**Descomplicando o Kubernetes - Expert Mode**

**DAY-2**

**Indice**

* O que iremos ver hoje?
* O que é um Pod?
* Criando um Pod
* Criando um Pod através de um arquivo YAML
* Criando um Pod com mais de um container
* Criando um container com limites de memória e CPU
* Adicionando um volume EmptyDir no Pod

**Inicio da aula do Day-2**

**O que iremos ver hoje?**

Durante a aula de hoje, iremos ver todos os detalhes importantes sobre o menor objeto do Kubernetes, o Pod. Vamos ver desde a criação de um simples Pod, passando por Pod com multicontainers, com volumes e ainda com limitação ao consumo de recursos, como CPU ou memória. E claro, vamos aprender como ver todos os detalhes de um Pod em execução e brincar bastante com nossos arquivos YAML.

**O que é um Pod?**

Primeiro coisa, o Pod é a menor unidade dentro de um cluster Kubernetes.

Quando estamos falando sobre Pod, precisamos pensar que o Pod é uma caixinha que contém um ou mais containers. E esses containers compartilham os mesmos recursos do Pod, como por exemplo, o IP, o namespace, o volume, etc.

Então, quando falamos de Pod, estamos falando de um ou mais containers que compartilham os mesmos recursos, ponto.

**Criando um Pod**

Temos basicamente duas formas de criar um Pod, a primeira é através de um comando no terminal e a segunda é através de um arquivo YAML.

Vamos começar criando um Pod através de um comando no terminal.

kubectl run giropops --image=nginx --port=80

O comando acima irá criar um Pod chamado giropops, com uma imagem do nginx e com a porta 80 exposta.

Para ver o Pod criado, podemos usar o comando:

kubectl get pods

O comando acima irá listar todos os Pods que estão em execução no cluster, na namespace default.

Sim, temos namespaces no Kubernetes, mas isso é assunto para outro dia. Por enquanto, vamos focar em Pods e apenas temos que saber que por padrão, o Kubernetes irá criar todos os objetos dentro da namespace default se não especificarmos outra.

Para ver os Pods em execução em todas as namespaces, podemos usar o comando:

kubectl get pods --all-namespaces

Ou ainda, podemos usar o comando:

kubectl get pods -A

Agora, se você quiser ver todos os Pods de uma namespace específica, você pode usar o comando:

kubectl get pods -n <namespace>

Por exemplo:

kubectl get pods -n kube-system

O comando acima irá listar todos os Pods que estão em execução na namespace kube-system, que é a namespace onde o Kubernetes irá criar todos os objetos relacionados ao cluster, como por exemplo, os Pods do CoreDNS, do Kube-Proxy, do Kube-Controller-Manager, do Kube-Scheduler, etc.

Caso você queira ver ainda mais detalhes sobre o Pod, você pode pedir para o Kubernetes mostrar os detalhes do Pod em formato YAML, usando o comando:

kubectl get pods <nome-do-pod> -o yaml

Por exemplo:

kubectl get pods giropops -o yaml

O comando acima irá mostrar todos os detalhes do Pod em formato YAML, praticamente igual ao que você irá ver no arquivo YAML que criamos para criar o Pod, porém com alguns detalhes a mais, como por exemplo, o UID do Pod, o nome do Node onde o Pod está sendo executado, etc. Afinal, esse Pod já está em execução, então o Kubernetes já tem mais detalhes sobre ele.

Uma outra saída interessante é a saída em formato JSON, que você pode ver usando o comando:

kubectl get pods <nome-do-pod> -o json

Por exemplo:

kubectl get pods giropops -o json

Ou seja, utilizando o parametro -o, você pode escolher o formato de saída que você quer ver, por exemplo, yaml, json, wide, etc.

Ahh, a saída wide é interessante, pois ela mostra mais detalhes sobre o Pod, como por exemplo, o IP do Pod e o Node onde o Pod está sendo executado.

kubectl get pods <nome-do-pod> -o wide

Por exemplo:

kubectl get pods giropops -o wide

Agora, se você quiser ver os detalhes do Pod em formato YAML, mas sem precisar usar o comando get, você pode usar o comando:

kubectl describe pods <nome-do-pod>

Por exemplo:

kubectl describe pods giropops

Com o describe você pode ver todos os detalhes do Pod, inclusive os detalhes do container que está dentro do Pod.

Agora vamos remover o Pod que criamos, usando o comando:

kubectl delete pods giropops

Fácil né? Agora, vamos criar um Pod através de um arquivo YAML.

