



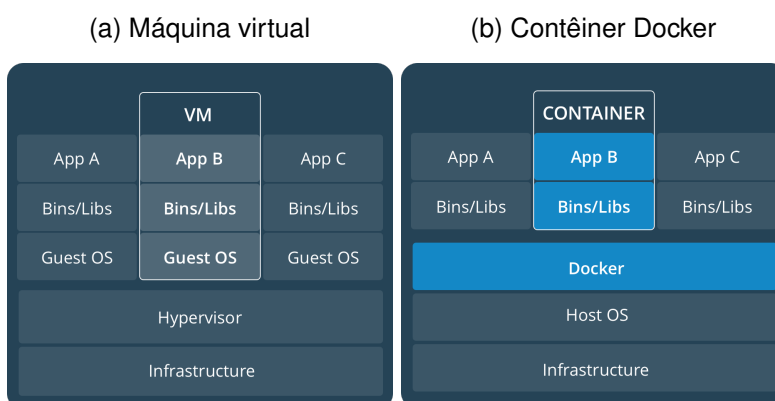
Laboratório 1: Contêineres Docker

14/04/2022

1 Conceitos sobre Docker

Na [Figura 1](#) é apresentado um comparativo entre máquinas virtuais e contêineres Docker. Uma máquina virtual requer um sistema operacional e o *hypervisor* virtualiza o acesso aos recursos de *hardware* do computador hospedeiro (*host*). Um contêiner Docker é executado de forma nativa no Linux e compartilha o kernel da máquina hospedeira com outros contêineres. Trata-se de uma solução mais leve, se comparada com a máquina virtual.

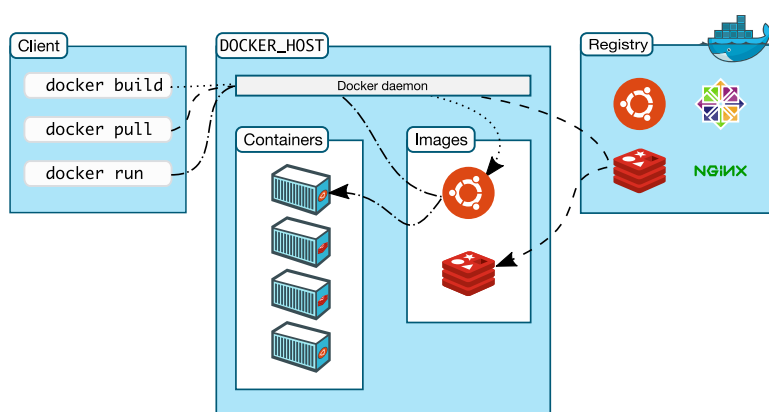
Figura 1: Contêineres vs máquinas virtuais



Fonte: docker.com


O Docker segue uma arquitetura cliente e servidor (Veja [Figura 2](#)). O cliente Docker envia comandos (usando API REST ou *sockets* UNIX) para que o *docker daemon* (servidor) execute as tarefas como construir, baixar imagens e executar contêineres. Cliente e servidor não precisam estar em execução na mesma máquina.

Figura 2: Arquitetura do Docker



Fonte: docker.com

2 Instalando docker

 Aqui serão apresentados os passos para instalar o Docker e Docker compose no Ubuntu Linux 20.04 LTS. Caso queira instalar em outra distribuição Linux, no macOS ou Windows, por favor, veja as instruções na página oficial do Docker, <https://www.docker.com/get-started>.

1. Remova as instalações (se possuir) do Docker

```
sudo apt-get remove docker docker-engine docker.io containerd runc
```

2. Atualize o repositório de pacotes e instale os pacotes necessários

```
sudo apt-get update

sudo apt-get install apt-transport-https ca-certificates curl

sudo apt-get install gnupg-agent software-properties-common
```

3. Adicione a chave GPG do Docker no apt

```
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
```

4. Adicione repositório de pacotes Docker no apt

```
sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu \
$(lsb_release -cs) stable"
```

5. Instale o Docker

```
sudo apt-get install docker-ce docker-ce-cli containerd.io
```

6. Permita que o seu usuário regular do sistema (sem ser o root) consiga usar o Docker


```
sudo usermod -aG docker login-do-teu-usuario
```

- Reinicie o computador

7. Verifique se a instalação foi concluída com sucesso

```
docker run hello-world
```

2.1 Instalando Docker Compose V2

 Abaixo são reproduzidas as principais instruções para instalar o Docker Compose no Ubuntu Linux 20.04 LTS. O documento completo está disponível em <https://docs.docker.com/compose/>. No macOS ou Windows o Docker Compose já é instalado juntamente com o Docker Desktop.

