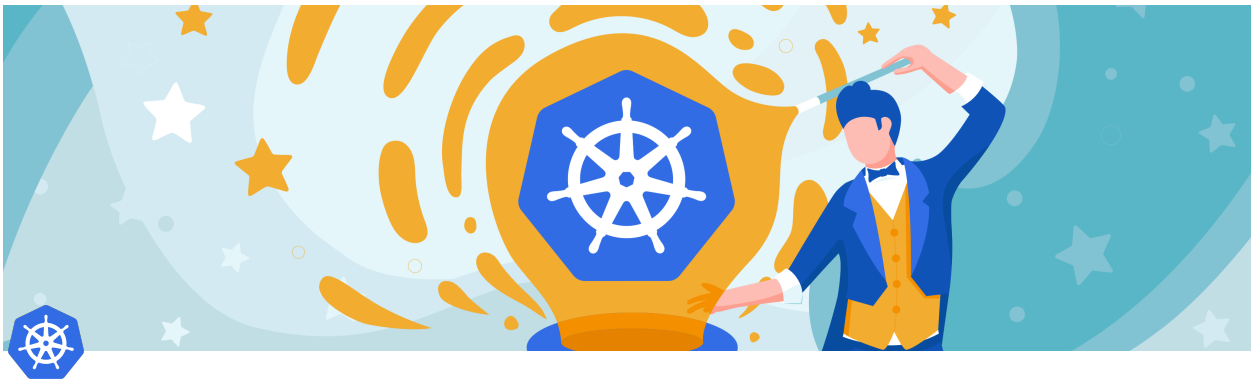


Kubernetes

https://www.freecodecamp.org/news/the-kubernetes-handbook/#introduction-to-container-orchestration-and-kubernetes

Usando localmente no linux

`instalar minikube`

https://minikube.sigs.k8s.io/docs/start/

`instalar kubectl` O kubectl tem que ser 1 versão a menos a do cluster.

https://kubernetes.io/docs/tasks/tools/install-kubectl-linux/

Minikube

O minikube é um sistema que cria uma VM dentro do computador usando o docker (ou driver especificado).

Esta VM é tratada como um cluster do Kubernetes, que é um PC que contém

Setando Hypervisor (Docker) no minikube

`minikube config set driver {docker}`

Iniciando minikube

`minukube start`

Isso iniciará uma sequência de ações para iniciar a Vm do minikube.

Depois de executar, uma vm com o kubernetes estará disponível para uso.

Você pode usar o `kubectl` pra usar esse minikube.

`minikube stop` para parar o minikube.

`minikube delete` para parar o minikube.

Usando localmente no wsl2 com windows

Installing Minikube | magda

- Instalar minikube somente no windows
- Fazer um script pra rodar o minikube do windows dentro do bash do wsl2
- Instalar docker e kubectl no wsl
- Linkar as configs do kubectl da pasta ~/.kube com a da pasta /mnt/c/Users/gabri/.kube
- Verificar se os diretórios que o config procura estão ok.
- Obs: é bem complicado.

Diretorios do config do kubectl

~/.kube/config

mnt/c/Users/gabri/

OBS: Sempre que reinciar o minikube, é preciso trocar a porta do servidor nas configurações do minikube do wls pela porta que tá na configuração do config do windows.

Arquitetura do Kubernetes

Clusters

Clusters são a estrutura macro do kubernetes, eles são caixas que contém os chamados **nodes.**

Nodes

Nodes são máquinas virtuais ou físicas, podem existir uma ou mais dessas máquinas dentro de um cluster do kubernetes.

O papel dos nodes é rodar containers, como se fosse um pc normal, e usa o docker para fazer isso.

Um node pode ter um dos dois papéis.

