Case Técnico - DevOps
Julho 2023

Gabriel Antonio Fogaça Vieira

Acesso externo a aplicação: http://35.247.231.37.nip.io/

Projeto GitHub: https://github.com/Gasbrieu/GKE_Nginx

Tecnologias utilizadas:

Docker - Utilizei o docker para gerar uma imagem python possibilitando rodar o programa no cluster Kubernetes,

Python - Através da lib Flask e Gunicorn foi criado uma aplicação web simples retornando “Hello World!”,

Cloud Build - Utilizei arquivos yaml para configuração da imagem Docker através do Cloud Build para facilitar na atualização do projeto hospedado no GitHub,

Cloud Deploy (CI/CD) - Criei um pipeline de entrega contínua no Cloud Deploy utilizando dois clusters Kubernetes (Staging e Production),

Kubernetes - Criei dois cluster Kubernetes para utilizar no pipeline de CI/CD,

Nginx - Utilizei para gerenciar o tráfego no Cluster Kubernetes de VPC Privado;

Artifact Registry - Para armazenar a imagem Docker utilizada no projeto.

Inicialmente utilizaremos o git clone para clonar o repositório do projeto no Google Cloud Platform:

```
git clone https://github.com/Gasbrieu/GKE_Nginx
```

No projeto configuramos uma pasta no Artifact Registry para armazenar a imagem Docker:

```
gcloud artifacts repositories \
      create docker-main-app \
      --repository-format=docker \
      --location=southamerica-east1 \
      --description="Docker repository"
```

Através desse repositório vamos configurar o Cloud Build com a criação da imagem através do Dockerfile do projeto:

Para este projeto, criei um Dockerfile que utiliza uma imagem Python3, fiz referência a sub-pasta com a aplicação e a copiei para o contêiner, logo após indiquei os comandos a serem executados como instalação das dependências e criação do servidor web :

Após a configuração do Dockerfile podemos executar o comando:

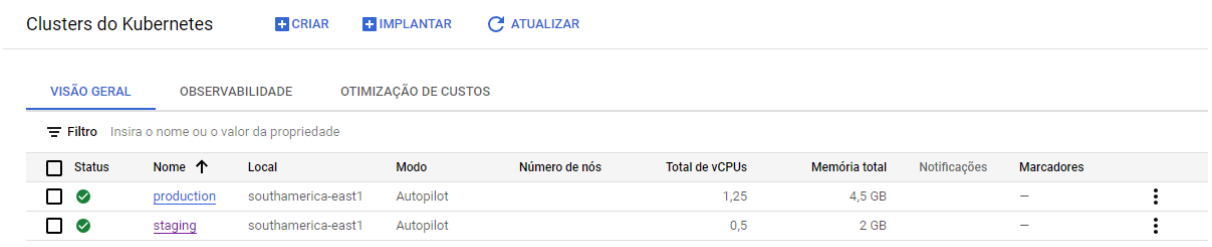
```
cd GKE_Nginx
gcloud builds submit
```

Com isso indicamos o acesso a pasta do projeto, e iniciamos o arquivo de configuração “cloudbuild.yaml”, fazendo o build do Dockerfile e salvando ele na pasta do Artifact Registry (Vale ressaltar que nesse primeiro momento possivelmente será necessário excluir as linhas 8-17 e nomear a versão manualmente).

Para criar os cluster segui o passo a passo presente na documentação do Google em relação ao Nginx e GKE para criação de regras para o firewall e IP, mas resumindo os comandos utilizados:

```
gcloud container clusters create-auto production
--zone=southamerica-east1 --enable-private-nodes
--master-ipv4-cidr=172.16.2.0/28
--enable-master-authorized-networks --master-authorized-networks
$CLOUDSHELL_IP/32

gcloud container clusters create-auto staging
--zone=southamerica-east1 --enable-private-nodes
--master-ipv4-cidr=172.16.2.0/28
--enable-master-authorized-networks --master-authorized-networks
$CLOUDSHELL_IP/32
```

Utilizei o Autopilot que é um serviço on-demand do Google Cloud para clusters Kubernetes.