Compose é uma ferramenta para facilitar o trabalho com aplicações que precisam ser executadas com múltiplos contêineres Docker.

1. Baixe o Compose

```
mkdir -p ~/.docker/cli-plugins

curl -SL https://github.com/docker/compose/releases/download/v2.0.1/docker-compose-linux-x86_64 -o ~/.docker/cli-plugins/docker compose

chmod +x ~/.docker/cli-plugins/docker compose
```

2. Verifique se a instalação foi concluída com sucesso

```
docker compose version
```

2.2 Docker images

No Docker, imagem (*image*) consiste em um conjunto de instruções para criar um contêiner que será executado pelo Docker. Ou seja, trata-se de um modelo, somente leitura, e que permite empacotar uma aplicação de forma que a mesma possa ser executada em um ambiente de desenvolvimento ou em um ambiente de produção (em um servidor).

As imagens podem ficar localmente em seu computador ou podem ser enviadas para repositórios remotos (*registry*), como o [Docker Hub](#). No repositório remoto você poderá compartilhar as imagens com outras pessoas e também poderá baixar imagens oficiais de sistemas operacionais, aplicações ou mesmo, imagens feitas pela comunidade.

```
# Procurando por imagens no Docker Hub (veja se a coluna OFFICIAL está com OK)
docker search openjdk

# Baixando uma imagem do Docker Hub
docker pull ubuntu

# Listando as imagens armazenadas (baixadas ou geradas) em seu computador
docker image ls

# Apagando as imagens que não são usadas por qualquer contêiner
# https://docs.docker.com/config/pruning/
docker image prune

# Listando o espaço ocupado em disco por imagens, contêineres, etc.
docker system df
```

Mais informações sobre o comando `docker image` podem ser obtidas na [documentação oficial](#).

2.3 Docker container

Um contêiner (*container*) consiste de uma instância de uma imagem Docker. Sendo assim, a partir de uma imagem é possível ter vários contêineres sendo executados de forma simultânea ou não. De acordo com a documentação oficial, um contêiner consiste de: (1) uma imagem; (2) um ambiente de execução; e (3) um conjunto padronizado de instruções. Ou seja, um contêiner Docker define uma forma padronizada para entregar *software*.

```
# Executando um contêiner, modo interativo, a partir da imagem 'ubuntu' oficial
docker run -it --rm --name primeiro ubuntu

# Execute o comando abaixo para ver qual a versão do Ubuntu
cat /etc/lsb-release
```

```
# Execute exit para sair da shell do contêiner
exit

# Executando compilador javac que está na imagem oficial adoptopenjdk/openjdk11 para compilar
# um arquivo chamado Teste.java que está no diretório atual do host
docker run --rm -v `pwd`:/app -w /app adoptopenjdk/openjdk11:alpine-slim javac Teste.java
```

Abaixo alguns comandos para trabalhar com contêineres.

```
# Para listar os contêineres que estão em execução
docker ps

# Para listar os contêineres que estão em execução e os inativos
docker ps -a

# Para verificar estatísticas (uso de CPU, memória, I/O) dos contêineres ativos
docker stats

# Para finalizar a execução de um contêiner
docker stop <container ID ou nome do contêiner>

# Para remover um contêiner
docker rm <container ID ou nome do contêiner>

# Para remover todos os contêineres inativos de uma só vez
docker container prune
```

Mais informações sobre o comando `docker container` podem ser obtidas na [documentação oficial](#).

2.4 Dockerfile

Dockerfile é um arquivo texto que contém instruções para construção de uma imagem Docker. Todas as instruções no Dockerfile são executadas de forma sequencial e a forma como as instruções são colocadas nesse arquivo poderá gerar uma imagem com mais ou menos camadas. Isto é, uma Docker image consiste de um conjunto de camadas empilhadas, e com a permissão de somente leitura, as quais representam as instruções contidas em um Dockerfile.

