DESCOMPLICANDO O KUBERNETES

O LIVRO

Componentes do K8s

O k8s tem os seguintes componentes principais:

- Master node
- Worker node
- Services
- Controllers
- Pods
- Namespaces e quotas
- Network e policies
- Storage

kube-apiserver é o central de operações do cluster k8s. Todas as chamadas, internas ou externas são tratadas por ele. Ele é o único que conecta no ETCD.

kube-scheduller usa um algoritmo para verificar em qual determinado pod deverá ser hospedado. Ele verifica os recursos disponíveis do node para verificar qual o melhor node para receber aquele pod.

No **ETCD** são armazenados o estado do cluster, rede e outras informações persistentes.

kube-controller-manager é o controle principal que interage com o kube-apiserver para determinar o seu estado. Se o estado não bate, o manager ira contactar o controller necessário para checar seu estado desejado. Tem diversos controller em uso como os endpoints, namespace e replication.

O **kubelet** interage com o Docker instalado no node e garante que os containers que precisavam estar em execução realmente estão.

O **kube-proxy** é o responsável por gerenciar a redes para os containers, é o responsável por expor portas dos containers

Supervisord é o responsável por monitorar e restabelecer, se necessário, o kubelet e o docker. Por esse motivo, quando existe algum problema em relação ao kubelet, como por exemplo o uso do cgroup driver diferente do que está rodando no Docker, você perceberá que ele ficará tentando subir o Kubelet frequentemente.

Pod é a menor unidade que você irá tratar no k8s. Você poderá ter mais de um container por Pod, porém vale lembrar que eles dividirão os mesmos recursos, como por exemplo IP. Uma das boas razões para se ter mais de um container em um Pod é o fato de você ter os logs consolidados..

O Pod, por poder possuir diversos containers, muito das vezes se assemelha a uma VM, onde você poderia ter diversos serviços rodando compartilhando o mesmo IP e demais recursos.

Services é uma forma de você expor a comunicação através de um NodePort ou LoadBalancer para distribuir as requisições entre diversos Pods daquele Deployment. Funciona como um balanceador de carga.

Container Network Interface

Para prover a rede para os containers, o K8s utiliza a especificação do **CNI**, Container Network Interface.
CNI é uma especificação que reúne alguma bibliotecas para o desenvolvimento de plugins para configuração e gerenciamento de redes para os containers. Ele provê uma comum interface entre as diversas soluções de rede para o k8s. Você encontra diversos plugins para AWS, GCP, Cloud Foundry entre outros.
https://github.com/containernetworking/cni

Enquanto o CNI define a rede dos pods, ele não te ajuda na comunicação entre os pods de diferentes nodes.
As características básicas da rede do k8s são:

- Todos os pods conseguem se comunicar entre eles em diferentes nodes
- Todos os nodes pode se comunicar com todos os pods
- Não utilizar NAT

Todos os IP dos pods e nodes são roteados sem a utilização de NAT. Isso é solucionado com a utilização de algum software que te ajudará na criação de uma rede Overlay.
Segue alguns:

- Weave
- Flannel
- Canal
- Calico
- Romana
- Nuage
- Contiv

https://kubernetes.io/docs/concepts/cluster-administration/addons/

Services

Criando um service ClusterIP

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

# kubectl get svc
NAME         TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)
AGE
kubernetes   ClusterIP   10.96.0.1        <none>        443/TCP
2h
nginx        ClusterIP   10.100.186.213   <none>        80/TCP
33s

# curl 10.100.186.213
...
Welcome to nginx!
...

# kubectl logs -f nginx-6f858d4d45-r9zpf
10.32.0.1 - - [11/Jul/2018:03:20:24 +0000] "GET / HTTP/1.1" 200
612 "-" "curl/7.52.1" "-"
10.32.0.1 - - [11/Jul/2018:03:20:32 +0000] "GET / HTTP/1.1" 200
612 "-" "curl/7.52.1" "-"

