# Índice



ESCOLA SUPERIOR DE REDES RNP

# Sessão 3: Agendamento no Kubernetes

## 1) Agendamento manual

1. Antes de iniciar, execute o comando abaixo:

```
# lab-3.1.1
```

2. Crie um pod com o nome `sched` e imagem `nginx:alpine`. Qual o estado desse pod?

▼ *Visualizar resposta*

Criar o pod é bastante fácil:

```
# kubectl run sched --image=nginx:alpine
pod/sched created
```

Mas... porque ele não está executando? Note que seu estado permanece como `Pending`, mesmo que aguardemos um bom tempo.

```
# kubectl get pod sched
NAME    READY   STATUS    RESTARTS   AGE
sched   0/1     Pending   0          21s
```

3. Qual o motivo disso?

▼ *Visualizar resposta*

O `kube-scheduler` é resposável por realizar o agendamento de pods no *cluster* Kubernetes. Como visto a seguir, nenhum pod com esse nome está em execução.

```
# kubectl get pod --all-namespaces | grep 'kube-scheduler'
```

4. Agende o pod manualmente no *node* `s2-node-1`. Verifique o funcionamento de sua configuração.

▼ *Visualizar resposta*

Primeiro, vamos remover o pod.

```
# kubectl delete pod sched
pod "sched" deleted
```

A seguir, criamos o arquivo `sched.yaml` com o conteúdo que se segue. Note que o atributo `.spec.nodeName` é utilizado para especificar em qual *node* o pod deverá executar.

```
1 apiVersion: v1
2 kind: Pod
3 metadata:
4   name: sched
5 spec:
6   nodeName: s2-node-1
7   containers:
8   - image: nginx:alpine
9     name: sched
```

Usando o arquivo acima, criamos o pod:

```
# kubectl create -f sched.yaml
pod/sched created
```

E, agora sim, ele está em execução. Confira com o comando `kubectl get pod`:

```
# kubectl get pod sched -o custom-
columns=NAME:.metadata.name,NODE:.spec.nodeName,STATUS:.status.phase
NAME    NODE        STATUS
sched   s2-node-1   Running
```

# 2) Etiquetas e seletores

1. Antes de iniciar, execute o comando abaixo:

```
# lab-3.2.1
```

2. Diversos pods foram publicados, com os *labels* `sys`, `pub` e `genre`. Quantos pods existem no sistema `nes`?

   ▼ *Visualizar resposta*

   Pode-se utilizar a opção `--selector` ou `-l` para aplicar seletores a uma listagem, como a fornecida pelo comando `kubectl get pod`. Usando os *labels* aplicados aos pods em execução, podemos fazer a filtragem solicitada:

   ```
   # kubectl get pod -l sys=nes
   NAME          READY   STATUS    RESTARTS   AGE
   castlevania   1/1     Running   0          72s
   megaman       1/1     Running   0          72s
   zelda         1/1     Running   0          72s
   ```

3. Quantos pods possuem o gênero `platform`?

   ▼ *Visualizar resposta*

   Assim como no item anterior, basta ajustar o seletor de acordo com o *label* objetivado.

   ```
   # kubectl get pod -l genre=platform
   NAME      READY   STATUS    RESTARTS   AGE
   megaman   1/1     Running   0          87s
   sonic     1/1     Running   0          87s
   ```

4. Quantos objetos existem no *publisher* `konami`, incluindo pods, deployments e demais?

   ▼ *Visualizar resposta*

   Novamente, utilizamos a funcionalidade de seletores de *labels*. Adicionalmente, para obter todos os objetos utilizamos `kubectl get all`:

   ```
   # kubectl get all -l pub=konami
   NAME              READY   STATUS    RESTARTS   AGE
   pod/castlevania   1/1     Running   0          104s

   NAME                         READY   UP-TO-DATE   AVAILABLE   AGE
   deployment.apps/metalgear    1/1     1            1           103s
   ```

5. Qual é o pod no sistema `ps1`, *publisher* `sony` e gênero `racing`?

   ▼ *Visualizar resposta*

   Pode-se especificar múltiplos *labels* separados por vírgula, como visto abaixo.

   ```
   # kubectl get pod -l sys=ps1,pub=sony,genre=racing
   NAME          READY   STATUS    RESTARTS   AGE
   granturismo   1/1     Running   0          2m4s
   ```

6. Exiba todos os pods que NÃO pertencem ao sistema `genesis`.

   ▼ *Visualizar resposta*

   Além de igualdade, pode-se também especificar desigualdade entre uma chave e valor de um *label*:

```
# kubectl get pod -l sys!=genesis
NAME                          READY   STATUS    RESTARTS   AGE
castlevania                   1/1     Running   0          8m34s
ff7                           1/1     Running   0          8m33s
granturismo                   1/1     Running   0          8m33s
megaman                       1/1     Running   0          8m34s
metalgear-5b86b6dbb9-wpmlf    1/1     Running   0          8m33s
zelda                         1/1     Running   0          8m34s
```