**Criando um Pod através de um arquivo YAML**

Vamos criar um arquivo YAML chamado pod.yaml com o seguinte conteúdo:

apiVersion: v1 # versão da API do Kubernetes

kind: Pod # tipo do objeto que estamos criando

metadata: # metadados do Pod

  name: giropops # nome do Pod que estamos criando

labels: # labels do Pod

  run: giropops # label run com o valor giropops

spec: # especificação do Pod

    containers: # containers que estão dentro do Pod

    - name: giropops # nome do container

        image: nginx # imagem do container

        ports: # portas que estão sendo expostas pelo container

        - containerPort: 80 # porta 80 exposta pelo container

Agora, vamos criar o Pod usando o arquivo YAML que acabamos de criar.

kubectl apply -f pod.yaml

O comando acima irá criar o Pod usando o arquivo YAML que criamos.

Para ver o Pod criado, podemos usar o comando:

kubectl get pods

Já que usamos o comando apply, acho que vale a pena explicar o que ele faz.

O comando apply é um comando que faz o que o nome diz, ele aplica o arquivo YAML no cluster, ou seja, ele cria o objeto que está descrito no arquivo YAML no cluster. Caso o objeto já exista, ele irá atualizar o objeto com as informações que estão no arquivo YAML.

Um outro comando que você pode usar para criar um objeto no cluster é o comando create, que também cria o objeto que está descrito no arquivo YAML no cluster, porém, caso o objeto já exista, ele irá retornar um erro. E por esse motivo que o comando apply é mais usado, pois ele atualiza o objeto caso ele já exista. :)

Agora, vamos ver os detalhes do Pod que acabamos de criar.

kubectl describe pods giropops

Outro comando muito util para ver o que está acontecendo com o Pod, mais especificamente ver o que o container está logando, é o comando:

kubectl logs giropops

Sendo que giropops é o nome do Pod que criamos.

Se você quiser ver os logs do container em tempo real, você pode usar o comando:

kubectl logs -f giropops

Simples né? Agora, vamos remover o Pod que criamos, usando o comando:

kubectl delete pods giropops

**Criando um Pod com mais de um container**

Vamos criar um arquivo YAML chamado pod-multi-container.yaml com o seguinte conteúdo:

apiVersion: v1 # versão da API do Kubernetes

kind: Pod # tipo do objeto que estamos criando

metadata: # metadados do Pod

  name: giropops # nome do Pod que estamos criando

labels: # labels do Pod

  run: giropops # label run com o valor giropops

spec: # especificação do Pod

  containers: # containers que estão dentro do Pod

  - name: girus # nome do container

    image: nginx # imagem do container

    ports: # portas que estão sendo expostas pelo container

    - containerPort: 80 # porta 80 exposta pelo container

  - name: strigus # nome do container

    image: alpine # imagem do container

    args:

    - sleep

    - "1800"

Com o manifesto acima, estamos criando um Pod com dois containers, um container chamado girus com a imagem nginx e outro container chamado strigus com a imagem alpine. Um coisa importante de lembrar é que o container do Alpine está sendo criado com o comando sleep 1800 para que o container não pare de rodar, diferente do container do Nginx que possui um processo principal que fica sendo executado em primeiro plano, fazendo com que o container não pare de rodar.

O Alpine é uma distribuição Linux que é muito leve, e não possui um processo principal que fica sendo executado em primeiro plano, por isso, precisamos executar o comando sleep 1800 para que o container não pare de rodar, adicionando assim um processo principal que fica sendo executado em primeiro plano.

Agora, vamos criar o Pod usando o arquivo YAML que acabamos de criar.

kubectl apply -f pod-multi-container.yaml

Para ver o Pod criado, podemos usar o comando:

kubectl get pods

Agora, vamos ver os detalhes do Pod que acabamos de criar.

kubectl describe pods giropops

Vamos conhecer dois novos comandos, o attach e o exec.

O comando attach é usado para se conectar a um container que está rodando dentro de um Pod. Por exemplo, vamos se conectar ao container do Alpine que está rodando dentro do Pod que criamos.

kubectl attach giropops -c strigus

Usando o attach é como se estivéssemos conectando diretamente em uma console de uma máquina virtual, não estamos criando nenhum processo dentro do container, apenas nos conectando a ele.

Por esse motivo se tentarmos utilizar o attach para conectar no container que está rodando o Nginx, nós iremos conectar ao container e ficaremos presos ao processo do Nginx que está em execução em primeiro plano, e não conseguiremos executar nenhum outro comando.

kubectl attach giropops -c girus

Para sair do container, basta apertar a tecla Ctrl + C.

Entendeu? Só vamos usar o attach para se conectar a um container que está rodando dentro de um Pod, e não para executar comandos dentro do container.

Agora, se você está afim de executar comandos dentro do container, você pode usar o comando exec.

