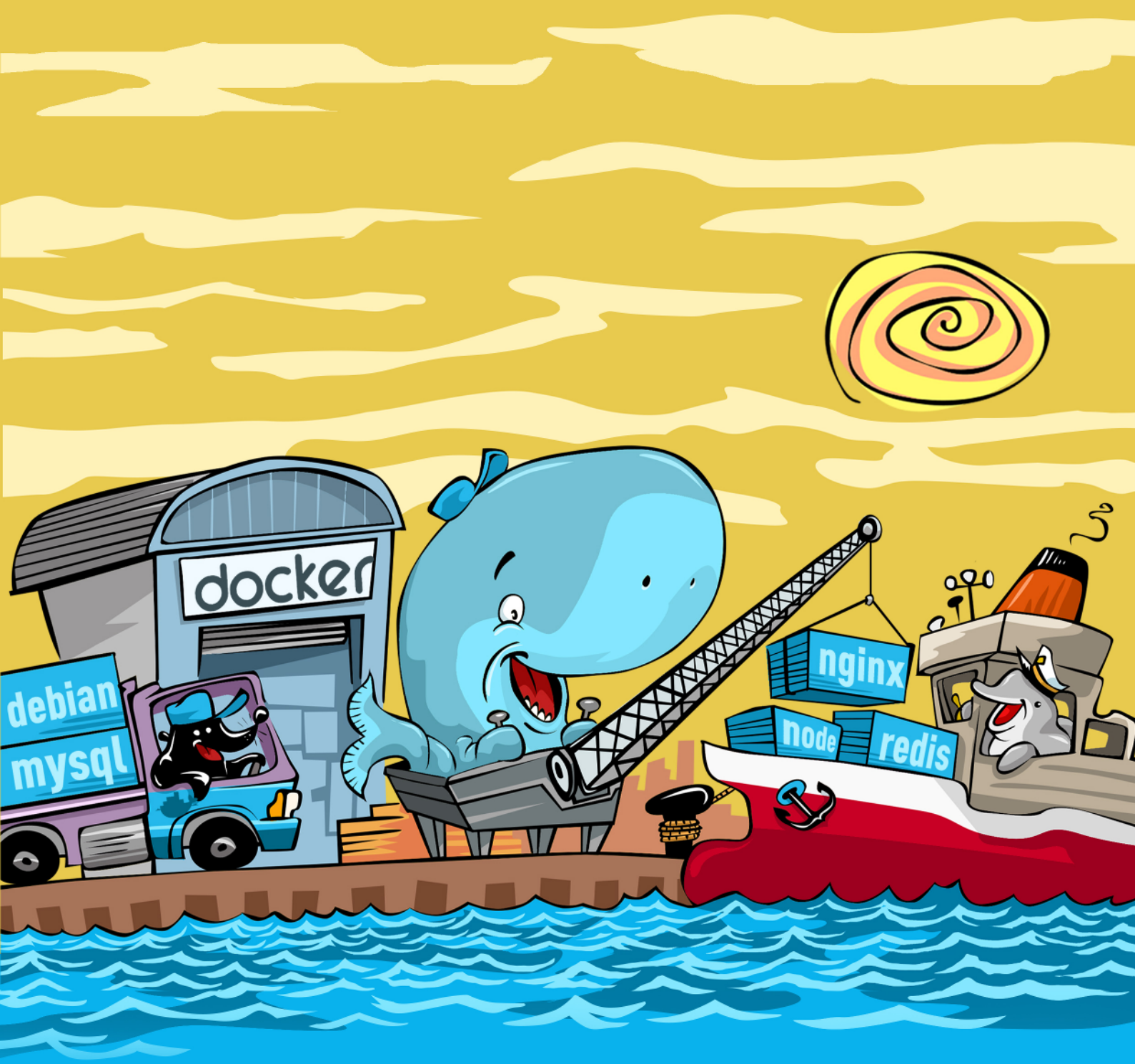


Curso de Docker

Autor: Juracy Filho

Co-autor: Leonardo Leitão

CODER .COM.BR

Índice

```
:source-highlighter: rouge
:toc: preamble
:pdf-stylesdir: /docs/commons/
:pdf-style: apostila-theme.yml
```

Sumário

Apostila do curso de Docker.

1. Conceitos

1.1. O que é Docker?

Docker

É uma ferramenta que se apoia em recursos existentes no *kernel*, inicialmente Linux, para isolar a execução de processos. As ferramentas que o Docker traz são basicamente uma camada de administração de *containers*, baseado originalmente no LXC.

Alguns isolamentos possíveis

- Limites de uso de memória
- Limites de uso de `CPU`
- Limites de uso de `I/O`
- Limites de uso de rede
- Isolamento da rede (que redes e portas são acessíveis)
- Isolamento do *file system*
- Permissões e Políticas
- Capacidades do *kernel*

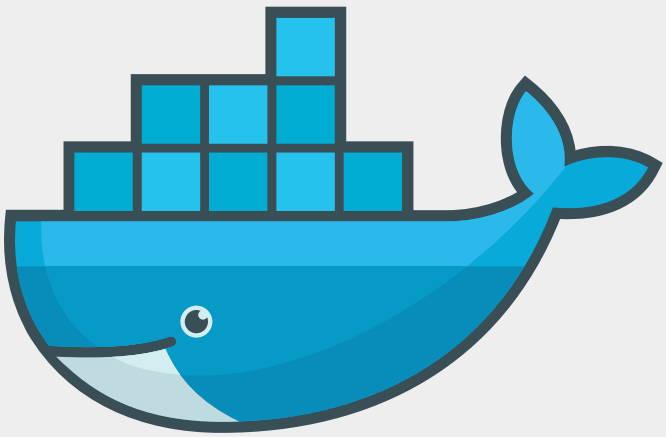
Podemos concluir dizendo que estes recursos já existiam no *kernel* a um certo tempo, o que o Docker nos trouxe foi uma maneira simples e efetiva de utiliza-los.

https://www.docker.com/what-docker

Definição oficial

Containers Docker empacotam componentes de *software* em um sistema de arquivos completo, que contêm tudo necessário para a execução: código, *runtime*, ferramentas de sistema - qualquer coisa que possa ser instalada em um servidor.

Isto garante que o *software* sempre irá executar da mesma forma, independente do seu ambiente.



1.2. Por que não uma VM?

O Docker tende a utilizar menos recursos que uma VM tradicional, um dos motivos é não precisar de uma pilha completa como vemos em Comparação VMs × Containers. O Docker utiliza o mesmo *kernel* do *host*, e ainda pode compartilhar bibliotecas.

Mesmo utilizando o mesmo *kernel* é possível utilizar outra distribuição com versões diferentes das bibliotecas e aplicativos.

