

container

um ambiente isolado, com um sistema de gerenciamento isolado tbm

mapeamento de portas

- É possível mapear tanto portas TCP como UDP diretamente para o host

exemplo **8080:80** onde:

- **8080:** é o host (onde iremos acessar pelo navegador)
- **80** é a porta usada no container

exemplo do comando: **docker container run -p 8080:80 nginx**

Docker Hub vs Docker Registry

- Docker registre - sistema server side para registrar e distribuir as imagens que criamos
- Docker Hub - É um 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.

Dockerfile

- O comando build requer a informação do diretório onde o build será executado bem como onde o arquivo de instruções se encontra.
- O comando **docker build** é o responsável por ler um Dockerfile e produzir uma nova imagem Docker.
- Uma boa dica é sempre fazer as alterações no final, deixar aquela layer que voce muda sempre no final para que quando o docker for rebuildar ele pegue desse ponto que foi alterado (por conta do cash) e rebuild o final ao invés de ter que rebuildar tudo pq tem muitos pontos modificados.
- **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.
- **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 21 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 u
- **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.

A. adicionando arquivo do host - repositório para dentro da imagem

criou um arquivo index.html de exemplo ` CONTEUDO DO SITE`

no docker file

FROM nginx:latest

LABEL maintainer 'aluno idilene'

RUN echo '`<h1> sem conteudo </h1>`' > /usr/share/nginx/html/conteudo.html

(esse html conteudo.html está sendo criado nesse arquivo do dockfile, ou seja ele esta escrevendo '`<h1> sem conteudo </h1>`' dentro desse arquivo do nginx)

COPY *.html /usr/share/nginx/html/ (esse comando copy é o justamente a copia que estamos fazendo do arquivo que está na raiz do container e que vamos mandar para o docker para que ele faça parte da pasta mencionada que é o /usr/share/nginx/html/)

se a gente der um ls no terminal wsl na pasta onde está esse repositório poderemos ver o arquivo index.ls

após isso é so dar um docker build -t <nome da tag (imagem)> .
docker container run -p 80:80 <nome da imagem>

se entrarmos no navegador com o localhost poderemos ver um link escrito **conteudo do site** e clicando nele nos levará para uma outra página escrito **sem conteudo**

B. Usando o volume

exemplo do dockerfile

FROM python:3.6

LABEL maintainer 'Juracy Filho '

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

USER www

VOLUME /log

WORKDIR /app

EXPOSE 8000

ENTRYPOINT ["/usr/local/bin/python"] (caminho no linux)

CMD ["run.py"]

rodando os seguintes códigos

docker build -t ex-build-dev .

```
docker run -it -v <diretorio atual>:/app -p 80:8000 --name python-server ex-build-dev  
# Serviço disponível em http://localhost
```

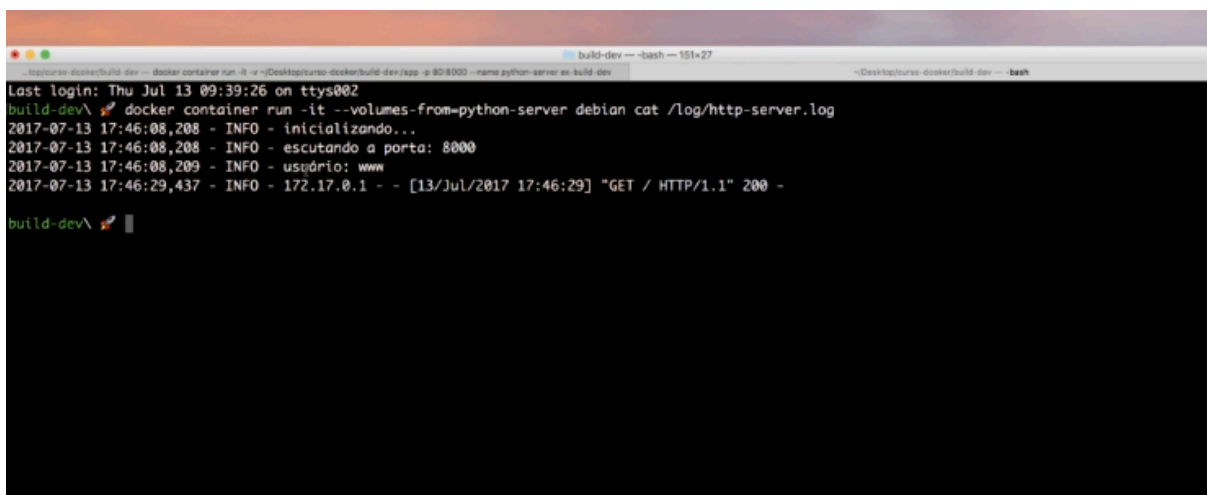
explicando o que esse dockfile está fazendo

