DESCOMPLICANDO O KUBERNETES

O LIVRO

O que preciso saber antes de começar?

Qual linux devo usar?

A maioria das distribuições você conseguirá seguir esse treinamento. Algumas ferramentas importantes começaram a se tornar padrão, o que fez com que as diferenças entre as distribuições diminuísse. Por exemplo, devemos citar o systemd e o journald, que hoje estão presente na maioria das distribuições Linux.

Alguns sites que devemos visitar:

https://kubernetes.io/
https://github.com/kubernetes/kubernetes/
https://github.com/kubernetes/kubernetes/issues
https://www.cncf.io/certification/cka/
https://www.cncf.io/certification/ckad/
https://12factor.net/pt_br/

E o K8s?

O Kubernetes significa Piloto ou Timoneiro em Grego.
Ele foi baseado no Borg, produto criado no Google e utilizado por muitos anos internamente.

O Borg deu origem, além do k8s, ao Mesos e o Cloud Foundry, por exemplo.

Como o kubernetes é difícil de pronunciar para quem fala inglês, eles apelidaram de k8s, que se pronuncia Kates

Ele tornou-se open source em Junho de 2014.
Possui milhares de desenvolvedores e mais de 66k commits.
A cada 3 meses sai uma nova release.

O melhor app para rodar em container, principalmente no k8s, são aplicações que seguem o 12 factors.

Para simplificar, a arquitetura do k8s é feita de um manager e diversos workers nodes. Veremos também como rodar tudo em uma única maquina, mais isso é interessante somente para estudos, nunca para produção. 😀

O manager roda um API server, um scheduler e alguns controllers e o ETCD, um sistema de armazenamento para manter o estado do cluster, configurações dos containers e de rede.

O k8s expõe uma API, através do kube-apiserver, e você se comunica com ela através do comando **kubectl,** você também pode criar o seu próprio client, evidentemente.

O kube-scheduller cuida das requisições vindas da kube-apiserver para a criação de novos containers e verifica em qual o melhor node para isso.

Cada worker node roda dois processos, o kubelet e o kube-proxy.

O kubelet é quem recebe as requisições para criação e gerenciamento dos containers, bem como de seus recursos.

O kube-proxy cria e gerencia as regras para expor os containers na rede.

Portas que devemos nos preocupar:

MASTER
kube-apiserver => 6443 TCP
etcd server API => 2379-2380 TCP
Kubelet API => 10250 TCP
kube-scheduler => 10251 TCP
kube-controller-manager => 10252 TCP
Kubelet API Read-only => 10255 TCP

WORKER
Kubelet API => 10250 TCP
Kubelet API Read-only => 10255 TCP
NodePort Services => 30000-32767 TCP

Caso utilize o Weave como pod network, conforme demonstrado no material, lembre-se de liberar as portas:

A porta TCP 6783 e as portas UDP 6783 e 6784.

Lembre-se de fazer as liberações em seu firewall.

Primeiras palavras chaves do K8s

Vamos conhecer algumas palavras chaves para que você possa se sentir em casa:

Pod => No k8s, containers não são tratados individualmente, ele são gerenciados através de pods, que são agrupamentos de um ou mais containers dividindo o mesmo endereço, isso normalmente é composto por uma app principal e outras app de suporte para essa primeira.

Controllers => Responsável pela orquestração, ele interage com o api server para saber o status de cada objeto.

Deployment => É um dos principais controllers, ele é responsável por garantir que possui recursos disponíveis como IP e storage para o deploy dos ReplicaSet e DaemonSet.

Jobs ou CronJobs => Responsáveis pelo gerenciamento de task isoladas ou recorrentes

Para acompanhar o treinamento, você pode possuir dois cenários. Poderá utilizar somente uma máquina com o minikube instalado, ou então, subir um cluster com 03 máquinas, pode ser virtual ou cloud.

Nosso foco será em dois comandos, o kubeadm e o kubectl.

