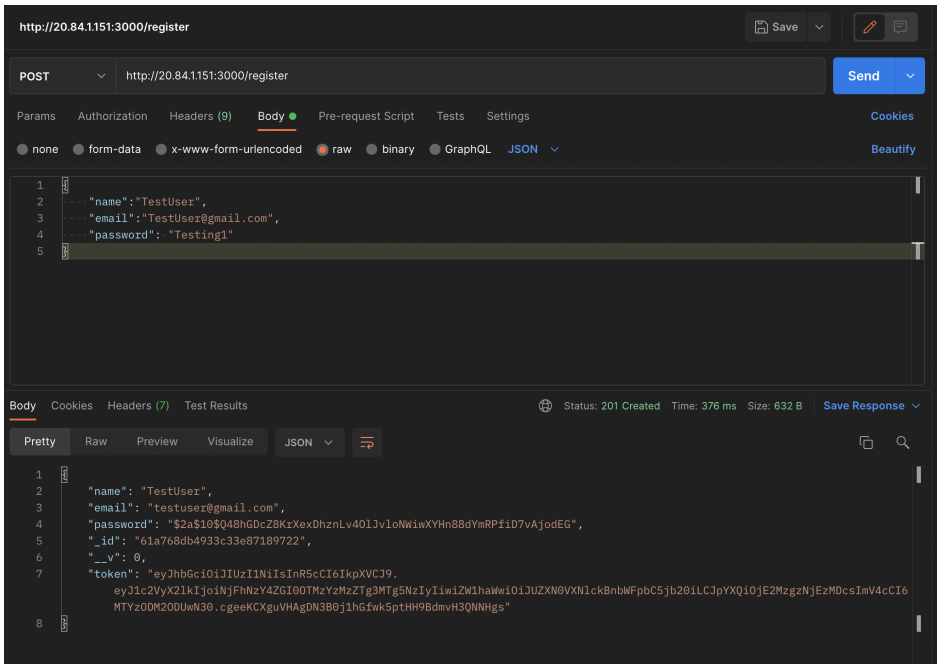


Figura 14 - Post Request com sucesso no Azure

5. Conclusão

A realização deste trabalho permitiu transitar a nossa aplicação por 3 cenários distintos.

O primeiro cenário foi a simples criação de uma *Docker image* que, e apesar de porventura ser o passo mais fácil de todo este projeto, traz consigo algumas decisões importantes que devem ser tomadas logo nesta fase, como por exemplo, não juntar as dependências do nosso serviço de *Node.js* com uma *base image* da base dados no mesmo *Dockerfile*. Ambos estes serviços têm de estar desacoplados pois caso não estejam podem vir a trazer sérios problemas aquando do *deployment* na *cloud* do nosso serviço (por exemplo problemas de sincronização dos dados da nossa base de dados).

De seguida temos o *deployment* local de um serviço que, no caso concreto da nossa aplicação, foi feito usando o *Docker Compose*. Este passo transferiu um grande valor ao grupo pois permitiu aplicar os conceitos teóricos das aulas e foi um pequeno passo introdutório à separação de serviços. Apesar de só ser requisitado uma *API* e uma base de dados para comunicar com essa *API* é de imediata compreensão como poderíamos, por exemplo, adicionar um novo serviço de *front-end* ao nosso projeto através do *Docker Compose*.

Por fim, o passo mais desafiante foi o *deployment* num *cluster* de *Kubernets*. Parte do trabalho foi atalhado com a utilização da ferramenta *Kompose,* mas consideramos mesmo assim que este segmento do projeto permitiu ao grupo enriquecer o seu conhecimento relativamente ao que tem sido a transição tecnológica para *deployments* na *cloud* e os seus principais benefícios (escalabilidade, rapidez, acesso ubíquo, resiliência, ...) assim como algumas das suas dificuldades (os ficheiros de configuração do *Kubernets* podem atingir uma complexidade elevada para utilizadores que não conheçam a tecnologia). Também acreditamos que esta breve introdução ao *Kubernets* ajuda no enriquecimento profissional pois trata-se de uma *trend* de mercado.

Gostaríamos também, de partilhar uma ferramenta que fomos utilizando durante a execução deste trabalho e que achamos relevante na execução do mesmo. A ferramenta explorada foi o *Okteto* (https://cloud.okteto.com). O *Okteto* permite criar gratuitamente um *cluster* para testes e o *deploy* é bastante simplificado pois é feito diretamente através do Github. Basta que o projeto presente no Github contenha o ficheiro "*docker-compose.yaml"* na raiz do projeto. Por ser uma versão gratuita apenas tínhamos 8gb de RAM, 4 CPUs e 5gb de espaço em disco. Tem também o inconveniente que o *Load Balancer e* os *Pods* são geridos automaticamente pela infraestrutura e até um máximo de 10 *Pods.*

6. Bibliografia

[1] IDG, “2020 Cloud Computing Study,” [Online]. Available: https://www.idg.com/tools-for-marketers/2020-cloud-computing-study/.

[2] RedHat, “Understanding cloud-native applications,” [Online]. Available: https://www.redhat.com/en/topics/cloud-native-apps.

[3] AWS, “What are Microservices?,” [Online]. Available: https://aws.amazon.com/microservices/.

[4] RedHat, “Why Kubernets native instead of cloud native?,” [Online]. Available: https://developers.redhat.com/blog/2020/04/08/why-kubernets-native-instead-of-cloud-native#.

[5] Kompose, “Kubernets + Compose = Kompose,” [Online]. Available: https://kompose.io/.

[6] Kubernets.io, “Deployments,” [Online]. Available: https://kubernets.io/docs/concepts/workloads/controllers/deployment/.

[7] Kubernets.io, “Services,” [Online]. Available: https://kubernets.io/docs/concepts/services-networking/service/.

[8] Kubernets.io, “Persistent Volumes,” [Online]. Available: https://kubernets.io/docs/concepts/storage/persistent-volumes/.

[9] Kubernets.io, “Secrets,” [Online]. Available: https://kubernets.io/docs/concepts/configuration/secret/.

[10] Kubernets.io, “Network Policies,” [Online]. Available: https://kubernets.io/docs/concepts/services-networking/network-policies/.

[11] Minikube, “https://minikube.sigs.k8s.io/docs/start/,” [Online]. Available: minikube start.