# kubectl delete svc nginx
service "nginx" deleted
```

Agora vamos criar nosso service ClusterIP, porém vamos criar um yaml com suas definições:

```
# vim primeiro-service-clusterip.yaml
apiVersion: v1
kind: Service
metadata:
  labels:
    run: nginx
  name: nginx-clusterip
  namespace: default
```

```
spec:
  ports:
  - port: 80
    protocol: TCP
    targetPort: 80
  selector:
    run: nginx
  type: ClusterIP
```

```
# kubectl create -f primeiro-service-clusterip.yaml
service/nginx-clusterip created
```

```
# kubectl get services
NAME             TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)
AGE
kubernetes       ClusterIP   10.96.0.1        <none>       443/TCP
2h
nginx-clusterip  ClusterIP   10.104.244.201   <none>       80/TCP
9s
```

```
# kubectl describe service nginx
Name:              nginx-clusterip
Namespace:         default
Labels:            run=nginx
Annotations:       <none>
Selector:          run=nginx
Type:              ClusterIP
IP:                10.104.244.201
Port:              <unset>  80/TCP
TargetPort:        80/TCP
Endpoints:         10.44.0.1:80
Session Affinity:  None
Events:            <none>
```

```
# kubectl delete -f primeiro-service-clusterip.yaml
service "nginx-clusterip" deleted
```

Agora vamos mudar um detalhe em nosso manifesto, vamos brincar com o nosso sessionaffinity:

```
# vim primeiro-service-clusterip.yaml
```

```
apiVersion: v1
kind: Service
metadata:
  labels:
```

```
    run: nginx
  name: nginx-clusterip
  namespace: default
spec:
  ports:
  - port: 80
    protocol: TCP
    targetPort: 80
  selector:
    run: nginx
  sessionAffinity: ClientIP
  type: ClusterIP
```

```
# kubectl create -f primeiro-service-clusterip.yaml
service/nginx-clusterip created

# kubectl get services
NAME             TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)
AGE
kubernetes       ClusterIP   10.96.0.1       <none>       443/TCP
2h
nginx-clusterip  ClusterIP   10.111.125.37   <none>       80/TCP
12s

# kubectl describe service nginx
Name:              nginx-clusterip
Namespace:         default
Labels:            run=nginx
Annotations:       <none>
Selector:          run=nginx
Type:              ClusterIP
IP:                10.111.125.37
Port:              <unset>  80/TCP
TargetPort:        80/TCP
Endpoints:         10.44.0.1:80
Session Affinity:  ClientIP
Events:            <none>
```

Com isso, agora temos como manter a sessão, ou seja, ele irá manter a conexão com o mesmo pod, respeitando o ip de origem, do cliente.

Agora podemos remover o service:

```
# kubectl delete -f primeiro-service-clusterip.yaml
service "nginx-clusterip" deleted
```

Criando um service NodePort

```
# kubectl expose deployment nginx --type=NodePort
service/nginx exposed

# kubectl get svc
NAME         TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)
AGE
kubernetes   ClusterIP   10.96.0.1      <none>        443/TCP
2h
nginx        NodePort    10.103.66.10   <none>        80:31059/TCP
11s

# kubectl delete svc nginx
service "nginx" deleted
```

Agora vamos criar um service NodePort, porém vamos criar um manifesto yaml com suas definições:

```
# vim primeiro-service-nodeport.yaml
apiVersion: v1
kind: Service
metadata:
  labels:
    run: nginx
  name: nginx-nodeport
  namespace: default
spec:
  externalTrafficPolicy: Cluster
  ports:
  - nodePort: 31111
    port: 80
    protocol: TCP
    targetPort: 80
  selector:
    run: nginx
  sessionAffinity: None
  type: NodePort
```

```
# kubectl create -f primeiro-service-nodeport.yaml
service/nginx-nodeport created
```

```
# kubectl get services
NAME             TYPE        CLUSTER-IP      ... PORT(S)
AGE
kubernetes       ClusterIP   10.96.0.1       ... 443/TCP        2h
nginx-nodeport   NodePort    10.100.250.181  ... 80:31111/TCP
14s
```