7. Exiba todos os pods que possuem gênero `racing` OU `fighting`.

   ▼ *Visualizar resposta*

   Pode-se também buscar por presença de uma chave em um grupo de valores. Veja:

```
# kubectl get pod -l 'genre in (racing, fighting)'
NAME            READY   STATUS    RESTARTS   AGE
granturismo     1/1     Running   0          10m
roadrash        1/1     Running   0          10m
streetfighter   1/1     Running   0          10m
```

# 3) Taints e tolerations

1. Remova os objetos criados na atividade anterior com o comando abaixo.

```
# kubectl delete pod,rs,deploy --all
```

2. Quantos *nodes* existem no sistema? Eles possuem algum *taint*?

   ▼ *Visualizar resposta*

   Primeiro, vamos determinar quantos *nodes* integram o *cluster*, e seus nomes:

```
# kubectl get node
NAME          STATUS   ROLES                  AGE   VERSION
s2-master-1   Ready    control-plane,master   17h   v1.23.5
s2-node-1     Ready    <none>                 17h   v1.23.5
```

   Para visualizar *taints* presentes em um *node*, basta utilizar o comando `kubectl describe node`. Faremos isso para cada um dos membros do *cluster*:

```
# kubectl describe node s2-master-1 | grep 'Taints:'
Taints:             node-role.kubernetes.io/master:NoSchedule
```

```
# kubectl describe node s2-node-1 | grep 'Taints:'
Taints:             <none>
```

3. Aplique um *taint* ao *host* `s2-node-1` com o par chave-valor `type=citric` e efeito `NoSchedule`.

   ▼ *Visualizar resposta*

   Para tanto, basta utilizar o comando `kubectl taint node`:

   ```
   # kubectl taint node s2-node-1 type=citric:NoSchedule
   node/s2-node-1 tainted
   ```

   A seguir, verificamos o funcionamento da configuração.

   ```
   # kubectl describe node s2-node-1 | grep 'Taints:'
   Taints:             type=citric:NoSchedule
   ```

4. Crie um pod com o nome `banana` e imagem `nginx:alpine`. Qual é o estado desse pod, e porque?

   ▼ *Visualizar resposta*

   Comaçamos criando o pod:

   ```
   # kubectl run banana --image=nginx:alpine
   pod/banana created
   ```

   Então, verificamos seu estado. Note que o mesmo encontra-se como `Pending`; por que será?

   ```
   # kubectl get pod banana
   NAME     READY   STATUS    RESTARTS   AGE
   banana   0/1     Pending   0          5s
   ```

   A última linha de eventos do pod, visualizável com `kubectl describe pod`, mostra a razão: os dois *nodes* do *cluster* possuem *taints* aos quais o pod `banana` não possui tolerância.

   ```
   # kubectl describe pod banana | tail -n1
     Warning  FailedScheduling  57s   default-scheduler  0/2 nodes are available: 1
   node(s) had taint {node-role.kubernetes.io/master: }, that the pod didn't
   tolerate, 1 node(s) had taint {type: citric}, that the pod didn't tolerate.
   ```

5. Crie um pod com o nome `orange` e imagem `nginx:alpine`, com tolerância ao *taint* `type=citric`. O

que ocorre com esse pod?

▼ *Visualizar resposta*

Para adicionar tolerância ao pod, iremos criá-lo via arquivo YAML. No arquivo novo `orange.yaml`, inclua o conteúdo que se segue:

```
 1 apiVersion: v1
 2 kind: Pod
 3 metadata:
 4   name: orange
 5 spec:
 6   containers:
 7   - image: nginx:alpine
 8     name: orange
 9   tolerations:
10   - key: "type"
11     operator: "Equal"
12     value: "citric"
13     effect: "NoSchedule"
```

A seguir, criamos o pod via arquivo.

```
# kubectl create -f orange.yaml
pod/orange created
```

Note que o pod `orange` entra então em execução, ao contrário de `banana`. Isso se deve à sua tolerância ao *taint* `type=citric` — não por acaso, ele é agendado no *node* `s2-node-1`, como visto abaixo.

```
# kubectl get pod orange -o wide
NAME     READY   STATUS    RESTARTS   AGE   IP           NODE        NOMINATED
NODE   READINESS GATES
orange   1/1     Running   0          8s    10.44.0.10   s2-node-1   <none>
<none>
```

6. Remova o *taint* do *node* `s2-master-1`. O que acontece com o pod `banana`?

▼ *Visualizar resposta*

Para remover *taints*, basta usar `kubectl taint node`:

```
# kubectl taint node s2-master-1 node-role.kubernetes.io/master:NoSchedule-
node/s2-master-1 untainted
```

Imediatamente o pod `banana` torna-se agendável, e é por conseguinte alocado no *node* `s2-master-1`.

```
# kubectl get pod banana -o wide
NAME     READY   STATUS    RESTARTS   AGE     IP          NODE          NOMINATED
NODE   READINESS GATES
banana   1/1     Running   0          6m43s   10.32.0.4   s2-master-1   <none>
<none>
```

7. Adicione um *taint* ao *node* `s2-master-1` com o par chave-valor `type=berry` e efeito `NoSchedule`. Qual o efeito dessa configuração no pod `banana`?

