Deploy de API utilizando Kubernetes

A melhor maneira de hospedar uma API é utilizando o **Kubernetes** ou **k8s**.

Kubernetes (**k8s**) é um ambiente extremamente complexo. Sua API é recheada de objetos e há várias maneiras de fazer a mesma coisa. Embora não seja objetivo do curso ensinar especificamente Kubernetes, eu preciso passar alguns conceitos básicos para que você não se perca.

Não se preocupe, pois o que precisa saber para fazer deploy de aplicações usando o k8s (Kubernetes) pode ser resumido em alguns conceitos básicos.

Para começar, temos um "Deployment". A API do Kubernetes nos permite interagir com ele basicamente criando objetos no Cluster. Os principais objetos são:

- **Pod**: O objeto básico do k8s (Kubernetes). Representa um processo rodando no Cluster. Um Pod contém um ou mais contêineres. Se um Pod possuir mais de um contêiner, eles compartilham os recursos do Pod;
- **Deployment**: Um objeto que agrega vários outros, como contêineres, por exemplo, que devem ser implantados e gerenciados em conjunto. O Deployment permite especificar a quantidade de réplicas dos Pods, permitindo criar um sistema de alta disponibilidade e escalabilidade;
- **StatefulSets**: Assim como o Deployment, agrega recursos (contêneres, Pods) e controla a ordenação e unicidade deles. Ele garante que os Pods não sejam alterados de modo a hospedar apps que precisam manter estado entre transações;
- **DaemonSet**: Distribui Pods entre os vários nós do Cluster, garantindo que os envolvidos sempre executem uma instância do Pod;
- **Service**: Expõe Pods como serviços de rede. Os Pods podem mudar de IP várias vezes, e há também o balanceamento de carga. Um Service abstrai isso para os clientes e permite que consigam encontrar o backend independentemente de como os Pods estão no momento;

Instalação do Kubernetes

A instalação do **k8s** é extremamente complexa e não vale a pena, se o objetivo for apenas aprendizado. Existem várias opções para desenvolvimento, como o **minikube**.

Minha recomendação é que instale o minikube em sua máquina e faça os exercícios.

O minikube pode ser instalado em **Windows**, **Linux** ou **MacOS**. Este material está sendo executado na versão Linux / Ubuntu.

Siga o **guia de instalação**. Infelizmente, está em inglês. Eu vou traduzir aqui a instalação.

Antes de prosseguir, certifique-se de que tenha instalado o **Docker** em sua máquina.

Instalação do minikube em Linux

Abra um terminal e execute os comandos:

```
curl -LO https://storage.googleapis.com/minikube/releases/latest/minikube-
linux-amd64
sudo install minikube-linux-amd64 /usr/local/bin/minikube
```

Inicie o **cluster k8s**:

```
minikube start
```

O **kubectl** é o principal utilitário para criar objetos no **cluster**. O minikube pode baixar para você com este comando:

```
minikube kubectl -- get po -A
```

Depois, você pode criar um **apelido** para ele:

```
alias kubectl="minikube kubectl --"
```

Agora, você pode utilizar o comando **kubectl** para criar objetos no seu cluster.

Instalação do minikube em MS Windows

Existe um **instalador para Windows**. Instale o minikube.

Inicie o **cluster k8s**:

```
minikube start
```

O **kubectl** é o principal utilitário para criar objetos no **cluster**. O minikube pode baixar para você com este comando:

```
minikube kubectl -- get po -A
```

Depois, você pode criar um **apelido** para ele:

```
alias kubectl="minikube kubectl --"
```

Agora, você pode utilizar o comando **kubectl** para criar objetos no seu cluster.

Instalação em MacOS

Se o seu computador for MacOS x86, execute os comandos abaixo:

```
curl -LO https://storage.googleapis.com/minikube/releases/latest/minikube-
darwin-amd64
sudo install minikube-darwin-amd64 /usr/local/bin/minikube
```

Agora, se for um MacOS ARM (**M1** ou superior), execute os comandos:

```
curl -LO https://storage.googleapis.com/minikube/releases/latest/minikube-
darwin-arm64
sudo install minikube-darwin-arm64 /usr/local/bin/minikube
```

Inicie o **cluster k8s**:

```
minikube start
```

O **kubectl** é o principal utilitário para criar objetos no **cluster**. O minikube pode baixar para você com este comando:

```
minikube kubectl -- get po -A
```

Depois, você pode criar um **apelido** para ele:

```
alias kubectl="minikube kubectl --"
```

Agora, você pode utilizar o comando **kubectl** para criar objetos no seu cluster.