```
# kubectl describe service nginx
Name:                     nginx-nodeport
Namespace:                default
Labels:                   run=nginx
Annotations:              <none>
Selector:                 run=nginx
Type:                     NodePort
IP:                       10.100.250.181
Port:                     <unset>  80/TCP
TargetPort:               80/TCP
NodePort:                 <unset>  31111/TCP
Endpoints:                10.44.0.1:80
Session Affinity:         None
External Traffic Policy:  Cluster
Events:                   <none>
```

```
# kubectl delete -f primeiro-service-nodeport.yaml
service "nginx-nodeport" deleted
```

Criando um service LoadBalancer

```
# kubectl expose deployment nginx --type=LoadBalancer
service/nginx exposed

# kubectl get svc
NAME         TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)
AGE
kubernetes   ClusterIP   10.96.0.1       <none>        443/TCP
2h
nginx        NodePort    10.109.184.120  <pending>     80:31111/TCP
14s

# kubectl delete svc nginx
service "nginx" deleted
```

Agora vamos criar service NodePort, porém vamos criar um yaml com suas definições:

```
# vim primeiro-service-loadbalancer.yaml
apiVersion: v1
kind: Service
metadata:
  labels:
    run: nginx
  name: nginx-loadbalancer
  namespace: default
spec:
  externalTrafficPolicy: Cluster
  ports:
  - nodePort: 31222
    port: 80
    protocol: TCP
    targetPort: 80
  selector:
    run: nginx
  sessionAffinity: None
  type: LoadBalancer

# kubectl create -f primeiro-service-loadbalancer.yaml
service/nginx-loadbalancer created

# kubectl get services
NAME         TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)
AGE
```

```
kubernetes  ClusterIP  10.96.0.1      <none>       443/TCP
2h
nginx       NodePort   10.96.172.176  <pending>    80:31111/TCP
14s
```

 # kubectl describe service nginx

```
Name:                     nginx-loadbalancer
Namespace:                default
Labels:                   run=nginx
Annotations:              <none>
Selector:                 run=nginx
Type:                     LoadBalancer
IP:                       10.96.172.176
Port:                     <unset>  80/TCP
TargetPort:               80/TCP
NodePort:                 <unset>  31222/TCP
Endpoints:                10.44.0.1:80
Session Affinity:         None
External Traffic Policy:  Cluster
Events:                   <none>
```

 # kubectl delete -f primeiro-service-loadbalancer.yaml

```
service "nginx-loadbalancer" deleted
```

EndPoint

Sempre que criamos um service, automaticamente é criado um endpoint. O endpoint nada mais é do que o IP do pod que o service irá utilizar, por exemplo, quando criamos um service do tipo ClusterIP temos o seu IP, correto?
Agora, quando batemos nesse IP ele redireciona a conexão para o Pod através desse IP, o EndPoint.

Para listar os EndPoints criados, execute:

```
# kubectl get endpoints
NAME         ENDPOINTS         AGE
kubernetes   10.142.0.5:6443   4d
```

Vamos verificar esse endpoint com mais detalhes:

```
# kubectl describe endpoints kubernetes
Name:         kubernetes
Namespace:    default
Labels:       <none>
Annotations:  <none>
Subsets:
  Addresses:          10.142.0.5
  NotReadyAddresses:  <none>
  Ports:
    Name   Port  Protocol
    ----   ----  --------
    https  6443  TCP

Events:  <none>
```

Vamos fazer um exemplo, para isso, vamos realizar a criação de um deployment e na sequência um service para que possamos ver com mais detalhes os endpoints que serão criados.

