Índice

Sessão 1: Introdução à conteinerização

1) Criação de máquina virtual no Virtualbox

Instale o Git, Vagrant e Virtualbox em sua máquina. Feito isso, abra o `cmd` ou `Windows PowerShell`, crie um diretório temporário de trabalho e execute os comandos abaixo. Note que o *prompt* `C:\>` mostrado é meramente ilustrativo.

```
C:\> git clone https://github.com/fbscarel/contorq-files.git
```

```
C:\> cd contorq-files\s1
```

```
C:\contorq-files\s1> vagrant up
Bringing machine 'default' up with 'virtualbox' provider...
==> default: Importing base box 'bento/debian-11'...
==> default: Matching MAC address for NAT networking...
==> default: Setting the name of the VM: docker

(...)
```

Podem demorar alguns minutos até que o prompt retorne a seu controle, seja paciente. Após a conclusão da configuração da VM, faça login como o usuário `vagrant`:

```
C:\contorq-files\s1> vagrant ssh
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@docker:~$
```

```
vagrant@docker:~$ hostname
docker
```

```
vagrant@docker:~$ whoami
vagrant
```

2) Instalação do Docker

```
vagrant@docker:~$ sudo -i
```

```
root@docker:~# whoami
root
```

A partir de agora iremos 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`).

Caso o diretório corrente seja relevante para a questão em curso, o comando `pwd` será utilizado para mostrar o caminho indicado na hierarquia de diretórios do sistema.

Vamos começar atualizando a lista de pacotes disponíveis para instalação.

```
# apt update
```

A seguir, instale os pré-requisitos do Docker.

```
# apt install -y \
                 ca-certificates     \
                 curl                \
                 gnupg               \
                 lsb-release
```

Depois, adicione o repositório oficial:

```
# curl -fsSL https://download.docker.com/linux/debian/gpg | sudo gpg --dearmor -o
/usr/share/keyrings/docker-archive-keyring.gpg
```

```
# echo \
  "deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyrings/docker-
archive-keyring.gpg] https://download.docker.com/linux/debian \
  $(lsb_release -cs) stable" | sudo tee /etc/apt/sources.list.d/docker.list >
/dev/null
```

Finalmente, vamos instalar o Docker propriamente dito.

```
# apt update
```

```
# apt install -y containerd.io=1.5.10-1                                  \
                 docker-ce=5:20.10.13~3-0~debian-$(lsb_release -cs)     \
                 docker-ce-cli=5:20.10.13~3-0~debian-$(lsb_release -cs)
```

```
<strong># cat << EOF >> /etc/docker/daemon.json
{
  "exec-opts": ["native.cgroupdriver=systemd"],
  "log-driver": "json-file",
  "log-opts": {
    "max-size": "100m"
  },
  "storage-driver": "overlay2"
}
EOF</strong>
```

```
# mkdir -p /etc/systemd/system/docker.service.d
```

```
<strong># systemctl daemon-reload            &&
            systemctl restart docker &&
            systemctl enable docker</strong>
```

Tudo pronto! Vamos ver se tudo está funcionando a contento:

```
# docker run hello-world

Unable to find image 'hello-world:latest' locally
```

```
latest: Pulling from library/hello-world
2db29710123e: Pull complete
Digest: sha256:6d60b42fdd5a0aa8a718b5f2eab139868bb4fa9a03c9fe1a59ed4946317c4318
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://hub.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/get-started/
```

3) Comandos básicos do Docker

3.1) Determinando a versão do daemon Docker

1. Utilizando o utilitário de linha de comando `docker`, determine a versão do *daemon* Docker em execução no sistema.

   ▼ *Visualizar resposta*

   ```
   # docker --version
   Docker version 20.10.13, build a224086
   ```

3.2) Determinando o número de containers em execução

1. Utilizando o utilitário `docker`, determine quantos containers estão em execução no sistema neste exato momento.

   ▼ *Visualizar resposta*

   ```
   # docker ps
   CONTAINER ID        IMAGE               COMMAND             CREATED
   STATUS              PORTS               NAMES
   ```

Pode-se também utilizar a ferramenta `wc` em conjunto com a opção `-q` (ou `--quiet`) do `docker` para mostrar apenas os identificadores numéricos dos containers, e assim contá-los de forma facilitada.

```
# docker ps -q | wc -l
0
```

3.3) Determinando o número de imagens no sistema

1. Usando o `docker`, determine quantas imagens de container estão presentes no sistema até o presente momento.

▼ *Visualizar resposta*

```
# docker images
REPOSITORY    TAG       IMAGE ID       CREATED        SIZE
hello-world   latest    feb5d9fea6a5   5 months ago   13.3kB
```

Assim como na atividade anterior, pode-se utilizar o `wc` para facilitar a contagem.

```
# docker images -q | wc -l
1
```

3.4) Executando um container de forma interativa

1. Execute um container com a imagem `nginx` usando o utilitário `docker`. Após sua execução, determine se o terminal encontra-se utilizável; caso negativo, pare o container para retomar o controle da sessão.

▼ *Visualizar resposta*

```
# docker run nginx
Unable to find image 'nginx:latest' locally
latest: Pulling from library/nginx

(...)

/docker-entrypoint.sh: Configuration complete; ready for start up
```

Após a execução do container, o terminal fica ocupado pois ele está operando em *foreground*. Para parar o container e retomar o controle da sessão, basta entrar com a combinação de teclas `CTRL + C`.

```
^C
root@docker:~#
```

3.5) Visualizando o estado de containers

1. Usando o comando `docker`, determine quantos containers estão em execução no momento no sistema (i.e., cujo estado é igual a `running`).

   ▼ *Visualizar resposta*

   ```
   # docker ps -q | wc -l
   0
   ```

2. Em seguida, execute o comando abaixo:

   ```
   # lab-1.3.5
   ```

   E agora? Quantos containers estão em execução?

   ▼ *Visualizar resposta*

   ```
   # docker ps -q | wc -l
   3
   ```

   Podemos ver detalhes sobre esses containers em execução omitindo a *flag* `-q`, como visto abaixo.

   ```
   # docker ps
   CONTAINER ID        IMAGE               COMMAND                  CREATED
   STATUS              PORTS               NAMES
   23468c73a6a0        nginx:alpine        "/docker-entrypoint.…"   14 seconds ago
   Up 12 seconds       80/tcp              web-3
   1ac90f8a3a01        nginx               "/docker-entrypoint.…"   14 seconds ago
   Up 13 seconds       80/tcp              web-2
   f88a62d2677f        nginx:alpine        "/docker-entrypoint.…"   14 seconds ago
   Up 13 seconds       80/tcp              web-1
   ```

3. Se considerarmos também os containers que já foram encerrados, quantos containers temos no total no sistema?

   ▼ *Visualizar resposta*

   ```
   # docker ps -aq | wc -l
   5
   ```

   Note que dois desses containers já foram encerrados (i.e. seu estado é `exited`), como visto na saída do comando abaixo. A *flag* `format` é utilizada para selecionar apenas os campos relevantes para a questão.

```
# docker ps -a --format "table {{.ID}}\t{{.Status}}"
CONTAINER ID        STATUS
23468c73a6a0        Up 36 seconds
1ac90f8a3a01        Up 36 seconds
f88a62d2677f        Up 37 seconds
b16dd2206755        Exited (0) 38 minutes ago
e9b14ff13624        Exited (0) About an hour ago
```

3.6) Visualizando imagens em uso

1. Como visto na atividade anterior, o comando `lab-1.3.5` criou três containers que encontram-se executando no *daemon* Docker. Qual é a imagem utilizada pelo container `web-3`?

   ▼ *Visualizar resposta*

   Observe que é possível utilizar a *flag* `filter` para selecionar certos elementos da saída do comando. No caso, utilizaremos o campo `name` para filtrar pelo nome do container.

   ```
   # docker ps -a --filter name=web-3 --format "table {{.Names}}\t{{.Image}}"
   NAMES               IMAGE
   web-3               nginx:alpine
   ```

2. Dos containers ativos no sistema (i.e. aqueles em estado `running`), qual — ou quais — deles utiliza a imagem `nginx`?

   ▼ *Visualizar resposta*

   ```
   # docker ps --format "table {{.Image}}\t{{.Names}}" | grep "^nginx "
   nginx               web-2
   ```

3. Qual o identificador numérico (*Container ID*) do container que utiliza a imagem `nginx` e está em estado `exited`?

   ▼ *Visualizar resposta*

   Também é possível aplicar filtros buscando por determinados estados de containers, como visto a seguir.

   ```
   # docker ps -a --filter status=exited --format "table {{.Image}}\t{{.ID}}" | grep
   "^nginx "
   nginx               b16dd2206755
   ```

3.7) Removendo elementos

1. Utilizando o utilitário `docker`, remova todos os containers do sistema (mesmo aqueles em estado `running`). Se necessário, invoque a parada dos containers antes de prosseguir com a remoção.

   ▼ *Visualizar resposta*

Ao tentar remover containers em execução encontramos um erro, como visto abaixo.

```
# docker rm 23468c73a6a0
Error response from daemon: You cannot remove a running container
23468c73a6a0ca579b15d0875e9773881ce29feaa927837a1a62c840511c4c81. Stop the
container before attempting removal or force remove
```

Neste caso, é primeiramente necessário parar os containers antes de removê-los (alternativamente, poderia-se utilizar a *flag* `-f`, ou `--force`).

```
# docker stop $( docker ps -q )
23468c73a6a0
1ac90f8a3a01
f88a62d2677f
```

Agora sim podemos remover os containers — além do método utilizado no comando anterior, e também possível usar o utilitário `xargs` para acelerar o processo:

```
# docker ps -aq | xargs docker rm
23468c73a6a0
1ac90f8a3a01
f88a62d2677f
b16dd2206755
e9b14ff13624
```

2. A seguir, remova a imagem `nginx:alpine`.

▼ *Visualizar resposta*

```
# docker image rm nginx:alpine
Untagged: nginx:alpine
Untagged:
nginx@sha256:a97eb9ecc708c8aa715ccfb5e9338f5456e4b65575daf304f108301f3b497314
Deleted: sha256:6f715d38cfe0eb66b672219f3466f471dda7395b7b7e80e79394508d0dccb5ef
Deleted: sha256:8ece31675e9808e3bf2f8eea8672b606cdef9bea7fd45f1663d6641b75fc4be2
Deleted: sha256:3b5e45e233591dd9ff90f920887bf6e097fb0ced3545072f5a15e43afeed8adf
Deleted: sha256:044b7060c689c20cb7803843978b38c565c0c8f2704452e9e367576b84dba3a4
Deleted: sha256:8dafd8d8363fd03a8b08d44b1a564600d5049ade3fc5fc2c0e146a4b313f80ea
Deleted: sha256:50644c29ef5a27c9a40c393a73ece2479de78325cae7d762ef3cdc19bf42dd0a
```

3.8) Executando containers em background

1. Faça o download da imagem deletada no passo anterior: `nginx:alpine`. Note que apenas o download deve ser feito, e nenhum container deve ser iniciado.

▼ *Visualizar resposta*

```
# docker image pull nginx:alpine
alpine: Pulling from library/nginx
df20fa9351a1: Pull complete
3db268b1fe8f: Pull complete
f682f0660e7a: Pull complete
7eb0e8838bc0: Pull complete
e8bf1226cc17: Pull complete
Digest: sha256:a97eb9ecc708c8aa715ccfb5e9338f5456e4b65575daf304f108301f3b497314
Status: Downloaded newer image for nginx:alpine
docker.io/library/nginx:alpine
```

2. Execute um container não-interativo com o nome `myapp` utilizando a imagem baixada no passo anterior.

▼ *Visualizar resposta*

```
# docker run -d --name myapp nginx:alpine
7f28dbec22c35a12904e0311196cd2266683f3199f412c05048256c03a2f9612
```

3. Finalmente, remova todas as imagens e containers existentes no sistema.

▼ *Visualizar resposta*

```
# docker rm -f $( docker ps -aq )
7f28dbec22c3
```

```
# docker image rm -f $( docker images -aq )
Untagged: nginx:latest
Untagged:
nginx@sha256:c628b67d21744fce822d22fdcc0389f6bd763daac23a6b77147d0712ea7102d0

(...)

Deleted: sha256:bf756fb1ae65adf866bd8c456593cd24beb6a0a061dedf42b26a993176745f6b
Deleted: sha256:9c27e219663c25e0f28493790cc0b88bc973ba3b1686355f221c38a36978ac63
```

```
# docker images -aq | wc -l ; docker ps -aq | wc -l
0
0
```

4) Comandos docker run

4.1) Determinando informações sobre um container em execução

1. Antes de iniciar, execute o comando abaixo:

```
# lab-1.4.1
```

Quantos containers estão em execução no sistema?