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 nesse caso é o python-server>debian cat /log/http-server.log**

esse caminho /log/http-server.log é o caminho do servidor criado no python para registrar esses logs

- 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



```
build-dev -- bash -- 151x27
Last login: Thu Jul 13 09:39:26 on ttys002
build-dev\ docker container run -it --volumes-from=python-server debian cat /log/http-server.log
2017-07-13 17:46:08,208 - INFO - inicializando...
2017-07-13 17:46:08,208 - INFO - escutando a porta: 8000
2017-07-13 17:46:08,209 - INFO - usuário: www
2017-07-13 17:46:29,437 - INFO - 172.17.0.1 - - [13/Jul/2017 17:46:29] "GET / HTTP/1.1" 200 -
build-dev\
```

passo a passo

1. Cria o arquivo docker file
2. roda o build : **docker image build -t <nome da tag .**
3. ver se a imagem foi gerada: **docker image ls**
4. para rodar a imagem: **docker container run -p 80:80 <nome da imagem>**

comandos

—COMPOSE: SUBIR CONTAINER—

- **docker compose up -d** (-d para desprender o terminal - modo interativo)
- **docker container run -it** (it modo interativo, desprende o terminal)

—LISTAR CONTAINERS / IMAGE / VOLUME—

- **docker image ls** ou **docker images**
- **docker container ls** ou **docker ps**
- **docker ps --no-trunc** - tira todas as reticencias, sem truncar ou cortar
- **docker volume ls**

—APAGAR CONTAINER / IMAGEM—

- **docker rm <id>** - remover container
- **docker image rm <id>** - remover imagem
- **docker image rm <nome da imagem> <nome da imagem>** : apagar mais de uma ao mesmo tempo separado por imagem
- **docker-compose down -v** - apaga os volumes

—STOP—

- **docker stop 9** - digamos que so tem 2 containers rodando e somente um deles o id começa com 9, você consegue referenciar somente usando o numero 9
- **docker stop <id>**
- **docker stop <nome do container>**

—LOGS—

- **docker compose logs <nome do container ou serviço>**
- **docker inspect <nome do container>** retorna varias informações sobre o container em formato json
- **docker inspect --format "{{json .State.Health }}" sqlserver | jq** (o jq ajuda a retornar os logs formatados, trazendo nesse caso os logs do state health)
- **docker-compose logs -f -t <nome do serviço>**

—TAG—

- **docker image tag <nome da imagem de origem> <nome da tag>** : exemplo
docker image tag redis:latest doc3r-radis

—IMAGE—

- **docker image pull** - baixar imagem do docker hub
- **docker image build / docker build**: gerar a imagem lendo um arquivo Dockerfile
- **docker image push** - para publicar o docker ou no hub ou na empresa

—BUILD—

- **docker image build -t <nome da tag>**

—EXEC—

o Comando exec é utilizado para rodar um comando no container exemplo

- **docker container exec -it container2 ifconfig**
- **docker-compose exec frontend cat /usr/share/nginx/html/index.html** - abre o arquivo index.html no terminal
- **docker-compose exec <nome do servico / container> psql -U postgres -d <banco> -c "<comando sql>"**

exemplo: **docker-compose exec db psql -U postgres -d email_sender -c "select * from emails"** → talvez precise add o sudo no inicio do comando

subindo imagem para o docker hub

após criar uma conta no docker hub

no terminal digite

docker image tag <nome da imagem> <nome do usuario>/<nome do repo>
exemplo

docker image tag ex-simple-build dockercod3r/simple-build:1.0

quando a gente der o comando **docker ls** deve aparecer o repositório

```
ex-build-arg    latest      28a1fcf35a42    16 hours ago    100MB
<none>          <none>      804c43fa0c7f    16 hours ago    100MB
ex-simple-build latest      674744d52c48    16 hours ago    108MB
node            8.1        d2ea9785e1cb    44 hours ago    665MB
nginx           latest     e4e6d42c70b3    47 hours ago    108MB
python          3.6        955d0c3b1bb2    5 days ago      684MB
mongo           3.4        57c67caab3d8    6 days ago      359MB
alpine           latest     7328f6f8b418    2 weeks ago     3.97MB
debian          latest     a2ff708b7413    3 weeks ago     100MB
hello-world     latest     1815c82652c0    4 weeks ago     1.84kB
build-dev\ ➤ docker image tag ex-simple-build dockercod3r/simple-build:1.0
build-dev\ ➤ docker image ls
REPOSITORY      TAG          IMAGE ID       CREATED        SIZE
ex-build-dev    latest      6a0d5e0b759a   34 minutes ago 684MB
ex-build-copy   latest      18eba9b20439   16 hours ago  108MB
ex-build-arg    latest      28a1fcf35a42   16 hours ago  100MB
<none>          <none>      804c43fa0c7f   16 hours ago  100MB
dockercod3r/simple-build 1.0         674744d52c48   16 hours ago  108MB
ex-simple-build latest      674744d52c48   16 hours ago  108MB
node            8.1        d2ea9785e1cb   44 hours ago  665MB
nginx           latest     e4e6d42c70b3   47 hours ago  108MB
python          3.6        955d0c3b1bb2    5 days ago    684MB
mongo           3.4        57c67caab3d8    6 days ago    359MB
alpine           latest     7328f6f8b418    2 weeks ago   3.97MB
debian          latest     a2ff708b7413    3 weeks ago   100MB
hello-world     latest     1815c82652c0    4 weeks ago   1.84kB
build-dev\ ➤
```

