Índice

Sessão 2: Conceitos centrais do Kubernetes

1) Preparação do ambiente

Antes de prosseguir, remova a máquina virtual utilizada na sessão anterior.

```
C:\contorq-files\s1> vagrant destroy -f
```

Agora, entre na pasta `contorq-files\s2` e inicie o provisionamento das VMs para esta sessão.

```
C:\> cd contorq-files\s2
```

```
C:\contorq-files\s2> vagrant up
Bringing machine 's2-master-1' up with 'virtualbox' provider...
Bringing machine 's2-node-1' up with 'virtualbox' provider...
==> s2-master-1: Importing base box 'bento/debian-11'...

(...)
```

O processo pode demorar alguns minutos, então seja paciente. Uma vez concluído, entre na VM mestre do *cluster* Kubernetes (`s2-master-1`).

```
C:\contorq-files\s2> vagrant ssh s2-master-1
Linux s2-master-1 5.10.0-10-amd64 #1 SMP Debian 5.10.84-1 (2021-12-08) x86_64

(...)
```

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

Vire o superusuário `root` e verifique que o *cluster* está operacional: ambos os nós `s2-master-1` e `s2-node-1` devem estar no estado `Ready`.

```
vagrant@s2-master-1:~$ sudo -i
```

```
root@s2-master-1:~# kubectl get nodes
NAME          STATUS   ROLES                  AGE     VERSION
s2-master-1   Ready    control-plane,master   5m25s   v1.23.5
s2-node-1     Ready    <none>                 102s    v1.23.5
```

Assim como na sessão anterior, iremos a partir de agora omitir os *prompts* `vagrant@docker:~$` e `root@docker:~ para maior claridade. Em seu lugar, serão mostrados apenas os caracteres $ (para o usuário não-privilegiado vagrant), e` (para o superusuário `root`).

2) Pods

2.1) Criando e removendo pods

1. Quantos pods estão em execução no sistema, neste momento? Considere apenas o *namespace default*, por ora.

   ▼ *Visualizar resposta*

   ```
   # kubectl get pods
   No resources found in default namespace.
   ```

2. Crie um novo pod com o nome `web` utilizando a imagem `nginx:alpine`, a partir de um arquivo YAML. Verifique que o pod está operacional.

   ▼ *Visualizar resposta*

   Começamos a partir da criação de um arquivo YAML com a definição do pod solicitado, com o conteúdo abaixo:

   ```yaml
   apiVersion: v1
   kind: Pod
   metadata:
     name: web
   spec:
     containers:
     - image: nginx:alpine
       name: web
   ```

Para criar o pod basta invocar o comando `kubectl create` com a *flag* `-f`, como visto a seguir.

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

Para verificar a correta execução do comando, utilize:

```
# kubectl get pods
NAME    READY   STATUS    RESTARTS   AGE
web     1/1     Running   0          17s
```

3. Remova o pod criado no passo anterior. Quantos pods estão em execução agora?

▼ *Visualizar resposta*

```
# kubectl delete pod web
pod "web" deleted
```

```
# kubectl get pods --no-headers=true | wc -l
No resources found in default namespace.
0
```

2.2) Verificando imagens e nodes utilizados

1. Antes de iniciar, execute o comando abaixo:

```
# lab-2.2.2
```

2. E agora, quantos pods estão em execução no sistema?

▼ *Visualizar resposta*

```
# kubectl get pods
NAME    READY   STATUS         RESTARTS   AGE
alpha   1/1     Running        0          29s
beta    1/2     ErrImagePull   0          29s
```

```
# kubectl get pods --no-headers | wc -l
2
```

3. Qual a imagem utilizada no pod `alpha`?

▼ *Visualizar resposta*

Pode-use utilizar o comando `kubectl describe pod` para esse fim, observando o campo `Image`.

```
# kubectl describe pod alpha | grep '^[[:space:]]*Image:'
    Image:          debian:stable-slim
```

Alternativamente, pode-se utilizar a saída em formato JSON do pod e pesquisar via JSONPath pelo caminho que informa a imagem de cada container no pod.

```
# kubectl get pod alpha -o jsonpath='{.status.containerStatuses[*].image}'
debian:stable-slim
```

4. Em quais *nodes* (nós) do *cluster* esses containers estão executando?

▼ *Visualizar resposta*

Pode-se informar o parâmetro `-o wide` para o comando `kubectl get pods`, que irá incluir colunas adicionais incluindo o *node* em que os pods estão executando.

```
# kubectl get pods -o wide
NAME    READY   STATUS         RESTARTS   AGE   IP          NODE        NOMINATED
NODE   READINESS GATES
alpha   1/1     Running        0          63s   10.44.0.1   s2-node-1   <none>
<none>
beta    1/2     ErrImagePull   0          63s   10.44.0.2   s2-node-1   <none>
<none>
```

Para formatar a saída de maneira mais precisa pode-se usar uma variedade de técnicas — a opção `-o custom-columns` é uma delas, demonstrada abaixo:

```
# kubectl get pods -o custom-columns=NAME:.metadata.name,NODE:.spec.nodeName
NAME    NODE
alpha   s2-node-1
beta    s2-node-1
```

2.3) Multi-container pods

1. Quantos containers fazem parte do pod `beta`?

▼ *Visualizar resposta*

Basta visualizar a coluna `READY` na saída do comando `kubectl get pods`:

```
# kubectl get pod beta
NAME   READY   STATUS         RESTARTS   AGE
beta   1/2     ErrImagePull   0          85s
```

2. Qual o nome das imagens utilizadas em cada um desse(s) container(s)?

   ▼ *Visualizar resposta*

   Pode-se obter a informação de diferentes formas. Por exemplo:

```
# kubectl describe pod beta | grep '^[[:space:]]*Image:'
    Image:          httpd:alpine
    Image:          bredis:alpine
```

```
# kubectl get pod beta -o jsonpath={.spec.containers[*].image}
httpd:alpine bredis:alpine
```

