Índice

Sessão 5: Gestão do ciclo de vida de aplicações no Kubernetes

1) Configuração de aplicações

1. Antes de iniciar, execute o comando abaixo:

```
# lab-5.1.1
```

2. Temos um pod executando no sistema. Determine seu nome, bem como o comando utilizado em sua invocação.

▼ *Visualizar resposta*

Bem, descobrir o nome do pod é bastante fácil:

```
# kubectl get pod
NAME      READY   STATUS    RESTARTS   AGE
sleeper   1/1     Running   0          2m36s
```

Já seu comando de invocação pode ser visto através de `kubectl get pod -o yaml` ou `kubectl describe`; abaixo, iremos usar a segunda opção, buscando pela seção `Args`:

```
# kubectl describe pod sleeper | grep 'Args:' -A2
    Args:
      sleep
      3600
```

O comando executado é, portanto, `sleep 3600`.

3. Crie um pod com o nome `sleeper-2`, usando a mesma imagem do pod analisado no passo anterior. Adicionalmente, execute o comando `sleep 7200` em sua invocação.

   Dentro do novo pod criado, utilize o comando `ps` para verificar o funcionamento de sua configuração.

   ▼ *Visualizar resposta*

   Vamos verificar qual a imagem usada pelo pod `sleeper`:

   ```
   # kubectl describe pod sleeper | grep 'Image:'
       Image:          alpine
   ```

   Certo. Então, basta executar um pod usando as configurações especificadas, como se segue.

   ```
   # kubectl run sleeper-2 --image=alpine -- sleep 7200
   pod/sleeper-2 created
   ```

   Vamos ver se o comando está de fato em execução com o comando `kubectl exec`:

   ```
   # kubectl exec -it sleeper-2 -- ps auxmw
   PID   USER     TIME  COMMAND
       1 root      0:00 sleep 7200
       6 root      0:00 ps auxmw
   ```

4. Agora, crie o container `sleeper-3`, desta vez utilizando a imagem `fbscarel/myapp-color`. Invoque o pod usando o comando `sleep 10800`.

   Utilize o comando `ps` para verificar o funcionamento de sua configuração.

   ▼ *Visualizar resposta*

   Diferentemente dos pods anteriores, que usavam a imagem `alpine` (cujo `ENTRYPOINT` é o *shell* `/bin/sh`), a imagem `fbscarel/myapp-color` possui o comando `python app.py` como `ENTRYPOINT`. Vimos isso na sessão 1, lembra-se?

   Assim, teremos que sobrescrever o `ENTRYPOINT` dessa imagem; para tanto, basta especificar a *flag* `--command` ao comando `kubectl run`, como se segue:

   ```
   # kubectl run sleeper-3 --image=fbscarel/myapp-color --command -- sleep 10800
   pod/sleeper-3 created
   ```

   Vejamos o comando em operação:

   ```
   # kubectl exec -it sleeper-3 -- ps auxmw
   PID   USER     TIME  COMMAND
       1 root      0:00 sleep 10800
   ```

```
    6 root      0:00 ps auxmw
```

5. Analise a definição YAML dos pods criados nos passos (c) e (d) e responda: qual a diferença entre eles? Que relação essas configurações possuem com as instruções `ENTRYPOINT` e `CMD` em arquivos `Dockerfile`?

   ▼ *Visualizar resposta*

   Veja que no caso de pod `sleeper-2` a seção que contém o comando `sleep` é a `Args` — isso se deve ao fato de que estamos apenas fornecendo argumentos adicionais ao `ENTRYPOINT` da imagem em uso, `alpine`. Isso equivale, para todos os efeitos, à instrução `CMD` em um arquivo `Dockerfile`.

   ```
   # kubectl describe pod sleeper-2 | grep 'sleep$' -A1 -B1
       Args:
         sleep
         7200
   ```

   Já no caso do pod `sleeper-3`, esse `ENTRYPOINT` teve que ser sobrescrito. Note, abaixo, que a seção que contém o comando `sleep` é a `Command`:

   ```
   # kubectl describe pod sleeper-3 | grep 'sleep$' -A1 -B1
       Command:
         sleep
         10800
   ```

6. Por padrão, a imagem `fbscarel/myapp-color` mostra um fundo azul. Crie um novo pod, de nome `myapp-green`, que mostra um fundo verde em seu lugar. Utilize o parâmetro de linha de comando `--color` para realizar essa configuração.

   Posteriormente, exponha o pod e visite a aplicação usando um navegador web em sua máquina física.

   ▼ *Visualizar resposta*

   Neste caso, não precisamos alterar o `ENTRYPOINT` do container, mas apenas fornecer argumentos ao comando padrão. Faremos isso através da entrada padrão com os caracters `--`:

   ```
   # kubectl run myapp-green --image=fbscarel/myapp-color -l app=myapp-green --
   --color=green
   pod/myapp-green created
   ```

   Para visualizar a aplicação usando um navegador web, iremos criar um serviço do tipo `NodePort`:

   ```
   # kubectl create svc nodeport myapp-green --tcp=80 --node-port=30080
   ```

```
service/myapp-green created
```

E, finalmente, iremos visualizar a aplicação.