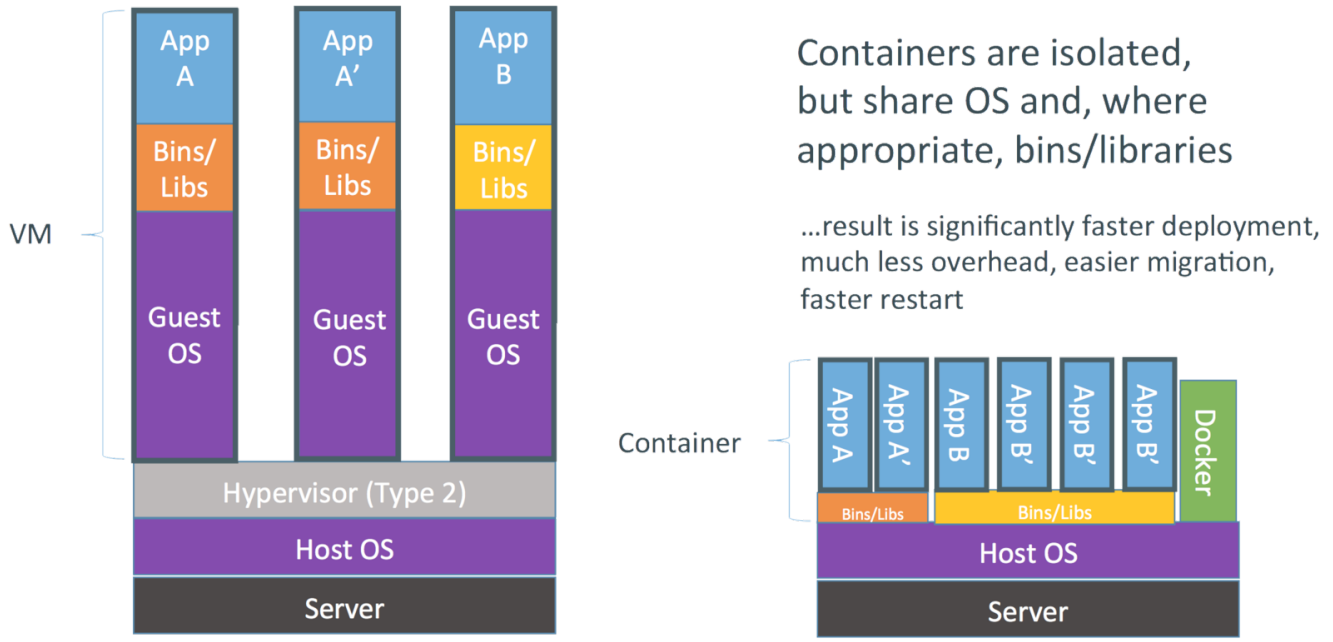


Figura 1. Comparação VMs × Containers

VM

> *Virtual Machine* (máquina virtual), recurso extremamente usado atualmente para isolamento de serviços, replicação e melhor aproveitamento do poder de processamente de uma máquina física.

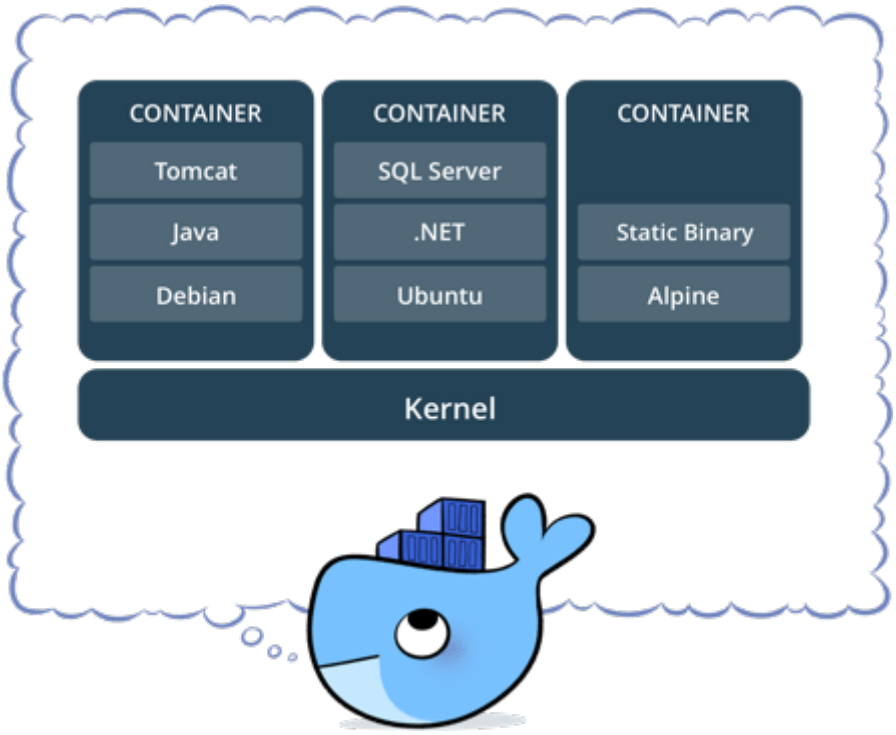
Devo trocar então minha VM por um *container*? Nem sempre, os *containers* Docker possuem algumas limitações em relação as VMs:

- **Todas as imagens são linux**, apesar do *host* poder ser qualquer SO que use ou emule um *kernel* linux, as imagens em si serão baseadas em linux.
- **Não é possível usar um *kernel* diferente do *host***, o **Docker Engine** estará executando sob uma determinada versão (ou emulação) do *kernel* linux, e não é possível executar uma versão diferente, pois as imagens não possuem *kernel*.

1.3. O que são *containers*?

Container é o nome dado para a segregação de processos no mesmo *kernel*, de forma que o processo seja isolado o máximo possível de todo o resto do ambiente.

Em termos práticos são *File Systems*, criados a partir de uma "imagem" e que podem possuir também algumas características próprias.



https://www.docker.com/what-container

1.4. O que são imagens Docker ?

Uma imagem Docker é a materialização de um modelo de um sistema de arquivos, modelo este produzido através de um processo chamado build.

Esta imagem é representada por um ou mais arquivos e pode ser armazenada em um repositório.

Docker *File Systems*

O Docker utiliza *file systems* especiais para otimizar o uso, transferência e armazenamento das imagens, *containers* e volumes.

O principal é o `AUFS`, que armazena os dados em camadas sobrepostas, e somente a camada mais recente é gravável.

- https://pt.wikipedia.org/wiki/Aufs
- https://docs.docker.com/engine/userguide/storagedriver/aufs-driver/

1.5. Arquitetura

De maneira simplificada podemos dizer que o uso mais básico do Docker consiste em:

- Ter o serviço `Docker Engine` rodando
- Ter acesso a `API Rest` do `Docker Engine`, normalmente através do `Docker Client`
- Baixar uma imagem do `Docker Registry`, normalmente do *registry* público oficial:

https://hub.docker.com

- Instanciar um *container* a partir da imagem baixada

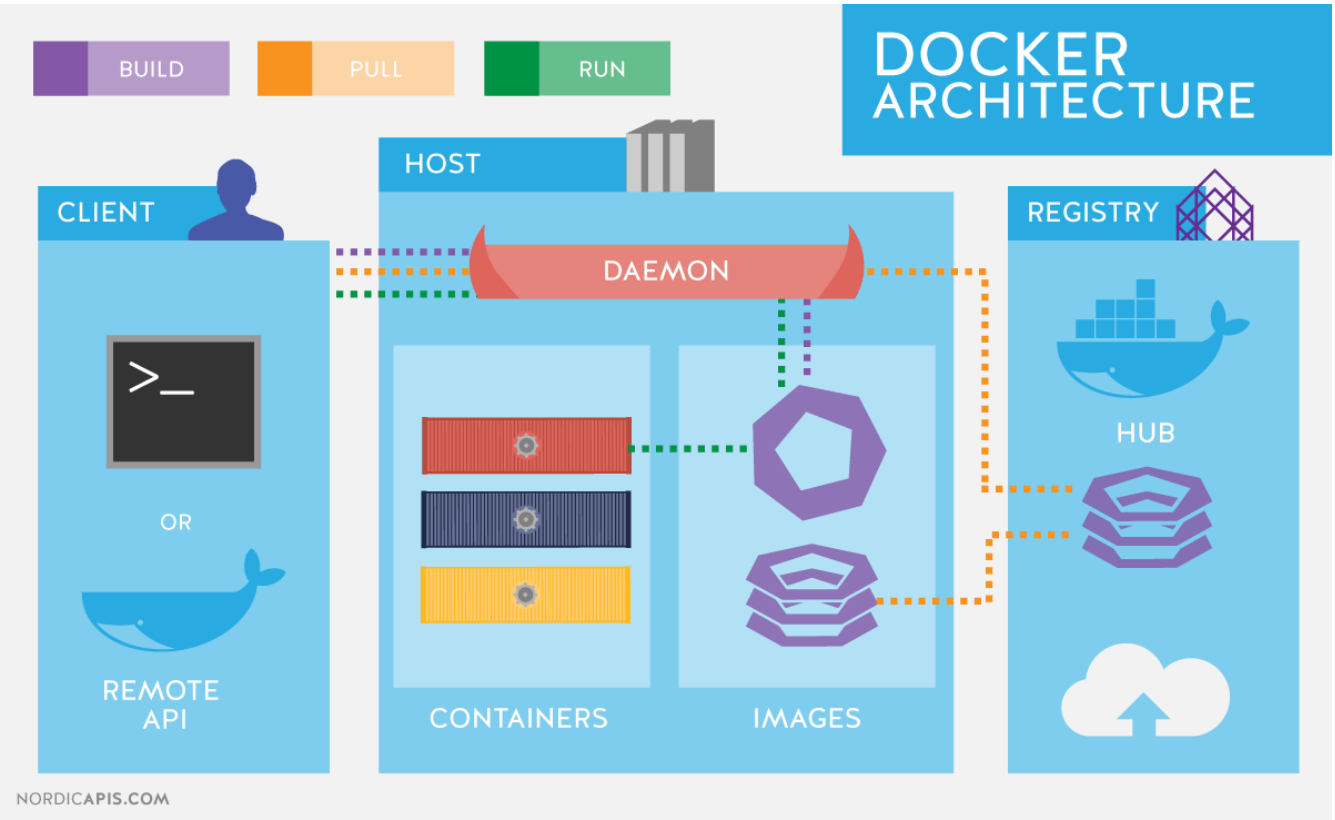


Figura 2. Arquitetura do Docker

1.6. Crescimento do Docker

A primeira versão do Docker é de 13 de março de 2013, tendo um pouco mais de 4 anos (*na epóca que este curso foi escrito*).

Nestes 4 anos ele tem se tornado cada vez mais popular e uma solução real para desenvolvedores (*manter o seu ambiente mais simples e próximo à produção*), administradores de sistema e ultimamente para uso *enterprise*, sendo avaliado pelos principais *players* do mercado uma alternativa mais econômica em relação as soluções atuais. Em sua maioria virtualização.

Figura 3. Ecossistema Docker

2. Instalação

2.1. Docker Engine e Docker Machine

- Instalação (`Linux`, `Microsoft Windows` e `MacOS`)
- Uso do Docker Machine
- Uso do Docker na nuvem, Amazon, possivelmente outros

3. Uso básico do Docker

3.1. Introdução ao Docker Client

Conforme vimos em Arquitetura, o `Docker Engine` expõe uma `API Rest` que pode ser consumida pelas mais diversas ferramentas. A ferramenta inicial fornecida com a própria *engine* é o `Docker Client`, utilitário de linha de comando.

3.2. *Hello World*: Meu Docker funciona !

Vamos confirmar o funcionamento do nosso Docker.

Exercício 1 - *Hello World*

`run.sh`

```
docker container run hello-world
```

Na documentação oficial, o passo para verificação da instalação é este *Hello World*, porém até a publicação deste curso a documentação ainda utilizava a sintaxe antiga: `docker run hello-world`

https://docs.docker.com/engine/getstarted/step_one/#step-3-verify-your-installation

Testar correto funcionamento do Docker, incluindo a recuperação de imagens e execução de *containers*.

3.3. Meu querido amigo `run`

O comando `run` é a nossa porta de entrada no Docker, agrupando diversas funcionalidades básicas, como:

- *Download* automático das imagens não encontradas: `docker image pull`
- Criação do `container`: `docker container create`
- Execução do `container`: `docker container start`
- Uso do modo interativo: `docker container exec`

A partir da versão 1.13, o Docker reestruturou toda a interface da linha de comando, para agrupar melhor os comandos por contexto.

Apesar dos comandos antigos continuarem válidos, o conselho geral é adotar a nova sintaxe.

https://blog.docker.com/2017/01/whats-new-in-docker-1-13/#h.yuluxi90h1om

Até a versão 17.03 (*corrente na publicação do curso*), ainda é possível utilizarmos a sintaxe antiga, porém precisamos pensar nela como atalhos:

```
docker pull
   docker image pull

docker create
   docker container create

docker start
   docker container start

docker exec
   docker container exec
```

3.4. Modo interativo

Podemos usar *containers* em modo interativo, isto é extremamente útil para processos experimentais, estudo dinâmico de ferramentas e de desenvolvimento.

Exemplos de Uso

- Avaliação do comportamento ou sintaxe de uma versão específica de linguagem.
- Execução temporária de uma distribuição `linux` diferente
- Execução manual de um script numa versão diferente de um interpretador que não a instalada no `host`.

Principais opções do Docker para este fim

- `docker container run -it`
- `docker container start -ai`
- `docker container exec -t`

Exercício 2 - Ferramentas diferentes

`run.sh`

```
bash --version
# GNU bash, versão 4.4.12(1)-release (x86_64-unknown-linux-gnu)

docker container run debian bash --version
# GNU bash, version 4.3.30(1)-release (x86_64-pc-linux-gnu)
```

Confirmar que o conjunto de ferramentas disponíveis em um *container* são diferentes das disponíveis no *host*.

Exercício 3 - `run` cria sempre novos *containers*

Modo interativo

```
docker container run -it debian bash
touch /curso-docker.txt
exit

docker container run -it debian bash
ls /curso-docker.txt
ls: cannot access /curso-docker.txt: No such file or directory
exit
```

Demonstrar que o `run` sempre irá instanciar um novo *container*.

Como vimos em Docker *File Systems*, o *container* e a imagem são armazenados em camadas, o processo de instanciar um *container* basicamente cria uma nova camada sobre a imagem existente, para que nessa camada as alterações sejam aplicadas.

Assim sendo o consumo de espaço em disco para instanciar novos *containers* é relativamente muito baixo.

Exercício 4 - *Containers* devem ter nomes únicos

Modo interativo

```
docker container run --help
  --memory-swappiness int   Tune container memory swappiness (0 to 100) (default
-1)
  --name string             Assign a name to the container
  --network string          Connect a container to a network (default "default")

docker container run --name mydeb -it debian bash
exit

docker container run --name mydeb -it debian bash
docker: Error response from daemon: Conflict. The container name "/mydeb" is
already in use by container
ad86038db5a3e8099d7c5a828c30520d26b0b98e35cbba46699ef25b7606b350. You have to
remove (or rename) that container to be able to reuse that name..
See 'docker run --help'.
```

Primeiro passo para reutilizar *containers*.

Exercício 5 - Reutilizar *containers*

Modo interativo

```
docker container ls
docker container ls -a

docker container start -ai mydeb
touch /curso-docker.txt
exit

docker container start -ai mydeb
ls /curso-docker.txt
/curso-docker.txt
exit
```

Demonstrar o uso do `start` em modo interativo, reutilizando um *container* previamente criado, além de confirmar que o mesmo consegue reter modificações em seu *file system*.

3.5. Cego, surdo e mudo, só que não !

Um *container* normalmente roda com o máximo de isolamento possível do `host`, este isolamento é

possível através do Docker Engine e diversas características provídas pelo *kernel*.

Mas normalmente não queremos um isolamento total, e sim um isolamento controlado, em que os recursos que o *container* terá acesso são explicitamente indicados.

Principais recursos de controle do isolamento

- Mapeamento de portas
- Mapeamento de volumes
- Copia de arquivos para o *container* ou a partir do *container*
- Comunicação entre os *containers*

3.5.1. Mapeamento de portas

É possível mapear tanto portas `TCP` como `UDP` diretamente para o *host*, permitindo acesso através de toda a rede, não necessitando ser a mesma porta do *container*. O método mais comum para este fim é o parâmetro `-p` no comando `docker container run`, o `-p` recebe um parâmetro que normalmente é composto por dois números separados por `:` (dois pontos). O primeiro é no *host* e o segundo é no *container*.