```
# kubectl run nginx --image=nginx --port=80 --replicas=3
deployment.apps/nginx created

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

```
# kubectl get endpoints
NAME         ENDPOINTS                                   AGE
kubernetes   10.142.0.5:6443                             4d
nginx        10.44.0.1:80,10.44.0.2:80,10.44.0.3:80      2m

# kubectl describe endpoints nginx
Name:         nginx
Namespace:    default
Labels:       run=nginx
Annotations:  <none>
Subsets:
  Addresses:          10.44.0.1,10.44.0.2,10.44.0.3
  NotReadyAddresses:  <none>
  Ports:
    Name     Port  Protocol
    ----     ----  --------
    <unset>  80    TCP

Events:  <none>

# kubectl get endpoints -o yaml
apiVersion: v1
items:
- apiVersion: v1
  kind: Endpoints
  metadata:
    creationTimestamp: 2018-07-11T00:53:42Z
    name: kubernetes
    namespace: default
    resourceVersion: "39"
    selfLink: /api/v1/namespaces/default/endpoints/kubernetes
    uid: db16e5ab-84a4-11e8-beea-42010a8e0005
...
    - ip: 10.44.0.3
      nodeName: elliot-03
      targetRef:
        kind: Pod
        name: nginx-6f858d4d45-gn2qx
        namespace: default
        resourceVersion: "532254"
        uid: bdbf1c59-885d-11e8-beea-42010a8e0005
    ports:
    - port: 80
      protocol: TCP
kind: List
metadata:
```

```
  resourceVersion: ""
  selfLink: ""

# curl <IP_ENDPOINT>

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

# kubectl delete service nginx
service "nginx" deleted
```

Limitando Recursos

Quando criamos um Pod podemos especificar a quantidade de CPU e Memória (RAM) que pode ser consumida em cada container. Quando algum container contém a configuração de limite de recursos o Scheduler fica responsável por alocar esse container no melhor nó possível de acordo com os recursos disponíveis.

Podemos configurar dois tipos de recursos, CPU que é especificada em unidades de núcleos e Memória que é especificada em unidades de bytes.

Vamos criar nosso primeiro Deployment com limite de recursos, para isso vamos subir a imagem de um nginx e copiar o yaml do deployment:

```
# kubectl run nginx --image=nginx --port=80 --replicas=1

# kubectl get deployments
deployment.apps/nginx created

# kubectl get deployment nginx -o yaml > deployment-limitado.yaml

# vim deployment-limitado.yaml
...
     spec:
       containers:
       - image: nginx
         imagePullPolicy: Always
         name: nginx
         ports:
         - containerPort: 80
           protocol: TCP
         resources: # remove o {}
          limits:
            # Adicione as linhas abaixo
            memory: "256Mi"
            cpu: "200m"
          requests:
            memory: "128Mi"
            cpu: "50m"
         terminationMessagePath: /dev/termination-log
         terminationMessagePolicy: File
...
```

Vamos Adicionar as linhas em negrito acima.

Agora vamos criar nosso deployment e verificar os recursos:

```
# kubectl replace -f deployment-limitado.yaml
deployment.extensions/nginx replaced
```

Vamos acessar um container e testar a configuração.

```
# kubectl get pod
NAME                     READY     STATUS    RESTARTS   AGE
nginx-7dcffc9bff-pd46r   1/1       Running   0          9s

# kubectl exec -ti nginx-7dcffc9bff-pd46r -- /bin/bash
```

Agora no container, instale e execute o stress para simular a carga em nossos recursos, no caso CPU e memória.

```
# apt-get update && apt-get install -y stress

# stress --vm 1 --vm-bytes 128M --cpu 1
stress: info: [221] dispatching hogs: 1 cpu, 0 io, 1 vm, 0 hdd
```

Aqui estamos *stressando* o container, utilizando 124M de RAM e um core de CPU. Brinque de acordo com os limites que você estabeleceu.
Quando ultrapassar o limite configurado, você receberá um erro como abaixo, pois ele não conseguirá alocar os recursos:

```
# stress --vm 1 --vm-bytes 512M --cpu 1
stress: info: [230] dispatching hogs: 1 cpu, 0 io, 1 vm, 0 hdd
stress: FAIL: [230] (415) <-- worker 232 got signal 9
stress: WARN: [230] (417) now reaping child worker processes
stress: FAIL: [230] (451) failed run completed in 0s

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

Namespaces

No kubernetes temos um cara chamado de Namespaces como já vimos anteriormente, mas o que é um Namespace , nada mais é do que um cluster virtual dentro do próprio cluster físico do Kubernetes.
Namespaces são uma maneira de dividir recursos de um cluster entre vários ambientes , equipes ou projetos.

Vamos criar nosso primeiro namespaces:

```
# kubectl create namespace primeiro-namespace
namespace/primeiro-namespace created
```

Vamos listar todos os namespaces do kubernetes:

```
# kubectl get namespaces
NAME                 STATUS    AGE
default              Active    10d
kube-public          Active    10d
kube-system          Active    10d
primeiro-namespace   Active    3m
```

Pegar mais informações:

```
# kubectl describe namespace primeiro-namespace
Name:         primeiro-namespace
Labels:       <none>
Annotations:  <none>
Status:       Active

No resource quota.

No resource limits.
```

Como podemos ver nosso namespace ainda está cru sem configurações, vamos incrementar esse namespace e colocar limite de recursos, para isso vamos utilizar o LimitRange.

Vamos criar o manifesto do LimitRange:

```
# vim limitando-recursos.yaml
```

```
apiVersion: v1
kind: LimitRange
metadata:
  name: limitando-recursos
spec:
  limits:
  - default:
      cpu: 1
```

```
      memory: 100Mi
    defaultRequest:
      cpu: 0.5
      memory: 80Mi
    type: Container
```

Agora vamos adicionar esse LimitRange ao Namespace:

```
# kubectl create -f limitando-recursos.yaml -n primeiro-namespace
limitrange/limitando-recursos created
```

Listando o LimitRange

```
# kubectl get limitranges
No resources found.
```

Opa, não encontramos não é mesmo? mas claro esquecemos de passar nosso namespace na hora de listar:

```
# kubectl get limitrange -n primeiro-namespace
NAME                 CREATED AT
limitando-recursos   2018-07-22T05:25:25Z
```

Ou

```
# kubectl get limitrange --all-namespaces
NAMESPACE            NAME                 CREATED AT
primeiro-namespace   limitando-recursos   2018-07-22T05:25:25Z
```

Vamos dar um describe no LimitRange:

```
# kubectl describe limitrange -n primeiro-namespace
Name:       limitando-recursos
Namespace:  primeiro-namespace
Type        Resource  Min  Max  Default Request  Default Limit
Max Limit/Request Ratio
----        --------  ---  ---  ---------------  -------------
-----------------------
Container   cpu       -    -    500m             1              -
Container   memory    -    -    80Mi             100Mi          -
```

Como podemos observar adicionamos limites de memória e cpu para cada container que subir nesse Namespace, se algum container for criado dentro do Namespace sem as configurações de Limitrange o container vai pegar essa configuração default com limite de recursos.

Vamos criar um pod para verificar se o limite se aplicará.

```
# vim pod-limitrange.yaml
apiVersion: v1
kind: Pod
metadata:
  name: limit-pod
spec:
  containers:
  - name: meu-container
    image: nginx
```

Agora vamos criar um pod fora do namespace limitado e outro dentro do namespace limitado (primeiro-namespace) e vamos observar os limites de recursos de cada container e como foram aplicados:

```
# kubectl create -f pod-limitrange.yaml
pod/limit-pod created

# kubectl create -f pod-limitrange.yaml -n primeiro-namespace
pod/limit-pod created
```

Vamos listar esses pods e na sequência ver mais detalhes :

```
# kubectl get pods --all-namespaces
NAMESPACE              NAME          READY    STATUS     RESTARTS    AGE
default                limit-pod     1/1      Running    0           1m
...
primeiro-namespace     limit-pod     1/1      Running    0
22s

