# Docker

# Instalação

O Docker deixou de ser apenas um software para virar um conjunto deles: um ecossistema.

Nesse ecossistema temos os seguintes softwares:

**Docker Engine:** É o software base de toda solução. É tanto o daemon responsável pelos containers como o cliente usado para enviar comandos para o daemon.

**Docker Compose:** É a ferramenta responsável pela definição e execução de múltiplos containers com base em arquivo de definição.

**Docker Machine:** é a ferramenta que possibilita criar e manter ambientes docker em máquinas virtuais, ambientes de nuvem e até mesmo em máquina física.

Não citamos o [Swarm](https://docs.docker.com/swarm/overview/) e outras ferramentas por não estarem alinhados com o objetivo desse livro: introdução para desenvolvedores.

## Instalando no GNU/Linux

Explicamos a instalação da forma mais genérica possível, dessa forma você poderá instalar as ferramentas em qualquer distribuição GNU/Linux que esteja usando.

### Docker engine no GNU/Linux

Para instalar o docker engine é simples. Acesse seu terminal preferido do GNU/Linux e torne-se usuário root:

su – root

ou no caso da utilização de sudo

sudo su – root

Execute o comando abaixo:

wget -qO- https://get.docker.com/ | sh

Aconselhamos que leia o script que está sendo executado no seu sistema operacional. Acesse [esse link](https://get.docker.com/) e analise o código assim que tiver tempo para fazê-lo.

Esse procedimento demora um pouco. Após terminar o teste, execute o comando abaixo:

docker container run hello-world

### Tratamento de possíveis problemas

Se o acesso a internet da máquina passar por controle de tráfego (aquele que bloqueia o acesso a determinadas páginas) você poderá encontrar problemas no passo do **apt-key**. Caso enfrente esse problema, execute o comando abaixo:

wget -qO- https://get.docker.com/gpg | sudo apt-key add -

### Docker compose no GNU/Linux

Acesse o seu terminal preferido do GNU/Linux e torne-se usuário root:

su – root

ou no caso da utilização de sudo

sudo su – root

Execute o comando abaixo:

curl -L https://github.com/docker/compose/releases/download/1.6.2/docker-compose-`uname -s`-`uname -m` > /usr/local/bin/docker-compose

chmod +x /usr/local/bin/docker-compose

Para testar, execute o comando abaixo:

docker-compose version

### Docker machine no GNU/Linux

Instalar o docker machine é simples. Acesse o seu terminal preferido do GNU/Linux e torne-se usuário root:

su – root

ou no caso da utilização de sudo

sudo su – root

Execute o comando abaixo:

$ curl -L https://github.com/docker/machine/releases/download/v0.7.0/docker-machine-`uname -s`-`uname -m` > /usr/local/bin/docker-machine && \

chmod +x /usr/local/bin/docker-machine

Para testar, execute o comando abaixo:

docker-machine version

Obs.: O exemplo anterior utiliza a versão mais recente no momento desta publicação. Verifique se há alguma versão atualizada consultando a [documentação oficial](https://docs.docker.com/machine/install-machine/).

## Instalando no MacOS

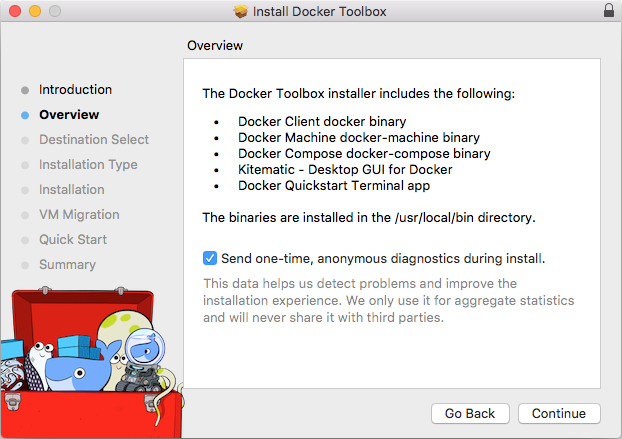
A instalação das ferramentas do Ecossistema Docker no MacOS é realizada através de um único grande pacote chamado **Docker Toolbox**.