3. Qual é o estado do container `beta-2` no pod `beta`? Há algum erro associado a ele?

   ▼ *Visualizar resposta*

   Para visualizar o estado de um container basta investigar o valor do campo `State` na saída do comando `kubectl describe pod`. Veja:

```
# kubectl describe pod beta | grep '^[[:space:]]*State:' -B6
  beta-1:
    Container ID:
docker://1cc6eaaeec32231c020c9e4769ad39010a5d2b9ac58c256e52331e914011c50e
    Image:          httpd:alpine
    Image ID:       docker-
pullable://httpd@sha256:d27f57dcfaf89612b95e5aedbe628feb3d358bac08f046585f287c4e3
1acfc87
    Port:           <none>
    Host Port:      <none>
    State:          Running
--
  beta-2:
    Container ID:
    Image:          bredis:alpine
    Image ID:
    Port:           <none>
    Host Port:      <none>
    State:          Waiting
```

   A razão para o estado `Waiting` pode ser obtida na seção `Events` do mesmo comando. Abaixo, as três últimas linhas da saída mostram o erro encontrado:

```
# kubectl describe pod beta | tail -n4
  Warning  Failed     38s (x3 over 83s)  kubelet            Failed to pull image
"bredis:alpine": rpc error: code = Unknown desc = Error response from daemon:
pull access denied for bredis, repository does not exist or may require 'docker
login': denied: requested access to the resource is denied
```

```
  Warning  Failed     38s (x3 over 83s)  kubelet            Error: ErrImagePull
  Normal   BackOff    12s (x4 over 82s)  kubelet            Back-off pulling
image "bredis:alpine"
  Warning  Failed     12s (x4 over 82s)  kubelet            Error:
ImagePullBackOff
```

Pode-se utilizar JSONPath para obter não apenas o estado mas também a razão, como visto a seguir.

```
# kubectl get pod beta -o jsonpath='{.status.containerStatuses[?(@.name=="beta-
2")].state}'
{"waiting":{"message":"Back-off pulling image
\"bredis:alpine\"","reason":"ImagePullBackOff"}}
```

2.4) Corrigindo a configuração de pods

1. Corrija a configuração do pod `beta` usando o comando `kubectl edit`.

▼ *Visualizar resposta*

Para editar a configuração de um recurso no Kubernetes basta invocar o comando `kubectl edit`:

```
# kubectl edit pod beta
```

Será aberto um editor com o arquivo YAML que descreve o objeto em questão. No caso, queremos alterar o nome da imagem do container na seção `spec.containers.image` do container `beta-2`, alterando a imagem para seu nome correto (`redis:alpine`).

```
 13 spec:
 14   containers:

(...)

 25   - image: bredis:alpine
 26     imagePullPolicy: IfNotPresent
 27     name: beta-2
```

Faça isso, salve e saia do editor. Imediatamente você verá a mensagem `pod/beta edited`. Após alguns segundos, consultando o estado do pod `beta`, constatamos que o problema foi solucionado.

```
# kubectl get pod beta
NAME   READY   STATUS    RESTARTS   AGE
beta   2/2     Running   0          64m
```

2. Crie um pod com o nome `oops` e imagem `nginx-oops` — o nome da imagem está incorreto, propositalmente. Verifique o estado do pod após sua criação.

   ▼ *Visualizar resposta*

   Criamos um arquivo YAML com o conteúdo solicitado:

   ```
   1 apiVersion: v1
   2 kind: Pod
   3 metadata:
   4   name: oops
   5 spec:
   6   containers:
   7   - image: nginx-oops
   8     name: oops
   ```

   A seguir, criamos o pod com o arquivo em questão.

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

   Após algum tempo, verificamos seu estado.

   ```
   # kubectl get pod oops
   NAME   READY   STATUS             RESTARTS   AGE
   oops   0/1     ImagePullBackOff   0          36s
   ```

3. Em lugar do comando `kubectl edit`, iremos corrigir o problema de forma diferente.

   Exporte o estado do objeto para um arquivo YAML usando o parâmetro `-o=yaml`, remova o objeto original, edite o arquivo YAML e recrie o pod.

   Verifique o estado do pod após a correção do problema.

   ▼ *Visualizar resposta*

   Primeiro, exportamos o pod para um arquivo YAML usando o parâmetro informado.

   ```
   # kubectl get pod oops -o yaml > oops-fixed.yaml
   ```

   A edição do arquivo YAML pode ser feita diretamente com um editor ou de forma automatizada, p.ex. via `awk` ou `sed`. Neste caso, usaremos o `sed` em modo *inline* para fazer a alteração do arquivo.

   ```
   # sed -i 's/nginx-oops/nginx/g' oops-fixed.yaml
   ```

   A seguir, removemos o pod.

```
# kubectl delete pod oops
pod "oops" deleted
```

E o recriamos usando o arquivo YAML corrigido:

```
# kubectl create -f oops-fixed.yaml
pod/oops created
```

Finalmente, podemos constatar que o pod está agora em estado `Running`.

```
# kubectl get pod oops
NAME   READY   STATUS    RESTARTS   AGE
oops   1/1     Running   0          43s
```

3) ReplicaSets

3.1) Trabalhando com ReplicaSets

1. Antes de iniciar, execute o comando abaixo:

```
# lab-2.3.1
```

2. Quandos Pods e ReplicaSets existem no sistema, no momento?

▼ *Visualizar resposta*

Para obter informações sobre pods já sabemos o comando, como visto na seção anterior. Já para visualizar ReplicaSets pode-se utilizar `kubectl get replicasets`, ou `rs` de forma mais curta:

```
# kubectl get pod --no-headers | wc -l
3
```

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

3. Quantos pods são objetivados (*Desired*) pelo ReplicaSet? E quantos estão prontos (*Ready*)?

▼ *Visualizar resposta*

As informações objetivadas constam nas colunas `DESIRED` e `READY`, como visto a seguir.

```
# kubectl get rs
NAME     DESIRED   CURRENT   READY   AGE
```

```
rs-app   3         3         0       52s
```

4. Qual é a imagem utilizada pelos pods do ReplicaSet `rs-app`?

   ▼ *Visualizar resposta*

   O `kubectl describe rs` é ideal para este cenário. Abaixo, utilizamos o comando `grep` para filtrar informações sobre a imagem utilizada pelos pods do ReplicaSet.

   ```
   # kubectl describe rs rs-app | grep '^[[:space:]]*Image:'
       Image:      alpine42
   ```

5. Como visto no item (c), não há imagens em estado `READY` no ReplicaSet. Por qual motivo os pods não foram criados?

   ▼ *Visualizar resposta*

   Podemos determinar esse informação observando os eventos de qualquer dos pods criados automaticamente pelo ReplicaSet. Esses nomes são auto-gerados e podem ser obtidos via `kubectl get pods`.