▼ *Visualizar resposta*

```
# docker ps -qf status=running | wc -l
1
```

É interessante obter também o nome desse container, se disponível.

```
# docker ps --format "table {{.ID}}\t{{.Names}}"
CONTAINER ID        NAMES
e69469798761        portal
```

2. Qual é a imagem usada por esse(s) container(s)?

▼ *Visualizar resposta*

```
# docker inspect portal -f '{{.Config.Image}}'
nginx:alpine
```

3. Quantas portas estão sendo publicadas por esse(s) container(s)?

▼ *Visualizar resposta*

Pode-se obter essa informação de algumas formas. A mais fácil é simplesmente executar o comando `docker ps -f name=CONTAINER` e contar o número de portas publicadas na coluna `PORTS`:

```
# docker ps -f name=portal
CONTAINER ID   IMAGE          COMMAND                  CREATED          STATUS
PORTS
NAMES
297b819cbb16   nginx:alpine   "/docker-entrypoint.…"   27 seconds ago   Up 25
seconds   0.0.0.0:30080->80/tcp, :::30080->80/tcp, 0.0.0.0:33306->3306/tcp,
:::33306->3306/tcp   portal
```

Como visto, são duas portas publicadas. Alternativamente, pode-se usar o comando `docker inspect`:

```
# docker inspect portal -f '{{.NetworkSettings.Ports}}'
map[3306/tcp:[{0.0.0.0 33306} {:: 33306}] 80/tcp:[{0.0.0.0 30080} {:: 30080}]]
```

4. Qual dessa(s) portas é exposta no container? E qual dessa(s) no *host* hospedeiro?

▼ *Visualizar resposta*

Segundo a documentação do comando `docker run` (https://docs.docker.com/engine/reference/commandline/run/), a sintaxe a ser informada para portas publicadas por containers segue o formato `HOST_IP:HOST_PORT:CONTAINER_PORT`.

No caso, temos portanto que o endereço IP `0.0.0.0` (ou seu equivalente em IPv6, `::`) está sendo exposto em ambas as ocasiões (i.e., todos os endereços IP do *host*), portas `30080/TCP` e `33306/TCP` no *host* hospedeiro, e portas `80/TCP` e `3306/TCP` no container, respectivamente.

4.2) Expondo portas de containers

1. Execute uma instância da imagem `fbscarel/myapp-color` em *background*, mapeando a porta `80/TCP` do container para a porta `31080` do *host*. Teste o funcionamento do container em seu navegador web.

▼ *Visualizar resposta*

Vamos primeiramente executar o container como solicitado:

```
# docker run -d -p 31080:80 fbscarel/myapp-color
Unable to find image 'fbscarel/myapp-color:latest' locally
latest: Pulling from fbscarel/myapp-color

(...)

Status: Downloaded newer image for fbscarel/myapp-color:latest
10241d5aa3bf156f2f0b80f7398bd382ed25349b67a7d56db07a8bd35455e028
```

O próximo passo é determinar o endereço IP no qual devemos nos conectar usando o navegador.

```
# hostname -I
10.0.2.15 192.168.68.10 172.17.0.1
```

Dos três endereços IP informados, o primeiro refere-se à interface NAT criada automaticamente pelo Vagrant quando do provisionamento da máquina e o terceiro consiste no endereço da *bridge* criada pelo Docker para permitir conectividade dos containers. Assim, o segundo endereço `192.168.68.10` é o que nos interessa.

Informamos o endereço IP juntamente com a porta mapeada no *host* (`31080`) no navegador instalado na máquina física, visualizando a página publicada.

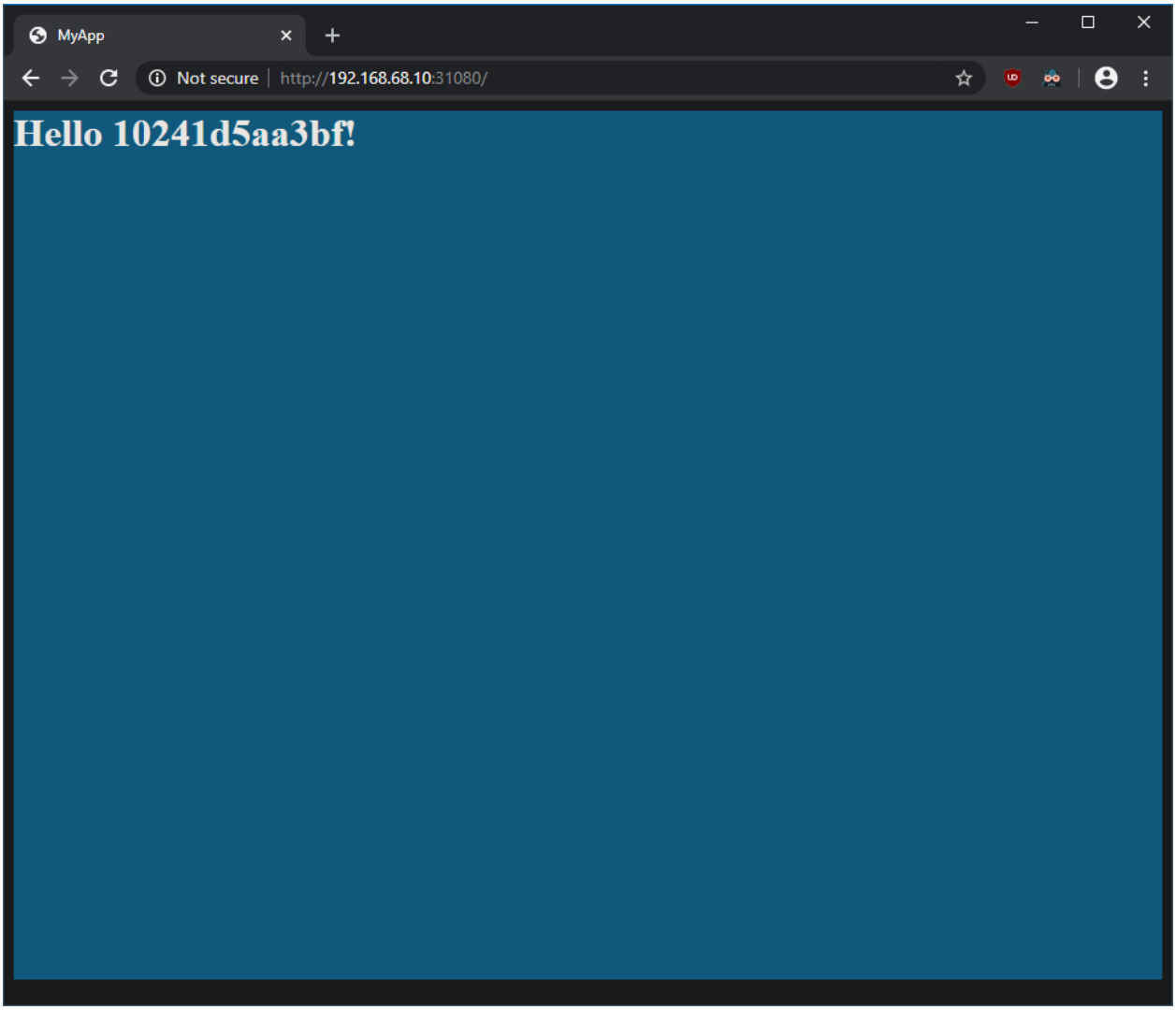

Figura 1. Container myapp-color publicado com sucesso

5) Variáveis de ambiente

5.1) Inspecionando variáveis de ambiente

1. Antes de iniciar, execute o comando abaixo:

```
# lab-1.5.1
```

Inspecione o valor das variáveis de ambiente configuradas para o container `myapp-1`. Qual o valor da variável `COLOR`?

▼ *Visualizar resposta*

Pode-se obter a informação solicitada de algumas formas — podemos, por exemplo, executar um comando interativo como `env` dentro do container para visualizar suas variáveis de ambiente:

```
# docker exec myapp-1 env | grep COLOR
COLOR=purple
```

Também é possível inspecionar informações de baixo nível sobre o container usando o comando `docker inspect`, buscando pela variável informada.

```
# docker inspect myapp-1 | grep COLOR
                "COLOR=purple",
```

5.2) Configurando variáveis de ambiente

1. Execute uma instância da imagem `fbscarel/myapp-color` em *background*, com o nome `myapp-red`, mapeando a porta `80/TCP` do container para a porta `32080` do *host*. Configure a variável de ambiente `COLOR` com o valor `red`. Finalmente, teste o funcionamento da configuração em seu navegador web.

   ▼ *Visualizar resposta*

   Primeiro, execute o container:

   ```
   # docker run -d --name myapp-red -p 32080:80 -e COLOR=red fbscarel/myapp-color
   632fab2582b63272a87e65c1a1614f7ff2e1f7392bbd47e351e1823d554a5fa7
   ```

   A seguir, acesse o *host* utilizando a porta informada para verificar o funcionamento da configuração:

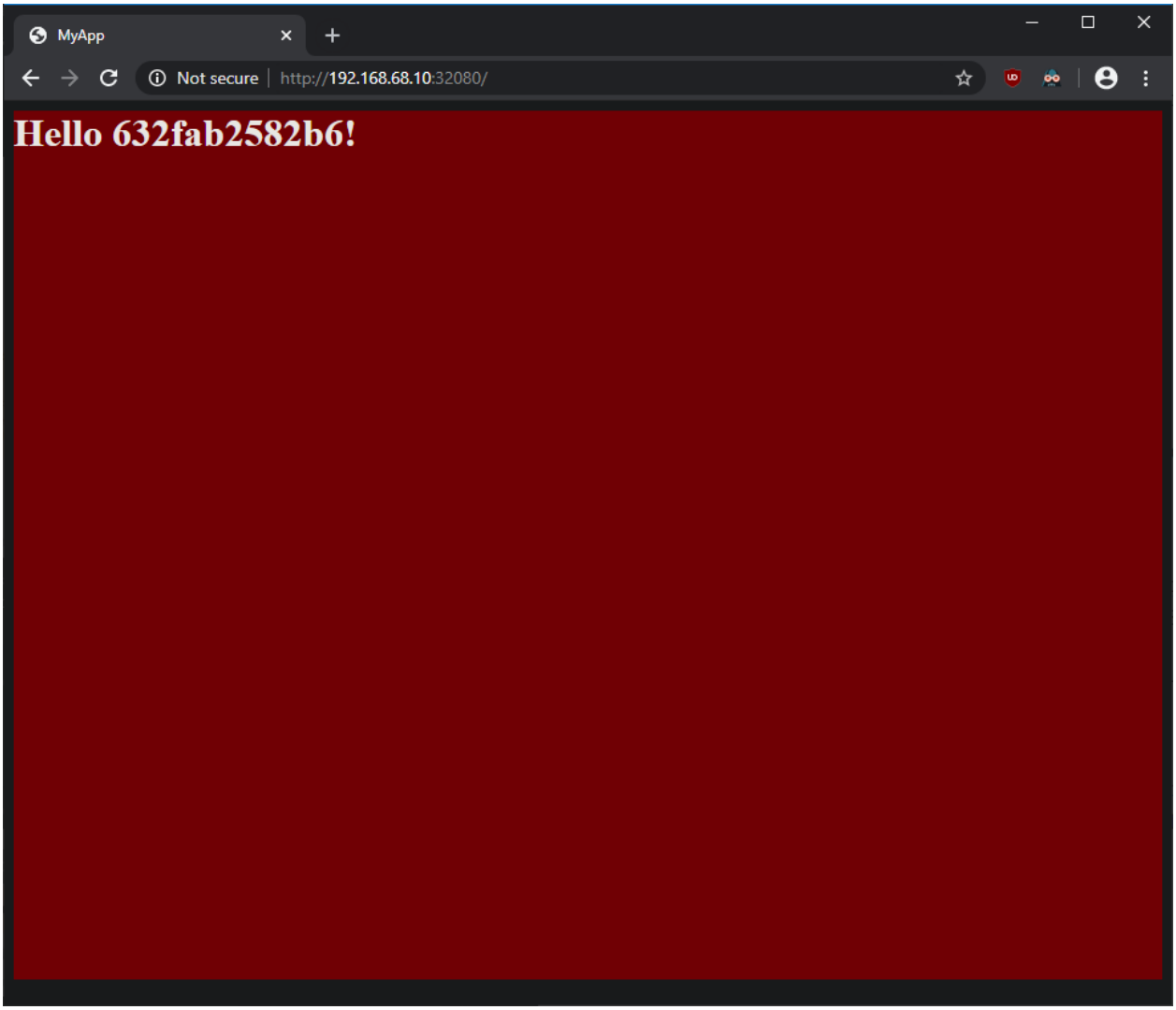

Figura 2. Variável de ambiente configurada com sucesso

2. Faça o *deployment* de uma base de dados MySQL em *background* usando a imagem de container `mysql`, com o nome `db`. Ajuste a senha do usuário `root` da base de dados para `qwerty`. Caso tenha dúvidas em como fazê-lo, consulte a página da imagem `mysql` no Docker Hub e verifique qual variável de ambiente deve ser usada nesse cenário.

   ▼ *Visualizar resposta*

   Como visto na documentação da imagem `mysql` no Docker Hub, a variável a ser utilizada para definir a senha do superusuário `root` é `MYSQL_ROOT_PASSWORD`.