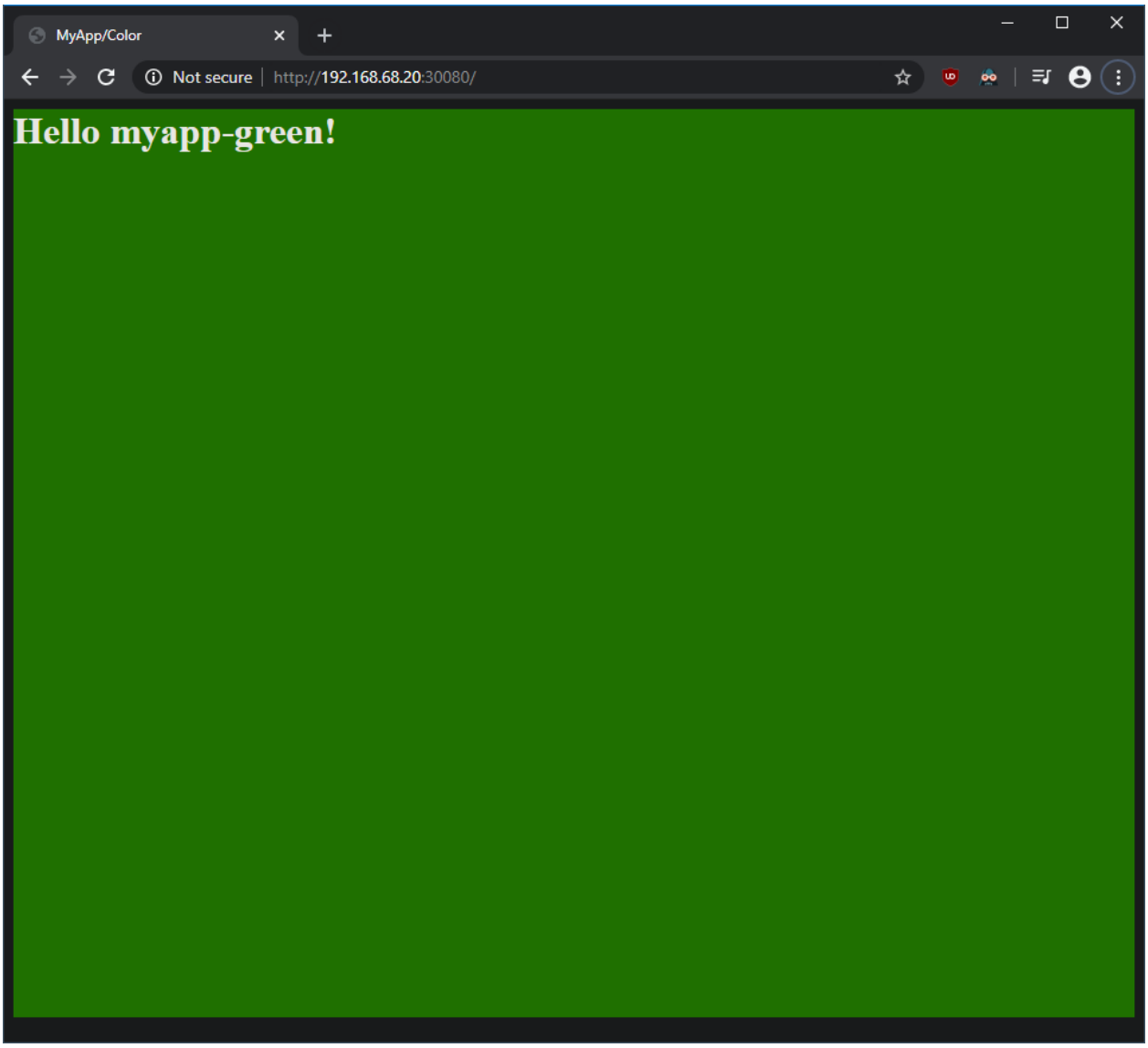

Figura 1. Aplicação publicada com a cor verde

2) Realizando updates e rollbacks

1. Remova os pods e serviços criados na atividade anterior com os comandos que se seguem.

```
# kubectl delete pod --all
```

```
# kubectl delete svc -l provider!=kubernetes
```

2. Crie um deployment com o nome `myapp`, usando a imagem `fbscarel/myapp-color:blue`, com 4 réplicas. Exponha a porta 80 dessas réplicas dentro do contexto do *cluster*.

▼ *Visualizar resposta*

Bem, com o conhecimento que adquirimos até aqui é bastante fácil atingir esses objetivos, vamos lá. Primeiro, cria-se o deployment:

```
# kubectl create deploy myapp --image=fbscarel/myapp-color:blue --replicas=4
--port=80
deployment.apps/color created
```

A seguir, o serviço. Note que como ele deve ser exposto apenas no contexto do *cluster*, o tipo `ClusterIP` é selecionado.

```
# kubectl create svc clusterip myapp --tcp=80
service/color created
```

Vamos verificar se o serviço foi publicado como esperado:

```
# kubectl get svc myapp --no-headers
color    ClusterIP    10.105.177.171    <none>    80/TCP    50s
```

3. Crie um pod com o nome `curl`, usando a imagem `curlimages/curl`. Faça com que o pod se mantenha em estado `Running` — para tanto, substitua o ponto de entrada do container com um *loop* do comando `sleep 60`.

   Em seguida, utilizando os comandos `curl` ou `wget` e partindo do pod recém-criado, acesse a URL `/color` do deployment realizado no passo anterior para verificar sua acessibilidade.

   ▼ *Visualizar resposta*

   Criar o pod com a imagem e nome solicitados é fácil; por outro lado, como mantê-lo rodando? Vamos sobrescrever o `ENTRYPOINT` do container com um *script shell* simples, que execute um *loop* do comando `sleep 60` continuamente:

   ```
   # kubectl run curl --image=curlimages/curl --command -- /bin/sh -c 'while true;
   do sleep 60; done'
   pod/curl created
   ```

   Agora que o pod está executando, podemos invocar comandos dentro dele. Iremos utilizar o `curl` para acessar a URL `/color` da aplicação publicada no passo anterior, constatando que sua cor, no momento, é azul.

   ```
   # kubectl exec -it curl -- curl http://myapp/color
   Hostname: myapp-68b775f85-wdvll ; Color: blue
   ```

4. Vamos monitorar o estado do deployment. Em um terminal dedicado, execute o comando `curl` ou `wget` invocado no passo anterior em um *loop* infinito. Faça com que o tempo máximo de espera do comando (*timeout*) seja de 3 segundos.

Sugestão: utilize um construto na forma `while true; do CURL-COMMAND; echo; sleep 1; done`.

▼ *Visualizar resposta*

Para atingir esse objetivo basta editar o comando anterior, colocando-o dentro de um *loop* com a *flag* `-m3` para ajustar o *timeout* do `curl`. Veja:

```
# kubectl exec -it curl -- /bin/sh -c 'while true; do curl -m3
http://myapp/color; echo; sleep 1; done'
Hostname: myapp-68b775f85-wdvll ; Color: blue
Hostname: myapp-68b775f85-kmnll ; Color: blue
Hostname: myapp-68b775f85-n6qwx ; Color: blue

(...)
```

5. Determine a estratégia de atualização utilizada no deployment `myapp`. Qual é o efeito dessa configuração, e qual o número máximo de pods inativos durante uma atualização?

▼ *Visualizar resposta*

O tipo de estratégia pode ser visualizada com o `kubectl describe deploy`, como visto abaixo.

```
# kubectl describe deploy myapp | grep StrategyType
StrategyType:           RollingUpdate
```

O tipo `RollingUpdate` possui configurações adicionais, visualizáveis no campo `RollingUpdateStrategy` desse mesmo comando.

```
# kubectl describe deploy myapp | grep RollingUpdateStrategy
RollingUpdateStrategy:  25% max unavailable, 25% max surge
```

Como visto acima, a estratégia está configurada para permitir no máximo 25% de pods indisponíveis. Como temos 4 pods no deployment, é fácil concluir que teremos no máximo 1 pod inativo durante uma atualização.

6. Atualize a imagem do deployment para `fbscarel/myapp-color:red`. O que ocorre? Verifique o estado dos pods e o monitoramento criado no passo (c) para responder essa pergunta.