   ▼ *Visualizar resposta*

   Vamos adicionar o *taint*:

   ```
   # kubectl taint node s2-master-1 type=berry:NoSchedule
   node/s2-master-1 tainted
   ```

   Note que ele não afeta o pod `banana`, já que seu efeito é de `NoSchedule` — isso impede o agendamento **futuro** de pods intolerantes a esse *taint*, mas não afeta pods previamente agendados.

   ```
   # kubectl get pod banana
   NAME     READY   STATUS    RESTARTS   AGE
   banana   1/1     Running   0          8m
   ```

8. Altere o efeito do *taint* aplicado no passo anterior para `NoExecute`. E agora, o que ocorre com o pod `banana`?

   ▼ *Visualizar resposta*

   Adicionamos o *taint*:

   ```
   # kubectl taint node s2-master-1 type=berry:NoExecute
   node/s2-master-1 tainted
   ```

   Veja que, diferentemente do *taint* anterior, o efeito deste é de `NoExecute`: isso força pods intolerantes ao *taint* a serem despejados (*evicted*) do *node* em questão.

   ```
   # kubectl get pod banana
   NAME     READY   STATUS        RESTARTS   AGE
   banana   0/1     Terminating   0          8m54s
   ```

   Note ainda que o pod, após despejado, é terminado e não mais existe. Para garantir sua recriação seria necessário o uso de um objeto hierarquicamente superior, como um ReplicaSet ou Deployment.

```
# kubectl get pod banana
Error from server (NotFound): pods "banana" not found
```

# 4) Afinidade em relação a nodes

1. Antes de iniciar, execute o comando abaixo:

```
# lab-3.4.1
```

2. Quantos *labels* estão associados ao *node* `s2-master-1`? E quanto ao *node* `s2-node-1`?

▼ *Visualizar resposta*

Para visualizar *labels*, podemos utilizar `kubectl describe node` em conjunção com o comando `grep`. Mas, para contar o número de *labels* como solicitado, pode ser mais interessante utilizar JSONPath para buscar o campo específico, como se segue:

```
# kubectl get node s2-master-1 -o jsonpath='{.metadata.labels}' | tr , '\n' | sed
's/}/\n/' | wc -l
8
```

Fazemos o mesmo procedimento para o outro *node* do *cluster*.

```
# kubectl get node s2-node-1 -o jsonpath='{.metadata.labels}' | tr , '\n' | sed
's/}/\n/' | wc -l
5
```

3. Qual é o valor do *label* `beta.kubernetes.io/os` no *host* `s2-node-1`?

▼ *Visualizar resposta*

Essa informação é trivialmente obtida via JSONPath. Veja:

```
# kubectl get node s2-node-1 -o jsonpath='{.metadata.labels}' | egrep -o
'"beta.kubernetes.io/os"[^,}]*'
"beta.kubernetes.io/os":"linux"
```

4. Aplique um novo *label* ao *host* `s2-node-1`, com o par chave-valor `beverage=soda`.

▼ *Visualizar resposta*

Para criar novos *labels*, basta usar o comando `kubectl label`, neste caso afetando objetos do tipo `node`:

```
# kubectl label node s2-node-1 beverage=soda
```

```
node/s2-node-1 labeled
```

5. Crie um novo deployment com o nome `fanta` e imagem `nginx:alpine`, com 5 réplicas. Em quais *nodes* seus pods estão executando?

   ▼ *Visualizar resposta*

   Vamos primeiramente criar o deployment:

   ```
   # kubectl create deploy fanta --image=nginx:alpine --replicas=5
   deployment.apps/fanta created
   ```

   Como ambos os *nodes* do *cluster* são agendáveis (lembre-se que removemos o *taint* do *node* `s2-master-1` na atividade anterior), os pods dividem-se entre eles.

   ```
   # kubectl get pod -o custom-columns=NAME:.metadata.name,NODE:.spec.nodeName
   NAME                     NODE
   fanta-74fb7bd575-2gvn7   s2-master-1
   fanta-74fb7bd575-qxtps   s2-master-1
   fanta-74fb7bd575-s6fzd   s2-master-1
   fanta-74fb7bd575-sckdc   s2-node-1
   fanta-74fb7bd575-tgcbz   s2-node-1
   ```

6. Altere a afinidade de nós do deployment `fanta` de modo que que ele execute **apenas** no *host* `s2-node-1`. Verifique o funcionamento de sua configuração.

   ▼ *Visualizar resposta*

   Para tanto, vamos editar o deployment `fanta`:

   ```
   # kubectl edit deploy fanta
   ```

   Na seção `.spec.template.spec`, adicionamos o excerto a seguir. Tenha **muita atenção** com a identação do documento resultante.