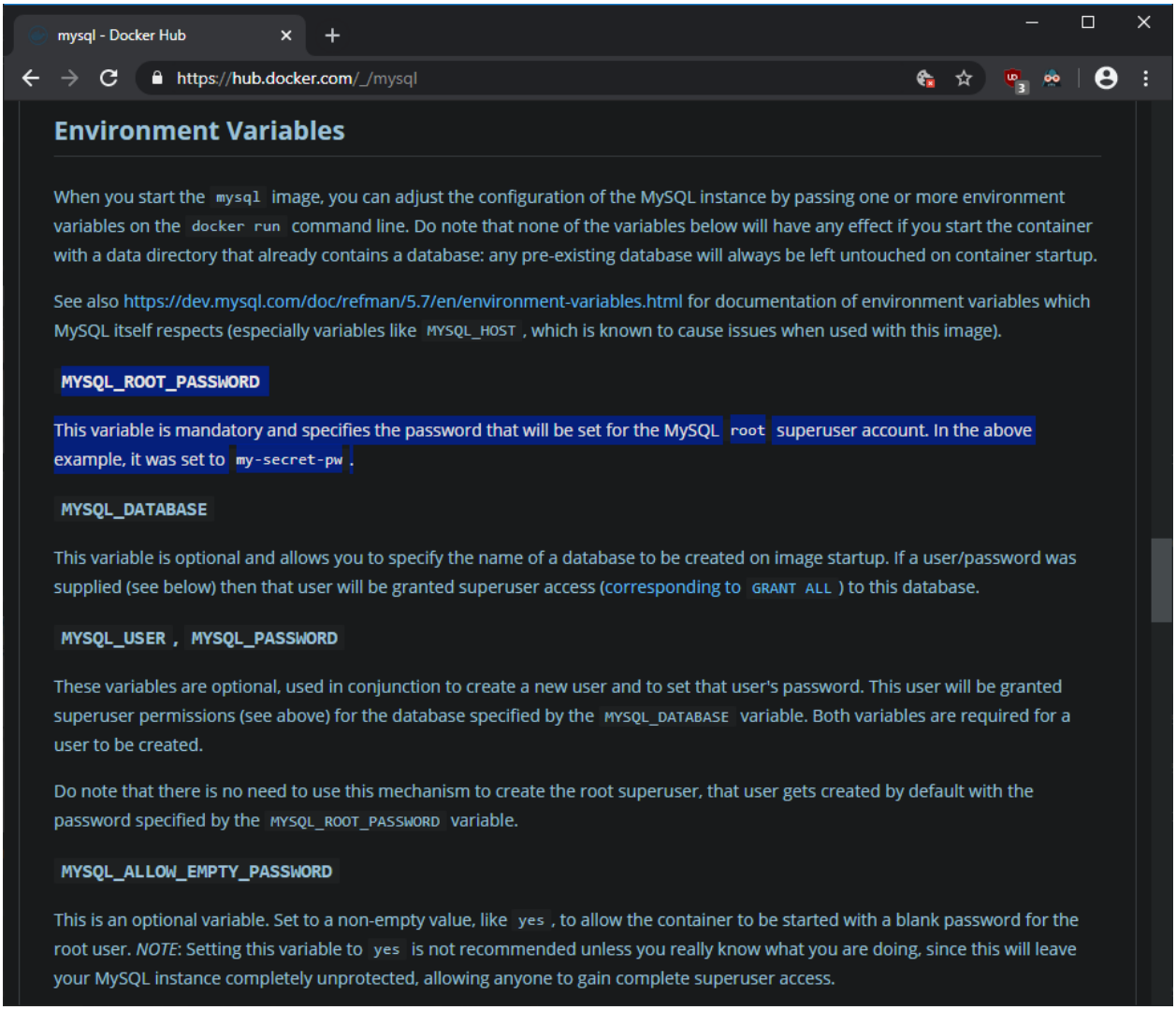

Figura 3. Documentação da imagem mysql no Docker Hub

Sabendo disso, basta invocar a execução do container com o comando:

```
# docker run -d --name db -e MYSQL_ROOT_PASSWORD=qwerty mysql
Unable to find image 'mysql:latest' locally
latest: Pulling from library/mysql

(...)

Status: Downloaded newer image for mysql:latest
8212503b3b9119d01696772b663f04b19593d72fb80b14a439117525e957f56e
```

6) Imagens Docker

6.1) Trabalhando com imagens e tags

1. Antes de iniciar, execute o comando abaixo:

```
# lab-1.6.1
```

Quantas imagens estão disponíveis no *host* local?

▼ *Visualizar resposta*

```
# docker images -q | wc -l
3
```

2. Qual o tamanho da imagem `debian`?

▼ *Visualizar resposta*

```
# docker images debian --format 'table {{.Repository}}\t{{.Size}}'
REPOSITORY          SIZE
debian              124MB
```

3. Qual a *tag* aplicada à imagem `nginx`?

▼ *Visualizar resposta*

```
# docker images nginx --format 'table {{.Repository}}\t{{.Tag}}'
REPOSITORY          TAG
nginx               alpine
```

6.2) Inspecionando Dockerfiles

1. Antes de iniciar, execute o comando abaixo:

```
# lab-1.6.2
```

Inspecione o arquivo `~/myapp-color/Dockerfile` e responda: qual é a imagem-base usada?

▼ *Visualizar resposta*

```
# grep FROM myapp-color/Dockerfile
FROM python:3.6
```

2. O código da aplicação é copiado para qual diretório dentro do container?

▼ *Visualizar resposta*

```
# cat myapp-color/Dockerfile | grep COPY
COPY . /opt/
```

3. Ao criar um container usando a imagem criada a partir desse `Dockerfile`, qual é o comando utilizado para executar a aplicação? Dentro de qual diretório esse comando é executado?

▼ *Visualizar resposta*

```
# cat myapp-color/Dockerfile | grep ENTRYPOINT
ENTRYPOINT ["python", "app.py"]
```

Como visto, o comando executado é `python app.py`. Esse comando é executando dentro de `WORKDIR`, como visto a seguir.

```
# cat myapp-color/Dockerfile | grep WORKDIR
WORKDIR /opt
```

4. Em qual porta a aplicação escutará, dentro do container criado?

▼ *Visualizar resposta*

```
# cat myapp-color/Dockerfile | grep EXPOSE
EXPOSE 80
```

6.3) Criando imagens

1. Crie uma imagem usando o `Dockerfile` estudado na atividade anterior. Utilize o nome `myapp-color`, sem qualquer *tag* especificada.

▼ *Visualizar resposta*

```
# cd ~/myapp-color/ ; docker build --tag myapp-color .
Sending build context to Docker daemon  80.38kB
Step 1/6 : FROM python:3.6
3.6: Pulling from library/python

(...)

Successfully built 4e3f9d11b523
Successfully tagged myapp-color:latest
```

Note que ao não especificarmos uma *tag* para a imagem produzida, ela é automaticamente marcada como `latest`. Isso também ocorre para imagens baixadas do Docker Hub — caso não seja especificada uma *tag*, a `latest` será utilizada.

Em geral, não recomenda-se o uso da *tag* `latest` em Dockerfiles e *builds*, por diversos razões. Este link (https://vsupalov.com/docker-latest-tag/) aponta diversos desses motivos em detalhe. Adicionalmente, a documentação do Kubernetes (https://kubernetes.io/docs/concepts/configuration/overview/#container-images) também desencoraja o uso da *tag* `latest` no *deployment* de imagens de container.

2. Execute uma instância da imagem `myapp-color` recém-criada, mapeando a porta 33080 do *host*

para a porta 80 do container. Acesse a aplicação em seu navegador e verifique o correto funcionamento de sua configuração.

▼ *Visualizar resposta*

```
# docker run -d -p 33080:80 myapp-color
d9366b093c35e51e13c8895310f17263d9d9bc011af04eac9478c8db330b35b8
```

Acessando o navegador na porta especificada, constatamos que a aplicação está de fato operacional:

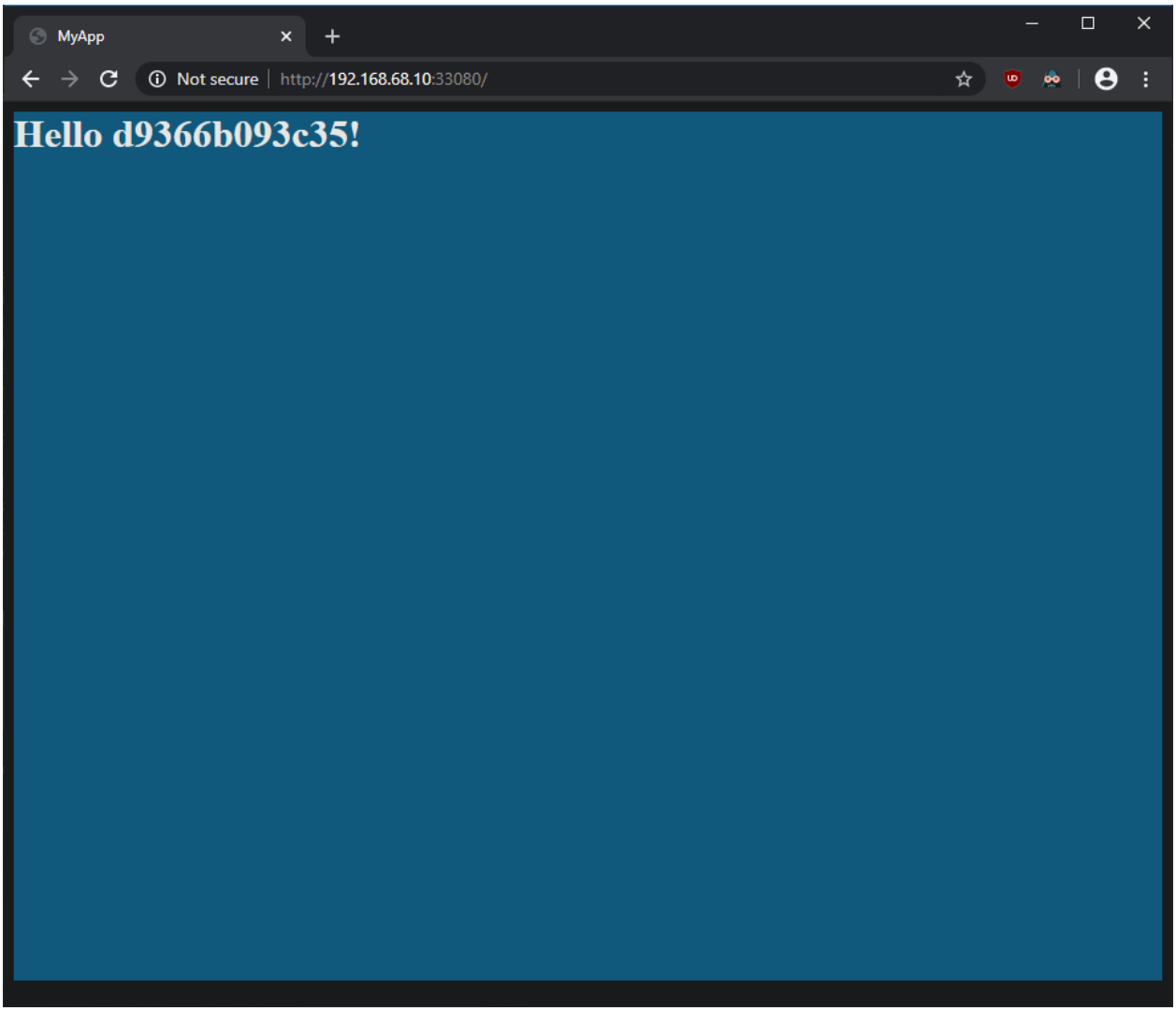

Figura 4. Imagem criada e container iniciado com sucesso

3. Qual é o sistema operacional utilizado como base para criação da imagem `python:3.6`?

▼ *Visualizar resposta*

```
# docker run python:3.6 grep PRETTY_NAME /etc/*release*
PRETTY_NAME="Debian GNU/Linux 11 (bullseye)"
```

4. Qual o tamanho da imagem `myapp-color`?

▼ *Visualizar resposta*

```
# docker images myapp-color --format '{{.Repository}}\t{{.Size}}'
myapp-color     913MB
```

5. Seria interessante conseguir reduzir o tamanho dessa imagem: pesquise o repositório `python`, versão 3.6, no Docker Hub e tente encontrar uma imagem-base mais enxuta. Alternativamente, use o comando `docker search`.

   Ao encontrar, edite o `Dockerfile` da aplicação para usar essa imagem-base. A seguir, faça o *build* de uma nova imagem com o mesmo nome da anterior, com a *tag* `small` aplicada.

   ▼ *Visualizar resposta*

   Existem diversas imagens mais enxutas disponíveis para o `python-3.6`, se consultarmos seu repositório no Docker Hub (https://hub.docker.com/_/python?tab=tags). Podemos utilizar as tags `3.6-slim` ou `3.6-alpine`, pra citar alguns exemplos. Abaixo, utilizaremos a imagem `python:3.6-alpine`.