▼ *Visualizar resposta*

Vamos atualizar a imagem usando o comando `kubectl edit`.

```
# kubectl edit deploy myapp
deployment.apps/myapp edited
```

Monitorando o estado do deployment, note que os pods são terminados à medida em que novos são criados.

```
# kubectl get pod -l app=myapp
NAME                     READY   STATUS              RESTARTS   AGE
myapp-68b775f85-5g55l    1/1     Running             0          2m1s
myapp-68b775f85-pfs2k    1/1     Running             0          2m1s
myapp-68b775f85-qltcx    1/1     Terminating         0          2m1s
myapp-68b775f85-zfw4g    1/1     Running             0          2m1s
myapp-6955c4f744-5scz7   0/1     ContainerCreating   0          4s
myapp-6955c4f744-8pqx9   0/1     ContainerCreating   0          4s
```

Como configurado, não há uma indisponibilidade total da aplicação em nenhum momento — o comando `curl` consegue atingir um pod do deployment mesmo durante a atualização.

```
Hostname: myapp-68b775f85-pfs2k ; Color: blue
Hostname: myapp-68b775f85-pfs2k ; Color: blue
Hostname: myapp-68b775f85-zfw4g ; Color: blue
Hostname: myapp-6955c4f744-5scz7 ; Color: red
Hostname: myapp-6955c4f744-5scz7 ; Color: red
Hostname: myapp-68b775f85-pfs2k ; Color: blue
Hostname: myapp-6955c4f744-8pqx9 ; Color: red
Hostname: myapp-6955c4f744-8pqx9 ; Color: red
Hostname: myapp-6955c4f744-s6g66 ; Color: red
```

7. Altere a estratégia de atualização para `Recreate`, e também altere a imagem do deployment para `fbscarel/myapp:green`. Qual o efeito dessa configuração na disponibilidade da aplicação?

   ▼ *Visualizar resposta*

   Vamos editar o deployment novamente. Altere a imagem, e a seguir...

   ```
   # kubectl edit deploy myapp
   deployment.apps/myapp edited
   ```

   Na seção `.spec.strategy`, altere o tipo de estratégia:

   ```
   strategy:
     type: Recreate
   ```

   E, ainda na seção `.spec.strategy`, remova o bloco `rollingUpdate`, que referencia especificamente o tipo de estratégia `RollingUpdate`.

   ```
   rollingUpdate:
     maxSurge: 25%
     maxUnavailable: 25%
   ```

Note que a atualização do deployment começa imediatamente. Desta vez, os pods são todos encerrados:

```
# kubectl get pod -l app=myapp
NAME                     READY   STATUS        RESTARTS   AGE
myapp-6955c4f744-5scz7   1/1     Terminating   0          2m23s
myapp-6955c4f744-8pqx9   1/1     Terminating   0          2m23s
myapp-6955c4f744-qvgxc   1/1     Terminating   0          2m17s
myapp-6955c4f744-s6g66   1/1     Terminating   0          2m16s
```

E, após algum tempo, são recriados conjuntamente.

```
# kubectl get pod -l app=myapp
NAME                     READY   STATUS    RESTARTS   AGE
myapp-665fd6ddbd-55fxs   1/1     Running   0          19s
myapp-665fd6ddbd-5ktqr   1/1     Running   0          19s
myapp-665fd6ddbd-6gp8s   1/1     Running   0          19s
myapp-665fd6ddbd-vt9vh   1/1     Running   0          19s
```

Monitorando o comando `curl`, perceba que a aplicação torna-se indisponível por algum tempo — enquanto os pods são recriados.

```
Hostname: myapp-6955c4f744-5scz7 ; Color: red
Hostname: myapp-6955c4f744-8pqx9 ; Color: red
Hostname: myapp-6955c4f744-qvgxc ; Color: red
Hostname: myapp-6955c4f744-5scz7 ; Color: red
curl: (28) Connection timed out after 3000 milliseconds
curl: (28) Connection timed out after 3001 milliseconds
(...)
curl: (28) Connection timed out after 3000 milliseconds
curl: (28) Connection timed out after 3001 milliseconds
Hostname: myapp-665fd6ddbd-55fxs ; Color: green
Hostname: myapp-665fd6ddbd-55fxs ; Color: green
Hostname: myapp-665fd6ddbd-6gp8s ; Color: green
Hostname: myapp-665fd6ddbd-vt9vh ; Color: green
```

3) Variáveis de ambiente e ConfigMaps

1. Remova os pods e serviços criados na atividade anterior com os comandos que se seguem.

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

```
# kubectl delete svc -l provider!=kubernetes
```

2. Vamos configurar a aplicação `fbscarel/myapp-color` usando variáveis de ambiente. Lance um pod com o nome `bobross` usando essa imagem, e ajustando a variável de ambiente `COLOR` com o valor `purple`.

   A seguir, exponha o pod na porta 31080 do *node* e utilize os comandos `curl` ou `wget` para consultar o caminho `/color` e verificar a cor da aplicação.

   ▼ *Visualizar resposta*

   Podemos usar a *flag* `--env` para configurar variáveis de ambiente para o pod, como solicitado:

   ```
   # kubectl run bobross --image=fbscarel/myapp-color -l app=bobross
   --env="COLOR=purple"
   pod/bobross created
   ```

   A seguir, exporemos o pod com um serviço do tipo `NodePort`, como solicitado.

   ```
   # kubectl create svc nodeport bobross --tcp=80 --node-port=31080
   service/bobross created
   ```

   E, finalmente, iremos consultar a cor atual da aplicação via `curl`:

   ```
   # curl http://localhost:31080/color
   Hostname: bobross ; Color: purple
   ```

3. Vamos alterar a cor da aplicação para amarelo. Primeiro, tente editar o pod; é possível?

   ▼ *Visualizar resposta*

   Vamos ver:

   ```
   # kubectl edit pod bobross
   ```

   Ao alterar a variável de ambiente, encontramos um erro. Não é possível editá-lo *on-the-fly*, portanto; teremos que recriar o pod.

   ```
   # pods "bobross" was not valid:
   # * 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. Alternativamente, exporte o pod em formato YAML, faça as alterações necessárias e recrie a aplicação. Qual é a seção que define as variáveis de ambiente do pod?

   Verifique, ainda, que a alteração de cor surtiu o efeito esperado.

▼ *Visualizar resposta*

Podemos fazer todos os passos com um *one-liner*, como se segue. Note o uso do comando `sed` para alterar a variável de ambiente que define a cor de fundo da aplicação.

```
# kubectl get pod bobross -o yaml > bobross.yaml ; sed -i 's/purple/yellow/'
bobross.yaml ; kubectl delete pod bobross ; kubectl create -f bobross.yaml
pod "bobross" deleted
pod/bobross created
```

A variável de ambiente é definida na seção `.spec.containers.env`, como visto com o comando `grep` abaixo:

```
# grep yellow bobross.yaml -B4
spec:
  containers:
  - env:
    - name: COLOR
      value: yellow
```

Vejamos se a aplicação mudou de cor, de fato:

```
# curl http://localhost:31080/color
Hostname: bobross ; Color: yellow
```

5. Vamos alterar o formato através do qual essas variáveis de ambiente são definidas — usando ConfigMaps. Crie um ConfigMap com o nome `cm-bobross` via arquivo YAML, definindo a variável `COLOR` com o valor `pink`.

   Em seguida, delete o pod `bobross` e recrie-o, desta vez definindo a variável de ambiente `COLOR` usando o ConfigMap `cm-bobross`. Teste o funcionamento de sua configuração.

   ▼ *Visualizar resposta*

   Para criar o ConfigMap, basta editar o arquivo YAML com o conteúdo abaixo:

   ```
   1 apiVersion: v1
   2 kind: ConfigMap
   3 metadata:
   4   name: cm-bobross
   5 data:
   6   COLOR: pink
   ```

   Vamos criá-lo via `kubectl create`:

   ```
   # kubectl create -f cm-bobross.yaml
   ```

```
configmap/cm-bobross created
```

E verificar que seu conteúdo é, de fato, o que esperamos. Note que podemos usar `kubectl describe cm` (em lugar da forma longa `kubectl describe configmaps`) para economizar tempo.