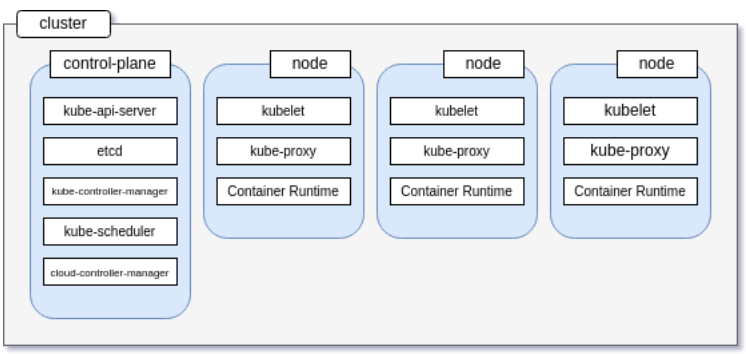
▼ Node de control plane, é o cérebro que atua na organização dos serviços

- Os control planes possuem inúmeros componentes avançados responsáveis por gerenciar o estado do cluster, de outros clusters, se ligar a cloud e se conectar com os nodes.
  - **kube-api-server** → forma de contato entre o control plane e os nodes por meio dos seus **kubelets.** A comunicação entre o terminal e o kube-api-server é pelo **kubectl**
  - **etcd** → Base de dados que armazena as verdades e dados daquele cluster
  - **kube-controller-manager** → Gerenciador de estados e atividados do cluster, você pede pra ele mudar o cluster e ele muda.
  - **kube-scheduler** → Resonsável por manter os nodes em controle, sem overload, e avalia se, dado recursos atuais, poderá ser feita uma atividade em um node
  - **cloud-controller-manager** → Componente que faz a ligação entre o cluster e serviços de cloud como o Google Kubernetes Engine (Não existe no control-plane do minikube)

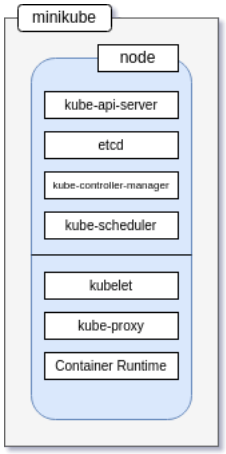
▼ Node de rodar serviços / Worker node.

- Por sua vez, esse node também possui componentes
  - **kubelet** → Plataforma que permite a comunicação do control plane com o node
  - **kube-proxy** → Proxy de todas as requisições que os containers fazem
  - **Container runtime** →Plataforma de rodar containers (Docker)

Cada node possui um ip externo.



No caso do **minikube,** só é rodado **um** cluster com **um** node, sem cloud-controller-manager. Esse node possue os dois papéis.



Pods

Pods são a menor unidade possível em uma arquitetura kubernetes, eles são responsáveis por rodar um ou mais containers.

O comum é que seja apenas um, pods rodam diferentes containers com diferentes serviços.

Pods são efêmeros, eles são criados e destruídos o tempo todo.

O ideal é manipular pods por meio de estruturas superiores, não diretamente.

Services

Uma vez que pods são efêmeros, se torna necessária uma entidade maior que os controle.

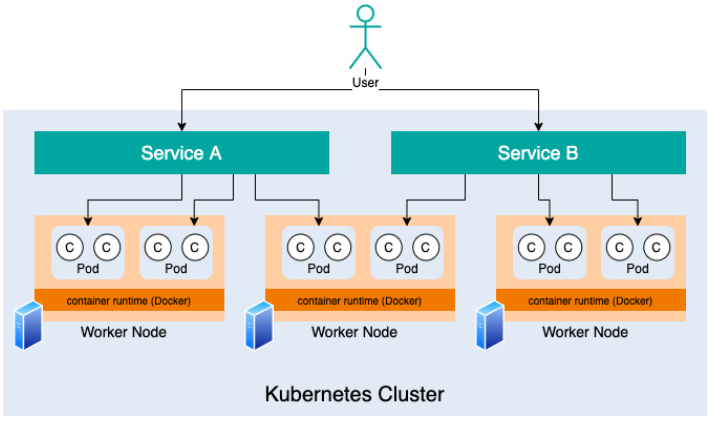
Os services são essa entidade, que junta os pods e os tratam como uma coisa só.