   ```
   # grep FROM ~/myapp-color/Dockerfile
   FROM python:3.6-alpine
   ```

   Também poderíamos utilizar o comando `docker search` para fazer buscas, como visto abaixo:

   ```
   # docker search python | head -n5
   NAME                              DESCRIPTION
   STARS               OFFICIAL            AUTOMATED
   python                            Python is an interpreted, interactive, objec…
   5500                [OK]
   django                            Django is a free web application framework, …
   996                 [OK]
   pypy                              PyPy is a fast, compliant alternative implem…
   251                 [OK]
   nikolaik/python-nodejs            Python with Node.js
   53
                     [OK]
   ```

   Uma das limitações do `docker search` é que ele não permite a busca por *tags*, apenas por repositórios no Docker Hub. Podemos suplantar essa limitação de algumas formas — uma delas, sugerida em https://stackoverflow.com/questions/24481564/how-can-i-find-a-docker-image-with-a-specific-tag-in-docker-registry-on-the-dock , envolve o *parsing* de saída JSON obtida a partir de uma busca feita diretamente ao Docker Hub. Veja um exemplo de uso abaixo, em que definimos a variável `$REPO` como `python`:

   ```
   # REPO=python
   url=https://registry.hub.docker.com/v2/repositories/library/${REP
   O}/tags/?page_size=100 ; \
   ```

```
( \
  while [ ! -z $url ]; do \
    >&2 echo -n "." ; \
    content=$(curl -s $url | python -c 'import sys, json; data =
json.load(sys.stdin); print(data.get("next", "") or "");
print("\n".join([x["name"] for x in data["results"]]))') ; \
    url=$(echo "$content" | head -n 1) ; \
    echo "$content" | tail -n +2 ; \
  done; \
  >&2 echo ; \
) | cut -d '-' -f 1 | sort --version-sort | uniq;
.............
2
2.7
2.7.7
2.7.8
2.7.9
2.7.10
2.7.11
2.7.12
2.7.13

(...)

slim
stretch
wheezy
windowsservercore
```

Vamos fazer o *build* da nova imagem com a *tag* aplicada.

```
# cd ~/myapp-color/ ; docker build --tag myapp-color:small .
Sending build context to Docker daemon  80.38kB
Step 1/6 : FROM python:3.6-alpine

(...)

Successfully built 3b117e5dc997
Successfully tagged myapp-color:small
```

6. Qual o tamanho da nova imagem, `myapp-color:small`?

▼ *Visualizar resposta*

```
# docker images myapp-color:small --format '{{.Repository}}\t{{.Size}}'
myapp-color     51.8MB
```

7. Execute uma instância da imagem `myapp-color:small`, mapeando a porta 34080 no *host* para a

porta 80 do container, e verifique que a aplicação ainda permanece funcional.

▼ *Visualizar resposta*

Para diferenciar essa execução da anterior iremos ajustar a variável de ambiente `COLOR`:

```
# docker run -d -p 34080:80 -e COLOR=yellow myapp-color:small
17ddf38bd99d2a78be7d7b314d62c1f582e2ead590f43d86754edb4f0f9d98f5
```

Acessando o navegador na porta configurada, verificamos que a aplicação permanece funcional e com a cor especificada via variável de ambiente:

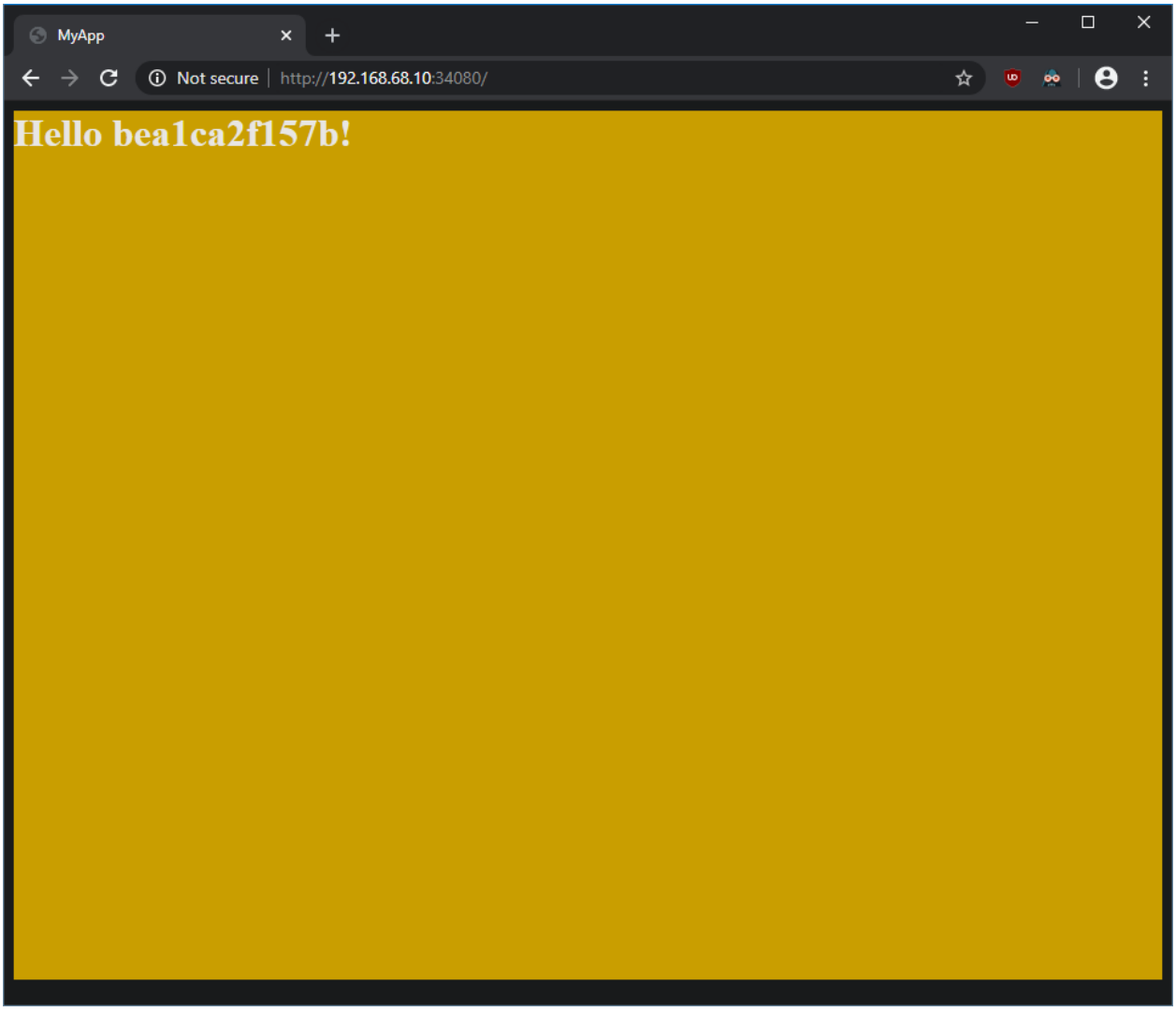

Figura 5. Imagem enxuta e funcional da aplicação myapp-color

7) Comandos e pontos de entrada

7.1) Comandos de inicialização em containers

1. Crie um diretório para testes usando o comando abaixo:

```
# mkdir ~/mydebian ; cd ~/mydebian
```

Dentro desse diretório, crie um arquivo `Dockerfile` com o seguinte conteúdo:

```
1 FROM debian
2
3 WORKDIR /opt
4
5 CMD ["echo", "Hello World"]
```

Crie uma imagem com a *tag* `mydebian` a partir desse `Dockerfile`. Em seguida, executa uma instância de container em *foreground* — qual mensagem é exibida na tela?

▼ *Visualizar resposta*

Para fazer o *build* da imagem basta executar:

```
# cd ~/mydebian/ ; docker build --tag mydebian .
Sending build context to Docker daemon  2.048kB
Step 1/3 : FROM debian

(...)

Successfully built 6ad80deda45b
Successfully tagged mydebian:latest
```

Executando o container como solicitado, obtemos:

```
# docker run mydebian
Hello World
```

2. Agora passe a *string* `Banana` como parâmetro para a linha de invocação do container. O que acontece, e porquê?

▼ *Visualizar resposta*

```
# docker run mydebian Banana
docker: Error response from daemon: failed to create shim: OCI runtime create
failed: container_linux.go:380: starting container process caused: exec:
"Banana": executable file not found in $PATH: unknown.
ERRO[0003] error waiting for container: context canceled
```

Encontramos um erro; isso se deve ao fato que os parâmetros passados ao final da linha de invocação do container são interpretados como comandos e **substituem** a configuração de `CMD` especificada no `Dockerfile`. Como não existe o binário `Banana` no `$PATH` do usuário, um erro é retornado.

7.2) Pontos de entrada em containers

1. Substitua o conteúdo do `Dockerfile` criado anteriormente pelo que se segue:

```
1 FROM debian
2
3 WORKDIR /opt
4
5 ENTRYPOINT ["echo", "Hello World"]
```

   Faça o *build* da imagem e execute o container normalmente, como feito no item (a) da atividade anterior. Em seguida, adicione a *string* `Banana` ao final do comando. O que acontece, e porquê?

   ▼ *Visualizar resposta*

```
# cd ~/mydebian/ ; docker build --tag mydebian .

(...)
```

   Feito o *build* da imagem, executamos o container normalmente:

```
# docker run mydebian
Hello World
```

   Nada de novo... e com a *string* `Banana` adicionada?

```
# docker run mydebian Banana
Hello World Banana
```

   O container funciona como esperado, e adiciona a *string* ao final da saída. Isso se deve ao fato de que a configuração de `ENTRYPOINT` define um ponto de entrada fixo para o container, e argumentos adicionais ao `docker run` são **adicionados ao final** (ou *appended*) do `ENTRYPOINT`.

2. Imagine que desejamos executar o comando `hostname` na imagem de container produzida no item anterior. Isso seria possível? E, se sim, como?

   ▼ *Visualizar resposta*

   Como vimos anteriormente, argumentos passados ao final do comando `docker run` são adicionados ao final do `ENTRYPOINT` do container — neste caso, no entanto, queremos alterar por completo o comando sendo executado.

   Podemos utilizar a *flag* `--entrypoint` para esse fim:

```
# docker run --entrypoint "hostname" mydebian
4009b0679ca0
```

7.3) Combinando pontos de entrada e comandos

1. Imagine que queremos produzir uma imagem de container que irá, por padrão, mostrar a mensagem `Banana` na tela. No entanto, se o usuário invocar o container com alguma *string* como argumento (por exemplo, `Chocolate`), essa palavra será mostrada no lugar de `Banana`.

   Como deveria ser configurado o `Dockerfile` para esse fim? Demonstre seu funcionamento fazendo o *build* da imagem e testando cada um dos casos apontados.

   ▼ *Visualizar resposta*

   Para atingir o objetivo apontado basta combinar as configurações de `ENTRYPOINT` e `CMD`, como visto abaixo:

   ```
   1 FROM debian
   2
   3 WORKDIR /opt
   4
   5 ENTRYPOINT ["echo"]
   6 CMD ["Banana"]
   ```

   Fazemos o *build* da imagem de container:

   ```
   # cd ~/mydebian/ ; docker build --tag mydebian .

   (...)
   ```

   E depois testamos — primeiro, sem nenhum parâmetro informado, e depois com uma *string* como `Chocolate`:

   ```
   # docker run mydebian
   Banana
   ```

   ```
   # docker run mydebian Chocolate
   Chocolate
   ```

8) Docker Compose

8.1) Criando serviços manualmente

1. Vamos começar devagar. Crie um container com a imagem `redis` e nome `redis`, rodando em *background*.

   ▼ *Visualizar resposta*

   ```
   # docker run -d --name redis redis
   ```

```
Unable to find image 'redis:latest' locally
latest: Pulling from library/redis

(...)

Status: Downloaded newer image for redis:latest
b6b1f657210949b33460fcae8b6f2826ef061da33f2aa471b22c0e436566e9e0
```

2. Agora, execute um container com a imagem `fbscarel/myapp-redis`, com o nome `web` e executando em *background*. Ligue-o com o container `redis` usando o *alias* `db` e exponha-o na porta 35080 do *host* e porta 80 do container.

   Feito isso, teste o funcionamento da aplicação: o que ela faz?

   ▼ *Visualizar resposta*

   Para ligar um container com outro temos diversas opções, uma delas a *flag* `--link` do comando `docker run`. Utilizando essa *flag*, o comando ficaria desta forma:

   ```
   # docker run -d --name web --link redis:db -p 35080:80 fbscarel/myapp-redis
   Unable to find image 'fbscarel/myapp-redis:latest' locally
   latest: Pulling from fbscarel/myapp-redis

   (...)

   Status: Downloaded newer image for fbscarel/myapp-redis:latest
   4c49128125c62aab07e430c9ffc0ae762c122baa231246ce34314a93a13267c9
   ```

   Publicada a aplicação, vamos acessá-la no navegador:

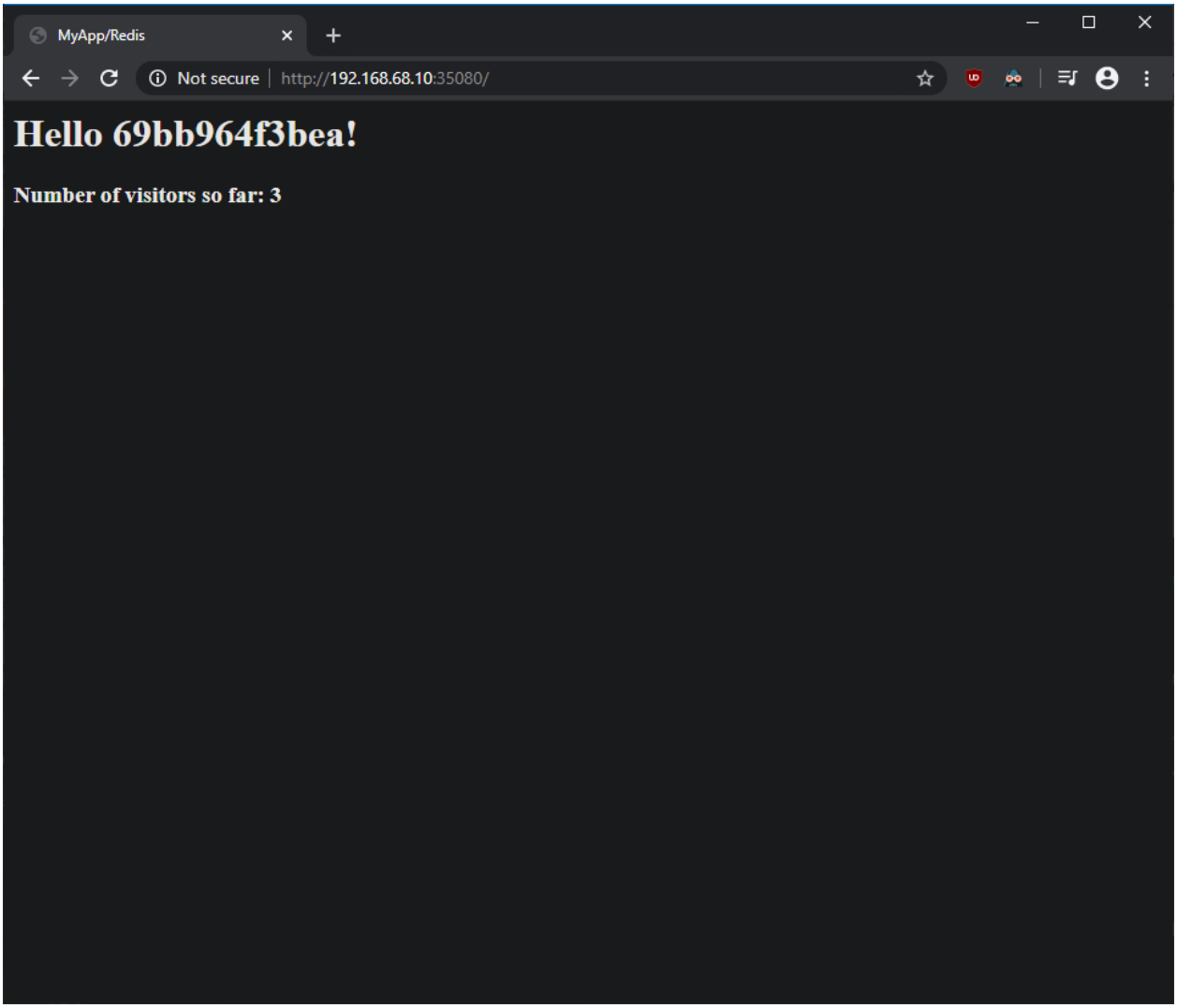

Figura 6. Aplicação myapp-redis funcional

Como visto acima, a aplicação mostra o *hostname* do container em que está operando, bem como um contador de visitantes na página. A cada vez que um novo acesso é realizado (por exemplo, usando a tecla `F5`), o contador é incrementado.

3. O que acontece com a aplicação se o container `redis` for encerrado? Porquê?

▼ *Visualizar resposta*

Vamos testar: primeiro, pare o container.

```
# docker stop redis
db
```

Note que agora a aplicação encontra um erro: *Cannot connect to Redis host 'db'*. Isso se deve ao fato de que o contador de visitantes é armazenado na base Redis, que agora encontra-se inacessível.

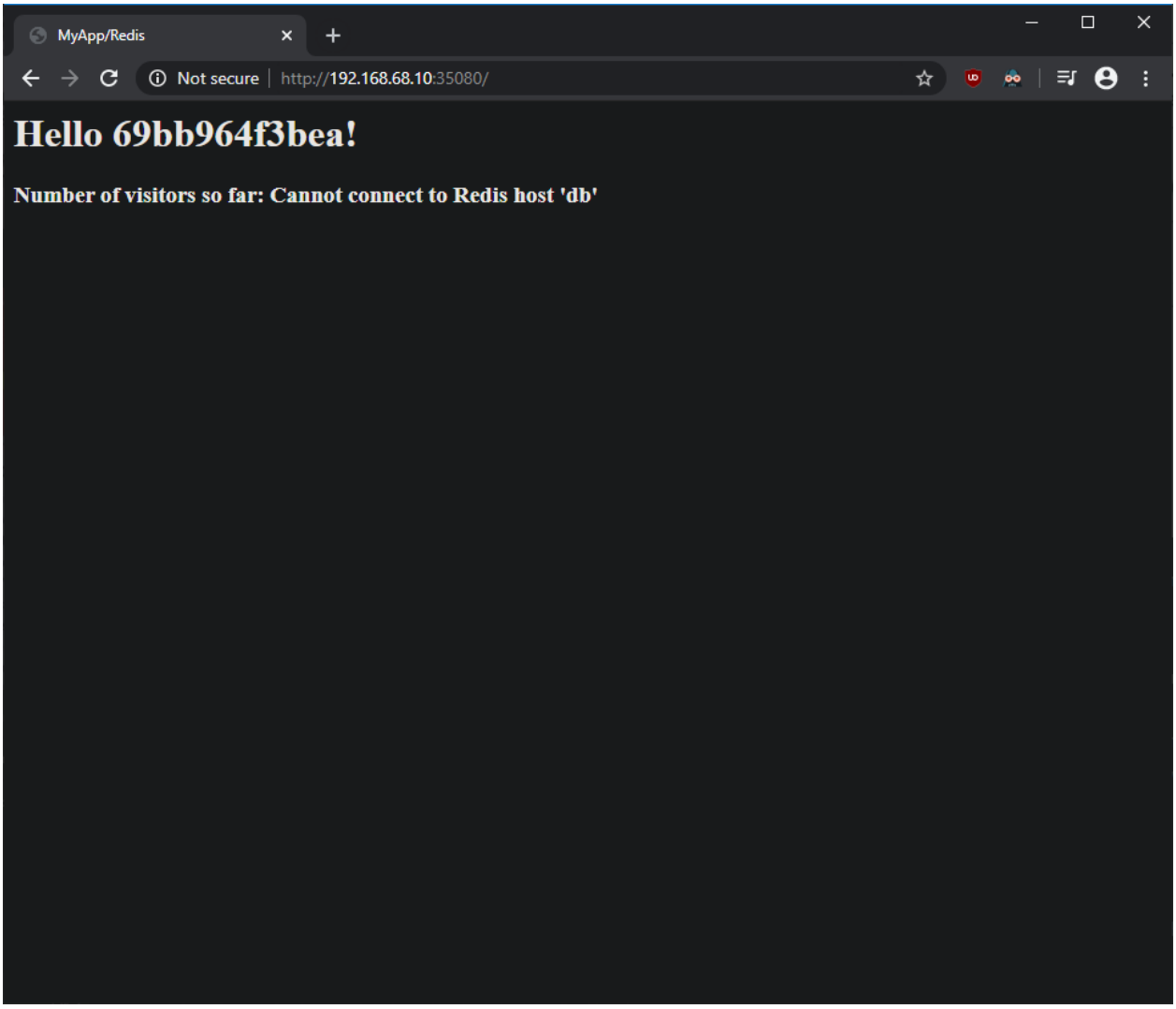

Figura 7. Base Redis inacessível

8.2) Usando o Docker Compose

1. Evidentemente, o método que utilizamos na atividade anterior para levantar serviços complexos não é sustentável: e se um dos containers cair, ou não for iniciado por engano? O ideal é que ambos sejam gerenciados conjuntamente, como um serviço — para esse fim, podemos usar o Docker Compose.

   O primeiro passo, evidentemente, envolve instalá-lo no sistema. Acesse https://github.com/docker/compose/releases/ e verifique a versão mais recente do Docker Compose — quando da escrita deste documento, a versão em questão é a 2.3.3.

   Para instalar o programa, execute o comando a seguir. Substitua o valor da variável `COMPOSE_VERSION` nesse comando com a versão desejada (por exemplo, a versão 2.3.3 mencionada anteriormente):

   ```
   <strong># export COMPOSE_VERSION=2.3.3 && curl -L
   "https://github.com/docker/compose/releases/download/v${COMPOSE_VERSION}/docker-
   compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose && chmod +x
   /usr/local/bin/docker-compose</strong>
   % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                  Dload  Upload   Total   Spent    Left  Speed
   ```

```
100   651  100   651    0     0   1276      0 --:--:-- --:--:-- --:--:--  1273
100 11.6M  100 11.6M    0     0  2324k      0  0:00:05  0:00:05 --:--:-- 3204k
```

Teste o funcionamento do binário.

```
# docker-compose --version
docker-compose version 2.3.3
```

2. Agora vamos à atividade propriamente dita. Antes de mais nada, pare e remova os containers criados na atividade (8.1).

   Em seguida, crie o diretório `/root/myapp-redis` e entre nele. Ali, crie um arquivo `docker-compose.yml` que automatize o lançamento dos dois containers criados na atividade anterior, com os mesmos nomes, ligações e configurações de exposição de portas. Se estiver em dúvida, consulte a documentação oficial do Docker Compose em https://docs.docker.com/compose/ .

   Finalmente, teste o funcionamento de sua configuração.

   ▼ *Visualizar resposta*

   Primeiro, vamos remover os containers criados anteriormente.

   ```
   # docker rm -f web db
   web
   db
   ```

   Vamos criar e entrar no diretório solicitado.

   ```
   # mkdir ~/myapp-redis ; cd ~/myapp-redis
   ```

   Dentro desse diretório criamos o arquivo `docker-compose.yml` com o conteúdo que se segue:

   ```
   1 version: '2.0'
   2 services:
   3   web:
   4     image: fbscarel/myapp-redis
   5     ports:
   6     - "35080:80"
   7     links:
   8     - db
   9   db:
   10    image: redis
   ```

   Agora, basta subir o serviço com o comando:

   ```
   # docker-compose up
   ```

```
Creating network "myapp-redis_default" with the default driver
Creating myapp-redis_db_1 ... done
Creating myapp-redis_web_1 ... done

(...)

web_1  |  * Running on http://0.0.0.0:80/ (Press CTRL+C to quit)
```

Vamos testar, acessando a porta especificada no arquivo `docker-compose.yml` em um navegador na máquina física.

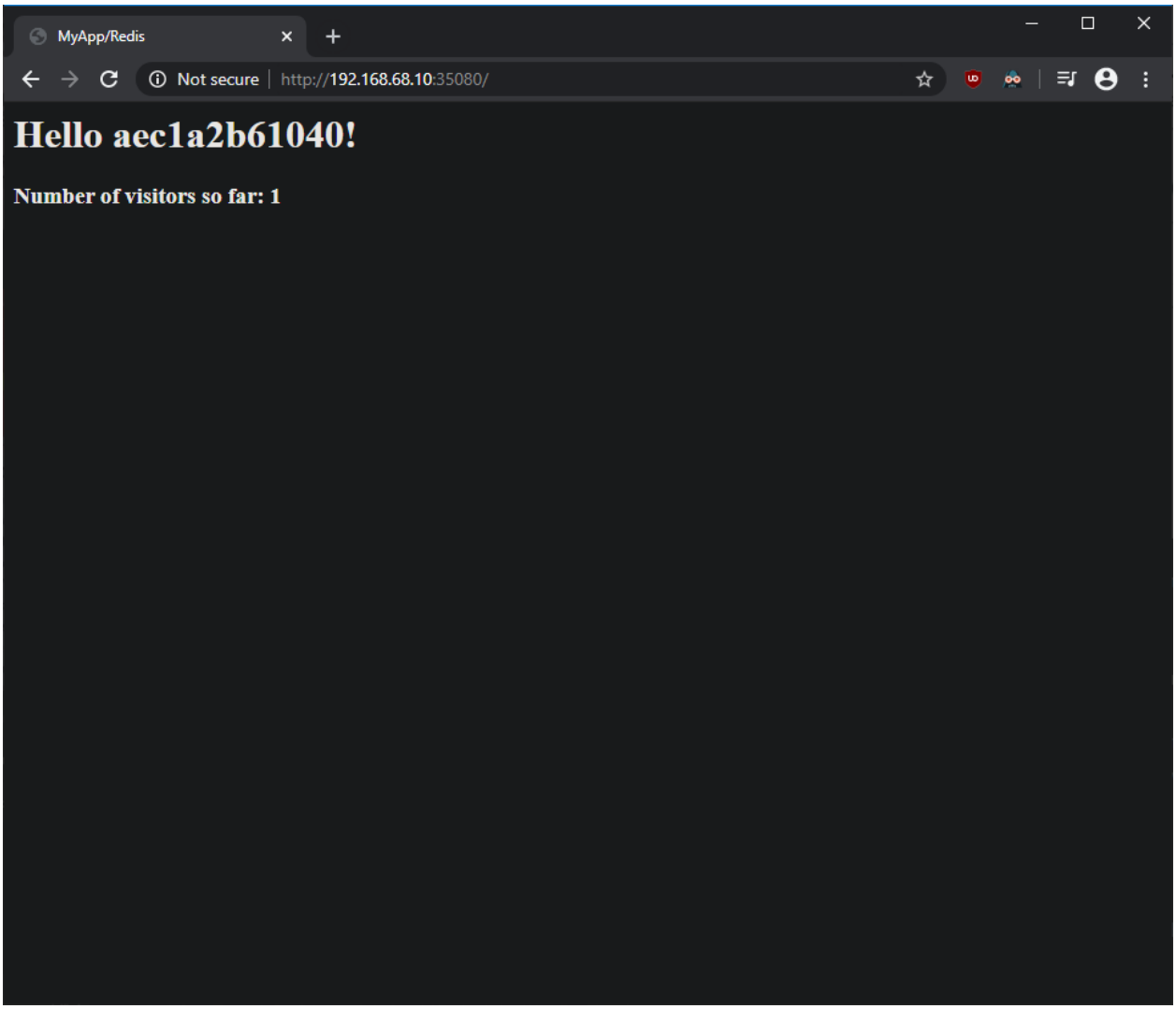

Figura 8. Serviço myapp-redis publicado via Docker Compose

8.3) Executando aplicações complexas

1. O Docker Compose permite publicar serviços muito mais complexos do que o visto na seção (8.2). Quer um exemplo, e um desafio? Acesse https://github.com/dockersamples/example-voting-app e faça o *deployment* da aplicação em seu ambiente usando o Docker Compose.