Você pode instalar via brew cask com o comando abaixo:

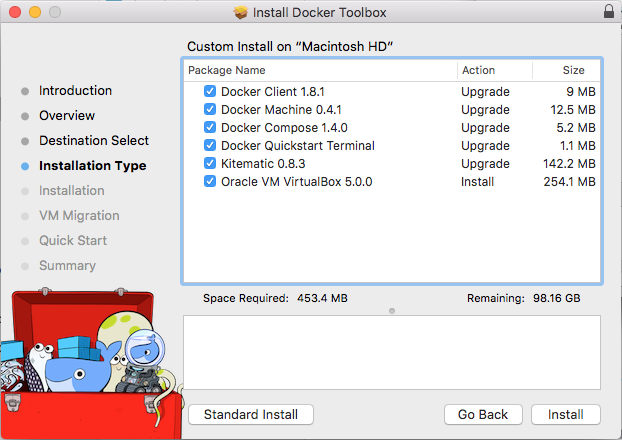
brew cask install docker-toolbox

Também pode instalar manualmente acessando [a página de download](https://www.docker.com/products/docker-toolbox) do **Docker toolbox** e baixando o instalador correspondente ao MacOS.

Após duplo clique no instalador, verá essa tela:



Apenas clique em **Continue**.



Marque todas as opções e clique **Install**.

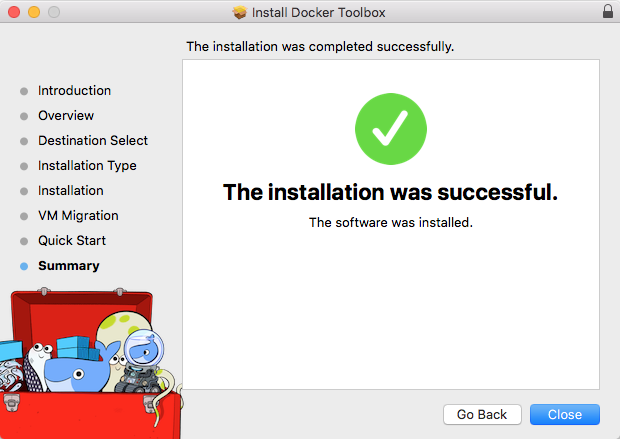
Será solicitado seu usuário e senha para liberar a instalação dos softwares. Preencha e continue o processo.

Na próxima tela, será apresentado as ferramentas que podem ser usadas para facilitar sua utilização do Docker no MacOS.



Apenas clique em **Continue**.

Essa é ultima janela que verá no processo de instalação.



Apenas clique em **Close** e finalize a instalação.

Para testar, procure e execute o software **Docker Quickstart Terminal**, pois ele fará todo processo necessário para começar a utilizar o Docker.

Nesse novo terminal execute o seguinte comando para teste:

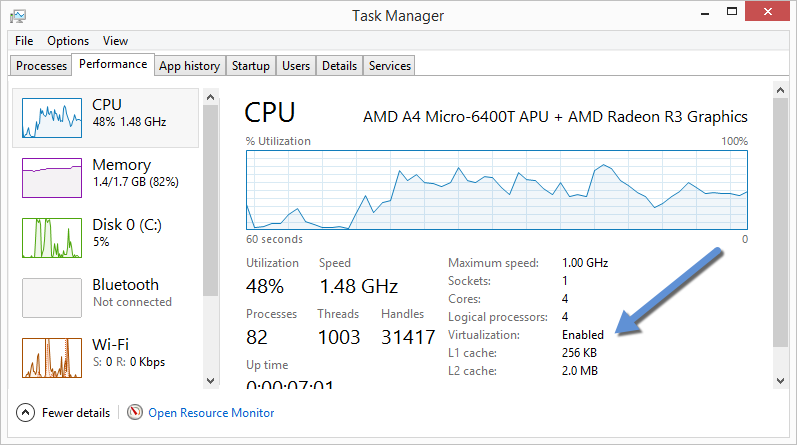
docker container run hello-world

## Instalando no Windows

A instalação das ferramentas do Ecossistema Docker no Windows é realizada através de um único grande pacote, que se chama **Docker Toolbox**.

O **Docker Toolbox** funciona apenas em [versões 64bit](https://support.microsoft.com/en-us/kb/827218) do Windows e somente para as versões superiores ao Windows 7.

É importante salientar também que é necessário habilitar o suporte de virtualização. Na versão 8 do Windows, é possível verificar através do **Task Manager**. Na aba **Performance** clique em **CPU** para visualizar a janela abaixo:



Para verificar o suporte a virtualização do Windows 7, utilize esse [link](http://www.microsoft.com/en-us/download/details.aspx?id=592) para maiores informações.