Os services são o porta voz de um conjunto de pods, eles fazem a ponte entre pods e o ambiente externo.

Diferentes tipos de services criam diferentes políticas de comunicação entre cada pod, como por exemplo o service **loadBalancer** que faz com que pods recebam requests vindas de fora do container (acredito que isso seja útil para APIs)

Se seu pod roda um container que a aplicação precisa ser acessada pelo ambiente externo, existe um serviço pra isso.

O papel de um serviço é conectar um pod ou conjunto de pods a uma rede e esta rede, fazendo com que esses se comuniquem com outros pods, serviços ou com requisições externas.



Outros componentes principais

Ingress

Componentes responsáveis por gerenciar o tráfego externo para dentro dos pods.

Volumes

Espaços de memória o qual podem ser mapeados para receber arquivos que precisam ser salvos nos containers, como em bancos de dados, por exemplo.

Configmap

Componente que recebe informações não sigilosas como variáveis que podem ser usadas dentro do código das aplicações conteinerizadas.

Secret

Igual ao configmap, mas usado para informações sigilosas, encriptadas em base64

Deployments

Caso um pod venha a cair, é necessário um blueprint/guia para que outro pod similar seja colocado em seu lugar, e isso é fornecido pelos deployments.

StatetefulSet

É uma espécie de blueprints para arquivos que possuem estados, por exemplo, banco de dados, multiplos containers de banco de dados podem ter inconsistencias ao escrever na mesma base de dados, por isso o statefulset é usado.

Primeiros comandos com Kubectl

Deployments

Nosso objetivo no momento é criar pods para que eles rodem nossas aplicações dentro de containers. E do que eu preciso pra rodar um container? Sua imagem, é claro. Logo, ao iniciar um pod, precisamos dizer que imagem ele irá rodar.

Além disso, o pod precisa ter especificado qual porta vai estar exposta e como irá mapeá-la ao container caso ele rode um serviço que precise de comunicação.

Todas essas informações montam uma espécie de **modelo** ou **imagem** para o próprio pod, é um **blueprint** de como os pods irão rodar.

Chamamos isso de **deployment**, um deployment é um modelo de criação de pods que pode ter inúmeras configurações pré-definidas.

A forma em que vamos criar pods é por meio desses deployments.

Por padrão, deployments só precisam de um **nome** e uma **imagem** especificada pra rodar, o resto é definido depois.

Um pod pode ser criado de 3 formas:

- `kubectl run {nome_do_pod} --image={imagem}` → Forma sem muitas especificações, rápida.
- `kubectl create deployment {nome_do_deployment} --image={imagem} {config_opcional}` → Essa opção cria um blueprint / deployment do pod com as nossas configurações desejadas já inicia esse pod com o nome do deployment + algum id. Como outros pods podem ser criados a partir dessa mesma imagem, o comando `kubectl get replicaset` mostra a quantidade de pods que têm aquele deployment.
- `kubectl apply -f {arquivo_de_config}.yaml` → Essa forma usa um arquivo .yaml que tem as configurações definidas de um deployment, incluindo o seu nome.

kubectl get {objeto}

Recebe algumas informações de objetos rodando

`kubectl get pods/pod` → Lista de pods rodando

`kubectl get all` → Lista todos os componentes

`kubectl get services` → Serviços rodando

`kubectl get deployments` → Lista de deployments

`kubectl get replicaset` → Lista quantos pods rodam de cada deployment

kubectl describe {objeto} {nome_do_objeto}

`kubectl describe pod {nome}` → Descreve o pod

`kubectl describe service {nome}` → Descreve o serviço

`kubectl describe deployment {nome}` → Descreve o

kubectl delete {objeto} {nome_do_objeto}

Assim como os de cima, é só repetir a estrutura com o objeto desejado que ele será apagado.

kubectl logs {nome_do_objeto}

Retorna um log com o que aconteceu com esse objeto nos ultimos tempos.

kubectl edit {objeto} {nome_do_objeto}

Edita configurações de um objeto específico.