   Abaixo, visualizamos as últimas quatro linhas de eventos de um dos pods. Note que a imagem `alpine42` não pôde ser baixada, pois inexiste no Docker Hub.

   ```
   # kubectl describe pod rs-app-mvb6f | tail -n4
     Warning  Failed     80s (x4 over 3m)     kubelet            Failed to pull
   image "alpine42": rpc error: code = Unknown desc = Error response from daemon:
   pull access denied for alpine42, repository does not exist or may require 'docker
   login': denied: requested access to the resource is denied
     Warning  Failed     80s (x4 over 3m)     kubelet            Error: ErrImagePull
     Normal   BackOff    69s (x6 over 2m59s)  kubelet            Back-off pulling
   image "alpine42"
     Warning  Failed     58s (x7 over 2m59s)  kubelet            Error:
   ImagePullBackOff
   ```

6. Corrija a imagem utilizada pelo ReplicaSet, seja via `kubectl edit` ou através da exportação do arquivo YAML do objeto e sua recriação.

   ▼ *Visualizar resposta*

   Invocando o comando `kubectl edit rs`, basta editar o nome da imagem `alpine42` para `alpine` e salvar o arquivo.

   ```
   # kubectl edit rs rs-app
   replicaset.apps/rs-app edited
   ```

7. O problema foi corrigido?

   ▼ *Visualizar resposta*

Ao contrário do previsto, ao visualizar os pods existentes no sistema constatamos que eles não foram recriados automaticamente, como visto abaixo. Portanto, o problema persiste mesmo após a correção do nome da imagem.

```
# kubectl get pod
NAME           READY   STATUS             RESTARTS   AGE
rs-app-mvb6f   0/1     ImagePullBackOff   0          5m14s
rs-app-pjx6q   0/1     ImagePullBackOff   0          5m14s
rs-app-xgx5r   0/1     ImagePullBackOff   0          5m14s
```

8. Remova todos os pods do ReplicaSet `rs-app`. O que acontece a seguir?

▼ *Visualizar resposta*

Podemos deletar os pods um a uma ou utilizar a flag `--all` para fazê-lo em um único passe.

Atente-se para o fato que esse comando irá apagar todos os pods do *namespace* em questão, e não apenas os do ReplicaSet `rs-all`. Veremos uma forma melhor de fazer essa filtragem na sessão 3 deste curso.

```
# kubectl delete pod --all
pod "rs-app-mvb6f" deleted
pod "rs-app-pjx6q" deleted
pod "rs-app-xgx5r" deleted
```

Após um curto tempo, listamos novamente os pods presentes no sistema. E, agora sim, constatamos que os três pods estão operacionais.

```
# kubectl get pod
NAME           READY   STATUS    RESTARTS   AGE
rs-app-56fsb   1/1     Running   0          64s
rs-app-hm8ps   1/1     Running   0          64s
rs-app-prkkt   1/1     Running   0          64s
```

9. Remova um dos três pods do ReplicaSet `rs-app`. E agora, o que ocorre? Porquê?

▼ *Visualizar resposta*

Vamos remover um dos três pods criados no passo anterior.

```
# kubectl delete pod rs-app-56fsb
pod "rs-app-56fsb" deleted
```

Feito isso, listamos os pods... e ainda há três presentes! O objetivo de um ReplicaSet é manter sempre o número de `replicas` solicitado em operação, recriando pods automaticamente se necessário: por isso, após a deleção manual de um pod, um novo foi criado em seu lugar sem qualquer intervenção.

```
# kubectl get pod
NAME           READY   STATUS    RESTARTS   AGE
rs-app-27mc7   1/1     Running   0          59s
rs-app-hm8ps   1/1     Running   0          3m51s
rs-app-prkkt   1/1     Running   0          3m51s
```

3.2) Escalando o número de réplicas

1. Escale o número de réplicas do ReplicaSet `rs-app` para 4 cópias. Verifique o funcionamento de sua configuração.

   ▼ *Visualizar resposta*

   A definição do número de réplicas é feita através do comando `kubectl scale`. Deve-se também informar o tipo de objeto a ser alterado, no caso, um `replicaset` (ou `rs`). O número de cópias é definido através da *flag* `--replicas`, que é obrigatória.

   ```
   # kubectl scale rs --replicas=4 rs-app
   replicaset.apps/rs-app scaled
   ```

   Pesquisando por informações do ReplicaSet, notamos que o número `Desired/Ready` já foi atualizado...

   ```
   # kubectl get rs rs-app
   NAME     DESIRED   CURRENT   READY   AGE
   rs-app   4         4         4       17h
   ```

   Bem como o número de pods em operação:

   ```
   # kubectl get pod
   NAME           READY   STATUS    RESTARTS   AGE
   rs-app-27mc7   1/1     Running   0          16h
   rs-app-9vzpf   1/1     Running   0          14s
   rs-app-hm8ps   1/1     Running   0          16h
   rs-app-prkkt   1/1     Running   0          16h
   ```

2. Agora, reduza o número de réplicas desse ReplicaSet para 2 cópias, e verifique o resultado.

   ▼ *Visualizar resposta*

   Para reduzir o número de réplicas pode-se utilizar o mesmo comando do item anterior, apenas fazendo os ajustes necessários.

   ```
   # kubectl scale rs --replicas=2 rs-app
   replicaset.apps/rs-app scaled
   ```

```
# kubectl get rs rs-app
NAME     DESIRED   CURRENT   READY   AGE
rs-app   2         2         2       17h
```

3.3) Corrigindo problemas com ReplicaSets

1. Antes de iniciar, execute o comando abaixo:

```
# lab-2.3.3
```

2. Considere o arquivo `~/labs/2.3.3/rs-nginx.yaml`. O que ocorre ao tentar criar um ReplicaSet a partir dele? Corrija-o, se necessário, e faça o *deployment* do objeto.

