Kubernetes









**Instalação Kubernetes (Aula 184)**

1. Instalar **minikube**
2. Instalar **kubectl**

**Comandos básicos minikube**

**minikube start –driver=[DRIVER\_NAME]** – Inicializa uma VM com o driver especificado. Drivers em: [Drivers | minikube (k8s.io)](https://minikube.sigs.k8s.io/docs/drivers/)

**minikube delete** – deleta a VM criada

**minikube status** – verifica o status da VM criada

**minikube dashboard** – abre uma aba no browser com um dashboard do Kubernetes

**Kubectl**

Kubectl (Kube control) é o controlador da máquina local que envia comandos ao Master Node do cluster Kubernetes.

**Comandos kubectl**

**kubectl get [ARG] –** Lista os objetos do argumento passado.

**Exemplos**

* **kubectl get deployments** – Lista todos os deployments do cluster
* **kubectl get pods** – Lista todas as Pods do cluster
* **kubectl get services** – Lista todos os serviços (objetos) criados no cluster
* **kubectl delete service [SERVICE\_NAME]** - Deleta o recurso
* **kubectl delete deployment [DEPLOYMENT\_NAME]** - Deleta o recurso

**kubectl create [OBJECT] [ARGS]** – Cria um objeto do Kubernetes.



**Objetos**

Os objetos pertencem ao ambiente Kubernetes e executam uma função nele.

**Objeto Deployments**

Serve para criarmos uma pod baseada em uma imagem do DockerHub.

**Criando um deployment**

**kubectl create deployment [DEPLOYMENT\_NAME] --image=[DEPLOYMENT\_IMAGE]** – cria um novo deployment e pod baseado na imagem informada.

**Exemplo de criação do deployment**

* **kubectl create deployment first-app --image=kub-first-app (COMANDO GERA ERRO!!!)**
* O comando acima gerará o erro **ErrImagePull**, já que os comandos do kubectl são executados dentro do cluster criado, que portanto **não terá acesso à uma imagem buildada no localhost**.
* Para que o comando execute corretamente, **é preciso que subamos a imagem para o DockerHub**, e utilizemos sua URL, como é demonstrado a seguir:
* **kubectl create deployment first-app --image=azold6/kub-first-app**

**Escalando um deployment**

**kubectl scale [OBJECT]/[OBJECT\_NAME] --replicas=[NUMBER\_OF\_DESIRED\_PODS]** - Escala um serviço (objeto kubernetes) para o número desejado de pods

**Exemplo**

* **kubectl scale deployment/first-app --replicas=3**

****

**Atualização de imagem em um Pod (deployment)**

1. Re-buildar a imagem desejada
2. Upá-la no DockerHub
3. Executar o comando **kubectl set image deployment/[DEPLOYMENT\_NAME] [CURRENT\_CONTAINER\_NAME]=[NEW\_IMAGE\_NAME\_ON\_DOCKERHUB]**

**IMPORTANTE!!!** A nova imagem será baixada **SOMENTE SE POSSUIR UM NOME OU TAG DIFERENTE DA IMAGEM ANTIGA DO POD.** Sendo assim, é uma boa prática versionar as imagens, como **imagem:1**, **imagem:2**...

1. Para visualizar o status da troca, executar **kubectl rollout status deployment/[DEPLOYMENT\_NAME]**

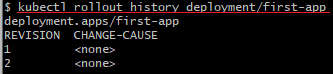
**Rollback de um deployment**

Para retornarmos para a versão anterior, devemos executar o comando abaixo:

* **kubectl rollout undo deployment/[DEPLOYMENT\_NAME]** - retorna à versão passada da Pod

Já para retornarmos para uma versão específica, devemos:

1. Executar **kubectl rollout history deployment/[DEPLOYMENT\_NAME]**

****

1. Selecionar a revisão para a qual deseja-se retornar. Pode-se verificar mais detalhes da revisão com o comando:

**kubectl rollout history deployment/[DEPLOYMENT\_NAME] --revision=[REVISION\_NUMBER]**

1. Executar **kubectl rollout undo deployment/[DEPLOYMENT\_NAME] --to-revision=[REVISION\_NUMBER]**

**Objeto Service**

Serve para agruparmos pods e atribuir a elas um IP fixo, além da possibilidade de expor as pods para a WWW.