Criando arquivos de deployments e serviços.

O flow básico para acesso de aplicações é o seguinte.

Requisição → Serviço → Pods.

A nossa requisição passa por um serviço, que irá gerenciar qual pod irá recebê-la, baseando-se em alguns critérios como workload.

Os pods são criados no modelo de um deployment, e as informações desse deployment são consumidas pelo arquivo do serviço para ele saber quais os nomes de pods ele terá que redirecionar as requisições e quais são as portas que ele deve se conectar no pod.

Lembrando que um pod roda um container, e esse container roda uma aplicação que ouve numa porta.

- O container está ouvindo na porta X e essa porta está exposta.
  - Se não estivessemos usando o kubernetes, era possível fazer um mapping dessa porta com uma porta do host (nosso pc) e poderiamos acessá-la pela internet mesmo.
  - Nesse caso, o host do container é o pod, e esse pod está dentro do node do cluster do kubernetes. Cada pod tem o seu próprio ip privado.
- A porta do container é mapeada a uma porta do pod dentro das configurações do deployment.
- O serviço recebe uma porta a qual ele deverá ouvir, nossas requisições irão entrar nessa porta. Ele também tem que saber a qual porta tem que se conectar aos pods, e isso também é informado no arquivo de serviços.
- Serviços também podem ser criados para fazer comunicação inter-pods, exemplo, a comunicação de uma api com o banco de dados é feita somente dentro do cluster, os pods de banco de dados não estão conectados ao serviço que liga à internet.

Código pra criar deployments

OBS: ARQUIVOS YAML NÃO PODEM CONTER TABS COMO IDENTAÇÃO, SOMENTE ESPAÇOS.

```
apiVersion: apps/v1
kind: Deployment
metadata: # Metadados do deployment
  name: {nome-do-deployment} #nginx-depl
  labels:
    app: {nome-do-app} #nginx
spec:
  replicas: 2 # Numero de pods a serem gerados ao dar apply
  selector:
    matchLabels:
      app: nginx # Label o qual é comum entre o deployment e os pods, labels iguais fazem com que o deployment identifique aquele pod como
```

```
  template: # Template do pod, especificação do pod.
    metadata: # Metadados do pod.
      labels:
        app: nginx #Label que faz com que os pods se liguem aos deployments.
    spec:
      containers: #Array com informações dos containers.
      - name: nginx
        image: nginx:1.16
        ports:
        - containerPort: 8080
```