   ```
   affinity:
     nodeAffinity:
       requiredDuringSchedulingIgnoredDuringExecution:
         nodeSelectorTerms:
         - matchExpressions:
           - key: beverage
             operator: In
             values:
             - soda
   ```

   Visualizando os *nodes* em que os pods do deployment são agendados, fica claro que a afinidade escolhida os força para o *node* `s2-node-1`:

```
# kubectl get pod -o custom-columns=NAME:.metadata.name,NODE:.spec.nodeName
NAME                     NODE
fanta-7f55c84456-9hpld   s2-node-1
fanta-7f55c84456-g52b8   s2-node-1
fanta-7f55c84456-jvbl4   s2-node-1
fanta-7f55c84456-kbdhr   s2-node-1
fanta-7f55c84456-ldbb5   s2-node-1
```

7. Agora, aplique um novo label ao *host* `s2-master-1`, com o par chave-valor `beverage=beer`.

   Faça com que um novo deployment, `bud`, também com a imagem `nginx:alpine` e 5 réplicas, execute **preferencialmente** (mas não exclusivamente) no *host* `s2-master-1`. Utilize peso `50` para a preferência dos *hosts* que casaram com o *label* objetivado. Verifique o funcionamento de sua configuração.

   ▼ *Visualizar resposta*

   Vamos primeiro aplicar o *label*:

   ```
   # kubectl label node s2-master-1 beverage=beer
   node/s2-master-1 labeled
   ```

   Agora, para a criação do deployment. Usaremos um arquivo YAML para esse fim e, diferentemente do que foi feito no deployment `fanta`, utilizaremos a afinidade `preferredDuringSchedulingIgnoredDuringExecution` — que faz com que os pods sejam preferencialmente agendados no node afim.

   ```yaml
   apiVersion: apps/v1
   kind: Deployment
   metadata:
     labels:
       app: bud
     name: bud
   spec:
     replicas: 5
     selector:
       matchLabels:
         app: bud
     template:
       metadata:
         labels:
           app: bud
       spec:
         affinity:
           nodeAffinity:
             preferredDuringSchedulingIgnoredDuringExecution:
             - weight: 50
               preference:
   ```

```
            matchExpressions:
            - key: beverage
              operator: In
              values:
              - beer
      containers:
      - image: nginx:alpine
        name: nginx
```

A seguir, criamos o deployment via arquivo.

```
# kubectl create -f bud.yaml
deployment.apps/bud created
```

Note que os pods foram todos agendados em `s2-master-1`, no exemplo abaixo. Isso se deve ao fato de que existem apenas dois *nodes* no *cluster*, e também porque o deployment `fanta` já ocupa boa parte dos recursos do *node* `s2-node-1`. Em um *cluster* mais realista, contudo, esse provavelmente não seria o caso.

```
# kubectl get pod -l app=bud -o custom-
columns=NAME:.metadata.name,NODE:.spec.nodeName
NAME                   NODE
bud-576f8958c9-bhxpj   s2-master-1
bud-576f8958c9-fbgk6   s2-master-1
bud-576f8958c9-ghfps   s2-master-1
bud-576f8958c9-ptz4b   s2-master-1
bud-576f8958c9-rp7lw   s2-master-1
```

# 5) Afinidade em relação a pods

1. Antes de iniciar, execute o comando abaixo:

   ```
   # lab-3.5.1
   ```

2. Em uma aplicação web típica, pode ser interessante fazer com que réplicas de atendimento ao usuário (*front-end*) não operem no mesmo *node*, para fins de redundância. De igual forma, é crítico que pods auxiliares a esses serviços (p.ex. um serviço de cache em memória como o Redis) estejam no mesmo *node* que a réplica de *front-end*. Para esse fim, podemos utilizar o recurso de afinidade e anti-afinidade entre pods.

   Crie um deployment de nome `cache` usando a imagem `redis:alpine`, com 2 réplicas. Usando anti-afinidade, garanta que esses pods não executem no mesmo *host*.

   ▼ *Visualizar resposta*

   Vamos criar o deployment via arquivo YAML, como se segue. Preste especial atenção na seção

`.spec.affinity.podAntiAffinity`:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: cache
spec:
  selector:
    matchLabels:
      app: store
  replicas: 2
  template:
    metadata:
      labels:
        app: store
    spec:
      affinity:
        podAntiAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
          - labelSelector:
              matchExpressions:
              - key: app
                operator: In
                values:
                - store
            topologyKey: "kubernetes.io/hostname"
      containers:
      - name: redis
        image: redis:alpine
```

Cria-se o deployment via arquivo:

```
# kubectl create -f cache.yaml
deployment.apps/cache created
```

E constata-se que, de fato, os pods foram agendados em *nodes* diferentes, como objetivado.