Exercício 6 - Mapear portas dos *containers*

`run.sh`

```
docker container run -p 8080:80 nginx
# acompanhar logs de acesso
# exemplo: 172.17.0.1 - - [09/Apr/2017:19:28:48 +0000] "GET / HTTP/1.1" 304 0 "-"
"Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/57.0.2987.133 Safari/537.36" "-"
# CTRL-C para sair
```

Mapeamento de portas

- Acessar a url http://localhost:8080 por um *browser*
- Receber a mensagem: **Welcome to nginx** no *browser*
- Verificar o *log* de acesso no terminal executando
- Tentar acessar a url http://localhost ou http://localhost:80
- Receber um erro do *browser*
- Parar a execução do *container*

3.5.2. Mapeamento de volumes

É possível mapear tanto diretórios no *host* como entidades especiais conhecidas como volumes para diretórios no *container*. Por enquanto vamos nos concentrar no mapeamento mais simples, uma diretório no *host* para um diretório no *container*. O método mais comum para este fim é o parâmetro `-v` no comando `docker container run`, o `-v` recebe um parâmetro que normalmente é

composto por dois caminhos absolutos separados por `:` (dois pontos). Assim como diversos outros parâmetros, o primeiro é no *host* e o segundo é no *container*.

Exercício 7 - Mapear diretórios para o *container*

`run.sh`

```
docker container run -p 8080:80 -v $(pwd)/not-found:/usr/share/nginx/html nginx
# <acessar via browser na porta 8080 e acompanhar logs de acesso>
# Exemplo: 2017/04/09 19:37:38 [error] 7#7: *1 directory index of
"/usr/share/nginx/html/" is forbidden, client: 172.17.0.1, server: localhost,
request: "GET / HTTP/1.1", host: "localhost:8080"
# CTRL-C para sair
```

`index.html`

```
<h1>Hello World</h1>
```

`run-alt.sh`

```
docker container run -p 8080:80 -v $(pwd)/html:/usr/share/nginx/html nginx
# <acessar via browser na porta 8080 e acompanhar logs de acesso>
# Exemplo: 172.17.0.1 - - [09/Apr/2017:19:40:03 +0000] "GET / HTTP/1.1" 200 21 "-
" "Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/57.0.2987.133 Safari/537.36" "-"
# CTRL-C para sair
```

Mapeamento de volumes

- Executar o `run.sh`
- Acessar a url http://localhost:8080 por um *browser*
- Receber o erro: **403 Forbidden**
- Verificar o *log* de acesso no terminal executando
- Parar a execução do *container*
- Executar o `run-alt.sh` (que mapea o diretório com o index.html)
- Tentar acessar a url http://localhost:8080
- Receber o texto: **Hello World**
- **Bônus**: Editar o `html/index.html` a partir de um editor de textos e atualizar o *browser*
- Parar a execução do *container*

3.6. Modo *daemon*

Agora sim, aonde o Docker começa a brilhar!

Antes de conhecer opções mais avançadas de compartilhamento de recursos, isolamento, etc, precisamos entender como rodar os *containers* em *background*. O parâmetro `-d` do `docker container run` indica ao Docker para iniciar o container em *background* (modo daemon).

Para entender melhor estes *containers* precisaremos conhecer um novo comando: `docker container ps`, que lista *containers* em execução.

Exercício 8 - Rodar um servidor web em *background*

`run.sh`

```
docker container run -d --name ex-daemon-basic -p 8080:80 -v $(pwd)/html:/usr/share/nginx/html nginx
# 20536baa3d861a1c8ed3a231f6f8466a442579390cdfa93b40eae2e441671a21
docker container ps
# CONTAINER ID        IMAGE               COMMAND                  CREATED              STATUS              PORTS
NAMES
# 20536baa3d86        nginx               "nginx -g 'daemon ..."   About a minute ago   Up About a minute
443/tcp, 0.0.0.0:8080->80/tcp   exercicio-07
```

Execução em background

- Levanta o container em *background*
- Tentar acessar a url http://localhost:8080 via *browser*
- Receber o texto: **Hello World**
- Verificar os *containers* em execução

Exercício 9 - Gerenciar o *container* em *background*

run.sh

```
docker container restart ex-daemon-basic
docker container ps
# STATUS: Up 5 seconds

docker container stop ex-daemon-basic
docker container ps
# <nenhum container em execução>

docker container start ex-daemon-basic
docker container ps
# CONTAINER ID        IMAGE               COMMAND                  CREATED
STATUS              PORTS                           NAMES
# 20536baa3d86        nginx               "nginx -g 'daemon ..."   15 minutes ago
Up 1 second         443/tcp, 0.0.0.0:8080->80/tcp   ex-daemon-basic
```

Reiniciar, parar e iniciar

- Reinicia o *container* e verifica que acabou de iniciar pelo tempo do *status*
- Para o *container* e através do `docker container ps` vemos que não está mais em execução
- Podemos também tentar acessar pelo *browser* a url http://localhost:8080, confirmando que não responde mais
- Inicia novamente o *container*, um observação importante é que não é mais necessário configurar os mapeamentos
- Verificar os *containers* em execução, também é possível confirmar a execução pelo *browser*.

3.7. Manipulação de *containers* em modo *daemon*

Existem diversos comandos que nos ajuda a acompanhar a execução dos *containers*, entre eles:

- `docker container ls`
- `docker container ls -a`
- `docker container inspect`
- `docker container exec`
- `docker container logs`

Vários comandos Docker possuem *aliases*, por exemplo o `container ls` tem os seguintes apelidos:

- `docker container list`
- `docker container ps`
- `docker ps` (antiga sintaxe)

3.8. Nova sintaxe do Docker Client

Agora que já vimos diversos comandos, incluindo tanto a sintaxe nova quanto a velha, podemos entender os principais motivos para esta mudança e a preferência pela nova sintaxe:

- Melhor utilização de comandos similares em contextos diferentes:
  - `docker container ls`
  - `docker image ls`
  - `docker volume ls`
- Maior clareza nos comandos:
  - `docker container rm` ao invés de `docker rm`
  - `docker image rm` ao inveś de `docker rmi`
  - `docker image ls` ao invés de `docker images`
- Mais simplicidade para criação de novos subcomandos
- Segregação da documentação e *helps* por contexto

4. Deixando de ser apenas um usuário

4.1. Introdução

Agora vamos ao conceitos necessários para deixar de apenas utilizar imagens prontas e começar a produzir nossas próprias imagens.

Exemplos de uso

- DevOps - Imagens para processos de integração contínua
- Portabilidade - Imagens de aplicação para ser utilizada em todos os ambientes (staging, prodution, etc)
- Desenvolvimento - Imagens que aproximam todos os devs do ambiente de produção e diminuem a curva de entrada de novos membros
- Imagens customizadas a partir de imagens públicas

4.2. Diferenças entre *container* e imagem

Utilizando uma analogia com `OOP`, podemos comparar um *container* a um objeto (instância), enquanto a imagem seria uma classe (modelo).

Todos os subcomandos relativos ao *container* podem ser listados através do parâmetro `--help`, bem como imagens:

- `docker container --help`
- `docker imagem --help`

4.3. Entendendo melhor as imagens

Toda imagem (bem como os *containers*) possuem um identificador único em formato *hash* usando `sha256`. Porém seu uso não é muito prático, então para simplificar isto o docker utiliza uma *tag* para identificar imagens.

A *tag* normalmente é formada por um nome, seguido de `:` dois pontos e depois uma versão. É extremamente comum utilizar uma versão chamada `latest` para representar a versão mais atual.

Exemplos de *tags* de imagens:

- `nginx:latest`
- `redis:3.2`
- `redis:3`
- `postgres:9.5`

Na prática uma *tag* é apenas um ponteiro para o *hash* da imagem, e várias *tags* podem apontar para o mesmo *hash*. Com isto é comum o uso de alguns apelidos nas tags, tomando como exemplo as imagens oficiais do redis. Existem 10 imagens e 30 *tags*.