**kubectl expose deployment [DEPLOYMENT\_NAME] –type=[TYPE] –port=[PORT]** - Cria um Service para expor uma porta do deployment informado

**Types:**

1. ClusterIP (só será ‘reachable’ de dentro do cluster)
2. NodePort (‘reachable’ de fora do cluster)
3. LoadBalancer (Gera um IP único para todas as pods e expõe ela, além de distribuir o tráfego)

Exemplo de exposição de portas de uma pod

**kubectl expose deployment first-app –type=LoadBalancer –port=8080**

Após a exposição, o comando **kubectl get services** irá expor os seguintes dados:



O **EXTERNAL-IP** é automaticamente atribuído ao serviço em um provedor cloud, mas no minikube, ainda é necessário executar o comando **minikube service first-app**

**‘Declarative Approach’**

Forma de evitar que precisemos executar múltiplos comandos para aplicar modificações no cluster. Comporta-se com um intuito similar ao docker-compose.

**Estrutura básica**

deployment.yaml

apiVersion: apps/v1

kind: Deployment ############## Tipo do objeto (neste caso, Deployment. Poderia ser um Service, Job, etc...)

metadata:

  name: second-app-deployment # Nome do objeto.

spec: ######################### ESPECIFICAÇÕES DO DEPLOYMENT

  replicas: 1 ################# Qtd. de pods do deployment.

  selector:

    matchLabels:

      app: second-app ######### Variável que auxilia o deployment a encontrar as pods que deve gerenciar.

      tier: backend ########### Neste caso, O deployment buscará pelo valor 'nginx' em spec.template.metadata.labels.app.

  ########### Se existir, funcionará adequadamente

template: ################################### **ESPECIFICAÇÕES DO POD. Em caso de confusão, consultar aula 198.**

    metadata:

      labels:

        app: second-app ##################### Local onde o selector.matchLabels busca pelas labels das aplicações.

        tier: backend

    spec:

      containers: ########################### Declaração dos contêineres que irão para as Pods (neste caso, 3)

        - name: second-node ################# Nome do contêiner

          image: academind/kub-first-app:2 ## Imagem a ser usada

        # - name: ... (Caso eu queira adicionar mais contêineres)

        #   image: ...

Para aplicar as definições descritas dentro do arquivo **deployment.yaml** acima, executamos o seguinte comando:

* **kubectl apply -f=[FILE\_NAME]**

Agora, precisamos de um objeto Service para expor as portas do Pod criado pelo **deployment** acima:

backend.yaml

apiVersion: v1

kind: Service ############## Tipo do objeto

metadata:

  name: backend ############ Nome do objeto

spec:

  selector:

    app: second-app ######## Selector para definir qual pod terá suas portas expostas por esse serviço.

  ports:            ######## O second-app foi definido no arquivo deployment.yaml

    - protocol: 'TCP'

      port: 80 ############# Porta externa

      targetPort: 8080 ##### Porta interna a ser exposta

  type: LoadBalancer ####### Tipo do serviço. Poderia ser ClusterIP, NodePort, etc...

Após executarmos **kubectl apply -f=[YAML\_NAME]**, basta executar **minikube service [EXPOSURE\_SERVICE\_NAME]** para acessarmos o Pod localmente.