Criação de objetos

Para fazer qualquer coisa no Minikube, precisamos do Kubectl (assim como no k8s de verdade). Bom, vamos criar um Pod, certo? Crie um arquivo chamado “simplepod.yml”, com este conteúdo:

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod
  labels:
    env: test
spec:
```

```
  containers:
  - name: nginx
    image: nginx
    imagePullPolicy: IfNotPresent
```

e execute o comando:

kubectl apply -f simplepod.yml

Vamos ver se o Pod está rodando?

kubectl get pod

```
NAME        READY   STATUS    RESTARTS   AGE
nginx-pod   1/1     Running   0          40s
```

Está sim. Podemos até abrir um shell para ele, como fazemos com os contêineres Docker:

```
kubectl exec -it  nginx-pod -- /bin/bash
```

Basta passarmos o nome do pod e, após os “--” o nome do comando. Neste caso, abrirá um shell para o Pod.

Pods criados desta maneira são chamados de “Pod estático”.

Como eu acesso o Pod?

Se você rodar este comando em sua máquina, deveria ver a porta 80 aberta, certo? Afinal, fazemos isso com o Docker:

```
netstat -a | grep :80
```

Simplesmente não há nenhum serviço escutando a porta 80! Mas, no shell do Pod, que te mostrei antes, podemos ver que o serviço está no ar:

```
curl http://localhost


<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
html { color-scheme: light dark; }
body { width: 35em; margin: 0 auto;
font-family: Tahoma, Verdana, Arial, sans-serif; }
```

```
</style>
…
```

Pods não podem ser acessados fora do cluster k8s!

Criando Deployment e Service

Vamos deletar esse Pod que não serve para nós. Aliás, a recomendação é que você evite criar Pods estáticos e prefira usar **Workload** objects, como Deployment, por exemplo.

```
kubectl delete -f simplepod.yml
```

Um Workload representa uma aplicação em execução, controlando diversos Pod, instanciados com base em um "template" passado. Os Workloads são associados a controladores que mantém sempre o número adequado de Pods em execução.

Vejamos uma versão de **Deployment**. Crie o arquivo "simpledep.yml":

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-nginx
spec:
  selector:
    matchLabels:
      run: my-nginx
  replicas: 2
  template:
    metadata:
      labels:
        run: my-nginx
    spec:
      containers:
      - name: my-nginx
        image: nginx
        ports:
        - containerPort: 80
```

Vamos criar esse Deployment com o comando:

```
kubectl apply -f simpledep.yml
```

E vamos ver nosso Pod:

```

```

```
kubectl get pod

NAME                        READY   STATUS    RESTARTS   AGE
my-nginx-5b56ccd65f-hq2bc   1/1     Running   0          8s
my-nginx-5b56ccd65f-qr5nm   1/1     Running   0          8s
```

Não temos 1, mas 2 Pods!

Quando criamos um Workload, como um Deployment, na verdade dizemos ao Kubernetes o que queremos que ele mantenha no Cluster e passamos um “gabarito” para que ele trabalhe. São Pods dinâmicos. No campo “spec” dissemos que queríamos 2 instâncias desse Pod, e no campo “template” informamos como é que ele vai criar os Pods.

Podemos ver também os deployments:

```
kubectl get deployment

NAME        READY   UP-TO-DATE   AVAILABLE   AGE
my-nginx    2/2     2            2           46s
```

E podemos ver o Deployment específico com o comando:

```
kubectl get deployment/my-nginx

NAME       READY   UP-TO-DATE   AVAILABLE   AGE
my-nginx   2/2     2            2           4m24s
```

Temos 2 de 2 Pods no ar. Podemos também descrever o Deployment, para ver mais detalhes:

```
kubectl describe deployment/my-nginx
```

Bom, temos 2 Pods… Como faremos para acessá-los? Precisamos saber seus IPs ou nomes? Podemos criar um Service, que identifica o Deployment e as portas. Há um comando simples do k8s que já cria um Service para nós:

```
kubectl expose deployment/my-nginx
```

Agora temos um Service criado e podemos verificá-lo com:

```
kubectl get service/my-nginx
```

```
NAME       TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)   AGE
my-nginx   ClusterIP   10.103.32.187   <none>        80/TCP    62s
```

E este serviço pode ser acessado pelo ClusteIP que ele criou. A gestão dos Pods fica avaliando se algum deles caiu, retirando-o da lista de Endpoints, que são os endereços dos Pods que compõem o Service. Podemos ver os Endpoints com:

```
kubectl describe service/my-nginx

Name:              my-nginx
Namespace:         default
Labels:            <none>
Annotations:       <none>
Selector:          run=my-nginx
Type:              ClusterIP
IP Family Policy:  SingleStack
IP Families:       IPv4
IP:                10.103.32.187
IPs:               10.103.32.187
Port:              <unset>  80/TCP
TargetPort:        80/TCP
Endpoints:         172.17.0.3:80,172.17.0.4:80
Session Affinity:  None
Events:            <none>
```

Expondo um Service

Mas e se quisermos acessar um Service de fora do Cluster Kubernetes?

