Kubernetes

Criado por Claudia Maia

O que é?

Ferramenta de orquestração de containers open-source. É utilizada para gerenciar aplicações em containers em diferentes ambientes.

Com a adoção dos microsserviços e a conteinirzação dessas aplicações, tornou muito difícil o gerenciamento desses múltiplos serviços usando scripts e soluções caseiras.

Vantagens de usar um orquestrador de containers:

- Alta **disponibilidade** ou nenhum downtime
- **escalabilidade** ou alta performance
- sistema de **recuperação** - backup e restore

🧩 Componentes:

- **Pod**: é a menor unidade do k8s. É uma abstração dos containers, um layer sobre os containers. O *pod* foi construído dessa forma para que não seja necessário trabalhar diretamente com uma tecnologia de containers, como o Docker.

  Normalmente, roda apenas uma aplicação

  Cada *pod* tem um endereço ip próprio

  Um *pod* pode morrer facilmente, ele é criado pra não ser persistido e, quando isso acontece, ele é recriado com um novo ip.

- **Serviço**: possuem ips permanentes. O ciclo de vida do *pod* e do *serviço* não são vinculados, então caso o *pod* morra, o *serviço* persistirá e você não precisará atualizar o ip's dos *pods*, caso haja uma aplicação e um db se comunicando dentro do seu *node*.

- **Ingress**: quando uma requisição externa é realizada, ela passa pelo *ingress* para passar pra baixo a requisição para o *serviço*. Dessa forma, ao invés de disponibilizar o ip do *node* + porta, a aplicação fica disponível por um dns usando o **https.**
- ConfigMap: configurações externas da sua aplicação. É nesse documento que vai informações como a url do seu serviço. Facilita alterações pequenas, sem a necessidade de ter que passar por todo o processo de fazer um novo build da imagem do container, subir para o repositório e baixar essa nova imagem para o pod. O pod lê as configurações do configMap.

  Informações sensíveis como usuário e senha não devem ser colocadas no configMap.
- Secret: documento que recebe as informações sensíveis. As informações salvas aqui não são salvas em texto, mas encodadas em base64. O pod lê as informações contidas nesse documento.
- Data Storage:
  - Volumes: são usados para garantir que o conteúdo do banco de dados seja persistido. Ele linka um disco rígido ao pod, esse storage pode ser na máquina local, ou seja na mesma máquina que o servidor ou remoto, fora do cluster de kubernetes, seja numa cloud ou num servidor on premise. Isso acontece porque o k8s não gerencia dados persistidos.

A vantagem dos sistemas distribuídos é caso seja necessário restartar o pod, é possível ter réplicas da aplicação em diferentes servidores de forma a não ter downtime nesses casos. Isso é feito pois os vários nós são ligados ao mesmo serviço. O serviço também funciona como um load balancer, ou seja, ele receberá a requisição e irá distribui-la para o pod que estiver de pé.

O deploy funciona como um blueprint para os pods, ou seja, ele é um abstração por cima dos pods, nele você irá especificar quantas réplicas serão feitas e quando escalar o aplicação ou quando fazer um downscale.

Não é possível fazer réplicas do banco de dados via deploy, porque o banco de dados trabalha com um estado que são os dados persistidos nele e, caso fosse possível trabalhar com réplicas do banco de dados, seria necessário um banco de dados externo compartilhado e seria necessário gerenciar quais pods estão

escrevendo nesse banco de dados e quais estão lendo para evitar inconsistência nos dados.

- StatefulSet: é usado em stateful apps ou banco de dados. Ele funciona similarmente ao deployment, no sentido de que também é responsável por criar réplicas dos pods e escalar ou fazer downscale do número de réplicas, mas tomando o cuidado de manter os dados de escrita e leitura sincronizados, de forma a não ter inconsistência nos dados. Embora esse componente tenha sido criado para essa função, não é uma atividade trivial gerenciar um banco de dados através de statefulset, é por isso que é mais comum o banco de dados ser hospedado fora do cluster do k8s.

🤖 Worker Machine no k8s cluster

Cada node tem vários pods nele

3 processos devem ser instalados em cada node

- container runtime: como o docker
- kubelet: faz a interface com o container runtime e o node
- Kube Proxy: é o responsável por direcionar a requisição. Ele possui uma lógica inteligente de direcionamento das requisições.

Worker nodes são quem realmente fazem o trabalho

👨‍✈️ Master process

Ele é o responsável por todos o gerenciamento de processos.

Serviços que rodam em todo master node

- api server: a interação é realizada por um cliente, seja a cli (através do kubelet) ou no k8s dashboard. É um gateway do cluster, ele recebe a requisição inicial através de uma alteração ou através de uma query. Ele também funciona como um gatekeeper se certificando que apenas requisições autenticadas e autorizadas chegam ao cluster. O pró dessa arquitetura é ter apenas um entrypoint para o cluster.
- scheduler: a api server recebe a requisição de schedule de um novo pod e transmite essa requisição para o scheduler. Esse, por sua vez, verifica pela requisição quanto de recurso será necessário e, de forma

inteligente, direciona a criação desse novo pod para o node mais adequado. O scheduler apenas decide em qual node o novo pod será criado, quem realmente executa essa requisição é o kubelet.