Para atualizar as pods com uma nova imagem, ou adicionar mais replicas, basta alterar o a yaml do deployment e executar **kubectl apply -f=[YAML\_NAME]**

**Junção de objetos no mesmo arquivo**

Com o divisor ‘---’, é possivel colocar múltiplas instruções em um mesmo arquivo yaml, como segue o exemplo:

  # SERVICE DEFINITIONS…

  # ...

  ---

  # DEPLOYMENT DEFINITIONS…

  # ...

---

  # MORE DEFINITIONS…

  # ...

**Liveness probe**

Podemos sobreescrever a forma ‘padrão’ de verificar a saúde do contêiner. Para isso, utilizamos o **LivenessProbe**.

#ESPECIFICAÇÕES DO OBJETO ACIMA...

template: ################################# ESPECIFICAÇÕES DO POD.

    metadata:

      labels:

        app: second-app

        tier: backend

    spec:

      containers:

        - name: second-node

          image: azold6/kub-first-app:2

          livenessProbe:

            httpGet:

              path: / # Caminho onde a requisição "verificadora de saúde" deve ser enviada

              port: 8080 # Porta para a requisição

            periodSeconds: 10 # Intervalo de verificação após o início

            initialDelaySeconds: 5 #Intervalo de verificação inicial

**Volumes**

O Kubernetes permite que escolhamos uma infinidade de tipos de volumes.

**emptyDir**

O emptyDir permite que os dados de um diretório não sejam perdido na reiniciação de um contêner, criando um diretório vazio no diretório desejado ao inicializar.

# ...

    spec:

      containers:

        - name: story-backend

          image: azold6/kub-deploy-story

          volumeMounts:

            - mountPath: /app/story # Caminho que não pode ser perdido no reiniciamento do contêiner

              name: story-volume # Linkagem do diretório desejado ao volume criado abaixo

      volumes:

        - name: story-volume # Nome do volume

          emptyDir: {} # Tipo do volume

**Problemática:** O emptyDir **inicializa atrelado a somente um pod**. Isso significa que, quando um pod precisa ser reiniciado, o dado não poderá ser acessado até que o Kubernetes o substitua por outra pod.

**hostPath**

Funciona como um **bind-mount**. Uma pasta no cluster é linkada a um diretório no contêiner.

# ...

    spec:

      containers:

        - name: story-backend

          image: azold6/kub-deploy-story

          volumeMounts:

            - mountPath: /app/story # Caminho que não pode ser perdido no reiniciamento do contêiner

              name: story-volume # Linkagem do diretório desejado ao volume criado abaixo

      volumes:

        - name: story-volume # Nome do volume

          hostPath:

path: /data # Pasta dentro do cluster onde o bind-mount será feito

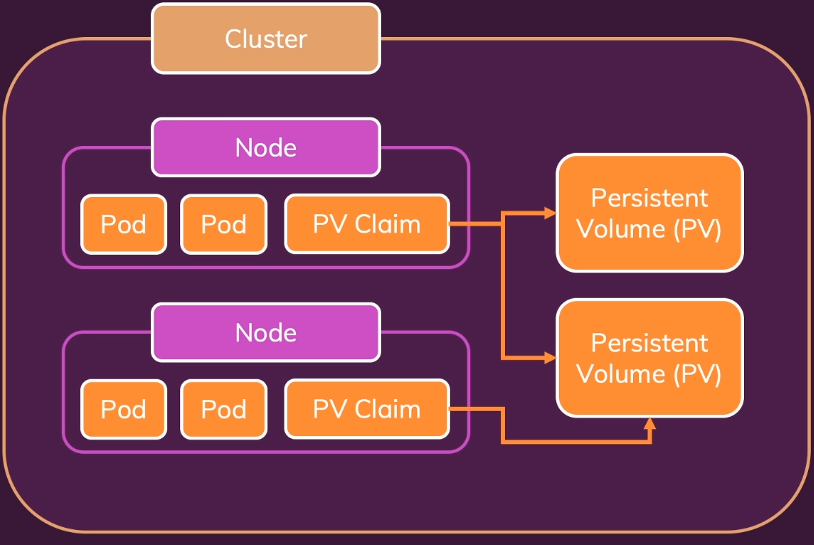
type: DirectoryOrCreate # Estratégia do bind-mount. Se o diretório /data não existir, ele será criado.