Tags do redis

- 3.0.7, 3.0
- 3.0.7-32bit, 3.0-32bit
- 3.0.7-alpine, 3.0-alpine
- 3.0.504-windowsservercore, 3.0-windowsservercore
- 3.0.504-nanoserver, 3.0-nanoserver
- 3.2.8, 3.2, 3, latest
- 3.2.8-32bit, 3.2-32bit, 3-32bit, 32bit
- 3.2.8-alpine, 3.2-alpine, 3-alpine, alpine
- 3.2.100-windowsservercore, 3.2-windowsservercore
- 3-windowsservercore, windowsservercore
- 3.2.100-nanoserver, 3.2-nanoserver, 3-nanoserver, nanoserver

fonte: https://hub.docker.com/_/redis/

4.4. Comandos básicos no gerenciamento de imagens

Já usamos de maneira implícita o recurso de *download* de imagens docker, agora vamos entender melhor o gerenciamento de imagens.

`docker image pull <tag>`
: Baixa a imagem solicitada, este comando pode ser executado implicitamente, quando o docker precisa de uma imagem para outra operação e não consegue localiza-la no *cache* local.

`docker image ls`
: Lista todas as imagens já baixadas, é possível ainda usar a sintaxe antiga: `docker images`

`docker image rm <tag>`
: Remove uma imagem do cache local, é possível ainda usar a sintaxe antiga: `docker rmi <tag>`

`docker image inspect <tag>`
: Extrai diversas informações utilizando um formato `JSON` da imagem indicada.

`docker image tag <source> <tag>`
: Cria uma nova *tag* baseada em uma *tag* anterior ou *hash*.

`docker image build -t <tag>`
: Permite a criação de uma nova imagem, como veremos melhor em build.

`docker image push <tag>`

Permite o envio de uma imagem ou *tag* local para um *registry*.

4.5. Docker Hub × Docker Registry

Docker Registry

É uma aplicação *server side* para guardar e distribuir imagens Docker.

Docker Hub

É uma serviço de registro de imagens Docker em nuvem, que permite a associação com repositórios para *build* automatizado de imagens. Imagens marcadas como **oficiais** no Docker Hub, são criadas pela própria **Docker Inc**. E o código fonte pode ser encontrado em: https://github.com/docker-library

A linha de comando possui o comando `docker search <tag>` para procurar imagens no Docker Hub.

4.6. Construção de uma imagem

Processo para gerar uma nova imagem a partir de um arquivo de instruções. O comando `docker build` é o responsável por ler um `Dockerfile` e produzir uma nova imagem Docker.

`Dockerfile`

Nome *default* para o arquivo com instruções para o *build* de imagens Docker. Documentação do `Dockerfile` — https://docs.docker.com/engine/reference/builder

Exercício 10 - Meu primeiro *build*

```
FROM nginx:1.13
RUN echo '<h1>Hello World !</h1>' > /usr/share/nginx/html/index.html
```

`run.sh`

```
docker image build -t ex-simple-build .
docker image ls
docker container run -p 80:80 ex-simple-build # Serviço disponível em
http://localhost
# CTRL-C para sair
```

Exemplo básico de um *build* e sua execução.

O comando `build` exige a informação do diretório aonde o *build* será executado bem como aonde o arquivo de instruções se encontra.

4.7. Instruções para a preparação da imagem

`FROM`

Especifica a imagem base a ser utilizada pela nova imagem.

`LABEL`

Especifica vários metadados para a imagem como o mantenedor. A especificação do mantenedor era feita usando a instrução específica, `MAINTAINER` que foi substituída pelo `LABEL`.

`ENV`

Especifica variáveis de ambiente a serem utilizadas durante o *build*.

`ARG`

Define argumentos que poderão ser informados ao *build* através do parâmetro `--build-arg`.

Exercício 11 - Uso das instruções de preparação

`Dockerfile`

```
FROM debian
LABEL maintainer 'Juracy Filho <juracy at gmail.com>'

ARG S3_BUCKET=files
ENV S3_BUCKET=${S3_BUCKET}
```

`run.sh`

```
docker image build -t ex-build-arg .
docker container run ex-build-arg bash -c 'echo $S3_BUCKET' # Saída esperada:
files
```

`run-alt.sh`

```
docker image build --build-arg S3_BUCKET=myapp -t ex-build-arg .
docker container run ex-build-arg bash -c 'echo $S3_BUCKET' # Saída esperada:
myapp
```

Exemplo de uso das instruções: `FROM`, `LABEL`, `ARG` e `ENV`.

Os *labels* podem ser extraídos futuramente da imagem, o comando abaixo extrai o mantenedor da imagem que acabamos de criar.

```
docker image inspect --format="{{index .Config.Labels \"maintainer\"}}"
ex-build-arg
```

4.8. Instruções para povoamento da imagem

`COPY`
: Copia arquivos e diretórios para dentro da imagem.

`ADD`
: Similar ao anterior, mas com suporte extendido a `URLs`. Somente deve ser usado nos casos que a instrução `COPY` não atenda.

`RUN`
: Executa ações/comandos durante o *build* dentro da imagem.

Exercício 12 - Uso das instruções de povoamento

`Dockerfile`

```
FROM nginx:1.13
LABEL maintainer 'Juracy Filho <juracy at gmail.com>'

RUN echo '<h1>Sem conteúdo</h1>' > /usr/share/nginx/html/conteudo.html
COPY *.html /usr/share/nginx/html/
```

`index.html`

```
<a href="conteudo.html">Conteúdo do site</a>
```

`run.sh`

```
docker image build -t ex-build-copy .
docker container run -p 80:80 ex-build-copy # Serviço disponível em
http://localhost
```

Exemplo de uso das instruções: `RUN` e `COPY`.

Para entendermos melhor é necessário executa-lo uma primeira vez e navegar na porta 80. E depois copiar o arquivo `exemplo-conteudo.html` para `conteudo.html`, executa-lo novamente e verificar o resultado no *browser*.

4.9. Instruções com configuração para execução do *container*

`EXPOSE`
: Informa ao Docker que a imagem expõe determinadas portas remapeadas no *container*. A exposição da porta não é obrigatória a partir do uso do recurso de redes internas do Docker. Recurso que veremos em Coordenando múltiplos containers. Porém a exposição não só ajuda a

documentar como permite o mapeamento rápido através do parâmetro `-P` do `docker container run`.

`WORKDIR`
: Indica o diretório em que o processo principal será executado.

`ENTRYPOINT`
: Especifica o processo inicial do *container*.

`CMD`
: Indica parâmetros para o `ENTRYPOINT`.

`USER`
: Especifica qual o usuário que será usado para execução do processo no *container* (`ENTRYPOINT` e `CMD`) e instruções `RUN` durante o *build*.

`VOLUME`
: Instrui a execução do *container* a criar um volume para um diretório indicado e copia todo o conteúdo do diretório na imagem para o volume criado. Isto simplificará no futuro, processos de compartilhamento destes dados para *backup* por exemplo.

Exercício 13 - Uso das instruções para execução do *container*

`Dockerfile`

```
FROM python:3.6
LABEL maintainer 'Juracy Filho <juracy at gmail.com>'

RUN useradd www && \
    mkdir /app && \
    mkdir /log && \
    chown www /log

USER www
VOLUME /log
WORKDIR /app
EXPOSE 8000

ENTRYPOINT ["/usr/local/bin/python"]
CMD ["run.py"]
```

`index.html`

```
<p>Hello from python</p>
```

Exercício 13 - Uso das instruções para execução do *container* (*continuação*)

run.sh

```
docker build -t ex-build-dev .
docker run -it -v $(pwd):/app -p 80:8000 ex-build-dev # Serviço disponível em
http://localhost
```

run.py

```
import logging
import http.server
import socketserver
import getpass


class MyHTTPHandler(http.server.SimpleHTTPRequestHandler):
    def log_message(self, format, *args):
        logging.info("%s - - [%s] %s\n" % (
            self.client_address[0],
            self.log_date_time_string(),
            format%args))


logging.basicConfig(
    filename='/log/http-server.log',
    format='%(asctime)s - %(levelname)s - %(message)s',
    level=logging.INFO)
logging.getLogger().addHandler(logging.StreamHandler())
logging.info('inicializando...')
PORT = 8000

httpd = socketserver.TCPServer(("", PORT), MyHTTPHandler)
logging.info('escutando a porta: %s', PORT)
logging.info('usuário: %s', getpass.getuser())
httpd.serve_forever()
```

Exemplo de uso das instruções: EXPOSE, WORKDIR, ENTRYPOINT, CMD, USER e VOLUME.