```
# kubectl get pod -l app=store -o custom-
columns=NAME:.metadata.name,NODE:.spec.nodeName
NAME                     NODE
cache-654dd5d647-m95vh   s2-master-1
cache-654dd5d647-x799g   s2-node-1
```

3. Continuando a atividade anterior, crie agora o deployment de uma aplicação web fictícia com o nome `webapp`, usando a imagem `nginx:alpine`, com 2 réplicas. Garanta que os pods de cada réplica não executem no mesmo *host*, usando anti-afinidade, e garanta **também** que cada um desses pods executem conjuntamente com um dos pods do deployment `cache`.

▼ *Visualizar resposta*

Assim como no caso anterior, usaremos um arquivo YAML para criar o deployment. Atente-se para as seções `.spec.affinity.podAntiAffinity` e `.spec.affinity.podAffinity`:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: webapp
spec:
  selector:
    matchLabels:
      app: web-store
  replicas: 2
  template:
    metadata:
      labels:
        app: web-store
    spec:
      affinity:
        podAntiAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
          - labelSelector:
              matchExpressions:
              - key: app
                operator: In
                values:
                - web-store
            topologyKey: "kubernetes.io/hostname"
        podAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
          - labelSelector:
              matchExpressions:
              - key: app
                operator: In
                values:
                - store
            topologyKey: "kubernetes.io/hostname"
      containers:
      - name: webapp
        image: nginx:alpine
```

A seguir, criamos o deployment e observamos o estado dos pods. Novamente, eles foram posicionados em *nodes* distintos.

```
# kubectl create -f webapp.yaml
deployment.apps/webapp created
```

```
# kubectl get pod -l app=web-store -o custom-
columns=NAME:.metadata.name,NODE:.spec.nodeName
NAME                    NODE
webapp-66b49bc49-sg6zv   s2-master-1
webapp-66b49bc49-zckv6   s2-node-1
```

4. Escale o número de réplicas do deployment `webapp` para 3. O que ocorre? Porquê?

▼ *Visualizar resposta*

Vamos escalar o deployment com `kubectl scale deployment`:

```
# kubectl scale deploy webapp --replicas=3
deployment.apps/webapp scaled
```

Observe que o novo pod fica em estado `Pending`... porquê?

```
# kubectl get pod -l app=web-store -o custom-
columns=NAME:.metadata.name,NODE:.spec.nodeName,STATUS:.status.phase
NAME                    NODE          STATUS
webapp-66b49bc49-sg6zv   s2-master-1   Running
webapp-66b49bc49-xztzq   <none>        Pending
webapp-66b49bc49-zckv6   s2-node-1     Running
```

A última linha de eventos do pod conta a história: não é possível atender os requisitos de afinidade e anti-afinidade do pod, já que não há como posicioná-lo com um pod de cache em memória e simultaneamente sem compartilhar o *node* com outro pod do tipo `webapp`.

```
# kubectl describe pod webapp-66b49bc49-xztzq | tail -n1
  Warning  FailedScheduling  102s  default-scheduler  0/2 nodes are available: 2
node(s) didn't match pod affinity/anti-affinity, 2 node(s) didn't match pod anti-
affinity rules.
```

# 6) Requisitos e limitações de recursos

1. Determine a capacidade disponível de CPU e memória em cada um dos *nodes* do *cluster*.

▼ *Visualizar resposta*

Os requisitos disponíveis em um *node* podem ser vistos com `kubectl describe nodes`; abaixo, usamos o `grep` para mostrar a seção relevante da saída.

```
# kubectl describe nodes s2-master-1 | grep '^Capacity:' -A5
Capacity:
  cpu:                1
  ephemeral-storage:  64255620Ki
```

```
  hugepages-2Mi:      0
  memory:             4025916Ki
  pods:               110
```

Fazemos o mesmo para o outro *node* do *cluster*:

```
# kubectl describe nodes s2-node-1 | grep '^Capacity:' -A5
Capacity:
  cpu:                1
  ephemeral-storage:  64255620Ki
  hugepages-2Mi:      0
  memory:             2030544Ki
  pods:               110
```

2. Faça o deployment do pod `hippo`, usando a imagem `httpd:alpine`, requerendo um mínimo de 2.5 unidades de CPU e um limite de 5. O que acontece?

▼ *Visualizar resposta*

Utilize o arquivo YAML abaixo para criar o pod:

```
 1 apiVersion: v1
 2 kind: Pod
 3 metadata:
 4   name: hippo
 5 spec:
 6   containers:
 7   - image: httpd:alpine
 8     name: hippo
 9     resources:
10       limits:
11         cpu: "5"
12       requests:
13         cpu: 2500m
```

A seguir, crie-o:

```
# kubectl create -f hippo.yaml
pod/hippo created
```

Observando o estado do pod, constatamos que ele se encontra como `Pending`.

```
# kubectl get pod hippo
NAME    READY   STATUS    RESTARTS   AGE
hippo   0/1     Pending   0          67s
```

Visualizando os eventos do pod, fica claro que não existem membros no *cluster* capazes de atender seus requisitos de CPU:

```
# kubectl describe pod hippo | tail -n1
  Warning  FailedScheduling  99s   default-scheduler  0/2 nodes are available: 2
Insufficient cpu.
```

3. Tente editar a configuração do pod, reduzindo o requerimento de recursos. É possível?

▼ *Visualizar resposta*

Vamos tentar:

```
# kubectl edit pod hippo
```

Após editar os requerimentos e salvar o arquivo, o seguinte erro é retornado — portanto, seria necessário deletar e re-criar o pod do zero, já com a configuração corrigida.

```
* spec: Forbidden: pod updates may not change fields other than
`spec.containers[*].image`, `spec.initContainers[*].image`,
`spec.activeDeadlineSeconds` or `spec.tolerations` (only additions to existing
tolerations)
```

4. Antes de prosseguir, execute o comando abaixo:

```
# lab-3.6.1
```

5. Um pod com o nome `rhino` foi criado. Qual o seu estado?

▼ *Visualizar resposta*

Vejamos:

```
# kubectl get pod rhino
NAME    READY   STATUS      RESTARTS   AGE
rhino   0/1     OOMKilled   0          5s
```

6. O que significa esse estado? Investigue as configurações do pod e determine os requerimentos e limites de recursos solicitados, bem como o comando invocado.