▼ *Visualizar resposta*

Ao tentar criar o ReplicaSet, encontramos um erro:

```
# kubectl create -f ~/labs/2.3.3/rs-nginx.yaml
error: unable to recognize "labs/2.3.3/rs-nginx.yaml": no matches for kind
"ReplicaSet" in version "v1"
```

Observando a primeira linha do arquivo, constatamos que o campo `apiVersion` está incorreto; como documentado em https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/ , deve-se utilizar `apps/v1`, e não `v1`.

```
# head -n1 ~/labs/2.3.3/rs-nginx.yaml
apiVersion: v1
```

Vamos corrigir o arquivo e fazer a criação do objeto.

```
<strong># sed -i 's/^\(apiVersion: \).&#42;/\1apps\/v1/' ~/labs/2.3.3/rs-
nginx.yaml && kubectl create -f ~/labs/2.3.3/rs-nginx.yaml</strong>
replicaset.apps/rs-nginx created
```

Finalmente, vamos verificar seu funcionamento.

```
# kubectl get rs rs-nginx
NAME       DESIRED   CURRENT   READY   AGE
rs-nginx   3         3         3       19s
```

3. Agora, considere o arquivo `~/labs/2.3.3/rs-redis.yaml`. Há algum erro a ser corrigido? Verifique o arquivo e faça a criação do ReplicaSet.

▼ *Visualizar resposta*

Ao tentar criá-lo, constatamos que o seletor `matchLabels` não casa com os `labels` definidos para o *template* do ReplicaSet.

```
# kubectl create -f ~/labs/2.3.3/rs-redis.yaml
The ReplicaSet "rs-redis" is invalid: spec.template.metadata.labels: Invalid
value: map[string]string(nil): `selector` does not match template `labels`
```

Para corrigir isso, basta aplicar o `label` apropriado ao *template* e, então, criar o objeto.

```
<strong># sed -i '/template:/a\    metadata:\n      labels:\n        tier: db'
~/labs/2.3.3/rs-redis.yaml && kubectl create -f ~/labs/2.3.3/rs-
redis.yaml</strong>
replicaset.apps/rs-redis created
```

Finalmente, verificamos sua criação como esperado.

```
# kubectl get rs rs-redis
NAME       DESIRED   CURRENT   READY   AGE
rs-redis   2         2         2       18s
```

4) Deployments

4.1) Trabalhando com deployments

1. Antes de iniciar, execute o comando abaixo:

```
# lab-2.4.1
```

2. Quantos Pods, ReplicaSets e Deployments existem no ambiente? Considere apenas o *namespace* `default`.

▼ *Visualizar resposta*

Podemos passar múltiplos parâmetros para o comando `kubectl get`, como visto abaixo, respondendo todos os questionamentos de uma única vez.

```
# kubectl get pod,rs,deploy
NAME                              READY   STATUS         RESTARTS   AGE
pod/deploy-web-68dccbc89f-9wmdd   0/1     ErrImagePull   0          12s
pod/deploy-web-68dccbc89f-qn6dl   0/1     ErrImagePull   0          12s
pod/deploy-web-68dccbc89f-t2p8d   0/1     ErrImagePull   0          12s

NAME                                    DESIRED   CURRENT   READY   AGE
replicaset.apps/deploy-web-68dccbc89f   3         3         0       12s
```

```
NAME                          READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/deploy-web    0/3     3            0           12s
```

3. Quantos pods estão em estado `READY`?

▼ *Visualizar resposta*

Além da contagem manual de pods a partir da saída do comando anterior, pode-se também utilizar a opção `-o custom-columns` para produzir uma saída mais filtrada:

```
# kubectl get pod -o custom-
columns=NAME:.metadata.name,READY:.status.containerStatuses[*].ready
NAME                           READY
deploy-web-68dccbc89f-9wmdd    false
deploy-web-68dccbc89f-qn6dl    false
deploy-web-68dccbc89f-t2p8d    false
```

4. Porquê? Faça as correções necessárias e garanta que o deployment `deploy-web` está operacional.

▼ *Visualizar resposta*

O erro `ErrImagePull` obtido no item (b) é um bom indicativo. Além disso, podemos observar os eventos de um dos pods do deployment para mais informações:

```
# kubectl describe pod deploy-web-68dccbc89f-9wmdd | tail -n4
  Warning  Failed     43s (x3 over 84s)  kubelet            Failed to pull image
"htttpd:alpine": rpc error: code = Unknown desc = Error response from daemon:
pull access denied for htttpd, repository does not exist or may require 'docker
login': denied: requested access to the resource is denied
  Warning  Failed     43s (x3 over 84s)  kubelet            Error: ErrImagePull
  Normal   BackOff    6s (x5 over 83s)   kubelet            Back-off pulling
image "htttpd:alpine"
  Warning  Failed     6s (x5 over 83s)   kubelet            Error:
ImagePullBackOff
```

Novamente, temos um caso de imagem com nome incorreto — desta vez, há uma letra `t` adicional no nome da imagem `httpd:alpine`. Vamos corrigir o deployment:

```
# kubectl edit deploy deploy-web
deployment.apps/deploy-web edited
```

Feito isso, vamos verificar seu estado:

```
# kubectl get deploy deploy-web
NAME         READY   UP-TO-DATE   AVAILABLE   AGE
deploy-web   3/3     3            3           5m22s
```

Note que, diferentemente do observado em ReplicaSets, os pods do deployment são recriados automaticamente após a efetivação da correção.

```
# kubectl get pod
NAME                          READY   STATUS    RESTARTS   AGE
deploy-web-588ff94457-2fdnl   1/1     Running   0          15s
deploy-web-588ff94457-98m4w   1/1     Running   0          13s
deploy-web-588ff94457-hh4n9   1/1     Running   0          11s
```

4.2) Criando deployments

1. Crie um novo deployment via arquivo YAML com o nome `deploy-nginx`, usando a imagem `nginx:alpine` e com 4 réplicas. Verifique o funcionamento de sua configuração.

   ▼ *Visualizar resposta*

   Vamos lá: primeiro, criamos um arquivo YAML com o conteúdo que se segue.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: deploy-nginx
spec:
  replicas: 4
  selector:
    matchLabels:
      tier: web
  template:
    metadata:
      labels:
        tier: web
    spec:
      containers:
      - name: nginx
        image: nginx:alpine
```

   A seguir, criamos o objetivo via `kubectl create`.

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

   E, finalmente, verificamos o estado do deployment.