   Essa é uma aplicação de votação com múltiplas camadas e containers, seguindo a arquitetura abaixo:

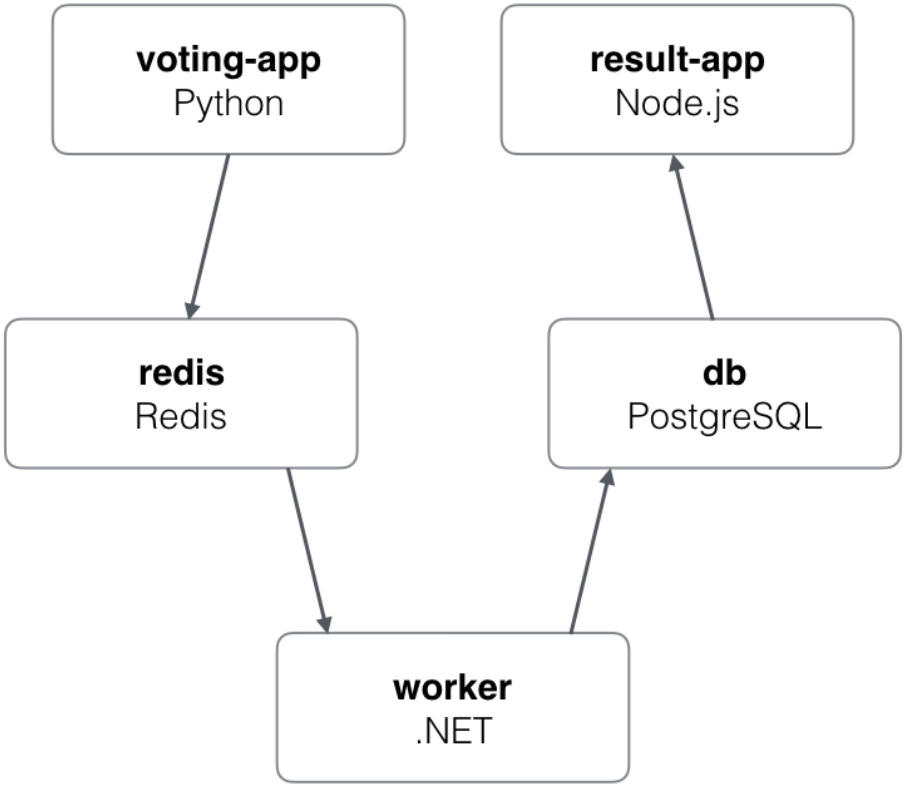

Figura 9. Arquitetura da aplicação example-voting-app

Uma vez realizado o *deployment*, você poderá votar e ver resultados nas portas 5000 e 5001, respectivamente.

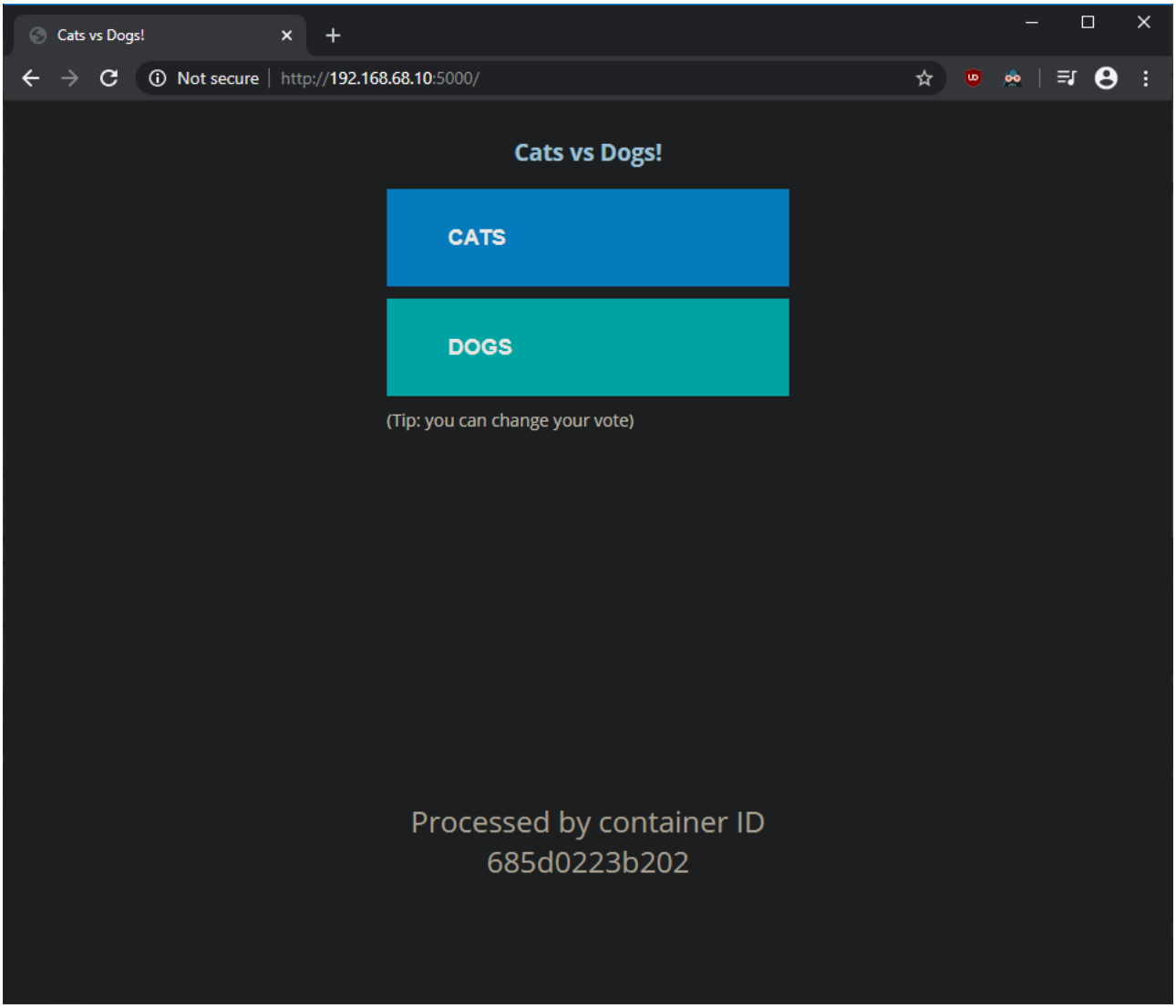

Figura 10. Interface de votação

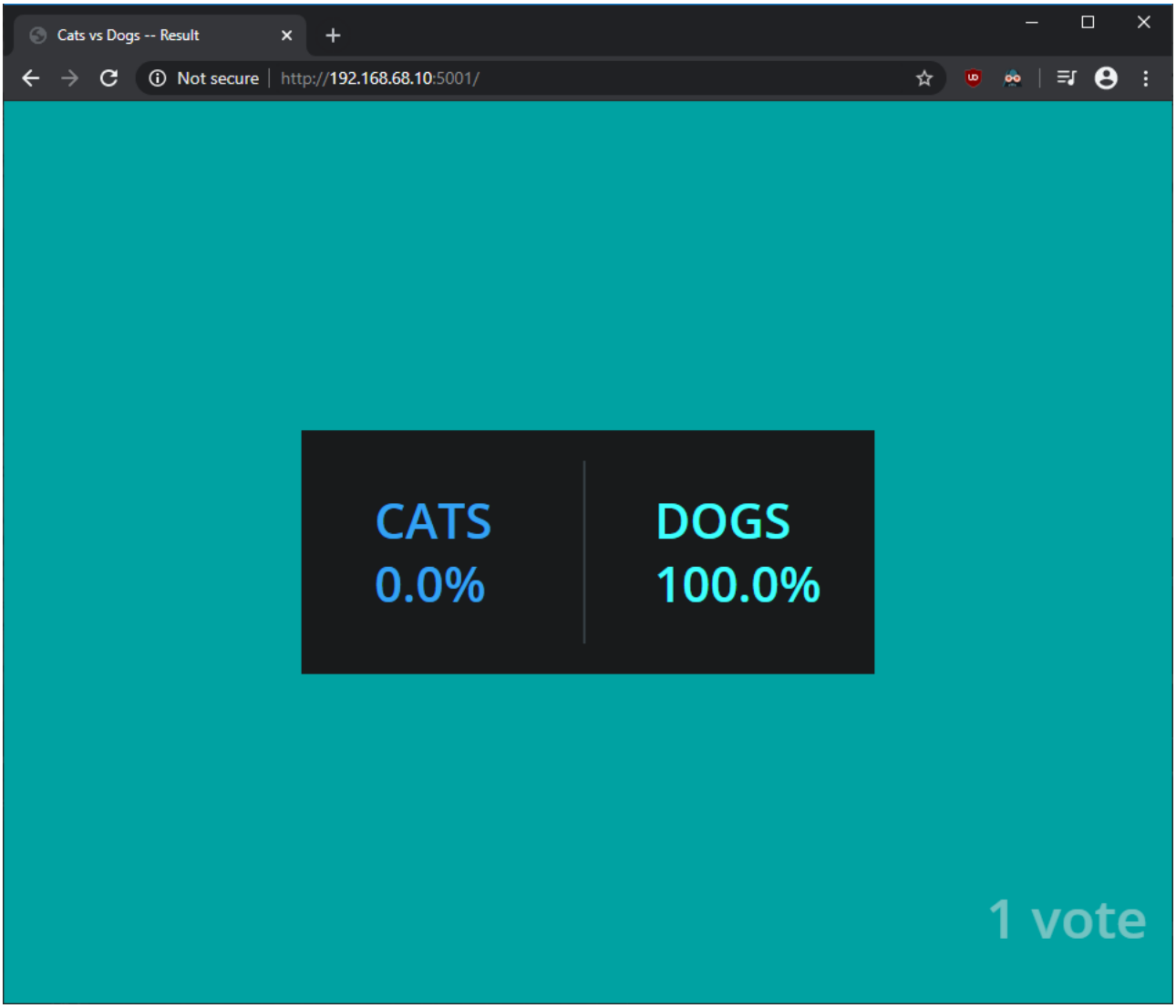

Figura 11. Interface de visualização de resultados

Explore o código-fonte das aplicações e seus `Dockerfiles` e arquivos `docker-compose.yml` para entender seu funcionamento, bem como aprender mais sobre aspectos avançados do Docker.

▼ *Visualizar resposta*

Para realizar a atividade, simplesmente execute:

```
<strong># cd ~ && git clone https://github.com/dockersamples/example-voting-app.git
&& cd example-voting-app && docker-compose up</strong>
Cloning into 'example-voting-app'...
(...)
```

Uma vez que todos os containers estejam operacionais, acesse a aplicação de voto através da URL http://192.168.68.10:5000/ , utilizando um navegador em sua máquina física. Para visualizas os resultados de votação, acesse http://192.168.68.10:5001/ .

9) Armazenamento no Docker

9.1) Diretórios relevantes na hierarquia do sistema

1. Em qual diretório do sistema são armazenados, por padrão, as imagens e containers Docker?

   ▼ *Visualizar resposta*

   Esse informação pode ser obtida via comando `docker info`, especificamente na entrada `Docker Root Dir`:

   ```
   # docker info 2> /dev/null | grep -i root
    Docker Root Dir: /var/lib/docker
   ```

   Nesse diretório ficam armazenados diversos objetos relevantes ao funcionamento do Docker, como containers, imagens, redes e volumes:

   ```
   # ls -1 /var/lib/docker/
   builder
   buildkit
   containers
   image
   network
   overlay2
   plugins
   runtimes
   swarm
   tmp
   trust
   volumes
   ```

2. Agora, execute o comando abaixo:

   ```
   # lab-1.9.1
   ```

   Em qual diretório são armazenados os arquivos relativos ao container `web-2`?

   ▼ *Visualizar resposta*

   Para responder essa pergunta, é necessário primeiro saber o identificador numérico do container em questão:

   ```
   # docker ps --format 'table {{.Names}}\t{{.ID}}'
   NAMES               CONTAINER ID
   web-3               e633f7677ec5
   web-2               ddfb2c342d0d
   web-1               bcbb25911c3f
   ```

   Note que os diretórios em `/var/lib/docker/containers` possuem um formato interessante — os primeiros caracteres desses diretórios coincidem com os identificadores numéricos obtidos

pelo comando anterior. Assim, fica fácil saber a qual container eles se referem.

```
# ls -1 /var/lib/docker/containers/
bcbb25911c3fe4670f411745cf787aca6266451dd88b8ebf0691b0d089809204
ddfb2c342d0d3be454cd3a6779597001c3cde207f15848c52009886bd42f4ec0
e633f7677ec5d71a4ffe303f8beea5760893b2dad57fdfce1e59e8959a933864
```

Podemos combinar esses conceitos para obter o diretório de forma direta, com o comando abaixo:

```
# ls -d /var/lib/docker/containers/$( docker ps -aqf name=web-2 )*
/var/lib/docker/containers/ddfb2c342d0d3be454cd3a6779597001c3cde207f15848c52009886bd42f4ec0
```

9.2) Persistência de dados

1. Execute um container não-nomeado com a imagem `redis:alpine` em *background*.

   Em seguida, use o utilitário de linha de comando `redis-cli` disponível dentro do container para inserir os seguintes valores na base: `fruit=banana`, `animal=zebra` e `vehicle=taxi`. Não se esqueça de salvar as modificações feitas na base em disco com o comando `save`.

