Deploy de aplicações com Docker, Kubernetes e Terraform

Comandos Docker

Antes de iniciarmos o estudo da sintaxe do **Dockerfile** estudaremos os principais comandos do **Docker**.

Imagens

Para criar **imagens** utilizamos o comando **build**:

```
docker build -t meu-postgres .
```

Este comando "compila" o **Dockerfile** gerando uma imagem de sistema que pode ser utilizada para criar conteineres. A sintaxe do comando **build** é simples:

```
docker build [OPTIONS] PATH | URL | -
```

Este comando constrói imagens a partir de um **Dockerfile** e uma pasta de **contexto**, indicados pelo argumento **PATH**. Você pode opcionalmente utilizar uma **URL** para indicar o contexto:

- A partir de um repositório **Git**.
- A partir de um arquivo **TAR**.

Veremos alguns exemplos:

```
docker build -t meu-postgres . ---> Contexto na pasta atual
docker build -t projeto
https://github.com/docker/rootfs.git#container:docker ---> Contexto no repo
Git, branch "container", pasta "docker"
docker build -t projeto http://server/context.tar.gz ---> Contexto no
arquivo TAR/GZ localizado naquele servidor.
```

Entre as opções do comando estão:

- **-t** | **--tag**: Nome da imagem com um tag de versão opcional. Exemplo: meu-postgres:v0.0.1
- **-f** | **--file**: Nome do **Dockerfile** caso seja diferente desse padrão: "Dockerfile"

As imagens ficam armazenadas localmente. Para criar conteineres, a imagem precisa estar baixada ou criada no local. O comando **pull** baixa uma imagem:

```
docker image pull debian
```

Listar e apagar imagens

Podemos listar imagens com o comando **images**:

```
docker images [OPTIONS] [REPOSITORY[:TAG]]

$ docker images

REPOSITORY                    TAG       IMAGE ID       CREATED         SIZE
gcr.io/k8s-minikube/kicbase   v0.0.42   dbc648475405   7 days ago
1.2GB
<none>                        <none>    52cd4c80d699   9 months ago
85.4MB
redis                         latest    19c51d4327cf   10 months ago
117MB
ubuntu                        latest    6b7dfa7e8fdb   11 months ago
77.8MB
gcr.io/k8s-minikube/kicbase   v0.0.28   e2a6c047bedd   2 years ago
1.08GB
```

Você reparou que as imagens vêm de repositórios? Podemos listar as imagens informando o nome e ou a tag:

```
docker images gcr.io/k8s-minikube/kicbase
REPOSITORY                    TAG       IMAGE ID       CREATED       SIZE
gcr.io/k8s-minikube/kicbase   v0.0.42   dbc648475405   7 days ago    1.2GB
gcr.io/k8s-minikube/kicbase   v0.0.28   e2a6c047bedd   2 years ago   1.08GB
```

E podemos apagar imagens se assim desejarmos:

```
docker image rm [OPTIONS] IMAGE [IMAGE...]

docker image rm redis
```

Existe a opção **--force** | **-f** para forçar a remoção de uma imagem que esteja sendo utilizada em um contêiner (ativo ou não).

Como as imagens ocupam espaço, você pode querer remover as imagens:

```
docker prune [-a]
```

Este comando remove as imagens que não têm tag associado e, se acrescido de **-a** remove as imagens que não tenham nenhum contêiner as utilizando.

Executar

Podemos executar uma imagem criando um contêiner e executando um comando nele. Pode ser algo simples como:

```
docker run ubuntu ls -la
total 56
drwxr-xr-x   1 root root 4096 Nov 29 12:58 .
drwxr-xr-x   1 root root 4096 Nov 29 12:58 ..
-rwxr-xr-x   1 root root    0 Nov 29 12:58 .dockerenv
lrwxrwxrwx   1 root root    7 Nov 30  2022 bin -> usr/bin
drwxr-xr-x   2 root root 4096 Apr 18  2022 boot
drwxr-xr-x   5 root root  340 Nov 29 12:58 dev
drwxr-xr-x   1 root root 4096 Nov 29 12:58 etc
...
```

Como eu sei que esse comando executou em um contêiner e não na shell do host?

```
docker ps -a
CONTAINER ID   IMAGE     COMMAND    CREATED          STATUS
PORTS     NAMES
1c8b2d97bc49   ubuntu    "ls -la"   17 seconds ago   Exited (0) 16 seconds
ago            optimistic_davinci
```

O comando **docker run** executa um comando a partir de uma imagem e o comando **docker ps** lista os contêineres em execução. Com a opção "-a" ele lista até os contêineres desativados.

A sintaxe geral do comando **docker run** é:

```
docker run [OPÇÕES] IMAGEM[:TAG|@DIGEST] [COMANDO] [ARGUMENTOS...]
```

Vejamos alguns exemplos:

- **Executar um shell em um contêiner**:

```
docker run -it ubuntu /bin/bash
root@c3f385170a37:/# ls
bin   dev  home  lib32  libx32  mnt  proc  run   srv  tmp  var
boot  etc  lib   lib64  media   opt  root  sbin  sys  usr
root@c3f385170a37:/# exit
exit
```

Executamos o comando **bash** em um novo contêiner utilizando a imagem **ubuntu**, vinda do **docker hub**.