Código pra criar serviços

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
spec:
  selector:
    app: nginx #Liga o serviço aos pods com esse nome, pois ele vai no deployment que tem essa label e se liga a todos os pods que estão li
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
# Se um serviço tiver mais de uma porta, então as portas têm que ser nomeadas.
```

3 Service Types

Os serviços podem ser de 3 tipos.

ClusterIP → ClusterIP é o tipo default de serviço, esse serviço é responsável por criar uma comunicação entre os pods.

Como funciona a comunicação entre pods?

Pra começar, cada pod possui um ip interno da sua rede, então, é possível a comunicação entre pods por meio dos seus ips.

Um pod poderia se comunicar a outro por meio dos seu ip privado, visto que está na mesma rede. Mas como pods são efêmeros, isso se torna impraticável.

Para contornar esse problema, os serviços são uma ponte de comunicação entre pods, eles fornecem um ip interno estático em que os pods podem se comunicar.

Esse ip é o nome do serviço, transformado em ip pelo DNS.

Como os serviços sabem a que pods se ligar?

Dentro da configuração .yaml do serviço, existe um parametro chamado selector, que deve ser igual ao label dos pods definidos dentro da configuração .yaml dos deployments.

NodePort → Esse tipo de serviço externo expoe o ip externo dos node workers e permite conexão externa diretamente para eles. As requisições são entregues de forma “aleatoria” para os nodes.

- Não seguro
- Necessário adicionar o NodePort na config do serviço, que vai de 30000 ate 32000 e algo. Essa porta estará aberta em todos os nodes.

LoadBalancer → O loadbalancer é um serviço excterno executado pela empresa fornecedora de cloud que expoe um ip externo e redireciona as requisições a esse ip aos pods especificados em selector a suas portas especificadas no target port

- Mais seguro
- Necessário adicionar o NodePort, que é a porta que esse serviço escutará em seu ip externo.

Como são usadas variáveis de ambiente dentro das aplicações.

Cada aplicação conteinerizada usa algumas variáveis de ambiente para funcionar.

Em caso de uma aplicação local, essas variáveis de ambiente seriam setadas no nosso sistema operacional.

No caso de containers, essas variáveis são setadas nos próprios containers.

Aplicativos têm na documentação de suas imagens o nome das variáveis de ambiente que devem ser setadas pra utilizá-los.

Como setar variáveis de ambiente nos containers dos pods com o arquivo de deployment.

Dentro do arquivo deployment é onde especificamos quais as variáveis de ambiente dos containers.

Mais uma vez o exemplo do codigo de deployment

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mongodb
  labels:
    app: mongodb
spec:
  replicas: 1
  selector:
    matchLabels:
      app: mongodb
  template:
    metadata:
      labels:
        app: mongodb
    spec:
      containers: #Nas informações dos containers colocaremos as variáveis de ambiente.
      - name: mongo
        image: mongodb
        ports:
        - containerPort: 8080
        env:
        - name: {NOME_DA_VARIAVEL_DE_AMBIENTE1}
          value: {VALOR_DA_VARIAVEL1}
        - name: {NOME_DA_VARIAVEL_DE_AMBIENTE2}
          value: {VALOR_DA_VARIAVEL2}
```

E se os valores das variáveis de ambiente forem sigilosos?

Se isso acontecer, nós utilizamos os chamados Secrets, que são componentes/objetos do cluster que servem para guardar informações sigilosas em base64.

Criando um secret

```
apiVersion: v1
kind: Secret
metadata:
  name: mongo-secret
type: Opaque
data:
  nome-da-key-secreta1: {value em base64}
  nome-da-key-secreta2: {value em base64}
```

Para pegar o valor de alguma coisa em base64, digite no terminal do linux:

`echo -n {palavra para codificar} | base64`

Referenciando o secret no deployment

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mongodb
  labels:
    app: mongodb
spec:
  replicas: 1
  selector:
    matchLabels:
      app: mongodb
  template:
    metadata:
      labels:
```

```
        app: mongodb
    spec:
      containers: #Nas informações dos containers colocaremos as variáveis de ambiente.
      - name: mongo
        image: mongodb
        ports:
        - containerPort: 8080
        env:
        - name: {NOME_DA_VARIAVEL_DE_AMBIENTE1}
          valueFrom: #Isso indica que o valor será puxado de algum arquivo configmap ou secret.
            secretKeyRef:
              name: {nome-do-arquivo-secret} #Nome do secret para buscar dados dentro dele.
              key: {nome-da-variavel-secret1} #Nome da key para buscar o seu value.

        - name: {NOME_DA_VARIAVEL_DE_AMBIENTE2}
          valueFrom:
            secretKeyRef:
              name: {nome-do-arquivo-secret}
              key: {nome-da-variavel-secret2}
```

Se os valores não forem sigilosos mas guardados por fora.

É interessante guardar os valores de algumas variáveis fora do deployment.

Quando esses valores não são sigilosos, usamos o configmap, que é um secret pra variaveis não-secretas.

Criando um ConfigMap

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: mongo-configmap
data:
  nome-da-key-secreta1: valor em texto
```

Se referenciar ao configmap é a mesma coisa que se referenciar ao Secret, o que muda é o nomezinho de `secretKeyRef` para `configMapKeyRef`

Criando serviços externos

Para criar um serviço que recebe requests de uma fonte externa e envia para uma api, por exemplo, usa-se o tipo LoadBalancer e nodePort em sua configuração.