para fazer login

docker login --username=<usuario>

depois ele vai pedir coloca a senha

–fazendo o push da imagem–

docker image push <nome da imagem que ficou que é composta pelo nome do usuario e o nome da imagem>

docker image push dockercod3r/simple-build:1.0

redes

existe 4 tipos de redes:

- none network
- bridge (padrão)
- host network
- overlay network (swarm) para fazer um cluster

1. None network

se eu tenho varios containers, esses containers ficam isolados, sem acesso entre si e sem acesso ao mundo exterior. so conseguiremos acessar via terminal

- seguro

docker container run -d --net none debian

-rm -> remove o container assim que ele terminar de rodar

2. Bridge network

cada container tem sua propria interface mas o bridge faz uma camada de isolamento com a rede do host com a camada de rede de cada container

a faixa de rede do bridge é 172.17.0.0 - o que quer dizer que esses dois zeros são os ips da maquina e que podem chegar ate 255 cada

inclusive conseguimos inspecionar o bridge network com o comando
docker network inspect bridge



```
[{"Name": "bridge",
  "Id": "27a1fab1b2e799e9ffada8286893757d86dee899f825a4a741d3098e026f24bd",
  "Created": "2024-04-03T12:57:04.401539205-03:00",
  "Scope": "local",
  "Driver": "bridge",
  "EnableIPv6": false,
  "IPAM": {
    "Driver": "default",
    "Options": null,
    "Config": [
      {
        "Subnet": "172.17.0.0/16",
        "Gateway": "172.17.0.1"
      }
    ]
  },
  "Internal": false,
  "Attachable": false,
  "Ingress": false,
  "ConfigFrom": {
    "Network": ""
  },
  "ConfigOnly": false,
  "Containers": {},
  "Options": {
    "com.docker.network.bridge.default_bridge": "true",
    "com.docker.network.bridge.enable_icc": "true",
    "com.docker.network.bridge.enable_ip_masquerade": "true",
    "com.docker.network.bridge.host_binding_ipv4": "0.0.0.0",
    "com.docker.network.bridge.name": "docker0",
    "com.docker.network.driver.mtu": "1500"
  },
  "Labels": {}
}]
```

-percebendo os endereços dos containers-

1. criando o primeiro container

docker container run -d - --name container1 alpine sleep 1000

- **sleep** deixara o container ativo e executando

após rodar o container

docker container exec -it container1 ifconfig

foi percebido que o endereço desse container foi o: 172.17.0.2

2. criando o segundo container

docker container run -d - --name container2 alpine sleep 1000

docker container exec -it container2 ifconfig

foi percebido que o endereço desse container foi o: 172.17.0.3

—tentando a conexão entre containers—

para testar a conectividade entre dois containers ja que eles estao na mesma maquina

docker container exec -it container1 ping 172.17.0.3 (que nesse caso é o endereço do container2

```
idilene@INOTE1461:~/curso-docker$ docker container exec -it container4 ping 172.17.0.3
PING 172.17.0.3 (172.17.0.3): 56 data bytes
64 bytes from 172.17.0.3: seq=0 ttl=64 time=0.184 ms
64 bytes from 172.17.0.3: seq=1 ttl=64 time=0.077 ms
64 bytes from 172.17.0.3: seq=2 ttl=64 time=0.115 ms
64 bytes from 172.17.0.3: seq=3 ttl=64 time=0.067 ms
64 bytes from 172.17.0.3: seq=4 ttl=64 time=0.076 ms
64 bytes from 172.17.0.3: seq=5 ttl=64 time=0.077 ms
64 bytes from 172.17.0.3: seq=6 ttl=64 time=0.077 ms
^C
--- 172.17.0.3 ping statistics ---
7 packets transmitted, 7 packets received, 0% packet loss
round-trip min/avg/max = 0.067/0.096/0.184 ms
```