Instalação

MiniKube

Primeiro iremos ver como realizar a instalação do k8s através de uma única máquina, onde iremos utilizar o minikube, que é o responsável por fazer rodar todos os componentes do k8s juntos. Ele também traz o Docker engine. 😀

Realizar antes a instalação do VirtualBox 5.0.12:
https://download.virtualbox.org/virtualbox/5.0.12/

Antes de instalar o minikube, precisamos realizar a instalação do kubectl:

LINUX

```
# curl -LO
https://storage.googleapis.com/kubernetes-release/release/$(curl -s
https://storage.googleapis.com/kubernetes-release/release/stable.txt
)/bin/linux/amd64/kubectl

# chmod +x kubectl && mv kubectl /usr/local/bin/
```

MACOS

```
# curl -LO
https://storage.googleapis.com/kubernetes-release/release/`curl -s
https://storage.googleapis.com/kubernetes-release/release/stable.txt
`/bin/darwin/amd64/kubectl

# chmod +x kubectl && mv kubectl /usr/local/bin/
```

No MacOS, você pode também realizar a instalação através do brew:

```
# brew install kubectl
```

WINDOWS

```
# curl -Lo
https://storage.googleapis.com/kubernetes-release/release/v1.13.7/bi
n/windows/amd64/kubectl.exe
```

Se você utiliza o PSGallery:

```
# Install-Script -Name install-kubectl -Scope CurrentUser -Force
install-kubectl.ps1 [-DownloadLocation <path>]
```

doc:
https://kubernetes.io/docs/tasks/tools/install-minikube/

LINUX

```
# curl -Lo minikube
https://storage.googleapis.com/minikube/releases/latest/minikube-lin
ux-amd64 \ && chmod +x minikube
# sudo cp minikube /usr/local/bin && rm minikube
```

MACOS

```
# curl -Lo minikube
https://storage.googleapis.com/minikube/releases/latest/minikube-dar
win-amd64 \ && chmod +x minikube
# sudo cp minikube /usr/local/bin && rm minikube
```

WINDOWS

https://storage.googleapis.com/minikube/releases/v1.1.1/minikube-windows-amd64.exe

Com isso, já podemos iniciar o nosso minikube e seus componentes:

```
# minikube start
Starting local Kubernetes v1.10.0 cluster...
Starting VM...
Downloading Minikube ISO
 153.08 MB / 153.08 MB
[============================================] 100.00% 0s
Getting VM IP address...
Moving files into cluster...
Downloading kubeadm v1.10.0
Downloading kubelet v1.10.0
Finished Downloading kubelet v1.10.0
Finished Downloading kubeadm v1.10.0
Setting up certs...
Connecting to cluster...
Setting up kubeconfig...
Starting cluster components...
Kubectl is now configured to use the cluster.
Loading cached images from config file.
```

Para visualizar todos os nodes:

```
# kubectl get nodes
NAME        STATUS    ROLES     AGE       VERSION
minikube    Ready     master    1m        v1.10.0
```

Nesse caso somente temos um, afinal estamos rodando o minikube justamente por esse motivo, para roda-lo quando temos somente uma máquina. Lembre-se, ele é recomendado somente para estudos, nunca em produção.

Instalação em cluster com 03 nodes.

Já vimos como realizar a instalação do K8s através do minikube, agora vamos realizar a instalação em um cenário com 03 máquinas, seja VM ou cloud.

O setup que iremos utilizar para o treinamento é de máquinas com, no mínimo, a seguinte configuração:

- Debian, Ubuntu, Centos, Red Hat, Fedora, SuSe.
- 2 Core CPU
- 2GB de memória RAM

Primeiro, faça a atualização de seus nodes:

Subir os seguintes módulos do kernel em todos os nodes:

```
# vim /etc/modules-load.d/k8s.conf
br_netfilter
ip_vs_rr
ip_vs_wrr
ip_vs_sh
nf_conntrack_ipv4
ip_vs
```

Debian e família:

```
# apt-get update -y && apt-get upgrade -y
```

Red Hat e família:

```
# yum upgrade -y
```

Agora, bora realizar a instalação do Docker

```
# curl -fsSL https://get.docker.com | bash
```

Assim, teremos a última versão do docker instalado em todos os nodes.

Agora vamos adicionar o repo do Kubernetes em nossos nodes:

Debian e Familia:

```
# apt-get update && apt-get install -y apt-transport-https

# curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg |
apt-key add -
```

```
# echo "deb http://apt.kubernetes.io/ kubernetes-xenial main" >
/etc/apt/sources.list.d/kubernetes.list

# apt-get update

# apt-get install -y kubelet kubeadm kubectl
```

Red Hat e Família

```
# vim /etc/yum.repos.d/kubernetes.repo
[kubernetes]
name=Kubernetes
baseurl=https://packages.cloud.google.com/yum/repos/kubernetes-el7
-x86_64
enabled=1
gpgcheck=1
repo_gpgcheck=1
gpgkey=https://packages.cloud.google.com/yum/doc/yum-key.gpg
https://packages.cloud.google.com/yum/doc/rpm-package-key.gpg


# setenforce 0

# systemctl stop firewalld

# systemctl disable firewalld

# yum install -y kubelet kubeadm kubectl

# systemctl enable kubelet && systemctl start kubelet
```