Após a criação dos clusters, configuramos um manifesto Kubernetes para o deploy e serviço(vamos utilizar o “service” principalmente em staging, no cluster de produção a ideia é usar o Nginx como controlador de acessos) do nosso cluster:

k8s > ! deployment.yaml

```yaml
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello-world-app
spec:
  selector:
    matchLabels:
      app: hello-world-app
  template:
    metadata:
      labels:
        app: hello-world-app
    spec:
      containers:
      - name: hello-world-app
        image: hello-world-app
        ports:
          - containerPort: 8080

```

k8s > ! service.yaml

```yaml
---
apiVersion: v1
kind: Service
metadata:
  name: hello-world-service
spec:
  ports:
  - port: 80
    targetPort: 8080
  type: LoadBalancer
  selector:
    app: hello-world-app

```

Em seguida também configuramos o arquivo de Cloud Deploy, especificando como funcionará o pipeline e o nome dos clusters utilizados:

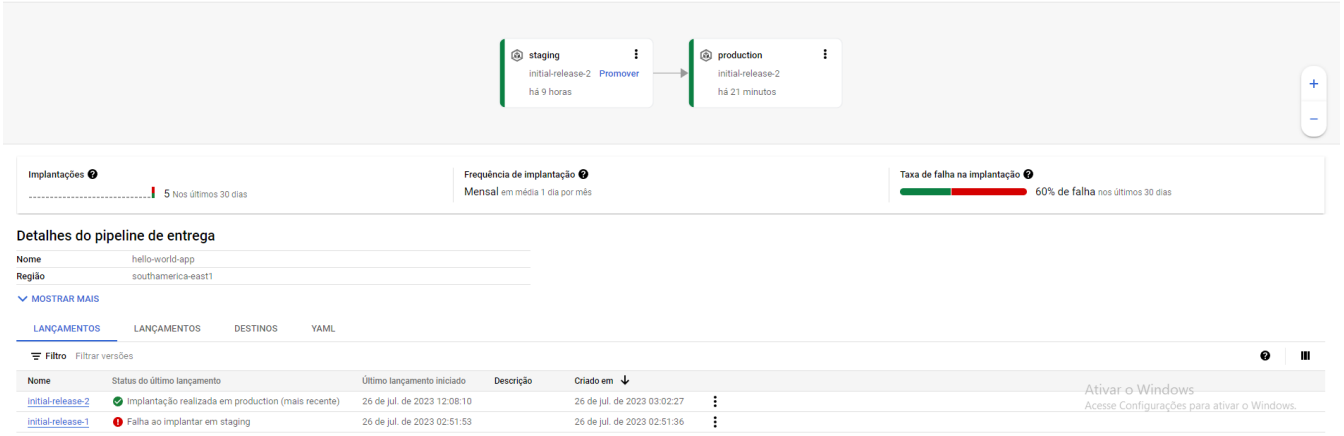
! clouddeploy.yaml

```yaml
---
apiVersion: deploy.cloud.google.com/v1beta1
kind: DeliveryPipeline
metadata:
  name: hello-world-app
description: Hello World Deployment Pipeline
serialPipeline:
  stages:
  - targetId: staging
  - targetId: production

---
apiVersion: deploy.cloud.google.com/v1beta1
kind: Target
metadata:
  name: staging
description: Staging Environment
gke:
  cluster: projects/devopssaocarlosmonks/locations/southamerica-east1/clusters/staging

---
apiVersion: deploy.cloud.google.com/v1beta1
kind: Target
metadata:
  name: production
description: Production Environment
gke:
  cluster: projects/devopssaocarlosmonks/locations/southamerica-east1/clusters/production


```

Definindo o pipeline, devemos executar o seguinte comando:

```
gcloud deploy releases create initial-release-1
--delivery-pipeline=hello-world-app
--region=southamerica-east1    --source=.
--images=hello-world-app=southamerica-east1-docker.pkg.dev/devopss
aocarlosmonks/docker-main-app/hello-world:v1
```

Vale ressaltar que o “v1” presente no campo “–images” corresponde a versão criada para a imagem no arquivo “cloudbuild.yaml”. Após a execução do comando acima vamos ter algo parecido com esse pipeline de entrega:



Após testar a aplicação em “staging” utilizamos o botão promover no pipeline para subir as alterações realizadas ao cluster “production” onde vamos configurar o acesso externo através do Nginx.