Neste exemplo temos um pequeno servidor *web* atendendo na porta 8000 e exposta via instrução `EXPOSE`.

Também temos o uso do `ENTRYPOINT` e `CMD` definindo exatamente que processo será executado ao subir o *container*, podemos notar que o *container* consegue encontrar o `run.py`, por conta da instrução `WORKDIR` que define o diretório aonde o processo principal será executado.

Ao executar o *container*, uma das informações colocados no *log* (*stdout* e arquivo em disco) é o usuário corrente, e podemos notar que o processo não está rodando como `root` e sim `www`, conforme foi definido pela instrução `USER`.

Por último temos o comando `VOLUME` que instrui o docker a expor o diretório `/log` como um volume, que pode ser facilmente mapeado por outro *container*. Podemos verificar isto seguindo os seguintes passos:

- Construir a imagem e executar o *container*: `run.sh`
- Acessar a `URL` http://localhost:8000 via *browser*
- Verificar o *log* gerado na saída do *container* criado
- Criar e rodar um segundo *container* mapeando os volumes do primeiro e checar o arquivo de log: `docker run -it --volumes-from=<container criado> debian cat /log/http-server.log`
- Importante substituir a referência do `volumes_from` pelo *hash* do primeiro *container* criado
- O resultado do `cat` será o mesmo log já gerado pelo primeiro *container*

5. Coordenando múltiplos containers

5.1. Introdução

Como já foi discutido um bom *container* Docker roda apenas um serviço, tendo um único processo principal, aplicações em geral são compostos por diversos processos específicos, como por exemplo:

- Banco de dados
- Gerenciamento de filas
- Servidor Web
- Aplicação em si
- `Workers` diversos

Estes serviços devem rodar cada qual em seu *container*. Porém carrega-los um a um, não só é enfadonho como propenso a erros, sejam eles:

- Sequência de inicialização
- Esquecimento de um dos serviços
- Parada e/ou reinicialização de um ou mais serviços

Para sanar este problema temos a disposição o `docker-compose`.

5.2. Gerenciamento de *micro service*

Antes de mergulharmos nos exemplos e comandos do `docker-compose`, vamos entender melhor o que são microsserviços.

A definition of this new architectural term

> The term "Microservice Architecture" has sprung up over the last few years to describe a particular way of designing software applications as suites of independently deployable services.
>
> While there is no precise definition of this architectural style, there are certain common characteristics around organization around business capability, automated deployment, intelligence in the endpoints, and decentralized control of languages and data.
>
> — Martin Fowler, Microservices: https://martinfowler.com/articles/microservices.html

Segundo Martin Fowler, microsserviços ou arquitetura de microsserviços é uma forma de *design* de aplicações de serviços independentes distribuíveis. Entre suas principais características:

- *Deploy* automatizado
- Inteligência no uso das `API`'s

- Controle descentralizado de dados
- Independência de linguagens

5.3. Docker compose

O Docker Compose é uma ferramenta para definir e gerenciar aplicações docker com múltiplos *containers*. Neste contexto os *containers* são chamados de serviços.

6. Projeto para envio de e-mails com *workers*

Exemplo completo de uma aplicação com múltiplos serviços em **docker**.

Componentes

- Servidor *web*
- Banco de dados
- Gerenciamento de filas
- *Workers* para envio de e-mail (escalável)
- Aplicação principal

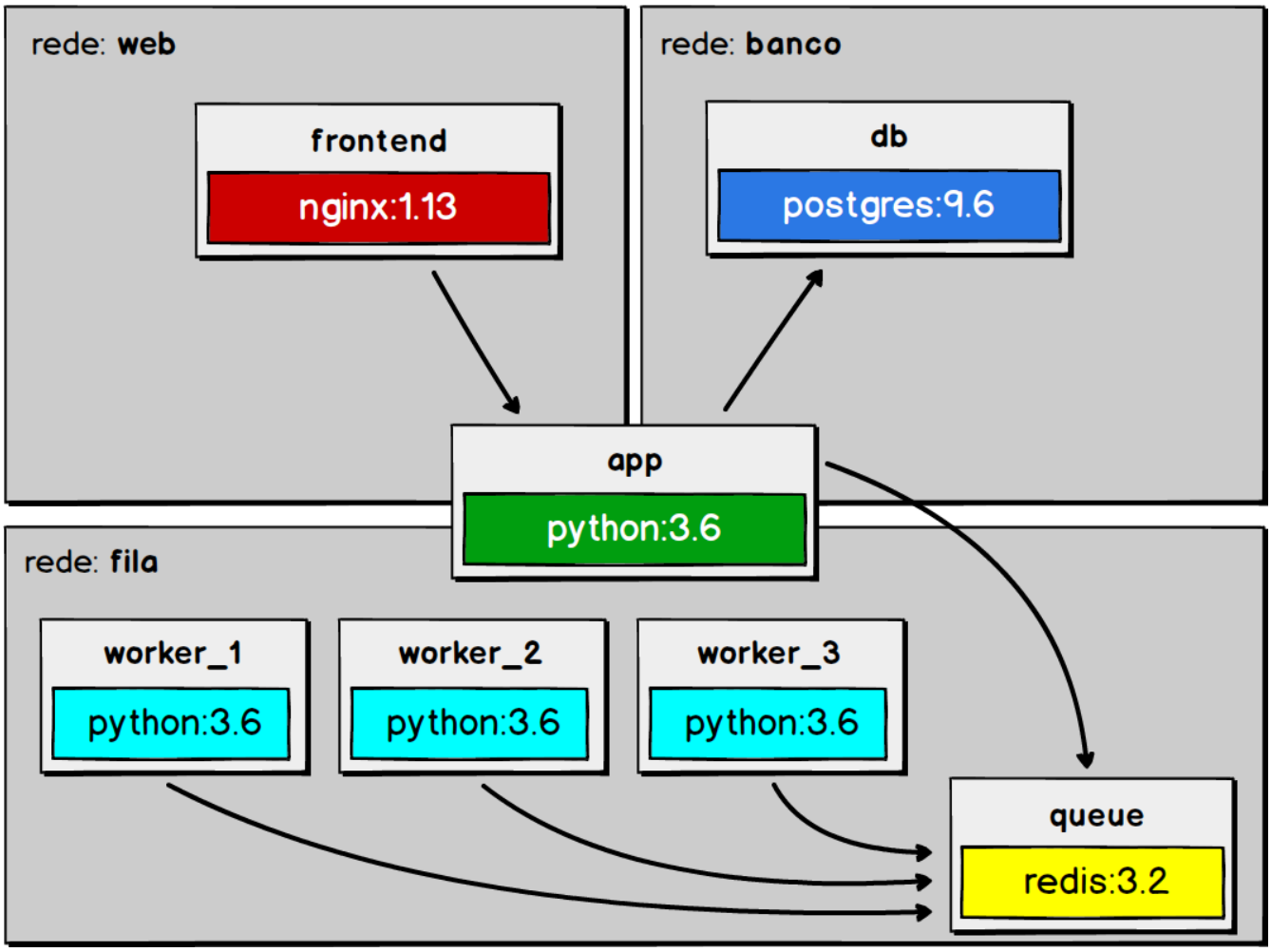


Figura 4. Diagrama final

6.1. Banco de dados

Exercício 14 - Iniciando a composição com o banco de dados

`docker-compose.yml`

```
version: '2'

services:
    db:
        image: postgres:9.6
```

`run.sh`

```
docker-compose up -d
docker-compose ps

echo "Aguardando a carga do postgres"
sleep 5

docker-compose exec db psql -U postgres -c '\l'
docker-compose down
```

Uso básico do docker-compose

- Estrutura básica do `docker-compose.yml` (format `YAML`)
- Versionamento (definindo as capacidades do `docker-compose`)
- Uso básico de serviços
- Comandos básicos: `up`, `ps`, `exec` e `down`

Neste exemplo levantamos um serviço de banco de dados e inspecionamos os `databases` disponíveis.