# kubectl describe pod limit-pod
Name:         limit-pod
Namespace:    default
Node:         elliot-03/10.142.0.6
Start Time:   Sun, 22 Jul 2018 05:35:01 +0000
Labels:       <none>
Annotations:  <none>
Status:       Running
IP:           10.44.0.1
Containers:
  meu-container:
    Container ID:
docker://3e5e3ca909fd83cdd59f48dd12663328c1ef7cda188fa0a590f3ffcea
d1c70dc
...
```

```
# kubectl describe pod limit-pod -n primeiro-namespace
Name:         limit-pod
Namespace:    primeiro-namespace
Node:         elliot-03/10.142.0.6
Start Time:   Sun, 22 Jul 2018 05:36:06 +0000
Labels:       <none>
Annotations:  kubernetes.io/limit-ranger=LimitRanger plugin set:
cpu, memory request for container meu-container; cpu, memory limit
for container meu-container
Status:       Running
IP:           10.44.0.2
Containers:
  meu-container:
    Container ID:
docker://4085b0c1e716f173378a9352213556f298e2caf3bf750919d9f803151
885e4d6
...
    Limits:
      cpu:     1
      memory:  100Mi
    Requests:
      cpu:        500m
      memory:     80Mi
```

Como podemos ver o Pod no Namespace primeiro-namespace está com limit de recursos configurados.

Kubectl taint

O Taint nada mais é do que adicionar propriedades ao nó do cluster para impedir que os pods sejam alocados em nós inapropriados.

Por exemplo, todo nó master do cluster é marcado para não receber pods que não sejam de gerenciamento do cluster.
O nó master está marcado com o taint NoSchedule assim o scheduler do Kubernetes não aloca pods no nó master , e procurar outros nós no cluster sem essa marca.

```
# kubectl describe node elliot-01 | grep -i taint
Taints:             node-role.kubernetes.io/master:NoSchedule
```

Vamos testar algumas coisas e permita que o nó master rode outros pods.

Primeiro vamos rodar 3 réplicas de nginx :

```
# kubectl run nginx --image=nginx --replicas=3
deployment.apps/nginx created

# kubectl get pods -o wide
NAME        READY   STATUS    RESTARTS   AGE   ...           NODE
nginx...    1/1     Running   0          1m    ...    elliot-02
nginx...    1/1     Running   0          1m    ...    elliot-02
nginx...    1/1     Running   0          1m    ...    elliot-03
```

Vamos adicionar a marca NoSchedule aos nós slave também para ver como eles se comportam.

```
# kubectl taint node elliot-02 key1=value1:NoSchedule
node/elliot-02 tainted

# kubectl taint node elliot-03 key1=value1:NoSchedule
node/elliot-03 tainted

# kubectl describe node elliot-02 | grep -i taint
Taints:             key1=value1:NoSchedule

# kubectl describe node elliot-03 | grep -i taint
Taints:             key1=value1:NoSchedule
```

Agora vamos aumentar a quantidade de réplicas:

```
# kubectl scale deployment nginx --replicas=5
deployment.extensions/nginx scaled

# kubectl get pods  -o wide
```

```
NAME        READY   STATUS    RESTARTS   AGE   ...    NODE
nginx...    1/1     Running   0          6m    ...    elliot-02
nginx...    1/1     Running   0          6m    ...    elliot-02
nginx...    0/1     Pending   0          26s   ...    <none>
nginx...    0/1     Pending   0          26s   ...    <none>
nginx...    1/1     Running   0          6m    ...    elliot-03
```

Como podemos ver , as nova replicas ficaram órfãs esperando aparece um nó com as prioridades adequadas para o Scheduler.

Vamos remover o esse Taint dos nossos nós slave:

```
# kubectl taint node elliot-02 key1:NoSchedule-
node/elliot-02 untainted

# kubectl taint node elliot-03 key1:NoSchedule-
node/elliot-03 untainted

# kubectl get pods  -o wide
NAME        READY   STATUS    RESTARTS   AGE   ...    NODE
nginx...    1/1     Running   0          20m   ...    elliot-02
nginx...    1/1     Running   0          20m   ...    elliot-02
nginx...    1/1     Running   0          14s   ...    elliot-03
nginx...    1/1     Running   0          14s   ...    elliot-02
nginx...    1/1     Running   0          20m   ...    elliot-03
```

Existem vários tipos de marcas que podemos usar para classificar os nós, vamos testar uma outra chamada NoExecute, que impede o Scheduler de agendar Pods nesses nós.