Ao executar uma imagem e gerar um contêiner, você adiciona uma camada com permissão de escrita no topo das camadas originais da imagem (somente leitura). Toda alteração feita em um contêiner (adicionar, alterar ou apagar arquivos) fica persistida nessa camada do contêiner. Se o contêiner for apagado, todas as alterações feitas dentro dele também serão.

A imagem definida por um arquivo Dockerfile deve gerar contêineres mais efêmeros possíveis. Isto é, o contêiner poderá ser parado, destruído e depois substituído por um novo contêiner com a menor intervenção possível. Veja na [documentação oficial](#) as melhores práticas para criar um arquivo Dockerfile.

2.4.1 Exemplo de como instalar pacotes e copiar arquivo para dentro do contêiner

Criaremos uma imagem baseada no Ubuntu 20.04 LTS. Nessa instalaremos o aplicativo `figlet` e depois copiaremos o arquivo `mensagem.txt`, que está na máquina *host*, para dentro da imagem.

1. Criando diretório e arquivo com a mensagem

```
mkdir std && cd std
echo "Sistemas Distribuídos" > mensagem.txt
```

2. Criando arquivo Dockerfile

```
# Nome da imagem que servirá de base
FROM ubuntu:latest
# Comandos que serão executados durante o 'docker build'
RUN apt-get update && apt-get install -y figlet && rm -rf /var/lib/apt/lists/*
COPY mensagem.txt /
# Comando que será executado durante o 'docker run'
CMD cat /mensagem.txt | figlet
```

3. Gerando a imagem a partir do Dockerfile

```
docker build -t stdfiglet .
```

4. Executando um contêiner a partir da imagem criada

```
docker run --rm --name segundo stdfiglet
```

2.4.2 Exemplo de um servidor www simples dentro de um contêiner

1. Criando diretório e arquivo index.html

```
mkdir -p std2/www && cd std2
echo "<html><h1>Sistemas Distribuidos</h1></html>" > www/index.html
```

2. Criando arquivo Dockerfile

```
# usando uma imagem do python3.9 sobre o linux Alpine
FROM python:3.10-alpine
# Diretório de trabalho para os comandos RUN, CMD, COPY, ADD, ENTRYPOINT
WORKDIR /www
# Porta que será exposta ao host
EXPOSE 80
# Comando que será executado durante o 'docker run'
CMD ["python3", "-m", "http.server", "80"]
```

3. Gerando a imagem a partir do Dockerfile

```
docker build -t stdwebserver .
```

4. Executando um contêiner a partir da imagem criada, fazendo mapeamento do diretório local¹ www para um diretório www dentro contêiner e o redirecionamento da porta 8000 do host para a porta 80 do contêiner. O parâmetro -d inicia o contêiner no modo *detached*²

```
docker run -v `pwd`/www:/www -dp 8000:80 --rm --name web stdwebserver
```

5. Abra o navegador *web* e aponte para a URL: <http://localhost:8000>

6. Interrompa a execução do contêiner com o comando: `docker stop web`

¹Veja na [documentação oficial](#) os motivos para optar por volumes a mapeamento de diretórios.

²<https://docs.docker.com/engine/reference/run/#detached-vs-foreground>

3 Colocando contêineres em uma mesma rede

Nessa seção são apresentados exemplos retirados da [documentação oficial do Docker](#), mas especificamente exemplos sobre [como criar uma rede bridge específica](#) para um conjunto de contêineres.

```
# Listando as redes atuais
docker network ls

# Criando uma nova rede com o driver bridge
docker network create --driver bridge rede-std

# Verificando informações da rede criada, como sua faixa de IPs (subnet)
docker network inspect rede-std
```

Iremos executar 4 contêineres a partir da imagem padrão do [Alpine Linux](#) (essa possui um tamanho de apenas 5 MB). No caso, iremos executar a *shell* `ash`, que é a padrão do Alpine, porém iremos desconectar o terminal interativo e deixar a contêiner em execução no segundo plano.

```
# 2 contêineres na mesma rede: rede-std
docker run -dit --rm --name alpine1 --network rede-std alpine ash
docker run -dit --rm --name alpine2 --network rede-std alpine ash

# 1 contêiner na rede bridge padrão
docker run -dit --rm --name alpine3 alpine ash

# 1 contêiner na rede: rede-std
docker run -dit --rm --name alpine4 --network rede-std alpine ash

# Conectando o alpine4 na rede bridge padrão
docker network connect bridge alpine4
```