- **Subir um contêiner com Nginx e expor a porta 80**:

```
docker run --name mynginx1 -p 80:80 -d nginx
Unable to find image 'nginx:latest' locally
latest: Pulling from library/nginx
1f7ce2fa46ab: Pull complete
9b16c94bb686: Pull complete
9a59d19f9c5b: Pull complete
9ea27b074f71: Pull complete
c6edf33e2524: Pull complete
84b1ff10387b: Pull complete
517357831967: Pull complete
Digest:
sha256:10d1f5b58f74683ad34eb29287e07dab1e90f10af243f151bb50aa5dbb4d62ee
Status: Downloaded newer image for nginx:latest
575165a2c80ff6a45febd3766b7074f072acaeebf516d8d56d2067acae22b180

~$ docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED          STATUS
PORTS                                  NAMES
575165a2c80f   nginx     "/docker-entrypoint.…"   48 seconds ago   Up 47
seconds   0.0.0.0:80->80/tcp, :::80->80/tcp   mynginx1

~$ docker rm -f mynginx1
mynginx1
```

Rodamos um contêiner com a imagem do **nginx**, vinda do **docker hub**, expondo a porta do contêiner 80 como porta 80 do localhost. A opção "-d" significa **detached**, não necessitando ficar preso ao terminal.

Como ele não possuía a imagem no repositório local, teve que baixar do **docker hub**, que é o repositório default de imagens **docker**.

Listamos o contêiner em execução com **docker ps** e depois o eliminamos com **docker rm**. A opção "-f" força a parada do contêiner.

CMD e Entrypoint

Vamos analisar o comando abaixo:

```
docker run --name mynginx1 -p 80:80 -d nginx
```

- "--name": Nome a ser aplicado ao contêiner.
- "-p 80:80": Mapeamento da porta do contêiner para a porta do localhost.
- "-d": Detached.
- "nginx": Nome da imagem.

Cadê o comando? Como o Docker sabe qual comando executar? Como ele sabe que tem que subir o **Nginx** no contêiner? Isso é determinado dentro da imagem. Quando criamos um **Dockerfile** para gerar uma imagem, expecificamos qual é o comando a ser executado quando um contêiner for criado a partir dela.

Veremos isso e maior detalhe quando falarmos da sintaxe do **Dockerfile**. Por enquanto, basta saber que há duas maneiras de fazer isso:

- Commando "CMD": Especifica um comando a ser executado quando um contêiner for criado. Pode ser substituído por um comando fornecido no **docker run**.
- Comando "ENTRYPOINT": Idem, mas não pode ser substituído.

Parar, reiniciar e remover

Vamos subir novamente o **nginx**:

```
docker run --name mynginx1 -p 80:80 -d nginx
```

Verificando se ele está rodando:

```
docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED         STATUS
PORTS                               NAMES
dc8f918a3c75   nginx     "/docker-entrypoint.…"   3 seconds ago   Up 3
seconds   0.0.0.0:80->80/tcp, :::80->80/tcp   mynginx1
```

Podemos até obter a página inicial do **nginx**:

```
curl -i http://localhost
HTTP/1.1 200 OK
Server: nginx/1.25.3
Date: Wed, 29 Nov 2023 13:21:33 GMT
```

```
Content-Type: text/html
Content-Length: 615
Last-Modified: Tue, 24 Oct 2023 13:46:47 GMT
Connection: keep-alive
ETag: "6537cac7-267"
Accept-Ranges: bytes

<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
...
```

Podemos parar esse contêiner com o comando **docker stop**:

```
~$ docker stop mynginx1
mynginx1

~$ docker ps
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS     NAMES

~$ docker ps -a
CONTAINER ID   IMAGE     COMMAND                  CREATED
STATUS                      PORTS     NAMES
dc8f918a3c75   nginx     "/docker-entrypoint.…"   About a minute ago
Exited (0) 6 seconds ago              mynginx1

~$ curl -i http://localhost
curl: (7) Failed to connect to localhost port 80: Connection refused
```

Quando paramos um contêiner, o processo dentro dele receberá um sinal **SIGTERM** do sistema operacional e, depois de um período, o sinal **SIGKILL**. Mas o contêiner continuará no docker e poderá ser reiniciado. Note que um contêiner parado não aparece no **docker ps**, pois é preciso utilizar o "-a". E o **curl** falhará, pois o processo não está mais escutando a porta 80.