```
# kubectl describe cm cm-bobross
Name:         cm-bobross
Namespace:    default
Labels:       <none>
Annotations:  <none>

Data
===
COLOR:
---
pink

BinaryData
===

Events:  <none>
```

Para fazer com que o pod utilize o ConfigMap, iremos criá-lo usando um arquivo YAML, com o conteúdo abaixo:

```
 1 apiVersion: v1
 2 kind: Pod
 3 metadata:
 4   labels:
 5     app: bobross
 6   name: bobross
 7 spec:
 8   containers:
 9   - image: fbscarel/myapp-color
10     name: bobross
11     env:
12       - name: COLOR
13         valueFrom:
14           configMapKeyRef:
15             name: cm-bobross
16             key: COLOR
```

A seguir, criamos o pod:

```
# kubectl delete pod bobross ; kubectl create -f bobross.yaml
pod/bobross created
```

E verificamos que a cor da aplicação foi de fato alterada:

```
# curl http://localhost:31080/color
Hostname: bobross ; Color: pink
```

6. Agora, vamos utilizar ConfigMaps de uma forma mais complexa. Realize as tarefas que se seguem:

   - Crie um pod com o nome `daytona-db`, usando a imagem `mysql:5.6` e defina a senha do usuário `root` como sendo `beach`. Publique os serviços da porta 3306 deste pod dentro do contexto do *cluster*.
   - Crie o ConfigMap `daytona-cm` que ajuste as variáveis `DBHOST`, `DBUSER` e `DBPASS` de acordo com os parâmetros do pod criado no passo anterior. Não utilize um arquivo YAML neste passo.
   - Crie e publique na porta 32080 dos *nodes* os serviços do pod `daytona-app`, usando a imagem `fbscarel/myapp-mysql`. Faça com que esse pod carrege todas as suas variáveis de ambiente a paritr do ConfigMap criado no passo anterior.
   - Acesse a interface da aplicação usando um navegador web em sua máquina física e verifique o funcionamento de sua configuração.

   ▼ *Visualizar resposta*

   Vamos lá: primeiro, criamos o pod `daytona-db` com as configurações solicitadas:

   ```
   # kubectl run daytona-db --image=mysql:5.6 --env="MYSQL_ROOT_PASSWORD=beach"
   --port=3306 --expose
   service/daytona-db created
   pod/daytona-db created
   ```

   Para criar o ConfigMap devemos usar o comando `kubectl create cm`. Como o enunciado solicita que não seja usado um arquivo YAML para este fim, iremos utilizar a opção `--from-literal` para passar pares chave-valor diretamente a partir da linha de comando.

   ```
   # kubectl create cm daytona-cm --from-literal="DBHOST=daytona-db" --from
   -literal="DBUSER=root" --from-literal="DBPASS=beach"
   configmap/daytona-cm created
   ```

   Vamos ver se funcionou:

   ```
   # kubectl describe cm daytona-cm
   Name:         daytona-cm
   Namespace:    default
   Labels:       <none>
   Annotations:  <none>

   Data
   ===
   ```

```
DBHOST:
----
daytona-db
DBPASS:
----
beach
DBUSER:
----
root
Events:  <none>
```

Perfeito. Agora, vamos criar o arquivo YAML do pod `daytona-app`. Note que, ao invés de informar quais variáveis devem ser carregadas a partir do ConfigMap, uma a uma, podemos utilizar a diretiva `envFrom` para carregar todas as variáveis disponíveis no ConfigMap.

```
 1 apiVersion: v1
 2 kind: Pod
 3 metadata:
 4   labels:
 5     app: daytona-app
 6   name: daytona-app
 7 spec:
 8   containers:
 9   - image: fbscarel/myapp-mysql
10     name: daytona-app
11     envFrom:
12     - configMapRef:
13         name: daytona-cm
```

Vamos criar o pod:

```
# kubectl create -f daytona-app.yaml
pod/daytona-app created
```

E em seguida, o serviço do tipo `NodePort` que irá publicá-lo:

```
# kubectl create svc nodeport daytona-app --tcp=80 --node-port=32080
service/daytona-app created
```

Finalmente, vamos testar o funcionamento de nossa configuração via navegador web, na máquina física.

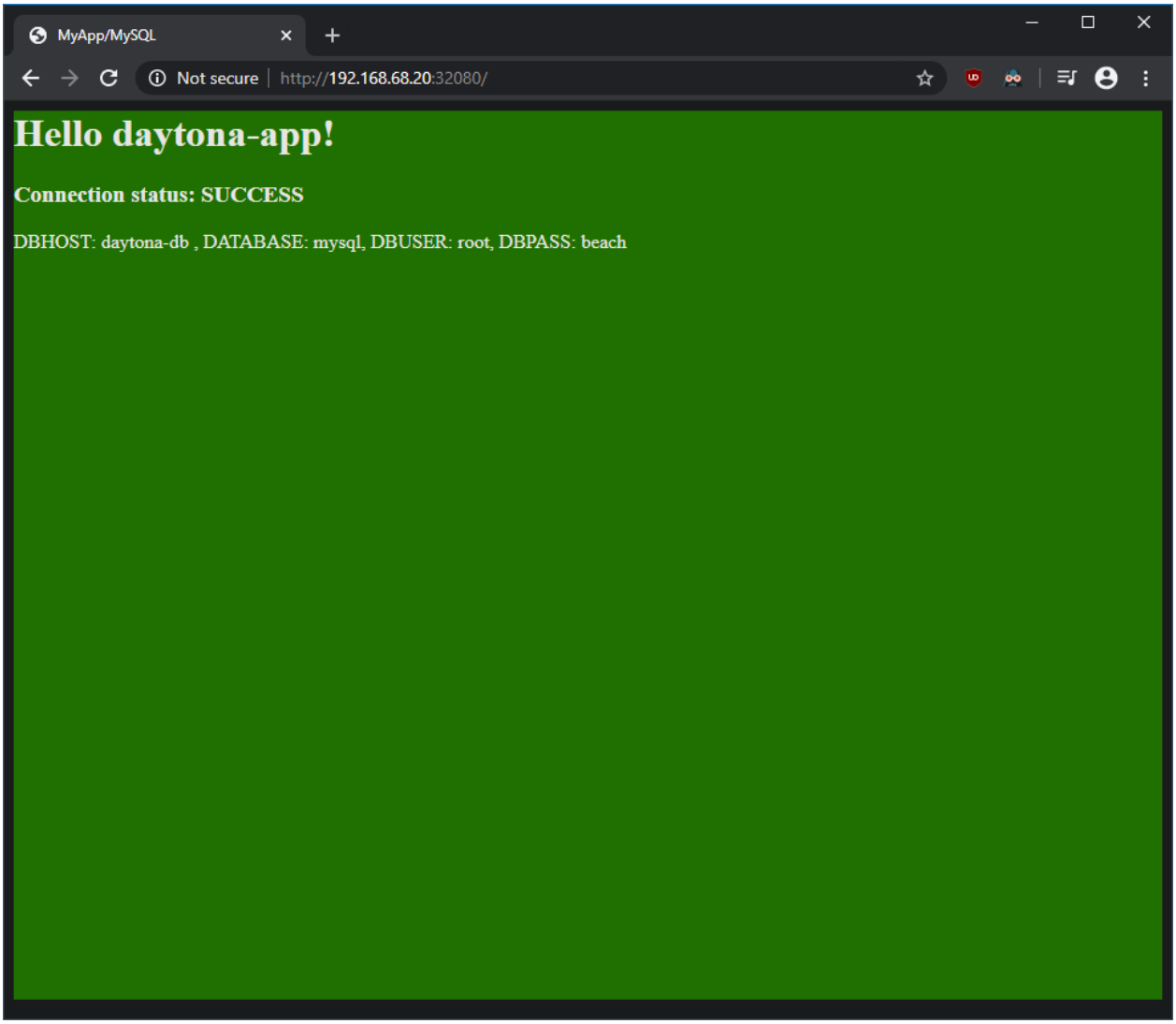

Figura 2. Conexão ao banco de dados ajustada via ConfigMap

4) Segredos (Secrets) em aplicações

1. Quantos Secrets existem no namespace *default*, no momento?

   ▼ *Visualizar resposta*

   É bastante fácil obter essa informação com o comando `kubectl get secrets`. Veja:

   ```
   # kubectl get secrets --no-headers | wc -l
   1
   ```

2. Quantos segredos estão definidos dentro desse(s) objeto(s)?

   ▼ *Visualizar resposta*

   Basta contar quantos pares chave-valor existem na seção `Data` na saída do comando `kubectl describe secrets`. No caso, temos três segredos definidos. Note que o nome do `default-token` em seu sistema provavelmente será diferente do exibido no comando abaixo — edite conforme necessário.