```
# kubectl get deploy deploy-nginx
NAME           READY   UP-TO-DATE   AVAILABLE   AGE
deploy-nginx   4/4     4            4           95s
```

5) Serviços

5.1) Serviços padrão do sistema

1. Quantos serviços existem no sistema, no momento?

   ▼ *Visualizar resposta*

   Para descobrir os serviços existentes no ambiente, basta utilizar o comando `kubectl get services` ou `svc`, para encurtar o comando.

   Abaixo, removemos o cabeçalho impresso por padrão na primeira linha da saída com a *flag* `--no-headers`, e usamos o comando `wc` para contar o número de linhas — e, portanto, de serviços.

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

2. Qual é o tipo do serviço padrão `kubernetes`?

   ▼ *Visualizar resposta*

   Essa informação pode ser visualizada no comando acima (removendo o comando `wc` adicional), ou mais especificamente:

   ```
   # kubectl get svc kubernetes -o custom-
   columns=NAME:.metadata.name,TYPE:.spec.type
   NAME         TYPE
   kubernetes   ClusterIP
   ```

3. Qual é a configuração de `port` utilizada para esse serviço? E quanto ao seu `targetPort`?

   ▼ *Visualizar resposta*

   Essas informações são listadas pelo comando `kubectl describe svc`, como visto abaixo.

   ```
   # kubectl describe svc kubernetes | grep '^Port:\|^TargetPort:'
   Port:              https  443/TCP
   TargetPort:        6443/TCP
   ```

4. Quantos `endpoints` estão configurados? Quais seus endereços IP e portas de destino?

   ▼ *Visualizar resposta*

   Assim como no item anterior, o mesmo comando também mostra os `endpoints`:

   ```
   # kubectl describe svc kubernetes | grep '^Endpoints:'
   Endpoints:         192.168.68.20:6443
   ```

5.2) Trabalhando com serviços

1. Antes de iniciar, execute o comando abaixo:

```
# lab-2.5.2
```

2. Quantos deployments existem no sistema neste momento? Qual (ou quais) as imagens de container usadas nesse(s) deployment(s)?

   ▼ *Visualizar resposta*

   Vamos primeiro descobrir o número de deployments.

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

   Sem o comando `wc` ao final, podemos observar também o nome do deployment, `myapp-color`. Para descobrir a imagem utilizada, basta invocar o `kubectl describe`:

```
# kubectl describe deploy myapp-color | grep 'Image:'
    Image:        fbscarel/myapp-color
```

3. Você consegue, a partir de sua máquina física, acessar a interface web da aplicação publicada no deployment? Porquê?

   ▼ *Visualizar resposta*

   Não, pois não há nenhum serviço que disponibilize acesso externo ao deployment `myapp-color`. Vamos corrigir isso.

4. Crie um novo serviço para permitir acesso à aplicação publicada pelo deployment em questão. Utilize a porta 80 em `port`/`targetPort`, e 30080 para o `NodePort` do serviço. Para a configuração de `spec.selector`, utilize o par chave-valor `app: myapp-color`.

   Não se esqueça de testar e validar o funcionamento de sua configuração.

   ▼ *Visualizar resposta*

   Vamos primeiro criar um arquivo YAML definindo o serviço de acordo com os parâmetros especificados no enunciado. Note que é necessário especificar o tipo do serviço, `NodePort` nesse caso.

   Observe ainda que o parâmetro `targetPort` não foi expressamente indicado, e irá herdar o valor de `port`, por padrão.

```
1 apiVersion: v1
2 kind: Service
3 metadata:
4   name: svc-myapp-color
```

```
 5 spec:
 6   selector:
 7     app: myapp-color
 8   ports:
 9     - protocol: TCP
10       port: 80
11       nodePort: 30080
12   type: NodePort
```

Perfeito! Vamos criar o serviço.

```
# kubectl create -f svc-myapp-color.yaml
service/svc-myapp-color created
```

Podemos utilizar o comando `kubectl describe svc` para ver detalhes sobre o mesmo. Há algumas informações interessantes a observar:

```
# kubectl describe svc svc-myapp-color | grep
'Selector\|Port\|TargetPort\|NodePort\|Endpoints'
Selector:                 app=myapp-color
Type:                     NodePort
Port:                     <unset>  80/TCP
TargetPort:               80/TCP
NodePort:                 <unset>  30080/TCP
Endpoints:                10.44.0.1:80,10.44.0.2:80,10.44.0.3:80
```

Veja que há três `endpoints` configurados para o serviço. Quais seriam esses pods? Ora, são as três réplicas criadas para o deployment `myapp-color`, como visto abaixo:

```
# kubectl get pod -o custom-columns=NAME:.metadata.name,IP:.status.podIP
NAME                            IP
myapp-color-54495db468-6prcj    10.44.0.1
myapp-color-54495db468-9hkp4    10.44.0.3
myapp-color-54495db468-cj2vg    10.44.0.2
```

Agora, vamos acessar o serviço externamente. Para isso, precisamos saber o endereço IP dos membros do *cluster*, as máquinas `s2-master-1` e `s2-node-1`. É fácil descobrir essa informação, seja via `kubectl get nodes -o wide` ou via uma filtragem mais específica:

```
# kubectl get node -o custom-columns=IP:.status.addresses[*].address
IP
192.168.68.20,s2-master-1
192.168.68.25,s2-node-1
```

Pode-se utilizar qualquer dos endereços acima, seja a máquina `s2-master-1` ou `s2-node-1`,

independentemente de onde o pod esteja executando. No exemplo, iremos acessar através do endereço do mestre do *cluster*.

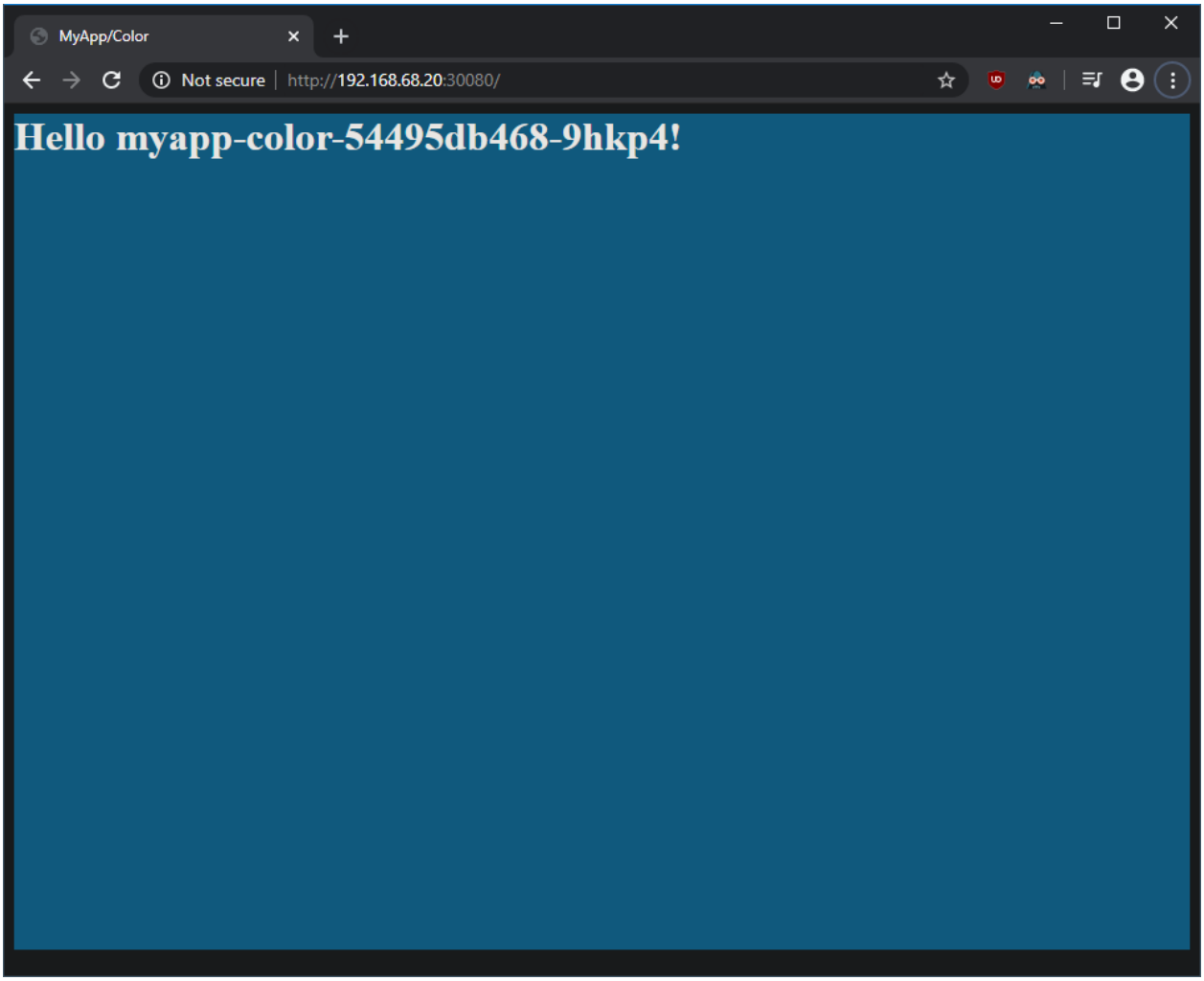

Figura 1. Acesso ao serviço svc-myapp-color realizado com sucesso

6) Namespaces

6.1) Trabalhando com namespaces

1. Antes de iniciar, execute o comando abaixo:

```
# lab-2.6.1
```

2. Quantos namespaces existem no ambiente?

▼ *Visualizar resposta*

Para descobrir os namespaces existentes no ambiente, basta utilizar o comando `kubectl get namespaces` ou `ns`, para encurtar o comando.

```
# kubectl get ns --no-headers | wc -l
9
```

3. Quantos pods existem no namespace `bar`?

   ▼ *Visualizar resposta*

   Já sabemos o comando para listar os pods existentes: `kubectl get pods`. Para aplicá-lo a um namespace específico, basta utilizar a opção `--namespace` ou `-n`.

   ```
   # kubectl -n bar get pod --no-headers | wc -l
   3
   ```

4. Crie um pod com o nome `hamlet` e imagem `httpd:alpine` no namespace `theater`. Verifique o funcionamento de sua configuração.

   ▼ *Visualizar resposta*

   Para criar um pod em um namespace específico, basta especificá-lo na seção `.metadata.namespace`, como visto abaixo.

   ```
   1 apiVersion: v1
   2 kind: Pod
   3 metadata:
   4   name: hamlet
   5   namespace: theater
   6 spec:
   7   containers:
   8   - image: httpd:alpine
   9     name: hamlet
   ```

   A seguir, criamos o objeto com o comando `kubectl create`:

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

   E, finalmente, visualizamos se ele foi criado com sucesso.

   ```
   # kubectl -n theater get pod
   NAME     READY   STATUS    RESTARTS   AGE
   hamlet   1/1     Running   0          5s
   ```

6.2) Comunicação entre objetos

1. Qual namespace contém o pod `zebra`?

   ▼ *Visualizar resposta*

   Ao contrário do que se poderia imaginar, não é possível selecionar um recurso (neste caso, um pod) por nome procurando em todos os namespaces. Veja o erro retornado:

```
# kubectl get pod zebra --all-namespaces
error: a resource cannot be retrieved by name across all namespaces
```

Para tanto, iremos listar todos os pods, em todos os namespaces, e então filtrar pelo nome usando um comando externo (como o `grep`). Para tornar a saída mais precisa, iremos imprimir apenas as colunas relevantes usando a opção `-o custom-columns`.