▼ *Visualizar resposta*

O estado `OOMKilled` indica que o pod foi encerrado por estar `OOM` (*Out Of Memory*, ou com falta de memória). Vamos ver seus requerimentos mínimos:

```
# kubectl describe pod rhino | grep 'Requests:' -A1
    Requests:
```

```
      memory:     50Mi
```

E limites:

```
# kubectl describe pod rhino | grep 'Limits:' -A1
    Limits:
      memory:  100Mi
```

Investigando o comando passado ao pod, parece que ele requer 250 MB de memória RAM — isso pode ser verificado consultando a página de manual do comando `stress`, disponível em https://linux.die.net/man/1/stress .

```
# kubectl describe pod rhino | grep 'Args:' -A7 | paste -s | sed -r
's/[[:space:]]+/ /g'
 Args: stress --vm 1 --vm-bytes 250M --vm-hang 1
```

Evidentemente, os requerimentos de 50/100 MB de memória não são suficientes para atender essa demanda.

7. Aumente o limite de recursos do pod e execute-o novamente. Houve sucesso?

▼ *Visualizar resposta*

Vamos obter a configuração do pod exportando-a para formato YAML, e redirecionando para um arquivo:

```
# kubectl get pod rhino -o yaml > rhino.yaml
```

A seguir, edite o arquivo (diretamente ou via `sed`), aumentando o limite de uso de memória do pod. Vamos deixar uma pequena margem de folga — abaixo, escolhemos um limite de 300 MB:

```
# sed -i 's/100Mi/300Mi/' rhino.yaml
```

Agora, deletamos o pod e o re-criamos via o arquivo editado acima.

```
<strong># kubectl delete pod rhino && kubectl create -f rhino.yaml</strong>
pod "rhino" deleted
pod/rhino created
```

Feito isso, verificamos o estado do pod: agora, perfeitamente funcional.

```
# kubectl get pod rhino
NAME    READY   STATUS    RESTARTS   AGE
```

```
rhino   1/1     Running   0          20s
```

8. Finalmente, delete o pod `rhino`.

```
# kubectl delete pod rhino
pod "rhino" deleted
```

# 7) DaemonSets

1. Quantos DaemonSets existem no *cluster*, em todos os namespaces?

   ▼ *Visualizar resposta*

   Para descobrir os DaemonSets existentes no ambiente, basta utilizar o comando `kubectl get daemonsets` ou `ds`, em forma curta.

   ```
   # kubectl get ds --all-namespaces --no-headers | wc -l
   2
   ```

2. Em qual namespace esses DaemonSets estão inseridos?

   ▼ *Visualizar resposta*

   Podemos verificar essa informação de diversas formas — abaixo, usaremos a *flag* `-o custom-columns` para imprimir apenas os campos relevantes.

   ```
   # kubectl get ds --all-namespaces -o custom-
   columns=NAME:.metadata.name,NAMESPACE:.metadata.namespace
   NAME         NAMESPACE
   kube-proxy   kube-system
   weave-net    kube-system
   ```

3. Em quais *nodes* estão presentes os pods do DaemonSet `weave-net`?

   ▼ *Visualizar resposta*

   Antes de descobrir essa informação, é interessante descobrir quais *labels* estão aplicados ao DaemonSet e a seus pods, permitindo o uso de seletores futuramente.

   ```
   # kubectl -n kube-system describe ds weave-net | grep '^[[:space:]]*Labels:'
   Labels:         name=weave-net
     Labels:           name=weave-net
   ```

   De posse dessa informação, iremos agora descobrir os *nodes* executando esses pods:

   ```
   # kubectl -n kube-system get pod -l name=weave-net -o custom-
   columns=NAME:.metadata.name,NODE:.spec.nodeName
   ```

```
NAME              NODE
weave-net-ptk5f   s2-node-1
weave-net-vzsr6   s2-master-1
```

4. Qual é a imagem utilizada pelos pods do DaemonSet `kube-proxy`?

   ▼ *Visualizar resposta*

   Basta utilizar o comando `kubectl describe daemonset`, como seria de imaginar:

   ```
   # kubectl -n kube-system describe ds kube-proxy | grep 'Image:'
       Image:      k8s.gcr.io/kube-proxy:v1.23.5
   ```

5. A função de DaemonSets é que todos (ou parte) dos *nodes* de um *cluster* executem uma cópia de um pod. Tipicamente, esse recurso é utilizado para tarefas como *daemons* de armazenamento do *cluster*, coletores de logs ou agentes de monitoramento.