**Problemática:** O **hostPath** é independente de pods, mas não de nodes. Ou seja, o **volume armazenado aqui poderá ser utilizado por todas as pods daquele worker node, mas não por pods de outros worker nodes**.

**Persistent Volumes**

Os volumes vistos até o momento pertenciam à **pod** ou ao **node** onde estavam inseridos. Isso nos gera problemas ao criar aplicações com múltiplos nodes, que precisam se conectar ao mesmo volume.

A idéia dos Persistent Volumes é que os volumes existam de forma independente ao pod/node. Os nodes, por sua vez, possuem uma **Claim**, que se comunica com os volumes desejados.



**Definindo um PersistentVolume**

apiVersion: v1

kind: PersistentVolume

metadata:

  name: host-pv

spec: #################### PERSISTENT VOLUME SPECIFICATIONS

  capacity:

    storage: 1Gi

  volumeMode: Filesystem # Filesystem, Block

  accessModes:

    - "ReadWriteOnce" # ReadWriteOnce, ReadOnlyMany, ReadWriteMany

  hostPath:

    path: /data

    type: DirectoryOrCreate

* **ReadWriteOnce:** Read/Write only no volume está disponível para somente um node (Once). **Pods que realizarem o Claim nesse volume devem estar no mesmo node.**
* **ReadOnlyMany:** Read only disponível para pods de nodes diferentes.
* **ReadWriteMany:** Read/Write only disponível para pods de nodes diferentes.

Para acessar os dados de um PersistentVolume, devemos criar um resource do tipo **PersistentVolumeClaim:**

apiVersion: v1

kind: PersistentVolumeClaim

metadata:

  name: host-pvc

spec:

  volumeName: host-pv # Nome do PersistentVolume a ser reinvindicado pela Claim

  accessModes:

    - ReadWriteOnce # Mesmo access mode do PersistentVolume

  resources: #Parte necessária. Ainda não sei o porquê.

    requests:

      storage: 1Gi

Após a criação da Claim, devemos referenciá-la no Deployment, para que o Worker Node possua uma Claim que acessa os dados daquele volume não-atrelado à nodes/pods.

# ...

    spec:

      containers:

        - name: story-backend

          image: azold6/kub-deploy-story

          volumeMounts:

            - mountPath: /app/story # Caminho que não pode ser perdido no reiniciamento do contêiner

              name: story-volume #### Linkagem do diretório desejado ao volume criado abaixo

      volumes:

        - name: story-volume ######## Nome do volume

          persistentVolumeClaim:

            claimName: host-pvc ##### Referenciando a Claim para acessar o volume.

Depois disso, informar o parâmetro storageClassName nas specs do **PersistentVolume** e da **PersistentVolumeClaim.**

Os Storage Class podem ser encontrados pelo comando **kubectl get sc**

****

**Incluindo parâmetro storageClassName nas specs dos resources necessários (PersistentVolume e PersistentVolumeClaim)**

# ...

spec:

  volumeName: host-pv

  storageClassName: standard # Adição necessária. Ainda não está claro o porquê.

  accessModes:

    - ReadWriteOnce

  resources:

    requests:

      storage: 1Gi

**Environment Variables**

Podemos utilizar variáveis de ambiente nas pods, simplismente adicionando o parâmetro **env** nas especificações do contêiner do depoyment (**NÃO ESPECIFICAÇÕES DO DEPLOYMENT!!!**)

# ...

    spec:

      containers:

        - name: story-backend

          image: azold6/kub-deploy-story

env:

name: STORY\_FOLDER # Definição das variáveis de ambiente

value: story

# ...

Alternativamente, podemos definir um yaml do objeto **ConfigMap** para conter as variáveis de ambiente.