```
# kubectl describe secrets default-token-w9ftb | sed -n -e '/^Data/,$p'
Data
====
token:
eyJhbGciOiJSUzI1NiIsImtpZCI6ImJtTC1pa3FNc0pPYzB3QnFpUkZjajI2S0lCTUQ0a29mT1Nmd3Jqd
0tnN00ifQ.eyJpc3MiOiJrdWJlcm5ldGVzL3NlcnZpY2VhY2NvdW50Iiwia3ViZXJuZXRlcy5pby9zZXJ
2aWNlYWNjb3VudC9uYW1lc3BhY2UiOiJkZWZhdWx0Iiwia3ViZXJuZXRlcy5pby9zZXJ2aWNlYWNjb3Vu
dC9zZWNyZXQubmFtZSI6ImRlZmF1bHQtdG9rZW4tdzc3ejciLCJrdWJlcm5ldGVzLmlvL3NlcnZpY2VhY
2NvdW50L3NlcnZpY2UtYWNjb3VudC5uYW1lIjoiZGVmYXVsdCIsImt1YmVybmV0ZXMuaW8vc2VydmljZW
FjY291bnQvc2VydmljZS1hY2NvdW50LnVpZCI6Ijg4ODc2NTM5LTNlOGMtNGEzZC05Zjc4LTc0ODI2MWI
4MmFkNCIsInN1YiI6InN5c3RlbTpzZXJ2aWNlYWNjb3VudDpkZWZhdWx0OmRlZmF1bHQifQ.CRHyR6E9d
70iPKY0cWCN3iVe4RJbc7aBeEJPtzpKDXWTS4sb81yd8vEinA1c1Qlldafw8UPuZujTiDQ8butJIuoqQn
2a2Poi2tUyvbMRmKkZ3EKa0YAP3AMAkwap2YG2LHlRG7lYtzhAwSdtDLLr6owhcbSaI7PkQHTbWMbOXp8
OrQgahicpxK8NKlKsHgDo0zKcSL6L_EGHNFSgA10plvVtD1kyGMbcVQbdI3rRVWm7tADLLko7-
Zd4DirTIbWiEMWbGG44-4O5GvC_0y2idj1r3vnlepdQIlyNUXh-
e_7IwjFuWHUK43tXepDAVvsS2Gj_LILiN7HkDGpPsDqPeQ
ca.crt:     1099 bytes
namespace:  7 bytes
```

3. Qual é o tipo desse(s) Secret(s)?

   ▼ *Visualizar resposta*

   Essa informação pode ser visualizada diretamente via `kubectl get secrets` ou, mais ao ponto, extraindo o campo específico via JSONPath.

   ```
   # kubectl get secrets default-token-w9ftb -o jsonpath={.type}
   kubernetes.io/service-account-token
   ```

4. Um grave problema com a configuração da aplicação `daytona-app` realizada na atividade anterior é que tanto o usuário quanto a senha de acesso ao banco de dados, ambas informações sensíveis, estão armazenadas em um ConfigMap sem qualquer segurança. Para solucionar isso, iremos mover essas duas variáveis para um Secret.

   Primeiro, crie um arquivo YAML definindo o secret `daytona-secret-yaml`, com as variáveis `DBUSER` e `DBPASS`. Os valores dessas variáveis devem ser os mesmos utilizados anteriormente.

   A seguir, crie um secret via `kubectl` com o nome `daytona-secret-kubectl`, com as mesmas variáveis e valores.

   Finalmente, compare ambos e verifique que são, de fato, idênticos.

   ▼ *Visualizar resposta*

   Para criar um Secret via arquivo YAML, devemos primeiramente obter os valores dos segredos em formato base64. Isso pode ser feito trivialmente, usando os comandos que se seguem:

   ```
   # echo -n 'root' | base64
   ```

```
cm9vdA==
```

```
# echo -n 'beach' | base64
YmVhY2g=
```

Feito isso, criamos o arquivo YAML descrevendo o Secret.

```
1 apiVersion: v1
2 kind: Secret
3 metadata:
4   name: daytona-secret-yaml
5 data:
6   DBPASS: YmVhY2g=
7   DBUSER: cm9vdA==
```

E o criamos via `kubectl create`:

```
# kubectl create -f daytona-secret-yaml.yaml
secret/daytona-secret-yaml created
```

Vamos fazer o mesmo procedimento, agora via linha de comando. Assim como no caso de ConfigMaps, podemos usar a opção `--from-literal` para informar os pares chave-valor diretamente no comando sendo executado. Atente-se para o fato de que essas informações poderão constar no histórico de comandos do usuário e nos logs do sistema, dependendo da configuração do ambiente — isso pode ser considerada uma exposição de informação sensível.