Ainda na família do Red Hat, é importante configurar alguns parâmetros de kernel no sysctl:

```
# vim /etc/sysctl.d/k8s.conf
net.bridge.bridge-nf-call-ip6tables = 1
net.bridge.bridge-nf-call-iptables = 1

# sysctl --system
```

Agora em ambas as distribuições é muito importante também verificar se o drive cgroup usado pelo kubelet é o mesmo usado pelo docker, para verificar isso execute o seguinte comando:

```
# docker info | grep -i cgroup
```

```
Cgroup Driver: cgroupfs
```

```
# sed -i "s/cgroup-driver=systemd/cgroup-driver=cgroupfs/g"
/etc/systemd/system/kubelet.service.d/10-kubeadm.conf

# systemctl daemon-reload

# systemctl restart kubelet
```

docs: https://kubernetes.io/docs/setup/independent/install-kubeadm/

Antes de iniciar o nosso cluster precisamos desabilitar nossa swap, portanto:

```
# swapoff -a
```

E comente a entrada referente a swap em seu arquivo fstab:

```
# vim /etc/fstab
```

Uma observação, quando iniciarmos o nosso cluster ele irá mostrar uma mensagem de warning falando que é recomendável a utilização do docker 17.03. Essa versão é bastante antiga e utilizando o K8s com a versão 18.03 não tive nenhum problema de incompatibilidade ou coisa do gênero, mas fica aqui o aviso. 😉

Agora vamos iniciar o nosso cluster

Antes de iniciarmos o nosso cluster, vamos fazer o pull das imagens que serão utilizadas para montar o nosso cluster.

```
# kubeadm config images pull
```

Executar o comando abaixo somente no nó principal (master-node).

```
# kubeadm init --apiserver-advertise-address $(hostname -i)
```

O comando acima irá iniciar o cluster e em seguida exibirá a linha que de comando que preciso executar em meus outros nodes.

```
      [WARNING SystemVerification]: docker version is greater than
the most recently validated version. Docker version: 18.05.0-ce.
Max validated version: 17.03
...
To start using your cluster, you need to run the following as a
regular user:

  mkdir -p $HOME/.kube
  sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
  sudo chown $(id -u):$(id -g) $HOME/.kube/config
...
kubeadm join --token 39c341.a3bc3c4dd49758d5 IP_DO_MASTER:6443
--discovery-token-ca-cert-hash sha256:37092
...
```

O kubeadm executa uma série de pré-verificações para garantir o bom funcionamento do kubernetes.
Agora, para que possamos iniciar o gerenciamento do nosso cluster, vamos criar a estrutura de diretórios de nossa configuração, o kubeadm gentilmente já informa os comandos necessários na hora que executamos o init.

```
# mkdir -p $HOME/.kube
# sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
# sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

Nesses diretório, teremos toda a configuração necessária para o funcionamento do kubeclt.

Se você ainda não reiniciou seu máquina para que os modulos do kernel fossem carregados, execute o seguinte comando:

```
# modprobe br_netfilter ip_vs_rr ip_vs_wrr ip_vs_sh
nf_conntrack_ipv4 ip_vs
```

Agora vamos criar o nosso podnetwork, mais pra frente iremos entrar em maiores detalhes sobre eles.
Nesse exemplo, vamos utilizar o Weave, sensacional opção juntamente com o Cálico, na minha opinião os mais completos.

```
# kubectl apply -f
"https://cloud.weave.works/k8s/net?k8s-version=$(kubectl version |
base64 | tr -d '\n')"
serviceaccount/weave-net created
clusterrole.rbac.authorization.k8s.io/weave-net created
clusterrolebinding.rbac.authorization.k8s.io/weave-net created
role.rbac.authorization.k8s.io/weave-net created
rolebinding.rbac.authorization.k8s.io/weave-net created
```