apiVersion: v1

kind: ConfigMap

metadata:

  name: myconfigmap

data:

  folder: 'story' # Variável key:value que será utilizada no yaml do resource Deployment

# Aqui, podem ser adicionadas mais veriáveis de ambiente do tipo key:value

# key2: 'value2'...

# key3: 'value3'...

Após a definição, basta referenciarmos o yaml no **deployment**

    #...

    spec:

      containers:

        - name: story-backend

          image: azold6/kub-deploy-story

          env:

            name: STORY\_FOLDER # Nome da variável de ambiente

            valueFrom: # Definindo o source do valor da variável

              configMapKeyRef: # Definindo que o valor virá de um objeto ConfigMap

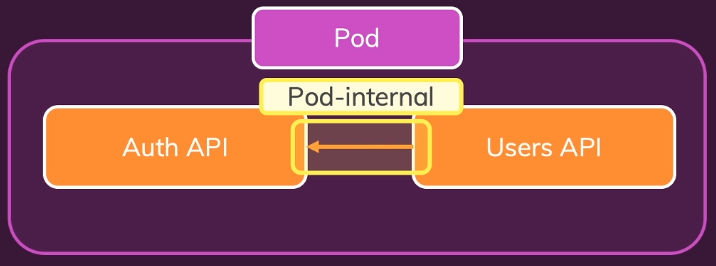
                name: myconfigmap # Nome do objeto

                key: story # Chave de onde o objeto virá, definido no [CONFIGMAP\_NAME].yaml

    #...

**Kubernetes Networking**

**Pod-Internal networking**.

****

Comunicação entre dois contêineres dentro de uma mesma pod. Acontece quando definimos dois contêineres no mesmo Deployment, desta forma:

    # ...

    spec:

      containers:

        - name:  cont-users

          image:  azold6/kub-demo-users:latest

        - name: cont-auth

          image: azold6/kub-demo-auth:latest

String de conexão para requisição:

`http://${process.env.AUTH\_ADRESS}/hashed-password/` + password

A variável de ambiente **AUTH\_ADRESS**, na string de conexão acima, deve ser definida como **localhost** em comunicações internas de uma pod, desta forma:

    # ...

    spec:

      containers:

        - name:  cont-users

          image:  azold6/kub-demo-users:latest

          env:

            - name: AUTH\_ADRESS

              value: localhost

        - name: cont-auth

          image: azold6/kub-demo-auth:latest

**Pod-to-Pod networking**.

Para realizarmos a comunicação entre duas pods, seguem informações:

1. O Kubernetes cria variáveis de ambiente automaticamente, seguindo o seguinte padrão:

* **Variável criada:** Nome do serviço que expõe as portas, em CAPS LOCK, **substituindo os - por \_**

**Exemplo:** O seguinte serviço poderia ser acessado por uma aplicação de outra pod pela variável de ambiente **SERV\_AUTH\_SERVICE\_HOST**

apiVersion: v1

kind: Service

metadata:

  name: serv-auth

spec:

  selector:

    app: app-auth

  ports:

    - protocol: TCP

      port: 80

      targetPort: 80

  type: ClusterIP

Isso é necessário, porque o IP da pod é dinâmico, e pode se alterar ao reiniciá-la ou ao escalá-la. Desta forma, a variável **[NOME\_SERVICO]\_SERVICE\_HOST** permite que adiquíramos o host da outra pod de forma dinâmica.

Abaixo, segue um exemplo de como ficaria a URL de conexão com outra pod no código

const hashedPW = await axios.get(`http://${process.env.SERV\_AUTH\_SERVICE\_HOST}/hashed-password/` + password);

Alternativamente, podemos consultar o IP do serviço de exposição pelo comando **kubectl get services** e utilizá-lo como host no código. O problema disso é que não pegaríamos o IP de forma dinâmica.