Acessando o cluster:

```
gcloud container clusters get-credentials production --zone
southamerica-east1
```

Iniciando o pod nginx-ingress:

```
helm install nginx-ingress ingress-nginx/ingress-nginx
```

Criando uma variável de ambiente que vamos utilizar no arquivo ingress-resource.yaml:

```
export NGINX_INGRESS_IP=$(kubectl get service
nginx-ingress-ingress-nginx-controller -ojson | jq -r
'.status.loadBalancer.ingress[].ip')
```

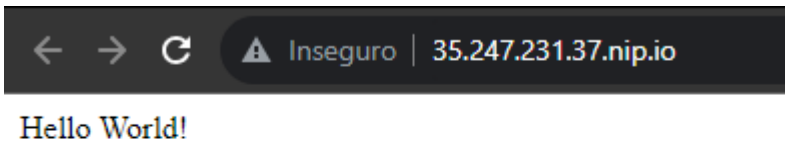
Criação do arquivo ingress-resource.yaml:

```
cat <<EOF > ingress-resource.yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ingress-resource
  annotations:
    kubernetes.io/ingress.class: "nginx"
    nginx.ingress.kubernetes.io/ssl-redirect: "false"
spec:
  rules:
  - host: "$NGINX_INGRESS_IP.nip.io"
    http:
      paths:
      - pathType: Prefix
        path: "/"
        backend:
          service:
            name: hello-world-service
            port:
              number: 8080
EOF
```

Aplicando as configurações definidas no arquivo:

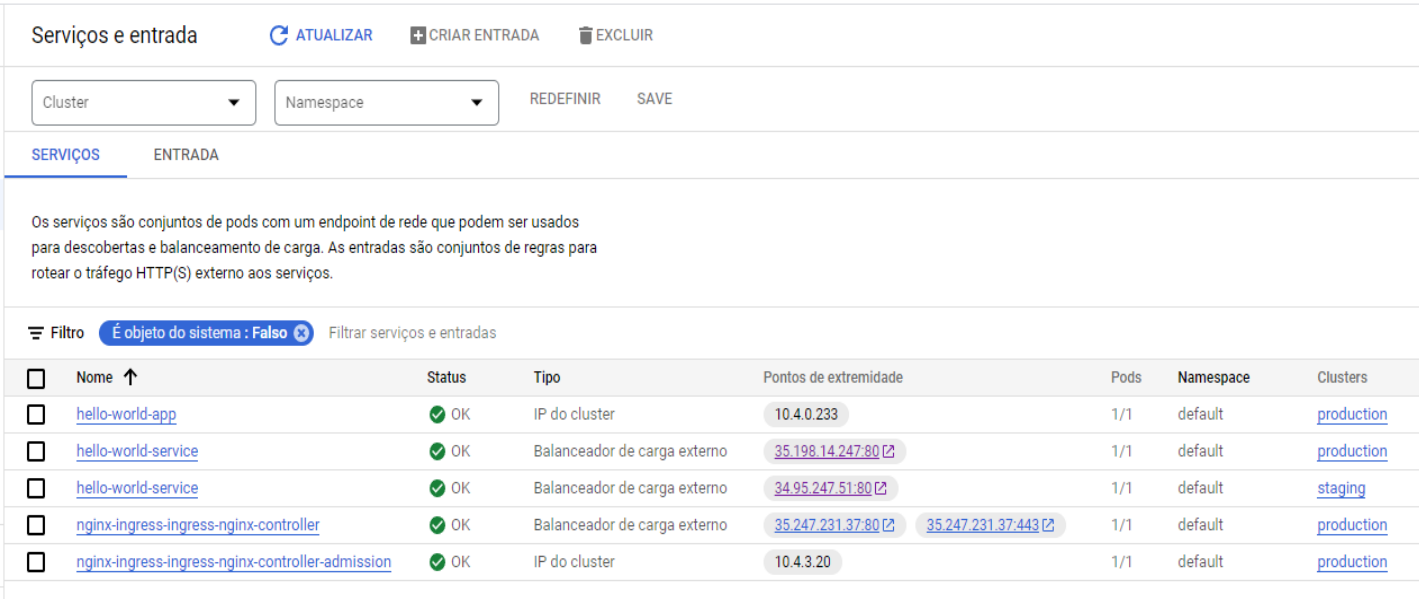
```
kubectl apply -f ingress-resource.yaml
```