- `docker-compose up -d`: Levanta todos os serviços em modo *daemon*
- `docker-compose ps`: Similar ao `docker ps`, mas se limitando aos serviços indicados no `docker-compose.yml`
- `docker-compose exec`: Similar ao `docker exec`, mas utilizando como referência o nome do serviço
- `docker-compose down`: Para todos os serviços e remove os *containers*

6.2. Volumes

Exercício 15 - Usando volumes e scripts de banco de dados

docker-compose.yml

```
version: '2'

volumes:
    dados:

services:
    db:
        image: postgres:9.6
        volumes:
            # Volume dos dados
            - dados:/var/lib/postgresql/data
            # Scripts
            - ./scripts:/scripts
            - ./scripts/init.sql:/docker-entrypoint-initdb.d/init.sql
```

run.sh — *Uso do script de verificação:* scripts/check.sql

```
docker-compose exec db psql -U postgres -f /scripts/check.sql
```

scripts/init.sql

```
create database email_sender;

\c email_sender

create table emails (
  id serial not null,
  data timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP,
  assunto varchar(100) not null,
  mensagem varchar(250) not null
);
```

scripts/check.sql

```
\l
\c email_sender
\d emails
```

6.3. Front-end

Exercício 16 - Começando nossa camada de *front-end*

`docker-compose.yml` — *Apenas o serviço de front-end*

```
frontend:
    image: nginx:1.13
    volumes:
        # Site
        - ./web:/usr/share/nginx/html/
    ports:
        - 80:80
```

`run.sh`

```
docker-compose up -d
docker-compose logs -f -t
```

Exercício 16 - Começando nossa camada de *front-end* (*continuação*)

web/index.html

```
<!DOCTYPE html>
<html lang="pt-BR">
  <head>
    <meta charset="utf-8">
    <meta http-equiv="x-ua-compatible" content="ie=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1">

    <title>Email Sender</title>

    <style>
      label { display: block; }
      textarea, input { width: 400px; }
    </style>
  </head>
  <body class="container">
    <h1>E-mail Sender</h1>
    <form>
      <div>
        <label for="assunto">Assunto</label>
        <input name="assunto" type="text">
      </div>

      <div>
        <label for="mensagem">Mensagem</label>
        <textarea name="mensagem" cols="50" rows="6"></textarea>
      </div>

      <div>
        <button type="submit">Enviar !</button>
      </div>
    </form>
  </body>
</html>
```

6.4. Filas

Exercício 17 - Aplicativo para enfileirar as mensagens

`docker-compose.yml` — *Apenas o novo serviço*

```
app:
    image: python:3.6
    volumes:
        # Aplicação
        - ./app:/app
    working_dir: /app
    command: bash ./app.sh
    ports:
        - 8080:8080
```

`app/app.sh`

```
#!/bin/sh

pip install bottle==0.12.13
python -u sender.py
```

`app/sender.py`

```
from bottle import route, run, request


@route('/', method='POST')
def send():
    assunto = request.forms.get('assunto')
    mensagem = request.forms.get('mensagem')
    return 'Mensagem enfileirada ! Assunto: {} Mensagem: {}'.format(
        assunto, mensagem)


if __name__ == '__main__':
    run(host='0.0.0.0', port=8080, debug=True)
```

`web/index.html` — *Ajustes na tag* `form`

```
    <h1>E-mail Sender</h1>
    <form action="http://localhost:8080" method="POST"> ①
      <div>
        <label for="assunto">Assunto</label>
```

① Novos atributos `action` e `method` para a *tag* `form`

6.5. Proxy reverso

Exercício 18 - Configurando um proxy reverso

`docker-compose.yml` — *Injeção da configuração do* `nginx`

```
app:
    image: python:3.6
    volumes:
        # Aplicação
        - ./app:/app
    working_dir: /app
    command: bash ./app.sh ①

frontend:
    image: nginx:1.13
    volumes:
        # Site
        - ./web:/usr/share/nginx/html/
        # Configuração do proxy reverso
        - ./nginx/default.conf:/etc/nginx/conf.d/default.conf
    ports:
        - 80:80
```

① Removido a publicação da porta 8080, já que agora é feito pelo *proxy* reverso

`nginx/default.conf`

```
server {
    listen       80;
    server_name  localhost;

    location / {
        root   /usr/share/nginx/html;
        index  index.html index.htm;
    }

    error_page   500 502 503 504  /50x.html;
    location = /50x.html {
        root   /usr/share/nginx/html;
    }

    location /api { ①
        proxy_pass http://app:8080/;
        proxy_http_version 1.1;
    }
}
```

① Mapeamento da `url` `/api` para a porta 8080 do serviço `db`

Exercício 18 - Configurando um proxy reverso (*continuação*)

`web/index.html` — *Mudança da* `URL` *do serviço*

```
    <h1>E-mail Sender</h1>
    <form action="http://localhost/api" method="POST"> ①
      <div>
        <label for="assunto">Assunto</label>
```

① Atributo `action` ajustado para `http://localhost/api`

6.6. Redes

Exercício 19 - Redes, dependência e banco de dados

`docker-compose.yml` — *Definição das redes*

```
networks: ①
    banco:
    web:

services:
    db:
        image: postgres:9.6
        volumes:
            # Volume dos dados
            - dados:/var/lib/postgresql/data
            # Scripts
            - ./scripts:/scripts
            - ./scripts/init.sql:/docker-entrypoint-initdb.d/init.sql
        networks: ②
            - banco

    app:
        image: python:3.6
        volumes:
            # Aplicação
            - ./app:/app
        working_dir: /app
        command: bash ./app.sh
        networks: ③
            - banco
            - web
        depends_on: ④
            - db
```

① Definição das redes

② Indicação da rede o serviço `db`

③ Indicação da rede o serviço `app`

④ Dependência para o serviço `app`

Exercício 19 - Redes, dependência e banco de dados (*continuação*)

`docker-compose.yml` — *Rede e dependência para o serviço* `frontend`

```
frontend:
    image: nginx:1.13
    volumes:
        # Site
        - ./web:/usr/share/nginx/html/
        # Configuração do proxy reverso
        - ./nginx/default.conf:/etc/nginx/conf.d/default.conf
    ports:
        - 80:80
    networks:
        - web
    depends_on:
        - app
```

`app/app.sh`

```
#!/bin/sh

pip install bottle==0.12.13 psycopg2==2.7.1
python -u sender.py
```

Exercício 19 - Redes, dependência e banco de dados (*continuação*)

app/sender.py

```
import psycopg2
from bottle import route, run, request

DSN = 'dbname=email_sender user=postgres host=db'
SQL = 'INSERT INTO emails (assunto, mensagem) VALUES (%s, %s)'


def register_message(assunto, mensagem):
    conn = psycopg2.connect(DSN)
    cur = conn.cursor()
    cur.execute(SQL, (assunto, mensagem))
    conn.commit()
    cur.close()
    conn.close()

    print('Mensagem registrada !')


@route('/', method='POST')
def send():
    assunto = request.forms.get('assunto')
    mensagem = request.forms.get('mensagem')
    register_message(assunto, mensagem)
    return 'Mensagem enfileirada ! Assunto: {} Mensagem: {}'.format(
        assunto, mensagem)


if __name__ == '__main__':
    run(host='0.0.0.0', port=8080, debug=True)
```