—criando uma nova rede—

docker network create - --driver bridge rede_nova

para listar as redes o comando é: **docker network ls**


```

idilene@INOTE1461:~/curso-docker$ docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
56012ce8b2e2        backend_default     bridge              local
27a1fab1b2e7        bridge              bridge              local
1e69fdd10b85        docker_gwbridge     bridge              local
3b828424e1d8        host                host                local
c285198b8036        minishop4_default   bridge              local
7544828f5813        none                null                local
744cc5276cb7        rede_nova           bridge              local
2ff0a6fc3807        restaurantes_api-master_default   bridge              local
89ee9195fe0b        restful-api-without-db_default     bridge              local
35492ef63372        videos_default      bridge              local
idilene@INOTE1461:~/curso-docker$

```

podemos perceber que uma nova faixa de ip foi criada

```

idilene@INOTE1461:~/curso-docker$ docker network inspect rede_nova
[
  {
    "Name": "rede_nova",
    "Id": "744cc5276cb7f5cb0b2d0a1129cd3ad1a66de116a9d79849b29a7ae5cfc1a4a1",
    "Created": "2024-04-04T16:34:17.561669142-03:00",
    "Scope": "local",
    "Driver": "bridge",
    "EnableIPv6": false,
    "IPAM": {
      "Driver": "default",
      "Options": {},
      "Config": [
        {
          "Subnet": "172.22.0.0/16",
          "Gateway": "172.22.0.1"
        }
      ]
    },
    "Internal": false,
    "Attachable": false,
    "Ingress": false,
    "ConfigFrom": {
      "Network": ""
    },
    "ConfigOnly": false,
    "Containers": {},
    "Options": {},
    "Labels": {}
  }
]

```

—adicionando um container a uma rede criada—

docker container run -d --name container5 --net rede_nova alpine sleep 1000

percebemos que o container criado tem o ip na mesma faixa que a rede

```

idilene@INOTE1461:~/curso-docker$ docker exec -it container6 ifconfig
eth0      Link encap:Ethernet  HWaddr 02:42:AC:16:00:02
          inet addr:172.22.0.2  Bcast:172.22.255.255  Mask:255.255.0.0
          inet6 addr: fe80::42:acff:fe16:2/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:12 errors:0 dropped:0 overruns:0 frame:0
          TX packets:7 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:1112 (1.0 KiB)  TX bytes:586 (586.0 B)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

idilene@INOTE1461:~/curso-docker$

```

um teste interessante é tentar fazer a comunicação entre o container 6 que está em uma nova rede com o container 3 que está em outra rede, e por estarem em redes diferentes o container 6 não consegue se comunicar com o container 3

```

idilene@INOTE1461:~/curso-docker$ docker exec -it container6 ping 172.17.0.3
PING 172.17.0.3 (172.17.0.3): 56 data bytes

```

adicionando o container 6 que já estava na rede_nova agora também estará na rede bridge

docker network connect bridge container6

e para verificar que deu certo e ver as redes que ela faz parte

docker exec -it container6 ifconfig

podemos ver que tanto ela faz parte da rede 22 como da 17 que é a padrão

```

idilene@INOTE1461:~/curso-docker$ docker exec -it container6 ifconfig
eth0      Link encap:Ethernet  HWaddr 02:42:AC:16:00:02
          inet addr:172.22.0.2  Bcast:172.22.255.255  Mask:255.255.0.0
          inet6 addr: fe80::42:acff:fe16:2/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:24 errors:0 dropped:0 overruns:0 frame:0
          TX packets:205 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:1784 (1.7 KiB)  TX bytes:19486 (19.0 KiB)

eth1      Link encap:Ethernet  HWaddr 02:42:AC:11:00:02
          inet addr:172.17.0.2  Bcast:172.17.255.255  Mask:255.255.0.0
          inet6 addr: fe80::42:acff:fe11:2/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:8 errors:0 dropped:0 overruns:0 frame:0
          TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:656 (656.0 B)  TX bytes:656 (656.0 B)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

```

—desconectando um container de uma rede —

docker network disconnect bridge container3

3. host network

nao ter um bridge para fazer uma ponte entre as interfaces de rede do container com o host, agora o container vai usar as interfaces de rede do host diretamente.

- tem uma camada de proteção baixa
- tem uma velocidade de rede maior ja que nao tem nenhum intermédio

docker container run -d - -name container7 --net host alpine sleep 1000

docker compose

listar banco de dados no postgres

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

dentro dessas aspas simples poderíamos passar um comando em sql

mostra os logs

docker-compose logs -f -t

docker worker

Após fazer as configurações no dockfile e compose esse comando serve para dizer quantos workers a gente quer startar

- **docker compose up -d - --scale worker=3**

Para ver as mensagens sendo enviadas em fila

- **docker compose logs -f -t worker**

docker-compose - override

criando um arquivo docker-compose-override.yml conseguimos sobrescrever variáveis do docker-compose.yml. por exemplo se eu colocar o nome do banco um que nao existe e no arquivo override colocar o nome correto, o build continuará funcionando porque o override vai sobrescrever.