```
# kubectl create secret generic daytona-secret-kubectl --from
-literal="DBUSER=root" --from-literal="DBPASS=beach"
secret/daytona-secret-kubectl created
```

Utilizando o comando `diff`, iremos agora comparar os dois Secrets criados, via arquivo YAML e via linha de comando. Note que os campos diferentes em ambos os objetos são apenas seus nomes e detalhes de *status*, como data de criação e versionamento do recurso. Para todos os efeitos, ambos são idênticos.

```
<strong># diff <(kubectl get secret daytona-secret-yaml -o=yaml) <(kubectl get
secret daytona-secret-kubectl -o=yaml)</strong>
7,8c7,8
&lt;   creationTimestamp: "2022-03-26T13:30:32Z"
&lt;   name: daytona-secret-yaml
---
&gt;   creationTimestamp: "2022-03-26T13:30:37Z"
&gt;   name: daytona-secret-kubectl
```

```
10,11c10,11
&lt;   resourceVersion: "100988"
&lt;   uid: c40024af-4b16-4376-8a44-0831d577c403
---
&gt;   resourceVersion: "100994"
&gt;   uid: 9caa6609-a61d-412e-bc56-5081a0608edc
```

5. Ajuste a aplicação `daytona-app` para que os valores das variáveis `DBUSER` e `DBPASS` sejam obtidos a partir de um dos secrets criados no passo anterior.

   Mantenha a variável `DBHOST` sendo definida a partir do ConfigMap `daytona-cm`. Adicionalmente, edite esse ConfigMap e remova as variáveis sensíveis.

   Verifique que a aplicação mantém-se funcional após seus ajustes.

   ▼ *Visualizar resposta*

   Vamos editar o arquivo YAML da aplicação `daytona-app` — iremos puxar todas as variáveis de ambiente do secret `daytona-secret-yaml` (poderia ser o outro, criado no passo anterior, sem qualquer prejuízo), e apenas a variável `DB_HOST` a partir do ConfigMap `daytona-cm`.

   ```yaml
   apiVersion: v1
   kind: Pod
   metadata:
     labels:
       app: daytona-app
     name: daytona-app
   spec:
     containers:
     - image: fbscarel/myapp-mysql
       name: daytona-app
       env:
         - name: DBHOST
           valueFrom:
             configMapKeyRef:
               name: daytona-cm
               key: DBHOST
       envFrom:
       - secretRef:
           name: daytona-secret-yaml
   ```

   Agora, iremos remover as variáveis sensíveis do ConfigMap.

   ```
   # kubectl edit cm daytona-cm
   configmap/daytona-cm edited
   ```

   E verificar que foram, de fato, removidas.

```
# kubectl describe cm daytona-cm
Name:         daytona-cm
Namespace:    default
Labels:       <none>
Annotations:  <none>

Data
===
DBHOST:
---
daytona-db

BinaryData
===

Events:  <none>
```

Finalmente, recriamos o pod `daytona-app` com o arquivo YAML atualizado.

```
# kubectl delete pod daytona-app ; kubectl create -f daytona-app.yaml
pod "daytona-app" deleted
pod/daytona-app created
```

Via navegador, constatamos que a aplicação permanece funcional, como esperado.

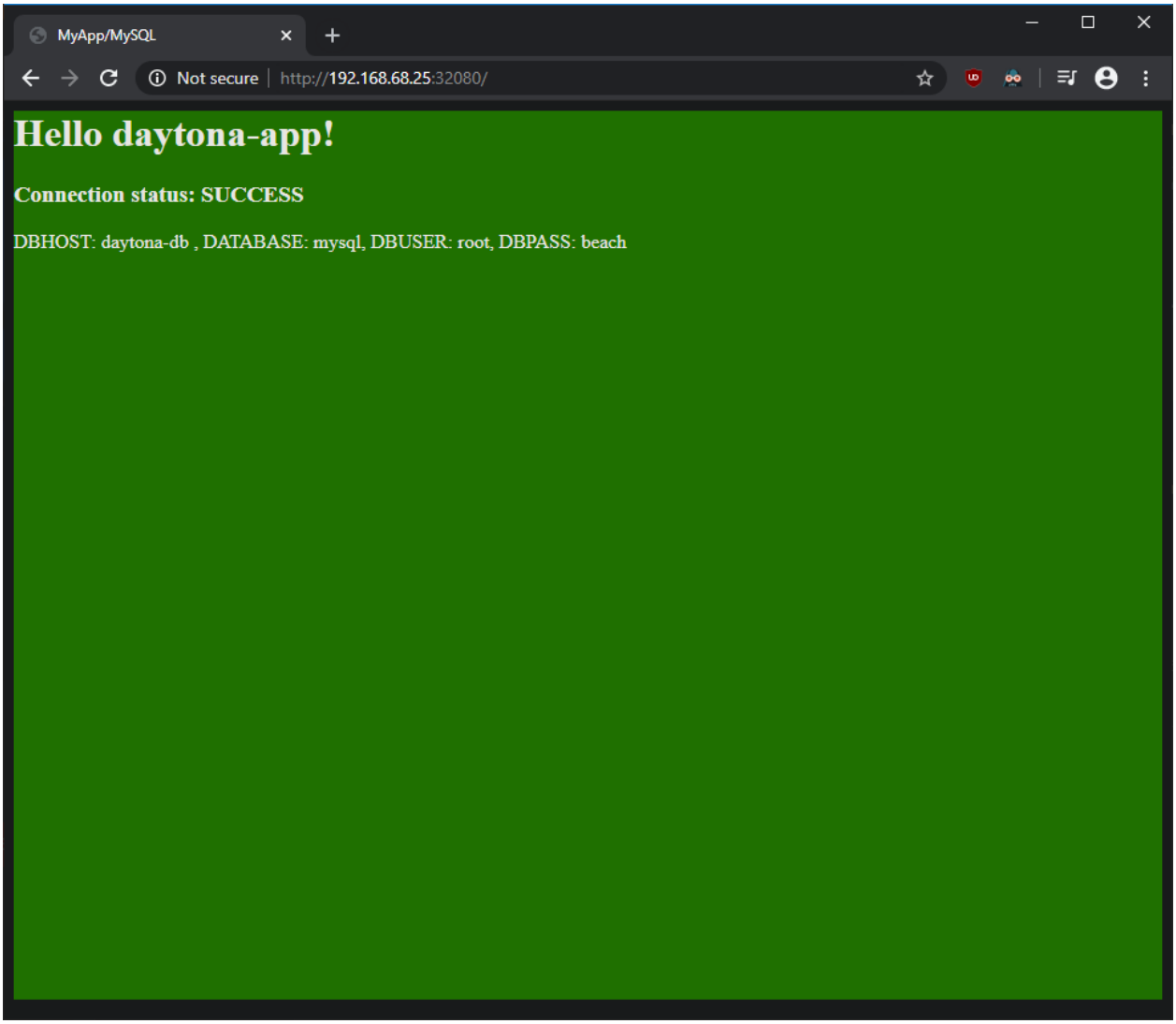

Figura 3. Configuração de usuário/senha definida via Secret

5) Multi-container Pods e Init Containers

5.1) Multi-container Pods

1. Antes de iniciar, execute o comando abaixo:

```
# lab-5.5.1
```

2. Quantos containers existem dentro do pod `flintstones`?

   ▼ *Visualizar resposta*

   Podemos determinar essa informação de diversas formas — por exemplo, contando o número de `Container IDs` do pod:

```
# kubectl describe pod flintstones | grep 'Container ID:' | wc -l
3
```

3. Qual é a imagem utilizada em cada um desses containers?

▼ *Visualizar resposta*

Vejamos:

```
# kubectl describe pod flintstones | grep 'Image:'
    Image:          nginx:alpine
    Image:          redis:alpine
    Image:          httpd:alpine
```

4. Qual o estado do pod `flintstones`? Por que motivo o(s) container(s) não estão sendo iniciados?

▼ *Visualizar resposta*

O pod não está em operação, apenas 2/3 containers foram executados com sucesso. Investigando o estado deles, fica claro que o container `barney` está tendo problemas:

```
# kubectl describe pod flintstones | grep 'barney:' -A8
  barney:
    Container ID:
docker://e1ba1c1588108c555126f7f49bb297826ef2e0a4f5e52f5bb121122c1587abc4
    Image:          httpd:alpine
    Image ID:       docker-
pullable://httpd@sha256:d27f57dcfaf89612b95e5aedbe628feb3d358bac08f046585f287c4e3
1acfc87
    Port:           <none>
    Host Port:      <none>
    State:          Waiting
      Reason:       CrashLoopBackOff
    Last State:     Terminated
```

Investigando os logs do mesmo, fica claro o motivo: ele não consegue escutar na porta 80, que já foi reservada pelo container que utiliza a imagem `nginx:alpine`.

```
# kubectl logs flintstones barney
AH00558: httpd: Could not reliably determine the server's fully qualified domain
name, using 10.44.0.3. Set the 'ServerName' directive globally to suppress this
message
(98)Address in use: AH00072: make_sock: could not bind to address 0.0.0.0:80
no listening sockets available, shutting down
AH00015: Unable to open logs
```

5. Crie um pod com o nome `jetsons`. Dentro dele, crie dois containers, um com o nome `george` e imagem `busybox`, rodando o comando `sleep 600`; o outro container deve se chamar `jane`, com a imagem `nginx:alpine`.

▼ *Visualizar resposta*

Vamos criar um arquivo YAML obedecendo às especificações informadas no enunciado:

```yaml
apiVersion: v1
kind: Pod
metadata:
  labels:
    run: jetsons
  name: jetsons
spec:
  containers:
  - image: busybox
    name: george
    args:
      - sleep
      - "600"
  - image: nginx:alpine
    name: jane
```

A seguir, criamos o pod:

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

E verificamos seu estado — note que ambos os containers estão operacionais, como objetivado.

```
# kubectl get pod jetsons
NAME      READY   STATUS    RESTARTS   AGE
jetsons   2/2     Running   0          3m45s
```

5.2) Init Containers

1. Antes de iniciar, execute o comando abaixo:

```
# lab-5.5.2
```

2. Qual dos pods criados possui um initContainer configurado?

▼ *Visualizar resposta*

Podemos visualizar as configurações de cada pod em detalhe via `kubectl describe pod`... ou, de forma mais direta, podemos buscar quais pods possuem a seção `.spec.initContainers`:

```
# kubectl get pod -o custom-
columns=NAME:.metadata.name,INITCONTAINERS:.spec.initContainers[*].name
NAME         INITCONTAINERS
scoobydoo    <none>
wackyraces   muttley
```

```
woodpecker   <none>
```

Veja que apenas o pod `wackyraces` possui um initContainer, de nome `muttley`.

3. Qual é a imagem utilizada pelo initContainer nesse pod?

▼ *Visualizar resposta*

Vejamos:

```
# kubectl get pod wackyraces -o jsonpath='{.spec.initContainers[*].image}'
alpine
```

4. Qual é o estado desse initContainer? Por que razão?

▼ *Visualizar resposta*

Ambas as informações figuram dentro do comando `kubectl describe pod`. Vamos selecionar as seções específicas via `grep`:

```
# kubectl describe pod wackyraces | grep 'muttley:' -A10 | grep 'State:'
    State:          Terminated
```

```
# kubectl describe pod wackyraces | grep 'muttley:' -A10 | grep 'Reason:'
      Reason:       Completed
```

5. Crie o pod `looney` com dois initContainers, `bugs` e `daffy`, ambos usando a imagem `busybox` e executando, respectivamente, os comandos `sleep 600` e `sleep 1200`. A seguir, inclua um container de nome `elmer` com a imagem `fbscarel/myapp-color`.

A seguir, responda:

- Qual é o estado do pod, após sua criação?
- Após quanto tempo o pod está pronto para atender requisições de clientes?

▼ *Visualizar resposta*

Vamos primeiramente criar o arquivo YAML segundo as especificações:

```
1 apiVersion: v1
2 kind: Pod
3 metadata:
4   labels:
5     app: looney
6   name: looney
7 spec:
8   initContainers:
9   - image: busybox
```

```
    name: bugs
    args:
      - sleep
      - "600"
  - image: busybox
    name: daffy
    args:
      - sleep
      - "1200"
  containers:
  - image: fbscarel/myapp-color
    name: elmer
```

A seguir, criamos o pod:

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

E verificamos seu estado. Note que ele se encontra como `Init:0/2`.

```
# kubectl get pod looney
NAME     READY   STATUS     RESTARTS   AGE
looney   0/1     Init:0/2   0          14s
```

Os initContainers são inicializados sequencialmente, um por vez. Portanto, temos que primeiro será executado o initContainer `bugs`, concluído após 600 segundos, e posteriormente o initContainer `daffy`, concluído após 1200 segundos. Só então o container `elmer` será executado — portanto, 600 + 1200 = 1800 segundos, ou 30 minutos.

6. Altere a configuração do pod de forma que o tempo total de espera até que a aplicação torne-se disponível seja de 30 segundos. Publique um serviço para viabilizar acesso à aplicação e teste seu funcionamento.

▼ *Visualizar resposta*

Vamos alterar ambos os tempos dos initContainers `bugs` e `daffy` para 15 segundos cada, via `sed`. A seguir, deletamos e recriamos o pod. Veja:

```
# sed -i 's/[1]*[2,6]00/15/' looney.yaml ; kubectl delete pod looney ; kubectl
create -f looney.yaml
pod "looney" deleted
pod/looney created
```

Após 30 segundos, o pod está ativo.

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

```
looney   1/1     Running   0          101s
```

Vamos visualizar o estado dos initContainers — note que ambos estão registrados como `Completed`.

```
# kubectl get pod looney -o custom-
columns=NAME:.spec.initContainers[*].name,STATUS:.status.initContainerStatuses[*]
.state.terminated.reason
NAME         STATUS
bugs,daffy   Completed,Completed
```

Crie o serviço do tipo `NodePort` para acessar o container...

```
# kubectl create svc nodeport looney --tcp=80 --node-port=32180
service/looney created
```

E valide seu funcionamento via `curl`.