O nodePort faz com que o objeto node abra uma porta para requisições vindas de fora.

```
apiVersion: v1
kind: Service
metadata:
  name: mongo-express
spec:
  selector:
    app: mongo-express
    type: LoadBalancer
  ports:
    - protocol: TCP
      port: 8081
      targetPort: 8081
      nodePort: 30000 #Essa porta vai de 30000 até 32000 e alguma coisa.
```

Executar serviço externo no minikube e fazer rodar no localhost

Nesse momento, o serviço externo deve ter um ip externo pelo qual podemos nos conectar.

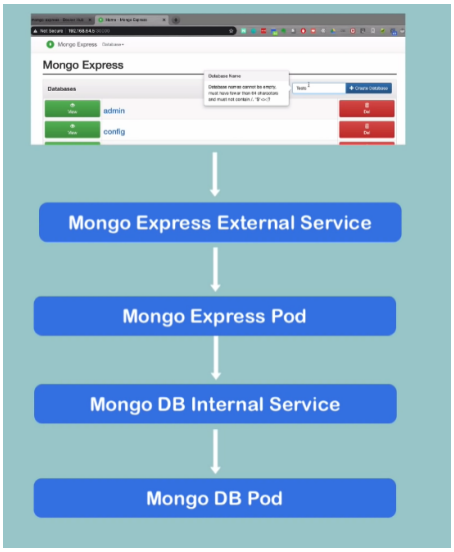
Mas no minikube, é necessário fazer uma coisa antes desse ip externo ser setado.

```
❯ kubectl get services
NAME                  TYPE           CLUSTER-IP      EXTERNAL-IP   PORT(S)          AGE
kubernetes            ClusterIP      10.96.0.1       <none>        443/TCP          167m
mongo-express-service LoadBalancer   10.102.15.103   <pending>     8081:30000/TCP   150m
mongodb-service       ClusterIP      10.107.205.86   <none>        27017/TCP        150m
 /mnt/c/Users/gabri/Desktop/kubernetes_test
❯
```

`minikube service {nome-do-serviço}`

Isso abrirá o navegador com o seu serviço rodando, ele te dará um ip externo e uma porta.

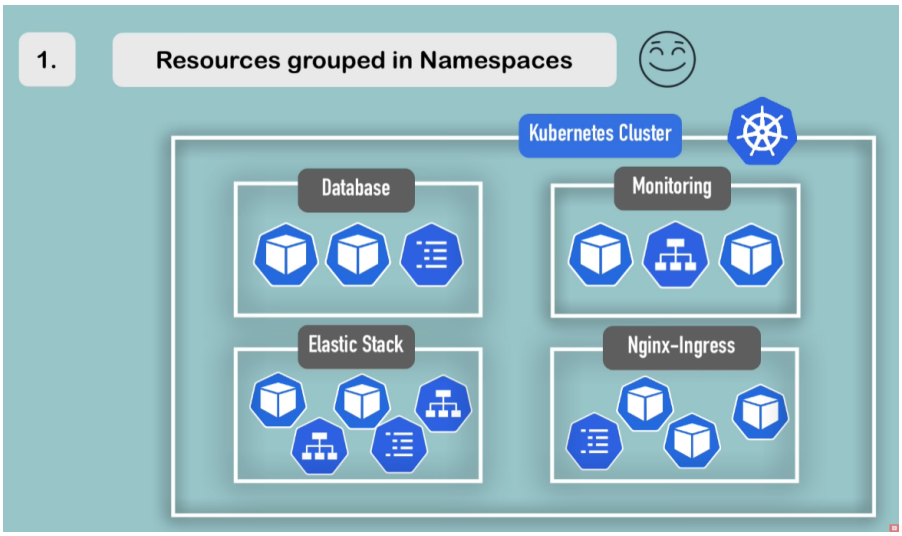
Request flow da aplicação criada ndo exemplo do vídeo com mongodb.