```
# kubectl get pod --all-namespaces -o custom-
columns=NAME:.metadata.name,NAMESPACE:.metadata.namespace | grep '^zebra '
zebra                                zoo
```

2. Acesse o pod `zebra` interativamente via *shell* `bash`. Qual nome de domínio deve ser utilizado para interagir com o serviço provido pelo pod `lion`, no mesmo namespace? Utilize as ferramentas de linha de comando `nslookup` ou `dig` para validar sua resposta.

   ▼ *Visualizar resposta*

   O enunciado fala sobre "interagir com o serviço provido pelo pod `lion`, no mesmo namespace". Mas, qual é esse serviço? Vamos descobrir seu nome, antes de mais nada:

   ```
   # kubectl -n zoo get svc
   NAME   TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)    AGE
   lion   ClusterIP   10.101.58.19   <none>        6379/TCP   8m9s
   ```

   Perfeito, vamos guardar esse nome. A seguir, precisamos de um *shell* interativo no pod `zebra`; isso pode ser feito através do comando `kubectl exec`, como visto a seguir.

   ```
   # kubectl -n zoo exec -it zebra -- /bin/bash
   root@zebra:/#
   ```

   Como ambos o pod `zebra` e o serviço `lion` encontram-se no mesmo namespace, não é necessário digitar o FQDN (*Fully Qualified Domain Name*) do serviço, mas apenas o nome curto. Iremos testar isso com o comando `nslookup`:

   ```
   root@zebra:/# nslookup lion
   Server:         10.96.0.10
   Address:        10.96.0.10#53

   Name:   lion.zoo.svc.cluster.local
   Address: 10.101.58.19
   ```

   Note que a resolução de nome funcionou, como esperado.

3. Ainda no pod `zebra`, tente interagir com o serviço provido pelo pod `casablanca`. Qual nome de domínio deve ser utilizado nesse caso?

▼ *Visualizar resposta*

A primeira pergunta que surge é: em qual namespace está o pod `casablanca`? Vamos descobrir:

```
# kubectl get pod --all-namespaces -o custom-
columns=NAME:.metadata.name,NAMESPACE:.metadata.namespace | grep '^casablanca '
casablanca                          cinema
```

Ok, o namespace `cinema` é diferente do namespace do pod `zebra`, `zoo`. E qual o nome do serviço provido pelo pod `casablanca`?

```
# kubectl -n cinema get svc
NAME         TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)    AGE
casablanca   ClusterIP   10.106.79.29   <none>        5432/TCP   9m40s
```

Perfeito. Voltando ao *shell* interativo dentro do pod `zebra`, iremos primeiramente tentar resolver o nome do serviço `casablanca` usando seu nome curto:

```
# kubectl -n zoo exec -it zebra -- /bin/bash
root@zebra:/#
```

```
root@zebra:/# nslookup casablanca
Server:         10.96.0.10
Address:        10.96.0.10#53

** server can't find casablanca: NXDOMAIN
```

Como esperado, não funcionou: para acessar pods e serviços em namespaces diferentes, é necessário utilizar seu FQDN ou, ao menos, o nome do namespace após o nome do serviço. Vamos testar.

```
root@zebra:/# nslookup casablanca.cinema
Server:         10.96.0.10
Address:        10.96.0.10#53

Name:   casablanca.cinema.svc.cluster.local
Address: 10.106.79.29
```

Perfeito! Note que além de usar o nome `casablanca.cinema`, poderíamos também utilizar o FQDN `casablanca.cinema.svc.cluster.local`, com igual efeito.

7) Comandos imperativos

Utilize **APENAS** comandos imperativos (p.ex. `kubectl run` ou `kubectl create`) em **TODAS** as atividades desta seção. Não utilize arquivos YAML na criação dos objetos.

7.1) Comandos imperativos básicos

1. Antes de iniciar, execute o comando abaixo:

```
# lab-2.7.1
```

2. Crie um pod com o nome `squirtle` e usando a imagem `nginx:alpine`.

▼ *Visualizar resposta*

Para tanto, basta executar o comando `kubectl run` como visto abaixo:

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

Verifique que o pod está, de fato, operacional.

```
# kubectl get pod squirtle
NAME       READY   STATUS    RESTARTS   AGE
squirtle   1/1     Running   0          5s
```

3. Crie um pod com o nome `bulbasaur` usando a imagem `postgres:9.6-alpine` e o *label* `type=pokemon`.

▼ *Visualizar resposta*

Novamente, usamos o comando `kubectl run`. Para definir *labels* adicionais (que serão vistos em maior detalhe na sessão 3 deste curso), utilize a opção `--labels` ou `-l`:

```
# kubectl run bulbasaur --image postgres:9.6-alpine -l type=pokemon
pod/bulbasaur created
```

Vamos checar se os *labels* foram aplicados corretamente.

```
# kubectl describe pod bulbasaur | grep '^Labels:'
Labels:       type=pokemon
```

4. Crie um serviço `bulbasaur-svc` que exponha o pod `bulbasaur` na porta 5432, apenas no contexto do *cluster*.

▼ *Visualizar resposta*

O comando `kubectl create svc` pode ser usado para criar serviços, como visto abaixo. A sintaxe da opção `--tcp` é `<port>:<targetPort>`, mas o segundo pode ser omitido caso ambos coincidam.

```
# kubectl create svc clusterip bulbasaur-svc --tcp=5432
service/bulbasaur-svc created
```

Vejamos se o serviço foi de fato criado.

```
# kubectl get svc bulbasaur-svc
NAME            TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)    AGE
bulbasaur-svc   ClusterIP   10.110.252.21   <none>        5432/TCP   8s
```

7.2) Comandos imperativos avançados

1. Crie um namespace com o nome `fire`.

▼ *Visualizar resposta*

Novamente, utiliza-se o comando `kubectl create`:

```
# kubectl create ns fire
namespace/fire created
```

```
# kubectl get ns fire
NAME   STATUS   AGE
fire   Active   83s
```

2. Crie dentro do namespace `fire` um deployment com o nome `charmander` e imagem `fbscarel/myapp-redis`, com três réplicas.

A seguir, crie um serviço do tipo `NodePort` e nome `charmander` que faça o mapeamento da porta 31080 no *node* para a porta 80 dos pods.

▼ *Visualizar resposta*

Primeiro, vamos criar o deployment com a imagem e número de réplicas especificado.

```
# kubectl -n fire create deploy charmander --image=fbscarel/myapp-redis
--replicas=3
deployment.apps/charmander created
```

Vejamos se ele está operacional:

```
# kubectl -n fire get deploy charmander
NAME         READY   UP-TO-DATE   AVAILABLE   AGE
charmander   3/3     3            3           5m45s
```

Perfeito. Agora, para a exposição do serviço — podemos usar o comando `kubectl create svc`, agora com o tipo `nodeport`. Para especificar o `NodePort` a ser utilizado a opção a ser passada é... isso mesmo, `--node-port`.

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

Vamos revisar as configurações desse serviço. Note que as três réplicas do deployment criado no passo anterior foram capturadas automaticamente, já que o deployment e o serviço possuem o mesmo nome.