```
daemonset.extensions/weave-net created
```

Vamos listar esses podnetwork:

 # kubectl get pods -n kube-system

NAME	READY	STATUS	RESTARTS	AGE
coredns-78fcdf6894-fzbp2	1/1	Running	0	9m
coredns-78fcdf6894-vp6td	1/1	Running	0	9m
etcd	1/1	Running	0	7m
kube-apiserver	1/1	Running	0	8m
kube-controller-manager	1/1	Running	0	8m
kube-proxy-smhxn	1/1	Running	0	9m
kube-scheduler-	1/1	Running	0	8m
weave-net-9b6kg	**2/2**	**Running**	**0**	**2m**

Agora já podemos adicionar os demais nodes ao cluster. Vamos pegar aquela linha de comando da saída do kubeadm init e executar nos outros dois nodes:

```
# kubeadm join --token 39c341.a3bc3c4dd49758d5 IP_DO_MASTER:6443
--discovery-token-ca-cert-hash sha256:37092
```

Para verificar todos os nodes do cluster execute:

```
# kubectl get nodes
NAME                STATUS    ROLES     AGE       VERSION
elliot-01    Ready     <none>    14s       v1.15.0
elliot-02    Ready     master    14m       v1.15.0
elliot-03    Ready     master    13m       v1.15.0
```

Primeiros Passos no K8s

Vamos ver alguns detalhes sobre o um dos nossos nodes:

```
# kubectl describe node elliot-03
Name:               elliot-03
Roles:              <none>
Labels:             beta.kubernetes.io/arch=amd64
                    beta.kubernetes.io/os=linux
                    kubernetes.io/hostname=elliot-03
Annotations:
kubeadm.alpha.kubernetes.io/cri-socket=/var/run/dockershim.sock
                    node.alpha.kubernetes.io/ttl=0

volumes.kubernetes.io/controller-managed-attach-detach=true
```

```
...

  Normal  NodeAllocatableEnforced  3m                kubelet,
elliot-03     Updated Node Allocatable limit across pods
  Normal  Starting                 3m                kube-proxy,
elliot-03  Starting kube-proxy.
  Normal  NodeReady                3m                kubelet,
elliot-03     Node elliot-03 status is now: NodeReady
```

Para visualizar novamente o token para inserção de novos nodes:

```
# kubeadm token create --print-join-command
kubeadm join --token 39c341.a3bc3c4dd49758d5 IP_DO_MASTER:6443
--discovery-token-ca-cert-hash sha256:37092
```

Está ruim de ficar digitando? use o auto-complete:

```
# source <(kubectl completion bash)
```

Deixe de forma definitiva:

```
# echo "source <(kubectl completion bash)" >> ~/.bashrc
```

Verificar os namespaces:

```
# kubectl get namespace
NAME          STATUS    AGE
default       Active    24m
kube-public   Active    24m
kube-system   Active    24m
```

Vamos listar os pods do namespace kube-system:

```
# kubectl get pods -n kube-system
NAME                           READY     STATUS    RESTARTS   AGE
coredns-78fcdf6894-fzbp2       1/1       Running   0          25m
coredns-78fcdf6894-vp6td       1/1       Running   0          25m
etcd-elliot-01                 1/1       Running   0
24m
kube-apiserver-linuxtips...    1/1       Running   0          24m
kube-controller-manager...     1/1       Running   0          24m
kube-proxy-fhz8p               1/1       Running   0          11m
kube-proxy-smhxn               1/1       Running   0          25m
kube-scheduler-linuxtips...    1/1       Running   0          24m
weave-net-9b6kg                2/2       Running   0          18m
weave-net-rlfpq                2/2       Running   0          11m
```

Tem pod escondido? Veja os pods de todos os namespaces:

```
# kubectl get pods --all-namespaces
```

Vamos executar o nosso primeiro exemplo, o nosso querido nginx de sempre:

```
# kubectl run nginx --image nginx
deployment.apps/nginx created
```

Vamos ver o objeto deployment:

```
# kubectl get deployments
NAME      DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
nginx     1         1         1            1           24s
```

Vamos ver a descrição do nosso deployment:

```
# kubectl describe deployment nginx
Name:                   nginx
Namespace:              default
CreationTimestamp:      Wed, 11 Jul 2018 01:21:41 +0000
Labels:                 run=nginx
Annotations:            deployment.kubernetes.io/revision=1
Selector:               run=nginx
Replicas:               1 desired | 1 updated | 1 total | 1
available | 0 unavailable
StrategyType:           RollingUpdate
MinReadySeconds:        0
RollingUpdateStrategy:  25% max unavailable, 25% max surge
Pod Template:
  Labels:  run=nginx
  Containers:
   nginx:
    Image:        nginx
    Port:         <none>
    Host Port:    <none>
    Environment:  <none>
    Mounts:       <none>
  Volumes:        <none>
Conditions:
  Type           Status  Reason
  ----           ------  ------
  Available      True    MinimumReplicasAvailable
  Progressing    True    NewReplicaSetAvailable
OldReplicaSets:  <none>
NewReplicaSet:   nginx-64f497f8fd (1/1 replicas created)
Events:
  Type    Reason             Age   From                   Message
  ----    ------             ----  ----                   -------
```