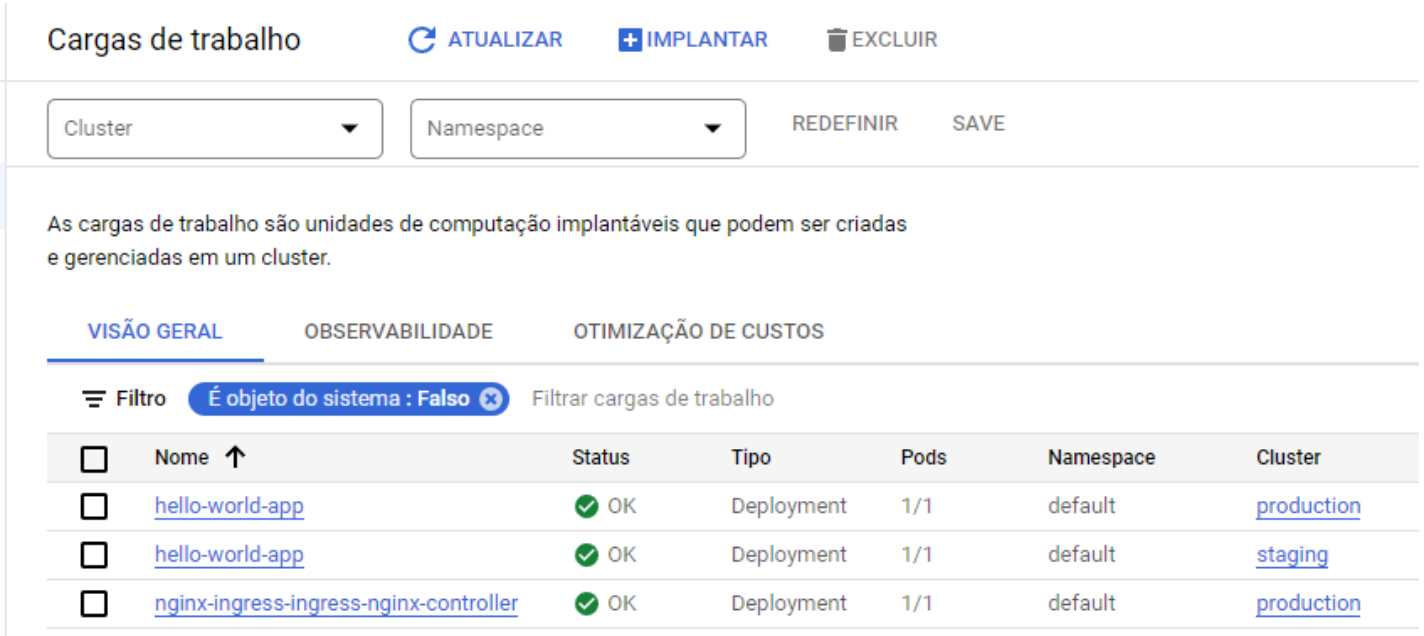
Para acessar a aplicação basta pegar o “host” presente no arquivo “ingress-resource.yaml”, segue um print de um acesso externo a aplicação rodando no cluster de VPC Privada criado nesse projeto:



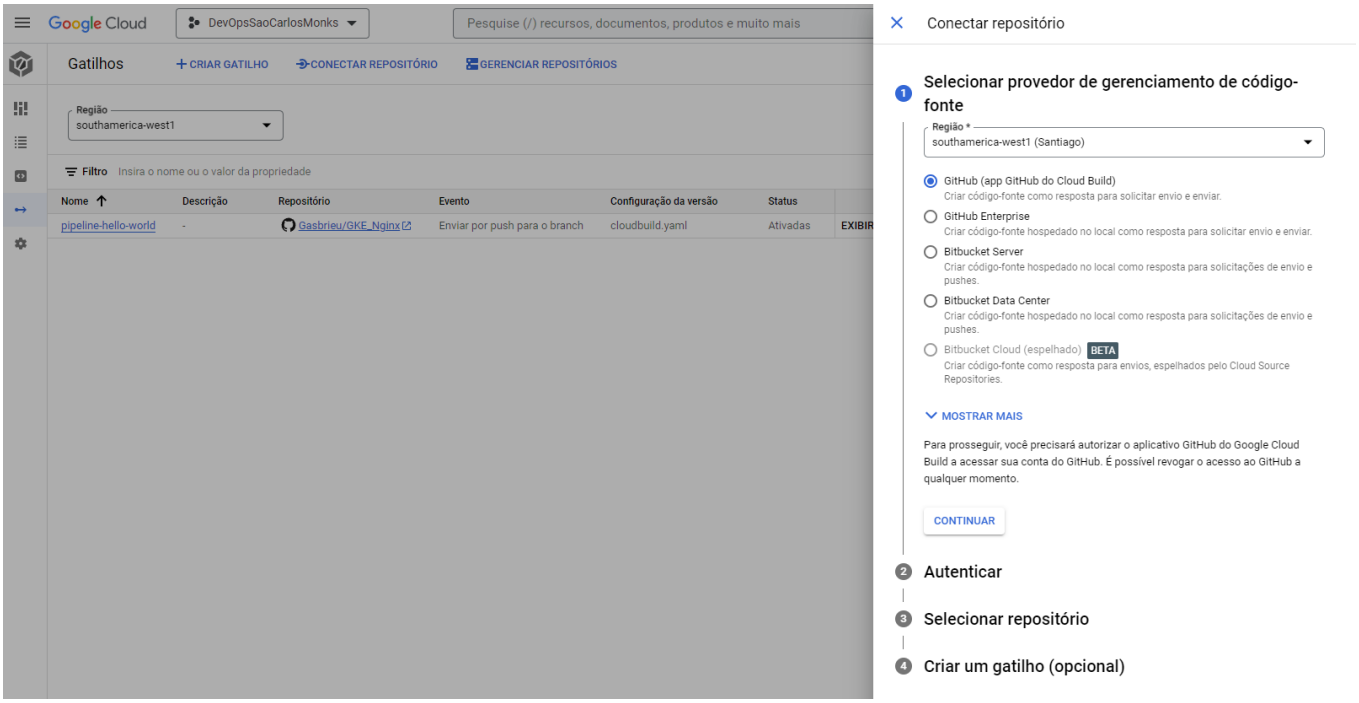
Hello World!

Segue os prints solicitados na documentação do case, vale ressaltar que utilizei a plataforma do GCP para facilitar a visualização, mas os pods podem ser listados com os comandos “kubectl get svc” e “kubectl get pod”:





Como último passo podemos criar um gatilho para atualizar o projeto sempre que um commit no github é enviado, para realizar essa etapa precisamos acessar a aba de gatilhos no Cloud Build > Conectar Repositório:



Realizando a autenticação e selecionando o repositório correto, por se tratar de um arquivo de configuração “cloudbuild.yaml” o gatilho vem como default já configurado.