```
# kubectl -n fire describe svc charmander
Name:                     charmander
Namespace:                fire
Labels:                   app=charmander
Annotations:              <none>
Selector:                 app=charmander
Type:                     NodePort
IP Family Policy:         SingleStack
IP Families:              IPv4
IP:                       10.99.1.175
IPs:                      10.99.1.175
Port:                     80  80/TCP
TargetPort:               80/TCP
NodePort:                 80  31080/TCP
Endpoints:                10.44.0.3:80,10.44.0.4:80,10.44.0.5:80
Session Affinity:         None
External Traffic Policy:  Cluster
Events:                   <none>
```

3. Crie dentro do namespace `fire` um pod com o nome `db` e imagem `redis:alpine`. Adicionalmente, exponha a porta 6379 desse pod dentro do contexto do *cluster*. Utilize apenas um comando para cumprir ambos os objetivos.

▼ *Visualizar resposta*

Vamos fazer tudo de uma vez! O segredo, nesse caso, é usar as opções `--port` e `--expose` para selecionar e expor a porta do pod. O tipo do serviço criado é o padrão, `ClusterIP`.

```
# kubectl -n fire run db --image=redis:alpine --port=6379 --expose
service/db created
pod/db created
```

Vamos conferir se o pod está operacional...

```
# kubectl -n fire get pod db
NAME   READY   STATUS    RESTARTS   AGE
db     1/1     Running   0          17s
```

Bem como o serviço:

```
# kubectl -n fire get svc db
NAME   TYPE        CLUSTER-IP    EXTERNAL-IP   PORT(S)    AGE
db     ClusterIP   10.103.81.2   <none>        6379/TCP   30s
```

4. Teste o acesso externo ao serviço `charmander`, conectando-se via navegador a partir de sua máquina física no endereço de um dos *nodes* do *cluster* na porta 31080.

▼ *Visualizar resposta*

No teste abaixo utilizamos o endereço IP do *host* `s2-node-1`, obtendo sucesso no acesso.

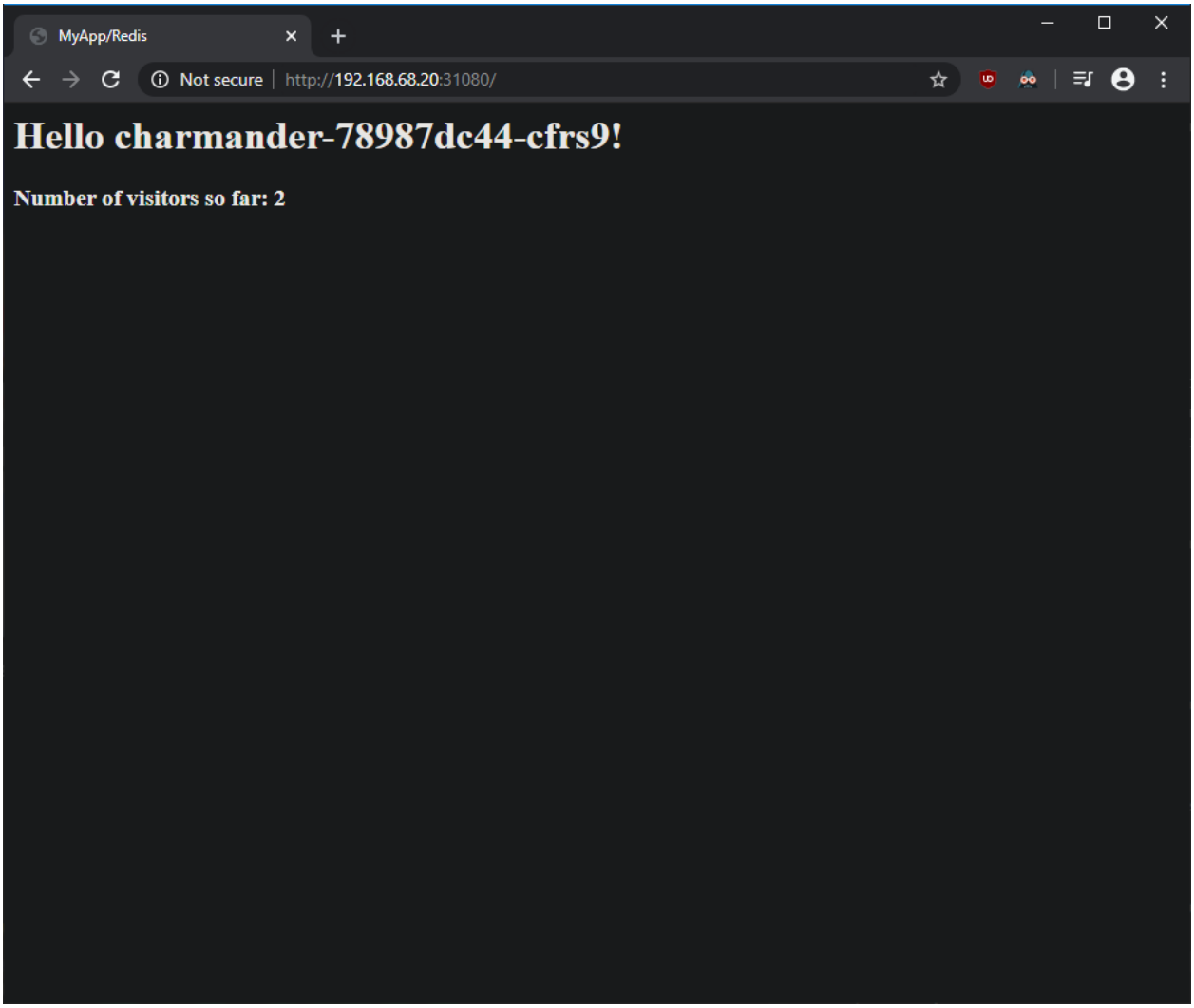

Figura 2. Publicação do serviço charmander realizado com sucesso

ENTREGA DA TAREFA

> Para que seja considerada entregue você deve anexar a esta atividade no AVA uma imagem (nos formatos .png ou .jpg) do seu navegador acessando a aplicação `myapp-color`, exposta na porta `30080` na cor azul.
>
> Utilize como referência a imagem mostrada na atividade 2.5.2 (d) deste roteiro.