Entraremos na seção interativa de cada contêiner e executaremos o comando `ping` para verificar quais contêineres poderão ser atingidos.

```
# Conectando terminal interativo no contêiner alpine1
docker container attach alpine1

# executando o comando o ping para verificar que é possível atingir alpine2 e alpine4.
# Porém, não é possível atingir alpine3, pois esse está em outra rede.
ping -c 2 alpine2
ping -c 2 alpine4
ping -c 2 alpine3 # tente usar o IP do contêiner no lugar desse nome
```

Para encerrar a seção interativa no `alpine1`, porém para manter o contêiner em execução, deixe a tecla `CTRL` pressionada e depois pressione as teclas **P** e **Q** (`CTRL + P + Q`). Abra uma seção interativa com os demais contêineres e veja quais cada um desses é capaz de atingir.

O contêiner `alpine4` deverá ser capaz de atingir `alpine1` e `alpine2` tanto pelo nomes quanto por seus endereços IPs. O contêiner `alpine3` só poderá ser atingido por meio de seu endereço IP. Para descobrir qual o IP que o `alpine3` recebeu, execute o comando `docker network inspect bridge`. Aproveite e veja o IP que o `alpine4` recebeu. Agora, entre no terminal interativo do `alpine4` e veja se consegue fazer um `ping` para o endereço IP do `alpine3`. Por fim, pare todos os contêineres com o comando `docker stop <nome contêiner>`.

4 Docker Compose

Nessa seção é apresentada uma adaptação do tutorial da [documentação oficial do Docker Compose](#). Trata-se de uma simples aplicação *web* em Python que persiste em um banco de dados [Redis](#) o total

de requisições HTTP que chegaram até ela e exibe esse total.

Temos assim duas aplicações, uma aplicação Python e o banco Redis. Nesse caso, seguindo as recomendações Docker, cada aplicação deverá ser executada em um contêiner separado. O Docker Compose facilita esse trabalho com múltiplos contêineres e usaremos ele aqui para subir todo o cenário.

4.1 Criando a estrutura de diretórios e arquivos

```
mkdir -p exemplo-compose/app
cd exemplo-compose
touch app/app.py
touch Dockerfile
touch docker-compose.yml
echo -e "flask\nredis" > requirements.txt
```

Ao término você deverá ter os seguintes arquivos:

```
exemplo-compose
|-- app
|   |-- app.py
|-- Dockerfile
|-- docker-compose.yml
`-- requirements.txt
```

O conteúdo do Dockerfile é apresentado a seguir.

```
FROM python:3.10-alpine
WORKDIR /app

# Definindo variáveis de ambiente que serão usadas pela aplicação
ENV FLASK_APP=app.py
ENV FLASK_RUN_HOST=0.0.0.0
ENV PYTHONUNBUFFERED=1

# instalando pacotes com o gerenciador do Alpine Linux
RUN apk add --no-cache gcc musl-dev linux-headers

# Iremos montar um volume e não copiar (COPY app/* .) o código para dentro da imagem.
# Isso permitirá alterar o código fonte do projeto sem a necessidade de alterar a imagem.

# copiando somente o arquivo requirements.txt
COPY requirements.txt requirements.txt

# instalando pacotes python
RUN pip install -r requirements.txt

# Flask coloca um servidor web para ouvir por padrão na porta 5000
EXPOSE 5000

CMD ["flask", "run"]
```

O conteúdo do arquivo `app/app.py` é apresentado abaixo:

```
import redis
from flask import Flask

app = Flask(__name__)
db = redis.Redis(host='redis', port=6379)

def incrementa_contador():
    try:
        return db.incr('contador')
    except redis.exceptions.ConnectionError:
        return -1

@app.route('/')
def inicial():
    contador = incrementa_contador()
    return f'Valor do contador: {contador}.\n'
```

O conteúdo do arquivo `docker-compose.yml` é apresentado a seguir. Atente-se que a sintaxe da linguagem **YAML** exige que o código seja indentado com 2 espaços, caso contrário, resultará em erro. No arquivo são descritos dois serviços (`webapp` e `redis`), sendo que um deles será construído (*build*) a partir do nosso `Dockerfile` e o outro (`redis`) baixará a imagem oficial do Redis, em cima do Alpine Linux, no Docker Hub.