   Consulte os valores para garantir que foram inseridos corretamente.

   ▼ *Visualizar resposta*

   Vamos começar iniciando o container:

   ```
   # docker run -d redis:alpine
   5d75609ac4d5717c4f05d0ca2716e28f7dc79666aa7f15157f6899969c6e9d06
   ```

   Para executar comandos dentro de um container é necessário utilizar seu nome; como este não foi especificado, podemos usar seu ID, que consiste nos 12 primeiros caracteres da *string* de saída do comando `docker run` executado acima.

   Para interagir com a base Redis podemos usar o comando `redis-cli`, documentado em https://redis.io/topics/rediscli . Podemos inserir dados com o parâmetro `set` — note que o nome do container nos comandos abaixo deve ser customizado para o criado pelo seu *daemon* Docker:

   ```
   # docker exec 5d75609ac4d5 redis-cli set fruit banana
   OK
   ```

   ```
   # docker exec 5d75609ac4d5 redis-cli set animal zebra
   OK
   ```

```
## docker exec 5d75609ac4d5 redis-cli set vehicle taxi
OK
```

Para consultar os dados, basta usar o parâmetro `get`:

```
# docker exec 5d75609ac4d5 redis-cli get fruit
banana
```

```
# docker exec 5d75609ac4d5 redis-cli get animal
zebra
```

```
# docker exec 5d75609ac4d5 redis-cli get vehicle
taxi
```

Como solicitado, iremos salvar o conteúdo gravado até aqui com o comando `save`:

```
# docker exec 5d75609ac4d5 redis-cli save
OK
```

2. Vamos imaginar que o sistema teve algum problema e os containers foram deletados. Remova o container criado no passo (a) e crie-o novamente. É possível acessar seus dados? Porquê?

▼ *Visualizar resposta*

Começamos pela remoção do container:

```
# docker rm -f 5d75609ac4d5
5d75609ac4d5
```

A seguir, iniciamos um novo container com as mesmas configurações:

```
# docker run -d redis:alpine
cfa02b50c204669cf8ed1fbd7e1fa4660e4f4535ce06a54cbca501379f45644f
```

Note que os dados não estão mais acessíveis, como demonstrado pelo comando abaixo. Isto ocorre porque o diretório de armazenamento de dados utilizado pelo container é temporário, criado sob o diretório `/var/lib/docker` e ligado ao seu identificador numérico. Quando o container é removido, esse ligação se perde.

```
# docker exec cfa02b50c204 redis-cli get fruit
```

3. E se o container fosse nomeado? Haveria alguma diferença?

▼ *Visualizar resposta*

Como visto pela sequência de comandos abaixo, o comportamento de volumes temporários é o mesmo entre containers nomeados e não-nomeados.

```
# docker run -d --name db-named redis:alpine
39747d12c54476974184a2f3579626fdde3b16980442bb9db9fdfcc42efaeca8
```

```
# docker exec db-named redis-cli set fruit banana
OK
```

```
# docker exec db-named redis-cli get fruit
banana
```

```
# docker exec db-named redis-cli save
OK
```

```
# docker rm -f db-named
db-named
```

```
# docker run -d --name db-named redis:alpine
424a62c4a4812dcabfaffabd83f9006077e9a2aeb5562a9ea1ba73f0873a4677
```

```
# docker exec db-named redis-cli get fruit
```

4. Execute o mesmo container, mas agora crie um mapeamento de volume de forma que os dados armazenados no diretório `/data` do container sejam armazenados em `/opt/data` no *host*. Utilize a imagem `redis:alpine` e o nome `db`.

   Feito isso, crie as chaves `fruit`, `animal` e `vehicle` com os mesmo valores utilizados no item (a) desta atividade. Não se esqueça de salvar as modificações feitas na base em disco com o comando `save`.

   ▼ *Visualizar resposta*

   O mapeamento de volumes pode ser feito através da opção `--volume`, ou `-v`. A sintaxe a ser utilizada é `DIR_HOST:DIR_CONTAINER`, como visto no comando abaixo.

   ```
   # docker run -d --name db -v /opt/data:/data redis:alpine
   c02759d357560ec0537672509563836475cc6f48bdb24d6722b39bf759461aaa
   ```

   Agora, basta inserir as chaves solicitadas.

```
# docker exec db redis-cli set fruit banana
OK
```

```
# docker exec db redis-cli set animal zebra
OK
```

```
# docker exec db redis-cli set vehicle taxi
OK
```

```
# docker exec db redis-cli save
OK
```

Veja que os dados gravados na base até aqui foram salvos no diretório `/opt/data` do *host*:

```
# ls /opt/data/
dump.rdb
```

5. Novamente, temos uma falha catastrófica no sistema. Desta vez, no entanto, temos os dados armazenados dentro do diretório `/opt/data`.

   Remova o container criado no passo (d) e crie-o novamente usando as mesmas opções. É possível acessar as chaves criadas anteriormente?

   ▼ *Visualizar resposta*

   ```
   # docker rm -f db
   db
   ```

   ```
   # docker run -d --name db -v /opt/data:/data redis:alpine
   b1bfcdf1bfe372e5bdd1d89e9e58a09ea0c998785bdb2ad4cc10ff08d1db86af
   ```

   Acima, o container foi removido e recriado com as mesmas opções. Como os dados estão armazenados externamente, em um diretório no *host*, eles estão íntegros e acessíveis:

   ```
   # docker exec db redis-cli get animal
   zebra
   ```

10) Gerência de redes no Docker

10.1) Angariando informações sobre redes

1. Antes de iniciar, execute o comando abaixo:

   ```
   # lab-1.10.1
   ```

   Agora, responda: quantas redes existem no ambiente Docker?

   ▼ *Visualizar resposta*

   ```
   # docker network ls -q | wc -l
   3
   ```

2. Qual é o identificador numérico associado à rede `host`?

   ▼ *Visualizar resposta*

   ```
   # docker network ls -qf name=host
   8ec6bab7d198
   ```

3. Aponte o identificador numérico da rede à qual o container `web-1` está conectado. Qual o nome dessa rede?

   ▼ *Visualizar resposta*

   O comando `docker inspect` pode ser usado para esse fim. Como sua saída é bastante extensa, podemos usar um filtro como o que se segue para obter apenas a informação pretendida.

   ```
   # docker inspect web-1 --format='{{range
   .NetworkSettings.Networks}}{{.NetworkID}}{{end}}'
   674a3d492d0e3dc1650a8dfa386d94516493c74a69f6a3bf8642e6c3246e0092
   ```

   O nome da rede pode ser obtido via `docker network ls`. Assim como no caso de containers, note que os 12 primeiros caracteres do identificador numérico são coincidentes com a saída do comando anterior. Podemos combinar os dois comandos da seguinte forma:

   ```
   # NETID="$( docker inspect web-1 --format='{{range
   .NetworkSettings.Networks}}{{.NetworkID}}{{end}}' )" ; docker network ls
   --format='table {{.ID}}\t{{.Name}}' | grep "${NETID:0:12}"
   674a3d492d0e        none
   ```

4. Qual a sub-rede configurada na rede Docker de nome `bridge`?

   ▼ *Visualizar resposta*

   As sub-redes configuradas em uma rede Docker podem ser visualizadas com o comando `docker network inspect`. As sub-redes em si ficam armazenadas como um *array* dentro de `$.IPAM.Config`. Podemos obter seu valor diretamente com o comando abaixo:

```
# docker network inspect bridge --format='{{range
.IPAM.Config}}{{.Subnet}}{{end}}'
172.17.0.0/16
```

5. Crie um container de nome `web-2` rodando em *background*, usando a imagem `nginx:alpine` e conectado à rede `none`.

   ▼ *Visualizar resposta*

   ```
   # docker run -d --name web-2 --network none nginx:alpine
   a9997e9712c534e2b1a881a3ae2f71aa893ef5e844d07d015aebd719abce6356
   ```

10.2) Conectando containers via redes Docker

1. Crie uma nova rede com o nome `myapp-mysql-network` usando o *driver* `bridge`. Aloque uma sub-rede na faixa 100.64.0.0/24, com o *gateway* em 100.64.0.1.

   ▼ *Visualizar resposta*

   ```
   # docker network create -d bridge --gateway 100.64.0.1 --subnet 100.64.0.0/24
   myapp-mysql-network
   0aafbb8d8b4c8213f56551b3a3eafa9373d2aaefe989130278f65cd2d548cf79
   ```

2. Faça o *deployment* de uma base de dados MySQL com o nome `db`, rodando em *background* e usando a senha `qwerty` para o usuário `root`. Conecte-a à rede criada no passo anterior.

   ▼ *Visualizar resposta*

   ```
   # docker run -d --name db --network myapp-mysql-network -e
   MYSQL_ROOT_PASSWORD=qwerty mysql
   ae428c32fc6fe357a0124a76183953565a92c9416309221e13dc0d5b9b99f25a
   ```

3. Agora, faça o *deployment* da aplicação de nome `app` usando a imagem `fbscarel/myapp-mysql`, rodando em *background*. Conecte o container à mesma rede criada no passo (a); além disso, também exponha a porta 36080 no *host* mapeada para a porta 80 do container.

   Teste o funcionamento de sua configuração.

   ▼ *Visualizar resposta*

   O *deployment* do container é bastante simples, como visto a seguir.

   ```
   # docker run -d --name app --network myapp-mysql-network -p 36080:80
   fbscarel/myapp-mysql
   0604e4b4ca355b75310e9503259f1cbe13ff6843f8a76e15fbbc7d966a484575
   ```

   A aplicação torna-se acessível, mas a conexão com o banco de dados não é completada com

sucesso:

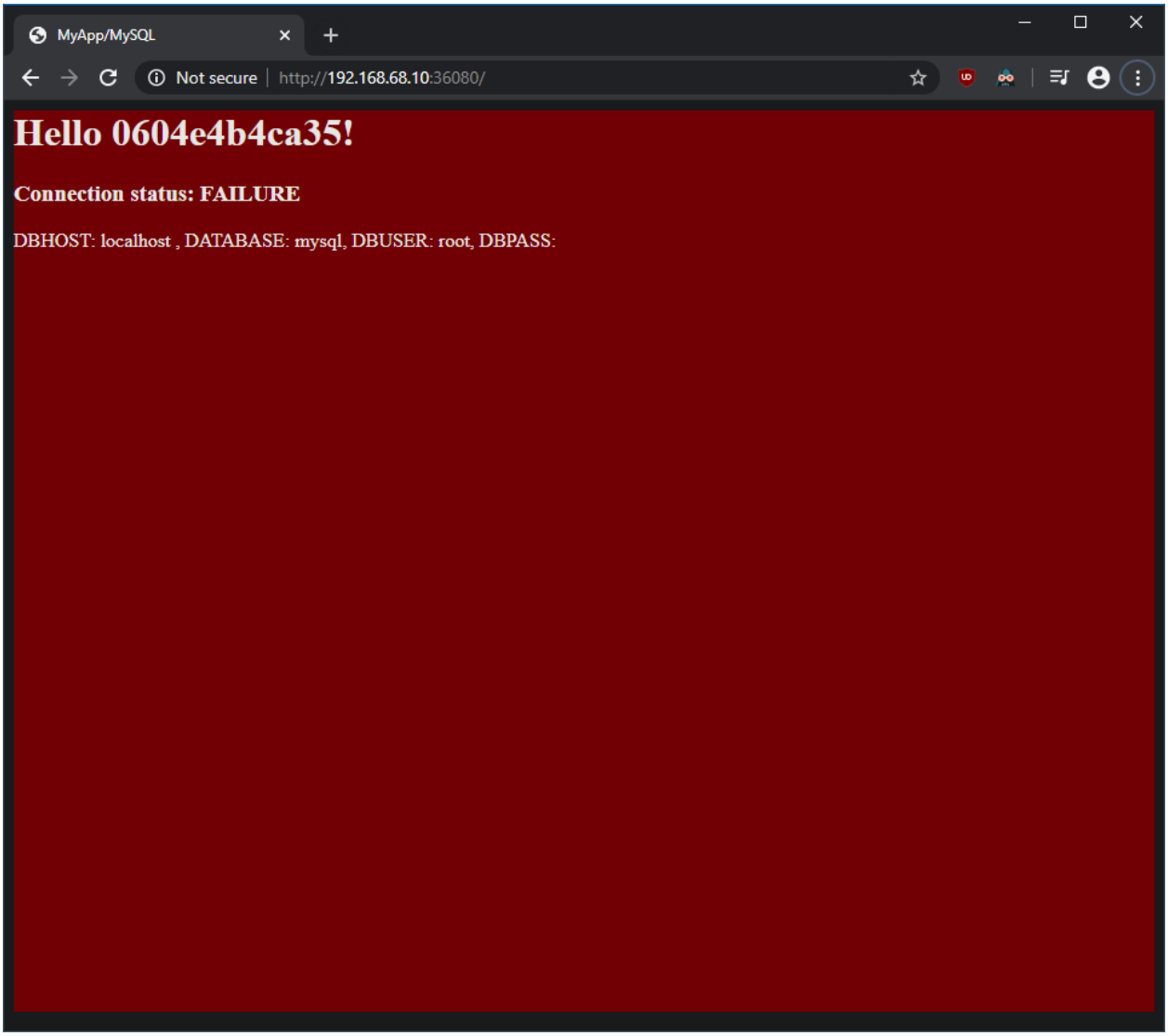

Figura 12. Conexão falha ao banco de dados

4. O que aconteceu, e qual o motivo? Inspecione o código-fonte da aplicação (disponível no caminho `/opt/app.py`, dentro do container recém-criado) para investigar uma solução para o problema.