Namespace

Dentro de um cluster, as vezes se torna necessário organizar os recursos (objetos) por algum tipo de grupo. Se mais de uma equipe estiver trabalhando no mesmo cluster em aplicações diferentes, é interessante que elas dividam os seus recursos em namespaces.

Namespaces são uma espécie de cluster dentro do próprio cluster, servindo de organização dos recursos, deployments/pods, configmaps, secrets, services, etc.



Namespaces também são úteis para definir acesso dentro do cluster, algumas pessoas não têm acesso a todos os namespaces disponíveis dentro do cluster.

Outro uso é definir um limite de recursos que os workers de cada namespace pode usar.

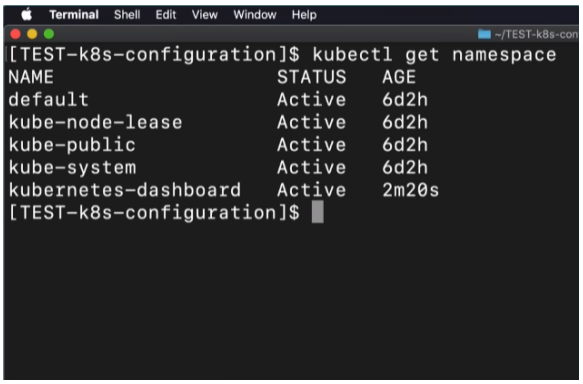
Alguns recursos que podem ser agrupados por namespace.

- Deployments / Pods
- Configmaps
- Secrets
- Services
- Ingresses

Recursos que não são agrupados por namespace.

- Nodes
- Volumes

Por padrão, o kubernetes tem alguns namespaces pré-definidos.



kubernetes-dashboard é exclusivo ao minikube

Existe o namespace `default` que representa o não-agrupamento dos recursos, sempre que criamos um objeto, ele será criado no namespace default, que representa o cluster inteiro, sem fronteiras.

Existe um parametro no kubectl get chamado `--namespace=` ou `-n` ele é utilizado para consultar namespaces especificos. Se não definirmos nada, ele procura pelo **namespace default.**

Criando namespaces

Namespaces podem ser criados com arquivos de configuração ou com `kubectl create namespace`

O recomendado é que o **namespace seja definido em cada arquivo** de deployment, service, configmap ou secret.

Exemplo da criação de serviço atribuido a um namespace.

```
apiVersion: v1
kind: Service
metadata:
  name: service
  namespace: my-namespace
spec:
  selector:
    app: service
  ports:
    - protocol: TCP
      port: 8081
      targetPort: 8081
```

Assim, o serviço será criado dentro do namespace, e isso também serve para outros objetos.

Kubens e kubectx para facilitar o uso de namespace. Pra nao ter que escrever o nome do namespace desejado sempre que queira consultar recursos de determinado namespace, use o kubens pra re-definir o namespace padrão, de default para o que você quiser.

Referindo-se a um serviço de outro namespace.

Dentro de configFiles que precisam se referir a algum outro serviço para se comunicar com ele, usamos `nomeDoServiço.namespaceDoServiço` dentro do configFile em questão.

Ingress

Em vez de usar serviços externos como o LoadBalancer para receber requests de fora, o ideal é que seja utilizado um componente chamado **Ingress** que, no lugar de ser um ip externo informado na url para fazer requests como era no caso do serviço externo, usamos domínios .com .net, etc.

O papel do ingress é receber requests http ou https do **domínio/rota** e ligá-lo a um serviço interno (backend), e esse serviço interno fazer contato com os pods que tem os containers que rodam a aplicação que cederão arquivos html, json, etc.