```
# kubectl taint node elliot-02 key1=value1:NoExecute
node/elliot-02 tainted

# kubectl taint node elliot-03 key1=value1:NoExecute
node/elliot-03 tainted

# kubectl get pods
NAME                     READY     STATUS    RESTARTS   AGE
nginx-64f497f8fd-7k9xb   0/1       Pending   0          40s
nginx-64f497f8fd-gvnm4   0/1       Pending   0          40s
nginx-64f497f8fd-m7hdc   0/1       Pending   0          40s
nginx-64f497f8fd-sgs5k   0/1       Pending   0          40s
nginx-64f497f8fd-sjf4x   0/1       Pending   0          40s
```

Como podemos ver todos os Pods estão órfãs. Porque o nó master tem a marca taint NoScheduler default do kubernetes e os nós Slave tem a marca NoExecute.

Vamos diminuir a quantidade de réplicas para ver o que acontece:

```
# kubectl scale deployment nginx --replicas=1
deployment.extensions/nginx scaled

# kubectl get pods
NAME                     READY     STATUS    RESTARTS   AGE
nginx-64f497f8fd-glxlf   0/1       Pending   0          40s
```

Vamos remover o taint NoExecute do nó slave:

```
# kubectl taint node elliot-02 key1:NoExecute-
node/elliot-02 untainted

# kubectl taint node elliot-03 key1:NoExecute-
node/elliot-03 untainted

# kubectl get pods
NAME                     READY     STATUS    RESTARTS   AGE
nginx-64f497f8fd-glxlf   1/1       Running   0          1m
```

Agora temos um nó operando normalmente.
Mas e se nossos Slaves ficarem indisponíveis, podemos rodar Pods no nó master? Claro que podemos, vamos configurar nosso nó master para que o Scheduler consiga agenda Pods nele.

```
# kubectl taint nodes --all node-role.kubernetes.io/master-
node/elliot-01 untainted


# kubectl describe node elliot-01 | grep -i taint
Taints:             <none>
```

Agora vamos aumentar a quantidade de réplicas do nosso nginx.

```
# kubectl scale deployment nginx --replicas=4
deployment.extensions/nginx scaled

# kubectl get pods -o wide
NAME       READY     STATUS    RESTARTS   AGE    ...    NODE
nginx...   1/1       Running   0          40s    ...   elliot-02
nginx...   1/1       Running   0          28m    ...   elliot-02
nginx...   1/1       Running   0          40s    ...   elliot-01
nginx...   1/1       Running   0          40s    ...   elliot-03
```

Vamos adicionar o Taint NoExecute nos nós slave para ver o que acontece:

```
# kubectl taint node elliot-02 key1=value1:NoExecute
node/elliot-02 tainted
```

```
# kubectl taint node elliot-03 key1=value1:NoExecute
node/elliot-03 tainted

# kubectl get pods -o wide
NAME      READY    STATUS    RESTARTS   AGE   ...   NODE
nginx...   1/1     Running   0          23s   ...  elliot-01
nginx...   1/1     Running   0          23s   ...  elliot-01
nginx...   1/1     Running   0          23s   ...  elliot-01
nginx...   1/1     Running   0          23s   ...  elliot-01

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

O Scheduler alocou tudo no nó master, como podemos ver o Taint pode ser usado para ajustar configurações de qual Pod deve ser alocado em qual nó.
vamos permitir que nosso Scheduler aloque e execute os Pods em todos os nós:

```
# kubectl taint node --all key1:NoSchedule-
node/elliot-01 untainted
node/elliot-02 untainted
node/elliot-03 untainted

# kubectl taint node --all key1:NoExecute-
node/elliot-01 untainted
node/elliot-02 untainted
node/elliot-03 untainted
```