E podemos reiniciar esse contêiner com o comando **docker start**:

```
~$ docker start mynginx1
mynginx1

~$ curl -i http://localhost
HTTP/1.1 200 OK
Server: nginx/1.25.3
Date: Wed, 29 Nov 2023 13:22:02 GMT
Content-Type: text/html
Content-Length: 615
Last-Modified: Tue, 24 Oct 2023 13:46:47 GMT
Connection: keep-alive
```

```
ETag: "6537cac7-267"
Accept-Ranges: bytes
...
```

Se você executou o contêiner sem atribuir um nome, pode utilizar o **contêiner ID** ou o nome de fantasia que o docker gera para você. Como boa prática, sempre use o "--name" e atribua um nome ao seu contêiner.

Dockerfile básico

Vamos ver uma pequena introdução à sintaxe do **Dockerfile** para criarmos uma imagem do zero. É claro que você não precisa disso, afinal de contas já existe uma imagem com **nginx** instalado. É apenas para começarmos a entender a sintaxe do **Dockerfile**.

Bom, vamos nos basear na imagem do **debian**, então nosso primeiro comando será:

```
FROM debian
```

Neste caso, estamos utilizando a imagem padrão do **linux debian** como nossa **base image**, ou seja, construiremos nossa imagem em cima dessa imagem, gerando uma nova imagem em nosso repositório. Poderia ser **ubuntu**, ou qualquer outra **distro** que possua imagem no **docker hub**.

Agora, o que precisamos fazer em todo sistema **linux** baseado em **debian**? Para começar, atualizar os pacotes e os repositórios configurados e depois instalar o **nginx**:

```
RUN apt-get update
RUN apt-get install nginx -y
```

O comando **RUN** executa instruções **shell** em uma camada gerando uma subimagem, que será utilizada na próxima etapa. Ao utilizar dois comandos RUN, tornamos o build mais lento. Uma forma mais otimizada seria:

```
RUN apt-get update && apt-get install nginx -y
```

Bom, agora, queremos mostrar uma página inicial nossa, em vez da página default do **nginx**. Eu criei um arquivo simples "index.html" com esse conteúdo:

```
<!doctype html>
<html>
  <head>
    <title>Meu site customizado!</title>
  </head>
  <body>
    <h1>Meu site costomizado criado com docker</h1>
```

```
  </body>
</html>
```

E deixei **na mesma pasta onde está o Dockerfile**! Isso é muito importante, já que a pasta onde está o Dockerfile é o **contexto** para o build. E então usei o comando **COPY** para copiar o arquivo do contexto para uma pasta na imagem:

```
COPY index.html /var/www/html/
ENV PATH=$PATH:/usr/sbin/
```

E vou colocar a pasta **sbin** no **PATH**, para poder executar o comando **nginx**.

Agora, uma coisa muito importante: Precisamos expor as portas que o **nginx** vai usar:

```
EXPOSE 80 443
```

Calma que falta pouco! Agora, vamos especificar o comando que será executado quando subirmos um contêiner com esta imagem. Vou usar o **CMD** para isto. Note que o CMD permite que sobrescrevamos o comando ao executar o **docker run**:

```
CMD ["nginx", "-g", "daemon off;"]
```

Eis o **Dockerfile** completo:

```
FROM debian
RUN apt-get update
RUN apt-get install nginx -y
COPY index.html /var/www/html/
ENV PATH=$PATH:/usr/sbin/
EXPOSE 80 443
CMD ["nginx", "-g", "daemon off;"]
```

Agora, façamos um build:

```
docker build -t mynginx1:0.0.1 .
```

Estou criando a imagem **mynginx1** com o **tag 0.0.1**. É boa prática atualizar o TAG sempre que houver uma mudança significativa na imagem, mantendo as anteriores.

E vamos subir um contêiner:

```
docker run -d --name nginx1 -p 80:80 mynginx1:0.0.1
```

Se você usar o **curl** (ou se abrir a página no navegador) poderá ver a resposta:

```
$ curl -i http://localhost
HTTP/1.1 200 OK
Server: nginx/1.22.1
Date: Thu, 30 Nov 2023 14:31:57 GMT
Content-Type: text/html
Content-Length: 161
Last-Modified: Wed, 29 Nov 2023 13:32:13 GMT
Connection: keep-alive
ETag: "65673d5d-a1"
Accept-Ranges: bytes

<!doctype html>
<html>
  <head>
    <title>Meu site customizado!</title>
  </head>
  <body>
    <h1>Meu site costomizado criado com docker</h1>
  </body>
```

Cache e progresso

Ao fazer um build, você pode querer ignorar o **cache** do Docker, re-executando comandos. Quando fazemos um build, o **docker** guarda as camadas intermediárias em um cache para agilizar o build, então ele não re-executa os comandos, pegando a imagem intermediária do cache.

Em certos casos, você precisa copiar novamente arquivos, ou precisa buscar coisas em sites remotos e precisa atualizar a imagem. Para evitar que ele use o cache basta especificar este flag no comando **docker build**:

```
--no-cache
```

E pode haver casos em que você está depurando um **Dockerfile** para resolver um problema e deseja ver os resultados intermediários dos comandos. Neste caso, use o flag abaixo no **docker build**:

```
--progress=plain
```