```
# curl http://localhost:32180/color
Hostname: looney ; Color: blue
```

6) Escalabilidade automática de aplicações

1. Remova os pods e serviços criados na atividade anterior com os comandos que se seguem.

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

```
# kubectl delete svc -l provider!=kubernetes
```

2. Crie um deployment com o nome `php-hpa`, usando a imagem `fbscarel/php-sqrt` e apenas uma réplica. Imponha um requerimento de uso de CPU de 100 milliCPU e limite de 200 mCPU.

   ▼ *Visualizar resposta*

   Vamos criar o deployment via arquivo YAML, já que queremos estabelecer limites de recursos. Use a configuração abaixo.

```
1 apiVersion: apps/v1
2 kind: Deployment
3 metadata:
4   name: php-hpa
5 spec:
6   selector:
7     matchLabels:
```

```
      app: php-hpa
  replicas: 1
  template:
    metadata:
      labels:
        app: php-hpa
    spec:
      containers:
      - name: php-hpa
        image: fbscarel/php-sqrt
        resources:
          limits:
            cpu: 200m
          requests:
            cpu: 100m
```

A seguir, criamos o deployment via `kubectl apply`.

```
# kubectl apply -f php-hpa.yaml
deployment.apps/php-hpa created
```

3. Exponha a porta 80 dos pods do deployment criado no passo anterior dentro do contexto do *cluster*.

▼ *Visualizar resposta*

Bem fácil, não é mesmo?

```
# kubectl create svc clusterip php-hpa --tcp=80
service/php-hpa created
```

4. Crie uma configuração de escalabilidade automática para o deployment usando o comando `kubectl autoscale`. Faça com que o *Horizontal Pod Autoscaler* mantenha entre 1 e 5 cópias dos pods do deployment `php-hpa`, objetivando um uso de 50% de CPU entre esses pods.

   Feito isso, visualize o estado atual do HPA e valide o funcionamento de sua configuração.

▼ *Visualizar resposta*

Vamos lá: felizmente, o comando `kubectl autoscale` é bastante auto-explicativo, e fica claro como converter as instruções do enunciado para parâmetros de linha de comando.

```
# kubectl autoscale deploy php-hpa --cpu-percent=50 --min=1 --max=5
horizontalpodautoscaler.autoscaling/php-hpa autoscaled
```

Terá funcionado? Vamos consultar o estado via `kubectl get hpa`:

```
# kubectl get hpa
```

```
NAME      REFERENCE              TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
php-hpa   Deployment/php-hpa     1%/50%    1         5         1          81s
```

5. Utilizando a imagem `busybox` e o comando `wget`, crie o pod `load-generator` que executa um *script* simples com requisições constantes à aplicação criada nos passos anteriores, gerando carga para a mesma.

   ▼ *Visualizar resposta*

   Vamos utilizar um *loop*, como fizemos antes. Note que, como ambos os pods estão dentro do mesmo namespace, podemos utilizar o nome curto do alvo:

   ```
   # kubectl run load-generator --image=busybox -- /bin/sh -c 'while true; do wget
   -q -O- http://php-hpa; done'
   pod/load-generator created
   ```

6. Monitore a evolução do *Horizontal Pod Autoscaler* e do deployment, visualizando o crescimento da carga demandada, bem como do número de réplicas criadas.

   ▼ *Visualizar resposta*

   Execute o comando `kubectl get hpa` periodicamente. Note que o número de réplicas vai subindo gradativamente para atender à carga aumentada: primeiro, temos 3 réplicas...

   ```
   # kubectl get hpa
   NAME      REFERENCE              TARGETS    MINPODS   MAXPODS   REPLICAS   AGE
   php-hpa   Deployment/php-hpa     191%/50%   1         5         3          10m
   ```

   Chegando a 4:

   ```
   # kubectl get hpa
   NAME      REFERENCE              TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
   php-hpa   Deployment/php-hpa     92%/50%   1         5         4          10m
   ```

   Do lado do deployment, o mesmo tipo de informação pode ser visualizada.

   ```
   # kubectl get deploy php-hpa
   NAME      READY   UP-TO-DATE   AVAILABLE   AGE
   php-hpa   4/4     4            4           14m
   ```

   Após algum tempo, o *Horizontal Pod Autoscaler* atinge seu limite de escalabilidade, levando o deployment a 5 réplicas.

   ```
   # kubectl get hpa
   NAME      REFERENCE              TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
   php-hpa   Deployment/php-hpa     46%/50%   1         5         5          63m
   ```

7. Encerre o pod `load-generator`, reduzindo a carga do *Horizontal Pod Autoscaler*. Acompanhe seu estado, bem como do deployment, e veja o que ocorre com o número de réplicas ativas.

   ▼ *Visualizar resposta*

   Vamos deletar o pod:

   ```
   # kubectl delete pod load-generator
   pod "load-generator" deleted
   ```

   Para facilitar o monitoramento, iremos criar um *loop* que, de 10 em 10 segundos, irá mostrar o estado do HPA e do deployment na tela:

   ```
   # while true; do k get hpa; k get deploy php-hpa; echo '---'; sleep 10; done
   NAME      REFERENCE            TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
   php-hpa   Deployment/php-hpa   51%/50%   1         5         5          75m
   NAME      READY   UP-TO-DATE   AVAILABLE   AGE
   php-hpa   5/5     5            5           79m
   ---
   NAME      REFERENCE            TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
   php-hpa   Deployment/php-hpa   15%/50%   1         5         5          76m
   NAME      READY   UP-TO-DATE   AVAILABLE   AGE
   php-hpa   5/5     5            5           79m
   ---
   NAME      REFERENCE            TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
   php-hpa   Deployment/php-hpa   1%/50%    1         5         5          76m
   NAME      READY   UP-TO-DATE   AVAILABLE   AGE
   php-hpa   5/5     5            5           79m

   (...)

   NAME      REFERENCE            TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
   php-hpa   Deployment/php-hpa   1%/50%    1         5         5          81m
   NAME      READY   UP-TO-DATE   AVAILABLE   AGE
   php-hpa   2/2     2            2           84m
   ---
   NAME      REFERENCE            TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
   php-hpa   Deployment/php-hpa   1%/50%    1         5         2          81m
   NAME      READY   UP-TO-DATE   AVAILABLE   AGE
   php-hpa   1/1     1            1           84m
   ---
   NAME      REFERENCE            TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
   php-hpa   Deployment/php-hpa   1%/50%    1         5         1          81m
   NAME      READY   UP-TO-DATE   AVAILABLE   AGE
   php-hpa   1/1     1            1           85m
   ```

   Observe que a carga vai diminuindo rapidamente, até chegar a 1% de CPU. Após alguns minutos, o HPA vai reduzindo o número de réplicas do deployment, indo do número máximo de 5 até o mínimo, 1 réplica.

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 o crescimento do número de réplicas do *Horizontal Pod Autoscaler* `php-hpa` à medida que a carga ao *deployment* `php-hpa` é aumentada.

Utilize como referência a saída de comando mostrada na atividade 5.6 (f) deste roteiro.