- controller manager: verifica mudanças de estado num cluster. Por exemplo, se um pod morre, é controller manager que faz essa monitoração e tenta recuperar o estado do cluster, o mais rápido possível. Enviando a requisição para o scheduler e seguindo o processo descrito anteriormente.
- etcd: é um key value store do estado do cluster. Todos os outros serviços listados acima funcionam lendo os dados salvos no etcd.

👮‍♂️👮‍♂️ Como adicionar novos master/node server?

- crie um novo servidor vazio
- instale todos processos de master/worker node
- adicione ao cluster

Existe diferença prática em criar um pod dessa forma:

`kubectl apply -f https://k8s.io/examples/pods/simple-pod.yaml`

ou dessa:

`kubectl create deployment nginx --image=nginx`?

Sim, há uma diferença prática significativa entre os dois comandos que você forneceu.

1. ***`kubectl apply -f https://k8s.io/examples/pods/simple-pod.yaml`***:
   - *Este comando cria um Pod usando um arquivo de manifesto YAML específico (`simple-pod.yaml`) fornecido pelo Kubernetes. Esse arquivo contém a definição do Pod, incluindo suas especificações, como nome, rótulos, contêineres, etc.*
   - *Ele é útil quando você tem um arquivo de manifesto personalizado que descreve exatamente como você deseja que o Pod seja configurado.*
2. ***`kubectl create deployment nginx --image=nginx`***:

- *Este comando cria um Deployment chamado "nginx" com um contêiner usando a imagem "nginx".*
- *O Deployment é uma abstração mais poderosa que gerencia a criação e escalonamento de réplicas de Pods. Ao criar um Deployment, você obtém funcionalidades adicionais, como rollouts, rollbacks, autoescalonamento, etc.*
- *O Deployment cria automaticamente os Pods conforme necessário para manter o número desejado de réplicas.*

Em resumo, enquanto ambos os comandos criam Pods, o segundo (`kubectl create deployment`) usa um nível de abstração mais alto (Deployment) que fornece gerenciamento avançado de replicação e recursos adicionais. O primeiro (`kubectl apply -f`) é mais direto, utilizando um arquivo de manifesto personalizado. A escolha entre eles dependerá das necessidades específicas do seu caso de uso. Se você precisar de recursos avançados de gerenciamento de replicação e rollout, o uso de Deployments será mais apropriado. Se você tiver uma configuração específica em um arquivo YAML, o uso de `kubectl apply -f` será a escolha certa.

📃 Configuration file:

- Metadata
- Especificação: todo tipo de configuração que você quer no seu componente. Cada kind tem seu atributo específico.
- status: é gerado automaticamente pelo k8s. O k8s checa as especificações do seu arquivo de configuração e as configurações do seu componente, caso as duas não sejam iguais, ele vai tentar igualar as duas, essa é a base do *self-healing* feature que o k8s tem. Esse status é atualizado continuamente. O k8s pega essa informação do status do etcd

💡Use um yaml validador online para checar indentação no arquivo yaml.

- O selector faz a conexão de um serviço com um deployment e o template dentro do spec do deployment lista as configurações para

os pods.

No secret file, o username e a senha são encodados em base64.

A ORDEM IMPORTA! O secret file tem que ser criado antes do arquivo do config file, pois as credenciais serão referenciadas lá através do secret file.

-/-/- : três traços (sem as barras) significam um novo documento na sintaxe no yaml. Então é possível criar um só yaml para o deployment e o service, por exemplo.

🚪 Como tornar um serviço externo?

- No objeto spec, especificar que o campo *type* é **LoadBalancer.**
- nodePort: tem que ser um valor entre 30000-32767

🌌 Namespace:

- Um espaço para organizar os recursos
- Um cluster virtual dentro do cluster
  - kube-system: não é criado para uso do admin do cluster. Lá são guardados processos da master e do kubectl
  - kube-public: informações públicas. Um configmap com as informações do cluster.
  - kube-node-lease: informações da saúde do node. Ele é responsável por determinar a disponibilidade do node.
  - default: é o namespace padrão. A menos que vc crie um novo namespace e direcione os recursos para esse novo namespace, é aqui que os recursos serão criados.

É possível criar um namespace pelo kubectl e também através de um arquivo de configuração.

❓ Quando usar um namespace?

- para agrupar recursos associados em um único namespace, por ex. um namespace: banco de dados, monitoração, etc
- muitos times usando a mesma aplicação: evitar conflitos quando dois times usam a mesma aplicação.
- reutilizar os componentes em ambiente de dev e produção
- reutilizar os componentes em dois ambientes de produção: o atual (blue) e novo (green)
- gerenciar permissionamento de times através de namespaces e também gerenciamento de recursos através dos namespaces.