6.7. *Workers*

Exercício 20 - Fila e workers

`docker-compose.yml` — *Nova rede*

```
networks:
    banco:
    web:
    fila: ①
# -------- <recorte>
    app:
        image: python:3.6
        volumes:
            # Aplicação
            - ./app:/app
        working_dir: /app
        command: bash ./app.sh
        networks:
            - banco
            - web
            - fila ②
        depends_on:
            - db
            - queue

    worker: ③
        image: python:3.6
        volumes:
            # Worker
            - ./worker:/worker
        working_dir: /worker
        command: ./app.sh
        networks:
            - fila
        depends_on:
            - queue
            - app
```

① Nova rede: `fila`

② Inclusão da nova rede no serviço `app`

③ Novo serviço `worker`

Exercício 20 - Fila e workers (*continuação*)

`docker-compose.yml` — *Nova rede*

```
    queue: ①
        image: redis:3.2
        networks:
            - fila
```

① Novo serviço `queue`

`app/app.sh` — *Nova dependência*

```
#!/bin/sh

pip install bottle==0.12.13 psycopg2==2.7.1 redis==2.10.5
python -u sender.py
```

Exercício 20 - Fila e workers (*continuação*)

app/sender.py — *Revisão geral e enfileiramento*

```
import psycopg2
import redis
import json
from bottle import Bottle, request


class Sender(Bottle):
    def __init__(self):
        super().__init__()
        self.route('/', method='POST', callback=self.send)
        self.fila = redis.StrictRedis(host='queue', port=6379, db=0)

        DSN = 'dbname=email_sender user=postgres host=db'
        self.conn = psycopg2.connect(DSN)

    def register_message(self, assunto, mensagem):
        SQL = 'INSERT INTO emails (assunto, mensagem) VALUES (%s, %s)'

        cur = self.conn.cursor()
        cur.execute(SQL, (assunto, mensagem))
        self.conn.commit()
        cur.close()

        msg = {'assunto': assunto, 'mensagem': mensagem}
        self.fila.rpush('sender', json.dumps(msg))
        print('Mensagem registrada !')

    def send(self):
        assunto = request.forms.get('assunto')
        mensagem = request.forms.get('mensagem')
        self.register_message(assunto, mensagem)
        return 'Mensagem enfileirada ! Assunto: {} Mensagem: {}'.format(
            assunto, mensagem)


if __name__ == '__main__':
    sender = Sender()
    sender.run(host='0.0.0.0', port=8080, debug=True)
```

Exercício 20 - Fila e workers (*continuação*)

worker/app.sh

```
#!/bin/sh

pip install redis==2.10.5
python -u worker.py
```

worker/worker.py

```
import redis
import json
from time import sleep
from random import randint


if __name__ == '__main__':
    r = redis.Redis(host='queue', port=6379, db=0)

    while True:
        mensagem = json.loads(r.blpop('sender')[1])
        print('Mandando a mensagem:', mensagem['assunto'])
        sleep(randint(15, 45))
        print('Mensagem', mensagem['assunto'], 'enviada')
```

6.8. Múltiplas instâncias

Exercício 21 - Escalar é preciso...

`docker-compose.yml` — *Uso de imagem customizada para o worker*

```
worker:
    build: worker ①
    volumes:
        # Worker
        - ./worker:/worker
    working_dir: /worker
    command: worker.py
    networks:
        - fila
    depends_on:
        - queue
        - app
```

① Uso da instrução `build` no lugar do `image` para indicar a necessidade de executar um `build`, neste caso do arquivo `worker/Dockerfile`

`run.sh` — *Escalando o worker e especializando o log*

```
docker-compose up -d
docker-compose scale worker=3 & ①
docker-compose logs -f -t worker ②
```

① Novo sub-comando: `scale` para levantar *n* instâncias de um serviço

② Queremos agora apenas acompanhar o *log* do serviço `worker`

`worker/Dockerfile` — *Build do worker*

```
FROM python:3.6
LABEL maintainer 'Juracy Filho <juracy at gmail.com>'

ENV PYTHONUNBUFFERED 1
RUN pip install redis==2.10.5

ENTRYPOINT ["/usr/local/bin/python"]
```

Exercício 21 - Escalar é preciso... (*continuação*)

`worker/worker.py` — *Mensagem de inicialização*

```
if __name__ == '__main__':
    r = redis.Redis(host='queue', port=6379, db=0)
    print('Aguardando mensagens ...') ①

    while True:
        mensagem = json.loads(r.blpop('sender')[1])
        print('Mandando a mensagem:', mensagem['assunto'])
        sleep(randint(15, 45))
        print('Mensagem', mensagem['assunto'], 'enviada')
```

① Mensagem de inicialização, para facilitar o acompanhamento do *log*

6.9. Boas práticas — Variáveis de ambiente

Exercício 22 - 12 Factors

`docker-compose.yml` — *Uso de variável de ambiente*

```
app:
    image: python:3.6
    environment:
        - DB_NAME=email_sender ①
    volumes:
        # Aplicação
        - ./app:/app
    working_dir: /app
    command: bash ./app.sh
    networks:
        - banco
        - web
        - fila
    depends_on:
        - db
        - queue
```

① Injeção no *container* de uma variável de ambiente chamada `DB_NAME`

`run.sh` — *Log do* `worker` *e da* `app`

```
docker-compose up -d
docker-compose scale worker=3 &
docker-compose logs -f -t worker app ①
```

① Acompanhar o *log* também do serviço `app`

Exercício 22 - 12 Factors (*continuação*)

`app/sender.py` — *Uso da variável* `REDIS_HOST` *e* `DB_*`

```
import os ①
import psycopg2
import redis
import json
from bottle import Bottle, request


class Sender(Bottle):
    def __init__(self):
        super().__init__()

        db_host = os.getenv('DB_HOST', 'db') ②
        db_user = os.getenv('DB_USER', 'postgres')
        db_name = os.getenv('DB_NAME', 'sender')
        dsn = f'dbname={db_name} user={db_user} host={db_host}' ③
        self.conn = psycopg2.connect(dsn)

        redis_host = os.getenv('REDIS_HOST', 'queue') ④
        self.fila = redis.StrictRedis(host=redis_host, port=6379, db=0)

        self.route('/', method='POST', callback=self.send)
```

① Módulo `os` é responsável por ler variáveis de ambiente

② Leitura de diversas variáveis de ambiente (incluindo o `DB_NAME`)

③ Uso das variáveis de ambiente para conexão com o banco de dados

④ Uso da variável de ambiente `REDIS_HOST` para o `redis`

Exercício 22 - 12 Factors (*continuação*)

`worker/worker.py` — *Uso da variável* `REDIS_HOST`

```
import redis
import json
import os ①
from time import sleep
from random import randint


if __name__ == '__main__':
    redis_host = os.getenv('REDIS_HOST', 'queue') ②
    r = redis.Redis(host=redis_host, port=6379, db=0)
    print('Aguardando mensagens ...')

    while True:
        mensagem = json.loads(r.blpop('sender')[1])
        print('Mandando a mensagem:', mensagem['assunto'])
        sleep(randint(15, 45))
        print('Mensagem', mensagem['assunto'], 'enviada')
```

① Módulo `os` é responsável por ler variáveis de ambiente

② Uso da variável de ambiente `REDIS_HOST` para o `redis`

6.10. Override

Exercício 23 - Sobreescrevendo localmente

`docker-compose.override.yml` — *Uso de variável de ambiente*

```
version: '2'

services:
    app:
        environment:
            - DB_NAME=email_sender
```

Appendix A: Tabela de Exercícios

Glossário

`LXC`

Conjunto de ferramentas, `API's` e bibliotecas para o uso de *containers* no Linux. Melhorando as capacidades de uma ferramenta chamada `chroot`, conhecida por "enjaular" processos em um sub-diretório.