O comando exec é usado para executar comandos dentro de um container que está rodando dentro de um Pod. Por exemplo, vamos executar o comando ls dentro do container do Alpine que está rodando dentro do Pod que criamos.

kubectl exec giropops -c strigus -- ls

Nós também podemos utilizar o exec para conectar em uma container que está rodando dentro de um Pod, porém, para isso, precisamos passar o parâmetro -it para o comando exec.

kubectl exec giropops -c strigus -it -- sh

O parametro -it é usado para que o comando exec crie um processo dentro do container com interatividade e com um terminal, fazendo com que o comando exec se comporte como o comando attach porém, com a diferença que o comando exec cria um processo dentro do container, no caso o processo sh. E por esse motivo que o comando exec é mais usado, pois ele cria um processo dentro do container, diferente do comando attach que não cria nenhum processo dentro do container.

Nesse caso, podemos até mesmo conectar no container do Nginx, pois ele vai conectar no container criando um processo que é o nosso interpretador de comandos sh, sendo possível executar qualquer comando dentro do container pois temos um shell para interagir com o container.

kubectl exec giropops -c girus -it -- sh

Para sair do container, basta apertar a tecla Ctrl + D.

**Criando um container com limites de memória e CPU**

Vamos criar um arquivo YAML chamado pod-limitado.yaml com o seguinte conteúdo:

apiVersion: v1 # versão da API do Kubernetes

kind: Pod # tipo do objeto que estamos criando

metadata: # metadados do Pod

  name: giropops # nome do Pod que estamos criando

labels: # labels do Pod

  run: giropops # label run com o valor giropops

spec: # especificação do Pod

  containers: # containers que estão dentro do Pod

  - name: girus # nome do container

    image: nginx # imagem do container

    ports: # portas que estão sendo expostas pelo container

    - containerPort: 80 # porta 80 exposta pelo container

    resources: # recursos que estão sendo utilizados pelo container

      limits: # limites máximo de recursos que o container pode utilizar

        memory: "128Mi" # limite de memória que está sendo utilizado pelo container, no caso 128 megabytes no máximo

        cpu: "0.5" # limite máxima de CPU que o container pode utilizar, no caso 50% de uma CPU no máximo

    requests: # recursos garantidos ao container

        memory: "64Mi" # memória garantida ao container, no caso 64 megabytes

        cpu: "0.3" # CPU garantida ao container, no caso 30% de uma CPU

Veja que estamos conhecendo alguns novos campos, o resources, o limits e o requests.

O campo resources é usado para definir os recursos que estão sendo utilizados pelo container, e dentro dele temos os campos limits e requests.

O campo limits é usado para definir os limites máximos de recursos que o container pode utilizar, e o campo requests é usado para definir os recursos garantidos ao container.

Simples demais!

Os valores que passamos para os campos limits e requests foram:

* memory: quantidade de memória que o container pode utilizar, por exemplo, 128Mi ou 1Gi. O valor Mi significa mebibytes e o valor Gi significa gibibytes. O valor M significa megabytes e o valor G significa gigabytes. O valor Mi é usado para definir o limite de memória em mebibytes, pois o Kubernetes utiliza o sistema de unidades binárias, e não o sistema de unidades decimais. O valor M é usado para definir o limite de memória em megabytes, pois o Docker utiliza o sistema de unidades decimais, e não o sistema de unidades binárias. Então, se você estiver utilizando o Docker, você pode usar o valor M para definir o limite de memória, mas se você estiver utilizando o Kubernetes, você deve usar o valor Mi para definir o limite de memória.

Agora vamos criar o Pod com os limites de memória e CPU.

kubectl create -f pod-limitado.yaml

Agora vamos verificar se o Pod foi criado.

kubectl get pods

Vamos verificar os detalhes do Pod.

kubectl describe pod giropops

Veja que o Pod foi criado com sucesso, e que os limites de memória e CPU foram definidos conforme o arquivo YAML.

Veja abaixo a parte da saída do comando describe que mostra os limites de memória e CPU.

Containers:

  girus:

    Container ID:   docker://e7b0c7b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0

    Image:          nginx

    Image ID:       docker-pullable://nginx@sha256:0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b

    Port:           80/TCP

    Host Port:      0/TCP

    State:          Running

      Started:      Wed, 01 Jan 2023 00:00:00 +0000

    Ready:          True

    Restart Count:  0

    Limits:

      cpu:     500m

      memory:  128Mi

    Requests:

      cpu:        300m

      memory:     64Mi

    Environment:  <none>

    Mounts:

      /var/run/secrets/kubernetes.io/serviceaccount from default-token-0b0b0 (ro)

Veja que na saída acima, ele mostra o campo CPU com o valor 500m, isso significa que o container pode utilizar no máximo 50% de uma CPU, afinal um CPU é igual a 1000 milliCPUs, e 50% de 1000 milicpus é 500 milliCPUs.

Então, se você quiser definir o limite de CPU em 50% de uma CPU, você pode definir o valor 500m, ou você pode definir o valor 0.5, que é o mesmo que definir o valor 500m.