Criando um ingress

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: mongo-express-ingress
  namespace: mongo
spec:
  rules:
    - host: mongo-express.com # Domínio
      http:
        paths:
          - path: /
            pathType: Exact
            backend:
              service:
                name: mongo-express-service
                port:
                  number: 80 #Essa porta é a porta que o serviço escuta.
```

OBS: O tipo de serviço que o ingress redireciona tem que ser NodePort, mas não necessáriamente expor uma porta.

Configurando Ingress - Implementação de Ingress

O ingress por si só não funciona, é necessária a criação de um pod com o papel de **Ingress Controller.**

O Ingress Controller é a porta de entrada do cluster, e ele irá validar nossas regras de requisição definidas no rules do .yaml do ingress.

Essa regras são formas de como o acesso ao domínio chegarão. Por exemplo, se o cluster for acessado pela rota my-cluster.com/backend1, então o papel do controller é avaliar essa regra e enviar para o ingress, que por sua vez, irá redirecionar para o serviço específicado na parte path.

Instalando ingress padrão NGINX no minikube

`minikube addons enable ingress`

Https no ingress

Para ter um certificado https no ingress, é necessário inserir um parâmetro `tls` no configFile do Ingress.

Esse parâmetro `tls` se refere a um Secret de tipo especial, é um arquivo secret que guarda as informações do `tls`, certificado ssl.

Esse Secret precisa ter os dados `tlc.crt (certificate)` e `tls.key` criptografados em base 64, seu type precisa ser kubernetes.io/tls

O namespace do secret deve ser o mesmo que o do ingress

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: mongo-express-ingress
  namespace: mongo
spec:
  rules:
    tls:
    - host: mongo-express.com # Domínio
      secretName: {nome do secret tls}
      http:
        paths:
          - path: /
            pathType: Exact
            backend:
              service:
                name: mongo-express-service
                port:
                  number: 80 #Essa porta é a porta que o serviço escuta.
```

Secret tls

```
apiVersion: v1
kind: Secret
metadata:
  name: {nome do secret tls}
data:
  tls.crt: valor base 64
  tls.key: valor base 64
type: kubernetes.io/tls
```

Helm - Package manager

Gerenciador de pacotes de arquivos YAML para o kubernetes.

As vezes fica muito desgastante sempre montar os mesmos conjuntos de arquivo yaml para rodar alguma aplicação no Kubernetes.

O Helm cuida disso, criando os chamados Charts, que são pacotes de yaml files que podem ser baixados para rodar aplicações específicas.

`helm repo add {nomeDoPacote} {link do repositorio do chart}`

`helm repo update`

`helm install nome_do_pacote`

Instalando JupyterHub

Installing JupyterHub — Zero to JupyterHub with Kubernetes documentation (zero-to-jupyterhub.readthedocs.io)

Ao instalar o JupyterHub, uma série de configurações podem ser ajustadas para ficar da maneira que quisermos.

Um pacote helm vem, geralmente, com uma lista de valores que podem ser alterados.

Esses valores é um yaml chamado values.yaml

Esse values é como se fosse um configmap para que os configFiles peguem suas informações de um local centralizado.

É possível verificar esse values usando.

`helm show values jupyterhub/jupyterhub`

É possivel jogar esse output pra um arquivo com:

`helm show values jupyterhub/jupyterhub > config.yaml`

Edita-se esse config.yaml e joga na instalação

Instalação de fato

Caso não tenha sido instalado antes:

- Criar namespace do jupyterhub caso inexistente
- Rodar o proximo codigo

```
helm upgrade --cleanup-on-fail \
  --install <helm-release-name> jupyterhub/jupyterhub \
  --namespace <namespace> \
  --create-namespace #se não tiver sido criado\
  --version=<versaoDochart> \
  --values config.yaml #seu arquivo config.yaml
```

Codigo puro

```
helm upgrade --cleanup-on-fail \
  --install jupyterhub jupyterhub/jupyterhub \
  --namespace jupyter \
  --create-namespace \
  --version=2.0.0 \
  --values config.yaml
```

Somente upgrade

```
helm upgrade --cleanup-on-fail jupyterhub jupyterhub/jupyterhub
  --namespace jupyter \
  --version=2.0.0 \
  --values config.yaml
```

Erros de timeout podem ser causados por nome de imagem errada ou docker não logado