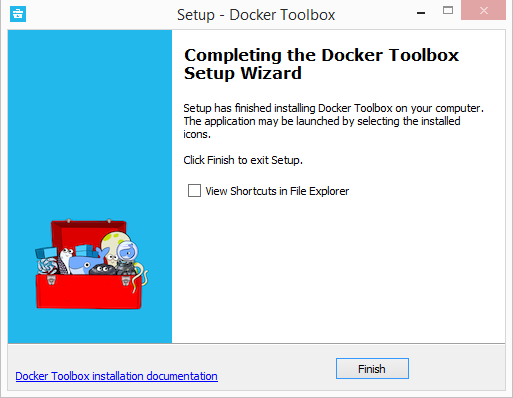
### Instalando o Docker Toolbox

Acesse [a página de download](https://www.docker.com/products/docker-toolbox) do **Docker toolbox** e baixe o instalador correspondente ao Windows.

Após duplo clique no instalador, verá essa tela:



Apenas clique em **Next**.



Por fim, clique em **Finish**.

Para testar, procure e execute o software **Docker Quickstart Terminal**, pois ele fará todo processo necessário para começar a utilizar o Docker.

Nesse novo terminal execute o seguinte comando para teste:

docker container run hello-world

# Comandos básicos

Para utilização do Docker é necessário conhecer alguns comandos e entender de forma clara e direta para que servem, assim como alguns exemplos de uso.

Não abordaremos os comandos de criação de imagem e tratamento de problemas (troubleshooting) no Docker, pois têm capítulos específicos para o detalhamento.

## Executando um container

Para iniciar um container é necessário saber a partir de qual imagem será executado. Para listar as imagens que seu **Docker host** tem localmente, execute o comando abaixo:

docker image list

As imagens retornadas estão presentes no seu **Docker host** e não demandam qualquer download da [nuvem pública do Docker](https://stack.desenvolvedor.expert/appendix/docker/hub.docker.com), a menos que deseje atualizá-la. Para atualizar a imagem basta executar o comando abaixo:

docker image pull python

Usamos a imagem chamada **python** como exemplo, mas caso deseje atualizar qualquer outra imagem, basta colocar seu nome no lugar de **python**.

Caso deseje inspecionar a imagem que acabou de atualizar, basta usar o comando abaixo:

docker image inspect python

O comando [inspect](https://docs.docker.com/engine/reference/commandline/inspect/) é responsável por informar todos os dados referentes à imagem.

Agora que temos a imagem atualizada e inspecionada, podemos iniciar o container. Mas antes de simplesmente copiar e colar o comando, vamos entender como ele realmente funciona.

docker container run <parâmetros> <imagem> <CMD> <argumentos>

Os parâmetros mais utilizados na execução do container são:

| **Parâmetro** | **Explicação** |
| --- | --- |
| -d | Execução do container em background |
| -i | Modo interativo. Mantém o STDIN aberto mesmo sem console anexado |
| -t | Aloca uma pseudo TTY |
| --rm | Automaticamente remove o container após finalização (**Não funciona com -d**) |
| --name | Nomear o container |
| -v | Mapeamento de volume |
| -p | Mapeamento de porta |
| -m | Limitar o uso de memória RAM |
| -c | Balancear o uso de CPU |

Segue um exemplo simples no seguinte comando:

docker container run -it --rm --name meu\_python python bash

De acordo com o comando acima, será iniciado um container com o nome **meu\_python**, criado a partir da imagem **python** e o processo executado nesse container será o **bash**.

Vale lembrar que, caso o **CMD** não seja especificado no comando **docker container run**, é utilizado o valor padrão definido no **Dockerfile** da imagem utilizada. No nosso caso é **python** e seu comando padrão executa o binário **python**, ou seja, se não fosse especificado o **bash**, no final do comando de exemplo acima, ao invés de um shell bash do GNU/Linux, seria exibido um shell do **python**.

## Mapeamento de volumes

Para realizar mapeamento de volume basta especificar qual origem do dado no host e onde deve ser montado dentro do container.

docker container run -it --rm -v "<host>:<container>" python

O uso de armazenamento é melhor explicado em capítulos futuros, por isso não detalharemos o uso desse parâmetro.

## Mapeamento de portas

Para realizar o mapeamento de portas basta saber qual porta será mapeada no host e qual deve receber essa conexão dentro do container.

docker container run -it --rm -p "<host>:<container>" python

Um exemplo com a porta 80 do host para uma porta 8080 dentro do container tem o seguinte comando:

docker container run -it --rm -p 80:8080 python

Com o comando acima temos a porta 80 acessível no Docker host que repassa todas as conexões para a porta 8080 dentro do container. Ou seja, não é possível acessar a porta 8080 no endereço IP do Docker host, pois essa porta está acessível apenas dentro do container que é isolada a nível de rede, como já dito anteriormente.