Para você testar os limites de memória e CPU, você pode executar o comando stress dentro do container, que é um comando que faz o container consumir recursos de CPU e memória. Lembre-se de instalar o comando stress, pois ele não vem instalado por padrão.

Para ficar fácil de testar, vamos criar um Pod com o Ubuntu com limitação de memória, e vamos instalar o comando stress dentro do container.

Chame o arquivo de pod-ubuntu-limitado.yaml.

apiVersion: v1

kind: Pod

metadata:

  name: giropops

spec:

  containers:

  - name: girus

    image: ubuntu

    args:

    - sleep

    - infinity

    resources:

    limits:

        memory: "128Mi"

        cpu: "0.5"

    requests:

        memory: "64Mi"

        cpu: "0.3"

Olha essa sacadinha do parâmetro infinity, ele faz o container esperar para sempre e assim, se manter em execução.

Agora vamos criar o Pod.

kubectl create -f pod-ubuntu-limitado.yaml

Agora vamos verificar se o Pod foi criado.

kubectl get pods

Agora vamos para dentro do container.

kubectl exec -it ubuntu -- bash

Agora vamos instalar o comando stress.

apt update

apt install -y stress

Agora vamos executar o comando stress para consumir memória.

stress --vm 1 --vm-bytes 100M

Até aqui tudo bem, pois definimos o limite de memória em 128Mi, e o comando stress está consumindo 100M, então está tudo certo.

Vamos aumentar o consumo de memória para 200M.

stress --vm 1 --vm-bytes 200M

Veja que o comando stress não consegue consumir 200M, pois o limite de memória é 128Mi, e 128Mi é menor que 200M e com isso tomamos o erro e o comando stress é interrompido.

Atigimos o nosso objetivo, atingimos o limite do nosso container! :D

Quer brincar um pouco mais com o comando stress? Veja o --help dele.

stress --help

Ele traz várias opções para você brincar com o consumo de memória e CPU.

**Adicionando um volume EmptyDir no Pod**

Primeira coisa, nesse momento não é o momento de entrar em maiores detalhes sobre volumes, nós teremos um dia inteiro para falar sobre volumes, então não se preocupe com isso agora.

O dia de hoje é para que possamos ficar bastante confortável com os Pods, desde sua criação, administração, execução de comandos, etc.

Então, vamos criar um Pod com um volume EmptyDir.

Antes, o que é um volume EmptyDir?

Um volume do tipo EmptyDir é um volume que é criado no momento em que o Pod é criado, e ele é destruído quando o Pod é destruído. Ou seja, ele é um volume temporário.

No dia-a-dia, você não vai usar muito esse tipo de volume, mas é importante que você saiba que ele existe. Um dos casos de uso mais comuns é quando você precisa compartilhar dados entre os containers de um Pod. Imagina que você tem dois containers em um Pod e um deles possui um diretório com dados, e você quer que o outro container tenha acesso a esses dados. Nesse caso, você pode criar um volume do tipo EmptyDir e compartilhar esse volume entre os dois containers.

Chame o arquivo de pod-emptydir.yaml.

apiVersion: v1 # versão da API do Kubernetes

kind: Pod # tipo de objeto que estamos criando

metadata: # metadados do Pod

  name: giropops # nome do Pod

spec: # especificação do Pod

  containers: # lista de containers

- name: girus # nome do container

    image: ubuntu # imagem do container

    args: # argumentos que serão passados para o container

    - sleep # usando o comando sleep para manter o container em execução

    - infinity # o argumento infinity faz o container esperar para sempre

    volumeMounts: # lista de volumes que serão montados no container

    - name: primeiro-emptydir # nome do volume

      mountPath: /giropops # diretório onde o volume será montado

volumes: # lista de volumes

- name: primeiro-emptydir # nome do volume

    emptyDir: # tipo do volume

      SizeLimit: 256Mi # tamanho máximo do volume

Agora vamos criar o Pod.

kubectl create -f pod-emptydir.yaml

Agora vamos verificar se o Pod foi criado.

kubectl get pods

Você pode ver a saída do comando kubectl describe pod giropops para ver o volume que foi criado.

kubectl describe pod giropops

Agora vamos para dentro do container.

kubectl exec -it ubuntu -- bash

Agora vamos criar um arquivo dentro do diretório /giropops.

touch /giropops/FUNCIONAAAAAA

Pronto, o nosso arquivo foi criado dentro do diretório /giropops, que é um diretório dentro do volume do tipo EmptyDir.

Se você digitar mount, vai ver que o diretório /giropops está montado certinho dentro de nosso container.

Pronto, agora você já sabe criar um Pod com um volume do tipo EmptyDir. :)

Lembrando mais uma vez que ainda vamos ver muito, mas muito mais sobre volumes, então não se preocupe com isso agora.