   Crie um DaemonSet que execute o coletor de logs FluentD. Utilize o nome `fluentd-elasticsearch` e imagem `quay.io/fluentd_elasticsearch/fluentd:v3.0.4`, alocando-o no namespace `kube-system`. Garanta que os pods executem em **todos** os *nodes* do *cluster*. Verifique sua configuração.

   ▼ *Visualizar resposta*

   Vamos criar o DaemonSet via arquivo YAML. Copie o conteúdo abaixo para o arquivo novo `fluentd.yaml` — atente-se para a tolerância aplicada aos pods do DaemonSet, permitindo que sejam agendados no *node* `s2-master-1`:

   ```
    1 apiVersion: apps/v1
    2 kind: DaemonSet
    3 metadata:
    4   name: fluentd-elasticsearch
    5   namespace: kube-system
    6 spec:
    7   selector:
    8     matchLabels:
    9       name: fluentd-elasticsearch
   10   template:
   11     metadata:
   12       labels:
   13         name: fluentd-elasticsearch
   14     spec:
   15       tolerations:
   16       - key: node-role.kubernetes.io/master
   17         effect: NoSchedule
   18       containers:
   19       - name: fluentd-elasticsearch
   20         image: quay.io/fluentd_elasticsearch/fluentd:v3.0.4
   ```

   A seguir, criamos o DaemonSet.

```
# kubectl create -f fluentd.yaml
daemonset.apps/fluentd-elasticsearch created
```

E, finalmente, verificamos que seus pods foram agendados em cada um dos *nodes* do *cluster*, como objetivado.

```
# kubectl -n kube-system get pod -l name=fluentd-elasticsearch -o custom-
columns=NAME:.metadata.name,NODE:.spec.nodeName
NAME                          NODE
fluentd-elasticsearch-hrtpm   s2-node-1
fluentd-elasticsearch-tfj5l   s2-master-1
```

# 8) Pods estáticos

1. Quantos pods estáticos existem no *cluster*, em todos os namespaces?

   ▼ *Visualizar resposta*

   Pods estáticos possuem como sufixo o nome do *node* em que estão executando. Isso fica bem claro ao visualizarmos o padrão de nomes dos pods existentes no *cluster* neste momento:

   ```
   # kubectl get pod --all-namespaces --no-headers -o custom-
   columns=NAME:.metadata.name
   coredns-f9fd979d6-2hpfw
   coredns-f9fd979d6-d7j8s
   etcd-s2-master-1
   fluentd-elasticsearch-gc4zk
   fluentd-elasticsearch-nmgvb
   kube-apiserver-s2-master-1
   kube-controller-manager-s2-master-1
   kube-proxy-7pq9k
   kube-proxy-trq9x
   kube-scheduler-s2-master-1
   weave-net-ptk5f
   weave-net-vzsr6
   ```

   Vamos filtrar apenas os pods que possuem esses sufixos, e contar as ocorrências, assim respondendo a pergunta postulada no enunciado.

   ```
   # kubectl get pod --all-namespaces --no-headers -o custom-
   columns=NAME:.metadata.name | grep 's2-master-1$\|s2-node-1$' | wc -l
   4
   ```

2. Dos pods publicados no namespace `kube-system`, quais deles são pods estáticos? E os demais, são publicados como quais tipos de objetos?

▼ *Visualizar resposta*

Já vimos como contar os pods estáticos'; para visualizar seus nomes, basta remover o comando `wc` do final do comando executado na atividade anterior:

```
# kubectl -n kube-system get pod --no-headers -o custom-
columns=NAME:.metadata.name | grep 's2-master-1$\|s2-node-1$'
etcd-s2-master-1
kube-apiserver-s2-master-1
kube-controller-manager-s2-master-1
kube-scheduler-s2-master-1
```

Observe que todos os pods estáticos estão publicados no *node* `s2-master-1`.

Para descobrir os pods que **não** são estáticos, basta realizar uma seleção complementar usando `grep -v`.

```
# kubectl -n kube-system get pod --no-headers -o custom-
columns=NAME:.metadata.name | grep -v 's2-master-1$\|s2-node-1$'
coredns-f9fd979d6-2hpfw
coredns-f9fd979d6-d7j8s
fluentd-elasticsearch-gc4zk
fluentd-elasticsearch-nmgvb
kube-proxy-7pq9k
kube-proxy-trq9x
weave-net-ptk5f
weave-net-vzsr6
```

Alguns desses pods são parte de DaemonSets, como vimos no tópico anterior:

```
# kubectl -n kube-system get ds --no-headers -o custom-
columns=NAME:.metadata.name
fluentd-elasticsearch
kube-proxy
weave-net
```

O `coredns`, por outro lado, consiste em um deployment que será analisado mais a fundo em sessões futuras deste curso.

```
# kubectl -n kube-system get deploy --no-headers -o custom-
columns=NAME:.metadata.name
coredns
```