## Gerenciamento dos recursos

Na inicialização dos containers é possível especificar alguns limites de utilização dos recursos. Trataremos aqui apenas de memória RAM e CPU, os mais utilizados.

Para limitar o uso de memória RAM que pode ser utilizada por esse container, basta executar o comando abaixo:

docker container run -it --rm -m 512M python

Com o comando acima estamos limitando esse container a utilizar somente 512 MB de RAM.

Para balancear o uso da CPU pelos containers, utilizamos especificação de pesos para cada container, quanto menor o peso, menor sua prioridade no uso. Os pesos podem oscilar de 1 a 1024.

Caso não seja especificado o peso do container, ele usará o maior peso possível, nesse caso 1024.

Usaremos como exemplo o peso 512:

docker container run -it --rm -c 512 python

Para entendimento, vamos imaginar que três containers foram colocados em execução. Um deles tem o peso padrão 1024 e dois têm o peso 512. Caso os três processos demandem toda CPU o tempo de uso deles será dividido da seguinte maneira:

O processo com peso 1024 usará 50% do tempo de processamento

Os dois processos com peso 512 usarão 25% do tempo de processamento, cada.

## Verificando a lista de containers

Para visualizar a lista de containers de um determinado Docker host utilizamos o comando [docker ps](https://docs.docker.com/engine/reference/commandline/ps/).

Esse comando é responsável por mostrar todos os containers, mesmo aqueles não mais em execução.

docker container list <parâmetros>

Os parâmetros mais utilizados na execução do container são:

| Parâmetro | Explicação |
| --- | --- |
| -a | Lista todos os containers, inclusive os desligados |
| -l | Lista os últimos containers, inclusive os desligados |
| -n | Lista os últimos N containers, inclusive os desligados |
| -q | Lista apenas os ids dos containers, ótimo para utilização em scripts |

## Gerenciamento de containers

Uma vez iniciado o container a partir de uma imagem é possível gerenciar a utilização com novos comandos.

Caso deseje desligar o container basta utilizar o comando [docker stop](https://docs.docker.com/engine/reference/commandline/stop/). Ele recebe como argumento o ID ou nome do container. Ambos os dados podem ser obtidos com o docker ps, explicado no tópico anterior.

Um exemplo de uso:

docker container stop meu\_python

No comando acima, caso houvesse um container chamado meu\_python em execução, ele receberia um sinal SIGTERM e, caso não fosse desligado, receberia um SIGKILL depois de 10 segundos.

Caso deseje reiniciar o container que foi desligado e não iniciar um novo, basta executar o comando [docker start](https://docs.docker.com/engine/reference/commandline/start/):

docker container start meu\_python

*Vale ressaltar que a ideia dos containers é a de serem descartáveis. Caso você use o mesmo container por muito tempo sem descartá-lo, provavelmente está usando o Docker incorretamente. O Docker não é uma máquina, é um processo em execução. E, como todo processo, deve ser descartado para que outro possa tomar seu lugar na reinicialização dele.*

# Criando sua própria imagem no Docker

Antes de explicarmos como criar sua imagem, vale a pena tocarmos em uma questão que normalmente confunde iniciantes do docker: “Imagem ou container?”

## Qual a diferença entre Imagem e Container?

Traçando um paralelo com o conceito de [orientação a objeto](https://pt.wikipedia.org/wiki/Orienta%C3%A7%C3%A3o_a_objetos), a **imagem** é a classe e o **container** o objeto. A imagem é a abstração da infraestrutura em estado somente leitura, de onde será instanciado o container.

Todo container é iniciado a partir de uma imagem, dessa forma podemos concluir que nunca teremos uma imagem em execução.

Um container só pode ser iniciado a partir de uma única imagem. Caso deseje um comportamento diferente, será necessário customizar a imagem.

## Anatomia da imagem

As imagens podem ser oficiais ou não oficiais.

## Imagens oficiais e não oficiais

As imagens oficiais do docker são aquelas sem usuários em seus nomes. A imagem **“ubuntu:16.04″** é oficial, por outro lado, a imagem [“nuagebec/ubuntu”](https://hub.docker.com/r/nuagebec/ubuntu/) não é oficial. Essa segunda imagem é mantida pelo usuário [nuagebec](https://hub.docker.com/u/nuagebec/), que mantém outras imagens não oficiais.