No serviço chamado *webapp* estamos fazendo o redirecionamento de portas do host para o contêiner e montando o volume com o diretório local `app` para dentro do contêiner no local `/app`. Também estamos passando uma variável de ambiente (*environment*) que será consumida pelo processo *flask* dentro do contêiner.

```
version: "3.8"
services:
  webapp:
    build: .
    ports:
      - "5000:5000"
    volumes:
      - ./app:/app
    environment:
      PYTHONUNBUFFERED: 1
      FLASK_ENV: development

  redis:
    image: redis:alpine
```

4.2 Executando os contêineres e consumindo a aplicação

Execute o comando `docker compose up` para subir os dois contêineres. Na primeira vez irá compilar a imagem para o serviço `webapp` e baixar a imagem do serviço `redis`. O terminal ficará recebendo todas as informações geradas pelos dois serviços.

Abra o navegador *web* no *host* e entre com a URL: <http://localhost:5000>. Você deverá ver o contador incrementar a cada atualização da página (pressione F5 para atualizar).

É possível subir somente um dos serviços, por exemplo, para subir somente o serviço `webapp`, digite: `docker compose up webapp`.

4.3 Subindo várias instâncias de um mesmo contêiner

1. Encerre todos os contêineres que estão em execução com `docker compose stop` no diretório do projeto.

2. Altere a linha 6 da listagem do docker-compose.yml substituindo o "5000:5000" por "5000".
3. Execute: `docker compose up --scale webapp=3` para subir 1 instância do serviço `redis` e 3 instâncias do serviço `webapp`.
4. Abra um outro terminal e execute o comando `docker compose ps` para ver quais portas do `host` foram mapeadas para a porta 5000 de cada contêiner.
5. No navegador *web* entre com a URL <http://localhost:<portadocontêiner>> e veja que todas as instâncias estão acessando o mesmo banco de dados Redis.

O ideal, para o usuário, seria apontar para uma única porta conhecida e não descobrir a porta que cada contêiner pegou. Isso pode ser feito por meio de um proxy, que seria mais um contêiner dentro no docker compose .yml. O serviço **NGINX** seria uma opção interessante para atuar como proxy e o servidor de DNS interno do Docker se encarregará de fazer o balanceamento de carga entre as instâncias do serviço `webapp`, pois a resolução segue a estratégia de varredura cíclica (*round robin*).

No diretório do projeto crie o arquivo `nginx.conf` com o seguinte conteúdo:

```
user  nginx;

events {
    worker_connections  500;
}

http {
    server {
        listen 4000;
        location / {
            proxy_pass http://webapp:5000;
        }
    }
}
```

Altera o arquivo `docker-compose.yml` de forma a ficar com o conteúdo como apresentado abaixo:

```
version: "3.8"
services:
  webapp:
    build: .
    ports:
      - "5000"
    volumes:
      - ./app:/app
    environment:
      PYTHONUNBUFFERED: 1
      FLASK_ENV: development

  redis:
    image: redis:alpine

  nginx:
    image: nginx:alpine
    volumes:
      - ./nginx.conf:/etc/nginx/nginx.conf:ro
    depends_on:
      - webapp
    ports:
      - "4000:4000"
```

Execute `docker compose up --scale webapp=3`, acesse a URL <http://localhost:4000>. Veja no console (onde executou o `docker compose`) qual instância de `webapp` atendeu o pedido. Pressione F5 no navegador e veja no console se outra instância atendeu o pedido.

Leitura obrigatória

- Contêineres de software
 - www.redhat.com/pt-br/topics/containers/whats-a-linux-container
 - www.redhat.com/pt-br/topics/containers/what-is-docker
 - https://docs.docker.com/develop/develop-images/dockerfile_best-practices
- Microserviços
 - www.redhat.com/pt-br/topics/microservices/what-are-microservices
- *The Twelve-Factor App*
 - https://12factor.net/pt_br/

© Este documento está licenciado sob Creative Commons “Atribuição 4.0 Internacional”.