```
  Normal  ScalingReplicaSet  45s   deployment-controller  Scaled
up replica set nginx-64f497f8fd to 1
```

Vamos conferir se nosso cluster está tudo ok:

```
# kubectl get events
LAST ... KIND ...  REASON    SOURCE      MESSAGE
1m       pod       Pulling   kubelet     pulling image "nginx"
1m       pod       Pulled    kubelet     successfully pulled
1m       Pod       created   kubelet     Created container
1m       pod       started   kubelet     Started container
```

Nas últimas linhas podemos ver nosso pod nginx sendo construído.

Olha a saída do get deployment, só que no formato yams:

```
# kubectl get deployment nginx -o yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  annotations:
    deployment.kubernetes.io/revision: "1"
  creationTimestamp: 2018-07-11T01:21:41Z
  generation: 1
  labels:
    run: nginx
  name: nginx
...
```

Nós podemos pegar essa saída e redirecionar para um arquivo:

```
# kubectl get deployment nginx -o yaml > meu_primeiro.yaml
```

Vamos dar uma olhada nesse arquivo, quer dizer, nesse manifesto que acabamos de criar, remova a parte em negrito conforme abaixo, pois essas informações são sobre o deployment atual e seu status. Não vamos precisar disso, afinal iremos criar outro deployment, baseado no atual. :)

```
# vim meu_primeiro.yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  annotations:
    deployment.kubernetes.io/revision: "1"
  creationTimestamp: 2018-07-11T01:21:41Z
  generation: 1
  labels:
    run: nginx
  name: nginx
```

```
  namespace: default
  resourceVersion: "2602"
  selfLink:
/apis/extensions/v1beta1/namespaces/default/deployments/nginx
  uid: c42bfabf-84a8-11e8-beea-42010a8e0005
spec:
  progressDeadlineSeconds: 600
  replicas: 1
  revisionHistoryLimit: 2
  selector:
    matchLabels:
      run: nginx
  strategy:
    rollingUpdate:
      maxSurge: 25%
      maxUnavailable: 25%
    type: RollingUpdate
  template:
    metadata:
      creationTimestamp: null
      labels:
        run: nginx
    spec:
      containers:
      - image: nginx
        imagePullPolicy: Always
        name: nginx
        resources: {}
        name: nginx
        resources: {}
        terminationMessagePath: /dev/termination-log
        terminationMessagePolicy: File
      dnsPolicy: ClusterFirst
      restartPolicy: Always
      schedulerName: default-scheduler
      securityContext: {}
      terminationGracePeriodSeconds: 30
status:
  availableReplicas: 1
  conditions:
  - lastTransitionTime: 2018-07-11T01:21:46Z
    lastUpdateTime: 2018-07-11T01:21:46Z
    message: Deployment has minimum availability.
    reason: MinimumReplicasAvailable
    status: "True"
    type: Available
```

```
  - lastTransitionTime: 2018-07-11T01:21:41Z
    lastUpdateTime: 2018-07-11T01:21:46Z
    message: ReplicaSet "nginx-64f497f8fd" has successfully
progressed.
    reason: NewReplicaSetAvailable
    status: "True"
    type: Progressing
  observedGeneration: 1
  readyReplicas: 1
  replicas: 1
  updatedReplicas: 1
```

Agora vamos remover o nosso deploy do nginx:

```
# kubectl delete deployment nginx
deployment.extensions "nginx" deleted
```

Agora vamos novamente criá-lo, só que utilizando o nosso manifesto:

```
# kubectl create -f meu_primeiro.yaml
deployment.extensions/nginx created

# kubectl get pods
NAME                     READY     STATUS    RESTARTS   AGE
nginx-64f497f8fd-ldkhc   1/1       Running   0          36s
```

Vamos criar um segundo arquivo, baseado no nosso novo deployment:

```
# kubectl get deployment nginx -o yaml > meu_segundo.yaml
```

Vamos ver a diferença entre eles:

```
# diff meu_primeiro.yaml meu_segundo.yaml
```

Idênticos concordam ?.

Agora vamos tentar expor a porta desse nosso pod:

```
# kubectl expose deployment/nginx
```

Não foi possível, tomamos o seguinte erro:

error: couldn’t find port via --port flag or introspection See ’kubectl expose -h’ for help and examples.

Precisamos ter a entrada *ports* no arquivo, vamos remover o deployment que está em execução e adicionar a entrada ports:

```
# kubectl delete -f meu_primeiro.yaml
deployment.extensions "nginx" deleted
```

Vamos alterar o arquivo e adicionar as linhas destacadas abaixo:

```
# vim meu_primeiro.yaml
...
      spec:
        containers:
        - image: nginx
          imagePullPolicy: Always
          ports:
          - containerPort: 80
          name: nginx
          resources: {}
...
```

Agora vamos criá-lo novamente esse arquivo:

```
# kubectl create -f meu_primeiro.yaml
deployment.extensions/nginx created
```

Verificando se nosso pod já está em execução:

```
# kubectl get deploy,pod
NAME          DESIRED   CURRENT   UP-TO-DATE   AVAILABLE    AGE
deployment... 1         1         1            1            3m


NAME                           READY     STATUS    RESTARTS   AGE
pod/nginx-6f858d4d45-qxjlh   1/1       Running   0          3m
```

Vamos tentar expor novamente:

```
# kubectl expose deployment/nginx
service/nginx exposed
```

Vamos pegar qual é o ip de nosso service do nginx que acabamos de criar:

```
# kubectl get svc nginx
NAME      TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)   AGE
nginx     ClusterIP   10.96.123.231   <none>        80/TCP    25s
```

Vamos ver os detalhes desse pod:

```
# kubectl describe pod nginx-6f858d4d45-qxjlh
Name:           nginx-6f858d4d45-qxjlh
Namespace:      default
Node:           elliot-03/10.142.0.6
Start Time:     Wed, 11 Jul 2018 02:43:27 +0000
Labels:         pod-template-hash=2941480801
                run=nginx
Annotations:    <none>
Status:         Running
IP:             10.44.0.1
Controlled By:  ReplicaSet/nginx-6f858d4d45
Containers:
  nginx:
    Container ID:
docker://d09ca10d048ac137de9538d13296684f4736c1a3e3f8b1e3643a4d64e
b37190d
    Image:          nginx
    Image ID:
docker-pullable://nginx@sha256:a65beb8c90a08b22a9ff6a219c2f363e16c
477b6d610da28fe9cba37c2c3a2ac
    Port:           80/TCP
    Host Port:      0/TCP
    State:          Running
      Started:      Wed, 11 Jul 2018 02:43:28 +0000
    Ready:          True
    Restart Count:  0
    Environment:    <none>
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from
default-token-gdt7w (ro)
Conditions:
  Type              Status
  Initialized       True
  Ready             True
  ContainersReady   True
  PodScheduled      True
Volumes:
  default-token-gdt7w:
    Type:        Secret (a volume populated by a Secret)
    SecretName:  default-token-gdt7w
    Optional:    false
QoS Class:       BestEffort
Node-Selectors:  <none>
Tolerations:     node.kubernetes.io/not-ready:NoExecute for 300s
```

```
                 node.kubernetes.io/unreachable:NoExecute for 300s
Events:
  Type    Reason     Age   From                         Message
  ----    ------     ----  ----                         -------
  Normal  Scheduled  10m   default-scheduler            Successfully
assigned default/nginx-6f858d4d45-qxjlh to elliot-03
  Normal  Pulling    10m   kubelet, elliot-03  pulling image
"nginx"
  Normal  Pulled     10m   kubelet, elliot-03  Successfully pulled
image "nginx"
  Normal  Created    10m   kubelet, elliot-03  Created container
  Normal  Started    10m   kubelet, elliot-03  Started container
```

Vamos ver os pods com mais detalhes:

```
# kubectl get pods -o wide
NAME         READY   STATUS    RESTARTS   AGE  IP          NODE
nginx-6f...  1/1     Running   0          11m  10.44.0.1   linux...
```

Vamos deletar um dos containers de nosso pod:

```
# kubectl delete pods nginx-6f858d4d45-qxjlh
pod "nginx-6f858d4d45-qxjlh" deleted
```

Veja os pods novamente:

```
# kubectl get pods
NAME                       READY     STATUS    RESTARTS   AGE
nginx-6f858d4d45-r9zpf     1/1       Running   0          20s
```