3. Qual é o diretório que contém os arquivos YAML desses pods estáticos? Quais pods estão ali definidos?

▼ *Visualizar resposta*

Esse configuração é realizada no `kubelet` — especificamente em seu arquivo de configuração, cujo caminho é indicado na linha de invocação do processo. Vamos visualizar essa linha com o comando `pgrep`:

```
# pgrep -a kubelet | egrep -o '\-\-config=[^ ]*'
--config=/var/lib/kubelet/config.yaml
```

A diretiva que define o diretório que contém os arquivos YAML de pods estáticos é a `staticPodPath`:

```
# grep '^staticPodPath:' /var/lib/kubelet/config.yaml
staticPodPath: /etc/kubernetes/manifests
```

Listando esse diretório é possível constatar que quatro pods estáticos estão definidos, todos eles relacionados ao *control plane* do Kubernetes.

```
# ls -1 /etc/kubernetes/manifests
etcd.yaml
kube-apiserver.yaml
kube-controller-manager.yaml
kube-scheduler.yaml
```

4. Crie um pod estático com o nome `static-nginx` que utiliza a imagem `nginx:alpine`.

▼ *Visualizar resposta*

Para criar um pod estático basta copiar o arquivo YAML para o diretório obtido no passo anterior, como vimos. Melhor ainda, podemos redirecionar esse arquivo YAML diretamente a partir da saída do comando `kubectl run`, desta forma:

```
# kubectl run static-nginx --image=nginx:alpine --dry-run=client -o=yaml >
/etc/kubernetes/manifests/static-nginx.yaml
```

Imediatamente, o pod estático é criado pelo `kubelet`:

```
# kubectl get pod
NAME                       READY   STATUS    RESTARTS   AGE
static-nginx-s2-master-1   1/1     Running   0          13s
```

5. Altere o pod criado no passo anterior para utilizar a imagem `nginx:1.19.2-alpine`.

▼ *Visualizar resposta*

Para editar um pod estático basta alterar o arquivo YAML que o define — as alterações são aplicadas automaticamente (e periodicamente) pelo `kubelet`. Vamos alterar a imagem via `sed`, como se segue.

```
# sed -i 's/\(image: nginx:\)alpine/\11.19.2-alpine/'
/etc/kubernetes/manifests/static-nginx.yaml
```

Usando `kubectl describe pod`, constatamos que a alteração foi realizada com sucesso.

```
# kubectl describe pod static-nginx-s2-master-1 | grep 'Image:'
    Image:          nginx:1.19.2-alpine
```

6. Antes de prosseguir, execute o comando abaixo:

```
# lab-3.8.1
```

7. Um novo pod estático foi criado. Descubra qual é esse pod e remova-o.

▼ *Visualizar resposta*

Vejamos qual a situação dos pods no cluster no momento:

```
# kubectl get pod
NAME                        READY   STATUS    RESTARTS   AGE
static-nginx-s2-master-1    1/1     Running   0          13m
turtlepower-s2-node-1       1/1     Running   0          2m16s
```

Temos um novo pod: `turtlepower-s2-node-1`. Pelo sufixo, é fácil intuir que ele se encontra no *node* `s2-node-1`.

Vamos logar nesse *node* via SSH, usando o Vagrant — execute o comando abaixo em sua máquina física, na pasta `contorq-files\s2`:

```
C:\contorq-files\s2> vagrant ssh s2-node-1
Linux s2-node-1 4.19.0-10-amd64 #1 SMP Debian 4.19.132-1 (2020-07-24) x86_64

(...)

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
vagrant@s2-node-1:~$
```

A seguir, vire o superusuário `root` com o comando `sudo -i`.

```
vagrant@s2-node-1:~$ sudo -i
root@s2-node-1:~#
```

Primeiro, vamos ver onde está o arquivo de configuração do `kubelet`.

```
# pgrep -a kubelet | egrep -o '\-\-config=[^ ]*'
--config=/var/lib/kubelet/config.yaml
```

E qual seria a configuração da diretiva `staticPodPath`?

```
# grep '^staticPodPath:' /var/lib/kubelet/config.yaml
staticPodPath: /etc/cowabunga
```

Perfeito! Vamos ver o conteúdo desse diretório:

```
# ls /etc/cowabunga/
turtlepower.yaml
```

Basta remover esse arquivo...

```
# rm -f /etc/cowabunga/turtlepower.yaml
```

E, de volta ao *node* `s2-master-1`, visualizar o estado dos pods em execução no *cluster*. Como esperado, o pod estático foi deletado após a remoção do arquivo YAML:

```
# kubectl get pod turtlepower-s2-node-1
Error from server (NotFound): pods "turtlepower-s2-node-1" not found
```

**ENTREGA DA TAREFA**

Para que seja considerada entregue você deve anexar a esta atividade no AVA uma imagem (nos formatos .png ou .jpg) do terminal mostrando a execução dos dois pods do *DaemonSet* `fluentd-elasticsearch`, cada um em um *node* do *cluster*.

Utilize como referência a saída de comando mostrada na atividade 3.7 (e) deste roteiro.