Vamos começar deletando tudo o que criamos:

```
kubectl delete -f simpledep.yml
```

E vamos criar outro arquivo: "exposed.yml":

```
apiVersion: v1
kind: Service
metadata:
  name: my-nginx
  labels:
    run: my-nginx
spec:
  type: NodePort
  ports:
  - port: 80
    targetPort: 80
    protocol: TCP
```

```
    name: http
  selector:
    run: my-nginx
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-nginx
spec:
  selector:
    matchLabels:
      run: my-nginx
  replicas: 1
  template:
    metadata:
      labels:
        run: my-nginx
    spec:
      containers:
      - name: nginx
        image: nginx
        ports:
        - containerPort: 80
```

Este arquivo está instalando dois objetos:

- Um **Service** do tipo **NodePort**, que permite expor uma porta (e um IP) utilizando o proxy do k8s;
- Um **Workload** do tipo **Deployment** que sobe um Pod Nginx.

Especificamos um YAML que cria múltiplos recursos e eles são separados por três hífens (“---”).

O Service NodePort se conecta com o Deployment através do seletor “run=my-nginx”.

Se fizermos o deploy deste arquivo, teremos 1 único Pod sendo executado. Assim, o NodePort pode simplesmente redirecionar o tráfego para ele. Eis o comando:

```
kubectl apply -f exposed.yml
```

Os Pods rodam na rede interna do cluster, e podem acessar utilizando o IP do serviço declarado nas variáveis de ambiente:

```
MY_NGINX_SERVICE_HOST=10.108.4.152
MY_NGINX_SERVICE_PORT_HTTP=8080
```

Mas os hosts externos ao Cluster precisam saber qual é o IP e porta externos expostos pelo NodePort. No Minikube podemos saber isso com este comando:

```
minikube service --url my-nginx
```

No meu caso o resultado foi:

http://192.168.49.2:32691

Se eu acessar isso em um navegador, vejo a página do Nginx.

Ué? Mas não queríamos expor a porta 80? Sim, mas com o k8s “não é assim que a banda toca”.

Você também pode saber a porta externa com o comando:

```
kubectl get service my-nginx --output='jsonpath="{.spec.ports[0]}"'

"
{"name":"http","nodePort":31965,"port":8080,"protocol":"TCP","targetPort":8
0}"
```

A demonstração

O código de exemplo tem duas rotas REST:

1. Postar novas notas, opcionalmente apagando depois da leitura:

```
curl -i --header "Content-Type: application/json" --request POST --data
'{"data" : "save this", "onetime" : false}' <url servidor>/api/note
```

No corpo da resposta está o identificador da nova nota criada. Copie-o para poder recuperar a nota.

2. Obter uma nota salva:

```
curl -i  <url servidor>/api/note/b61bc30d-8b2c-41e7-8df7-36a262826f44
```

Etapas para executar a demonstração

1. Instale o **minikube** e inicie:

```
minikube start
```

2. Instale o **Kong** ingress controller no minikube: https://docs.konghq.com/kubernetes-ingress-controller/latest/deployment/minikube/.

Um ingress controller é um **proxy reverso** e **balanceador de carga** que gerencia os acessos aos serviços dentro de um **cluster**. Precisamos de um componente destes para podermos permitir acesso ao nosso serviço, através de uma regra de entrada (**ingress rule**) que criaremos.

- Crie a variável de ambiente com o endereço do proxy:

```
export PROXY_IP=$(minikube service -n kong kong-proxy --url | head -1)
```

3. Compile a API:

```
cd code
go build -o ../api.bin cmd/main.go
```

4. Crie uma imagem **Docker**:

Para poder utilizar a imagem dentro de um **cluster k8s**, é preciso que ela esteja em um repositório, como o **Harbor**. Mas, como você está utilizando o **minikube** local, é possível apontar o **cliente Docker** da sua máquina para a implementação do **minikube**, então a imagem será gerada dentro da **VM** do **k8s**:

```
eval $(minikube -p minikube docker-env) --- use the same shell.
```

Agora faça o build da imagem normalmente:

```
cd ..
docker build -t api:v001 .
```

5. Instale a aplicação no **k8s** criando os objetos necessários:

- O **deployment** e um **service** para o **Redis**. Assim, poderemos acessar o servidor Redis pelo nome do serviço que criamos.
- O **deployment** e um **service** para o nosso servidor **Go**.
- Uma regra de entrada (**ingress rule**) para permitir acesso externo ao **service** da nossa aplicação Go.

```
kubectl apply -f redis.yaml
kubectl apply -f serviceDeployment.yaml
kubectl apply -f ingress-rule.yaml
```

Se quiser apagar os objetos, basta enviar o **delete** com o nome dos arquivos:

```
kubectl delete -f redis.yaml
kubectl delete -f serviceDeployment.yaml
```

```
kubectl delete -f ingress-rule.yaml
```

6. Teste o acesso utilizando a variável de ambiente **PROXY_IP** que você criou:

Poste uma nota:

```
curl -i --header "Content-Type: application/json" --request POST --data
'{"data" : "save this", "onetime" : false}' $PROXY_IP/api/note
```

Anote o id da nota e consulte:

```
curl -i  $PROXY_IP/api/note/<id da nota>

por exemplo: curl -i  $PROXY_IP/api/note/b61bc30d-8b2c-41e7-8df7-
36a262826f44
```