   Uma vez entendido o problema, volte a executar o container com as correções integradas. Verifique seu funcionamento.

   ▼ *Visualizar resposta*

   A conexão com o banco de dados não funcionou por dois motivos:

   1. O nome utilizado para conectar-se ao banco é `localhost`, e não o nome correto `db`.
   2. A senha de conexão com o usuário `root` está em branco, como visto no item anterior.

   Investigando o código fonte da aplicação, percebemos que esses parâmetros podem ser configurados via linha de comando ou variáveis de ambiente, como visto abaixo.

   ```
   # docker exec -it app cat /opt/app.py | grep 'add_argument\|argtest' | grep -v
   '^def'
   ```

```
  parser.add_argument('-h', '--hostname', required=False)
  parser.add_argument('-d', '--database', required=False)
  parser.add_argument('-u', '--username', required=False)
  parser.add_argument('-p', '--password', required=False)

  DBHOST   = argtest (args.hostname, os.environ.get('DBHOST'),   'localhost')
  DBUSER   = argtest (args.username, os.environ.get('DBUSER'),   'root')
  DBPASS   = argtest (args.password, os.environ.get('DBPASS'),   '')
  DATABASE = argtest (args.database, os.environ.get('DATABASE'), 'mysql')
```

Note que os parâmetros de *hostname*, base de dados, usuário e senha de conexão podem ser configurados. No caso de uso de argumentos em linha de comando, pode-se utilizar respectivamente as opções `--hostname`, `--database`, `--username` e `--password`. Já no caso de variáveis de ambiente, os valores esperados são `DBHOST`, `DATABASE`, `DBUSER` e `DBPASS`.

Sabendo disso, basta invocar o container com as configurações ajustadas. Primeiro, vamos encerrar sua operação:

```
# docker rm -f app
app
```

Agora, vamos iniciá-lo. No exemplo abaixo, utilizaremos variáveis de ambiente para corrigir a configuração.

```
# docker run -d --name app --network myapp-mysql-network -p 36080:80 -e DBHOST=db
-e DBPASS=qwerty fbscarel/myapp-mysql
7d0c3f697dc06843e299bff0be07c026c82cbbcca76687764fb3280305355e69
```

Testando o acesso via navegador, podemos constatar que a conexão é completada com sucesso.

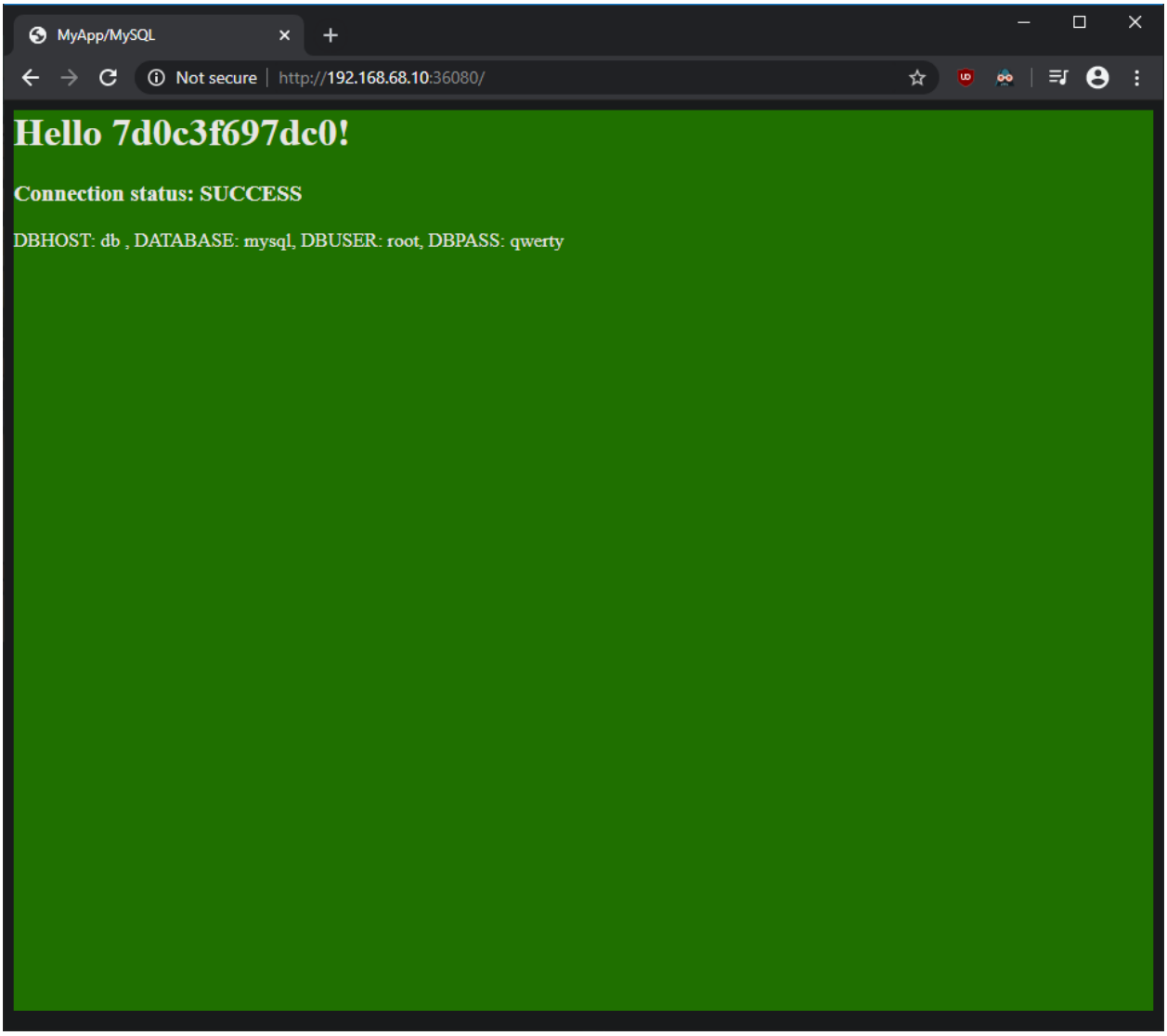

Figura 13. Conexão ao banco de dados realizada com sucesso

11) Registros e política de reinício

11.1) Visualizando registros de containers

1. Invoque a execução de um container para criação de registros aleatórios com o comando abaixo:

```
# docker run -d --name logger chentex/random-logger:latest 100 400
```

Como seria possível visualizar as mensagens de log geradas por esse container desde sua criação?

▼ *Visualizar resposta*

O comando `docker logs` cumpre esse papel:

```
# docker logs logger
2020-09-23T05:00:18+0000 ERROR An error is usually an exception that has been
caught and not handled.
```

```
2020-09-23T05:00:19+0000 WARN A warning that should be ignored is usually at this
level and should be actionable.
2020-09-23T05:00:19+0000 ERROR An error is usually an exception that has been
caught and not handled.
2020-09-23T05:00:19+0000 DEBUG This is a debug log that shows a log that can be
ignored.
2020-09-23T05:00:19+0000 WARN A warning that should be ignored is usually at this
level and should be actionable.

(...)
```

2. São muitas mensagens! Como visualizar as cinco mais recentes?

   ▼ *Visualizar resposta*

   Basta usar a opção `--tail`, como visto abaixo.

   ```
   # docker logs logger --tail=5
   2020-09-23T05:04:13+0000 WARN A warning that should be ignored is usually at this
   level and should be actionable.
   2020-09-23T05:04:14+0000 WARN A warning that should be ignored is usually at this
   level and should be actionable.
   2020-09-23T05:04:14+0000 ERROR An error is usually an exception that has been
   caught and not handled.
   2020-09-23T05:04:14+0000 DEBUG This is a debug log that shows a log that can be
   ignored.
   2020-09-23T05:04:14+0000 INFO This is less important than debug log and is often
   used to provide context in the current task.
   ```

3. Note que o container em questão produz mensagens constantemente — poderia ser interessante acompanhá-las à medida que são geradas, interativamente no terminal. Como fazê-lo?

   ▼ *Visualizar resposta*

   A opção `--follow` (ou `-f`) pode ser usada para esse fim. Contudo, sem o uso conjunto da opção `--tail`, isso resultará na exibição de todos os logs desde a criação do container antes de passar a acompanhar as mais recentes. É, portanto, bastante apropriado combinar os dois conceitos, como visto abaixo.

   ```
   # docker logs logger --tail=0 -f
   2020-09-23T05:07:02+0000 INFO This is less important than debug log and is often
   used to provide context in the current task.
   2020-09-23T05:07:03+0000 WARN A warning that should be ignored is usually at this
   level and should be actionable.
   2020-09-23T05:07:03+0000 ERROR An error is usually an exception that has been
   caught and not handled.
   2020-09-23T05:07:03+0000 INFO This is less important than debug log and is often
   used to provide context in the current task.
   2020-09-23T05:07:04+0000 WARN A warning that should be ignored is usually at this
   ```

```
level and should be actionable.
```

11.2) Monitorando o estado de containers

1. Ocasionalmente um container pode encerrar sua execução, seja por um erro inesperado ou por exaustão de recursos. Nesses casos, pode ser interessante reiniciar o container de forma automática. Quais são as opções disponíveis nesse sentido, usando o `docker run`?

   ▼ *Visualizar resposta*

   Temos quatro opções de reinício automático para containers, como documentado na página de manual do `docker run`: `no`, `on-failure`, `always` e `unless-stopped`. Elas são descritas em detalhe abaixo:

```
|-------------------------|-----------------------------------|
 | Policy                  | Result                            |
|-------------------------|-----------------------------------|
 | no                      | Do not automatically restart the  |
 |                         | container when it exits.          |
|-------------------------|-----------------------------------|
 | on-failure[:max-retries] | Restart only if the container     |
 |                         | exits with a non-zero exit        |
 |                         | status. Optionally, limit the     |
 |                         | number of restart retries the     |
 |                         | Docker daemon attempts.           |
|-------------------------|-----------------------------------|
 | always                  | Always restart the container      |
 |                         | regardless of the exit status.    |
 |                         | When you specify always, the      |
 |                         | Docker daemon will try to restart |
 |                         | the container indefinitely. The   |
 |                         | container will also always start  |
 |                         | on daemon startup, regardless of  |
 |                         | the current state of the          |
 |                         | container.                        |
|-------------------------|-----------------------------------|
 | unless-stopped          | Always restart the container      |
 |                         | regardless of the exit status,    |
 |                         | but do not start it on daemon     |
 |                         | startup if the container has been |
 |                         | put to a stopped state before.    |
|-------------------------|-----------------------------------|
```

2. Usando o container `alpine` e o comando `sleep`, teste o funcionamento de uma das opções de reinício exploradas no item anterior. Verifique que sua solução produz o efeito desejado.

   ▼ *Visualizar resposta*

   Vamos testar abaixo a opção `unless-stopped`. Para todos os efeitos, neste caso, a opção `always` teria comportamente semelhante.

```
# docker run -d --restart unless-stopped --name sleeper alpine sleep 10
Unable to find image 'alpine:latest' locally
latest: Pulling from library/alpine

(...)

d72a0f83b2255928f8947feb96bb01b656239a556396760109bde10a8f13a523
```

O container está operacional. Para verificar o funcionamento da diretiva de `restart` podemos monitorar constantemente o estado do container, usando o comando abaixo por exemplo:

```
# while true; do docker ps -f name=sleeper --format '{{.Status}}'; sleep 1; done
Up Less than a second
Up 1 second
Up 2 seconds
Up 3 seconds
Up 4 seconds
Up 5 seconds
Up 6 seconds
Up 7 seconds
Up 8 seconds
Up 10 seconds
Up Less than a second
Up 1 second
```

Observe que o container permanece funcional por dez segundos, e logo em seguida é encerrado — isso ocorre porque o comando `sleep` termina sua execução. Imediatamente, ele é reiniciado pelo *daemon* Docker e o processo se repete.

3. Como determinar o número de vezes que um container foi reiniciado?

▼ *Visualizar resposta*

Essa informação pode ser obtida através da inspeção detalhada do container via `docker inspect`. Para obter diretamente a informação desejada, pode-se executar:

```
# docker inspect sleeper --format '{{.RestartCount}}'
45
```

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 página de resultados de votação do software `example-voting-app`.

Utilize como referência a segunda imagem mostrada na atividade 1.8.3 (a) deste roteiro.