🌄 Características dos namespaces:

- Você não pode acessar a maioria dos recursos de um namespace para outro
- Cada namespace deve ter o seu próprio configmap e secrets
- Serviços de outros namespaces podem ser acessados. Isso é feito acessando <nome-do-serviço>.<nome-do-namespace>
- Alguns componentes não podem ser criados dentro de um namespace, como: volume e node

🌐 Ingress

- Uma forma de abrir a aplicação para o mundo externo através de um dns amigável
- kind: Ingress
- Routing rules: domain address que redireciona internalmente para um serviço
- O serviceName e o servicePort declarados no Ingress devem corresponder ao nome do internal service e a targetPort do service
- Necessário ter um Ingress Controller: entrypoint para o cluster. Avalia as regras para redirecionar as chamadas.
- Existe diferentes implementações third-party

- É possível configurar apenas um host no ingress e redirecionar para diversos paths
- Também é possível criar sub-domínios, tais como analytics.myapp.com através de um único ingress. Nesse caso, cada sub-domínio será registrado como host.
- É possível configurar Certificado TLS:
    - adicionar tls no spec, adicionar um secretName e adicionar um tls.crt (tls certificate) e tls.key encodado em base 64

🪖Helm

- Gerenciador de pacotes para o k8s
- Templates yaml de stacks comumente usadas em k8s, como por exemplo, Elastic Search Stack. São chamados Helm Charts
- É possível criar seu próprio Helm Chart com a ferramenta
- Também é possível baixar e usar um existente
- Template engine: criação de template reutilizáveis para serviços similares dentro do seu cluster, usando {{.Values...}}, conjuntamente com a criação de values.yaml
- Criação de templates para diferentes ambientes
- Como criar um Helm Chart:
    - Estrutura de diretório:
        - nome do chart
            - Chart.yaml: meta info sobre o chart, tais como: nomer, versão, dependencias
            - values.yaml: default values que poderão ser sobrescritos depois
            - pasta charts: onde serão armazenados as dependências do charts
            - pasta templates: onde ficarão os templates de fato

💽 Como persistir dados usando volumes?

Não existe persistência de dados nativa no k8s, é necessário configurar um sistema de persistência de dados independente do ciclo de vida do pod.

O armazenamento deve estar disponível em todos os nodes, porque não é possível saber em que node o novo pod vai ser criador.

É necessário que o armazenamento não se perca mesmo que o cluster inteiro se perca.

- Persistent Volume:
  - um recurso do cluster
  - criado via YAML
    - kind: PersistentVolume
    - spec:
      - capacity:
        - storage: quanto de armazenamento?
  - precisa de um volume físico
  - Persistent volumes não podem ser separados em namespaces
  - Depois de ser disponibilizado, o persistent volume tem que ser chamado pela aplicação, logo é necessário configurar um persistent volume claim
    - Persistent Volume Claim:
      - kind: PersistentVolumeClaim
      - accessModes: read/write/readandwrite
      - O PVC é usado na configuração do pod
      - Claims tem estar no mesmo namespace do pod que está chamando o PVC
- Storage Class:
  - provisiona persistent volumes dinamicamente, assim que o PersistentVolumeClaim chama
  - é criado via YAML
  - kind: StorageClass

- é requerido pelo PVC

📍 StatefulSet

- guarda o estado salvando-o em algum volume
- Stateful apps são deployadas no k8s através do StatefulSet
- StatefulSet é mais difícil de ser replicado, em comparação ao deployment
- atribui uma identidade para cada pod
- Existem master pods (que podem escrever e ler dados) e worker pods (que só podem ler dados)
- Um novo pod clona os dados do pod anterior e depois continua sendo sincronizado conjuntamente com os outros
- É necessário configurar um PV para cada StatefulSet para evitar que os dados se percam quando o cluster morrer, por exemplo
- É necessário usar armazenamento remoto para poder realizar a transferência de dados para um novo pod.
- Seu nome é criado sequencialmente <nome-do-ST>-<numero-ordenado>
- A deleção ocorre do último pod criado para frente
- Quando o pod reinicia:
  - terá um novo ip
  - mas o nome e o endpoint será o mesmo
- Não é o componente mais adequado para aplicações conteinerzadas

🐂 Services

- Possui um ip estável
- tem a possibilidade de ter loadbalance
- ideal para comunicação dentro do cluster ou externa ao cluster
  - ClusterIP:

        - serviço default
        - o serviço sabe para qual pod endereçar uma requisição através do selector.
        - o serviço sabe qual porta direcionar uma requisição através da targetPort
    - Multi-Port:
        - em um caso de serviços com muitas portas abertas, é necessário nomear cada uma delas
    - Headless Services:
        - Caso de uso: para stateful apps
        - quando é necessário se comunicar com um pod específico
        - 3 tipos de atributos dos serviços:
            - clusterIP: default
            - LoadBalancer: É acessado externamente através um cloud provider. Um NodePort e um clusterIP são criados automaticamente.
            - NodePort: acessível por tráfego externo com uma porta fixa. Essa porta é definida no atributo NodePort. Não é seguro, pois abre uma porta para tráfego externo