As imagens oficiais são mantidas pela empresa docker e [disponibilizadas](https://hub.docker.com/explore/) na nuvem docker.

O objetivo das imagens oficiais é prover um ambiente básico (ex. debian, alpine, ruby, python), um ponto de partida para criação de imagens pelos usuários, como explicaremos mais adiante, ainda nesse capítulo.

As imagens não oficiais são mantidas pelos usuários que as criaram. Falaremos sobre envio de imagens para nuvem docker em outro tópico.

### ****Nome da imagem****

O nome de uma imagem oficial é composto por duas partes. A primeira, a [documentação](https://docs.docker.com/engine/userguide/containers/dockerimages/) chama de **“repositório”** e, a segunda, é chamada de **“tag”**. No caso da imagem **“ubuntu:14.04”**, **ubuntu** é o repositório e **14.04** é a tag.

Para o docker, o **“repositório”** é uma abstração do conjunto de imagens. Não confunda com o local de armazenamento das imagens, que detalharemos mais adiante. Já a **“tag”**, é uma abstração para criar unidade dentro do conjunto de imagens definidas no **“repositório”**.

Um **“repositório”** pode conter mais de uma **“tag”** e cada conjunto repositório:tag representa uma imagem diferente.

Execute [o comando](https://docs.docker.com/engine/reference/commandline/images/) abaixo para visualizar todas as imagens que se encontram localmente na sua estação, nesse momento:

docker image list

## Como criar imagens

Há duas formas de criar imagens customizadas: com **commit** e com **Dockerfile**.

### Criando imagens com commit

É possível criar imagens executando o comando [commit](https://docs.docker.com/engine/reference/commandline/commit/), relacionado a um container. Esse comando usa o status atual do container escolhido e cria a imagem com base nele.

Vamos ao exemplo. Primeiro criamos um container qualquer:

docker container run -it --name containercriado ubuntu:16.04 bash

Agora que estamos no bash do container, instalamos o nginx:

apt-get update

apt-get install nginx -y

exit

Paramos o container com o comando abaixo:

docker container stop containercriado

Agora, efetuamos o **commit** desse **container** em uma **imagem**:

docker container commit containercriado meuubuntu:nginx

No exemplo do comando acima, **containercriado** é o nome do container criado e modificado nos passos anteriores; o nome **meuubuntu:nginx** é a imagem resultante do **commit**; o estado do **containercriado** é armazenado em uma imagem chamada **meuubuntu:nginx** que, nesse caso, a única modificação que temos da imagem oficial do ubuntu na versão 16.04 é o pacote **nginx** instalado.

Para visualizar a lista de imagens e encontrar a que acabou de criar, execute novamente o comando abaixo:

docker image list

Para testar sua nova imagem, vamos criar um container a partir dela e verificar se o nginx está instalado:

docker container run -it --rm meuubuntu:nginx dpkg -l nginx

Se quiser validar, pode executar o mesmo comando na imagem oficial do ubuntu:

docker container run -it --rm ubuntu:16.04 dpkg -l nginx

Vale salientar que o método **commit** não é a melhor opção para criar imagens, pois, como verificamos, o processo de modificação da imagem é completamente manual e apresenta certa dificuldade para rastrear as mudanças efetuadas, uma vez que, o que foi modificado manualmente não é registrado, automaticamente, na estrutura do docker.

### Criando imagens com Dockerfile

Quando se utiliza Dockerfile para gerar uma imagem, basicamente, é apresentada uma lista de instruções que serão aplicadas em determinada imagem para que outra imagem seja gerada com base nas modificações.

Dockerfile

Podemos resumir que o arquivo Dockerfile, na verdade, representa a exata diferença entre uma determinada imagem, que aqui chamamos de **base**, e a imagem que se deseja criar. Nesse modelo temos total rastreabilidade sobre o que será modificado na nova imagem.

Voltemos ao exemplo da instalação do nginx no ubuntu 16.04.

Primeiro crie um arquivo qualquer para um teste futuro:

touch arquivo\_teste

Crie um arquivo chamado **Dockerfile** e dentro dele o seguinte conteúdo:

FROM ubuntu:16.04

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

COPY arquivo\_teste /tmp/arquivo\_teste

CMD bash

No arquivo acima, utilizamos quatro [instruções](https://docs.docker.com/engine/reference/builder/):

**FROM** para informar qual imagem usaremos como base, nesse caso foi **ubuntu:16.04**.

**RUN** para informar quais comandos serão executados nesse ambiente para efetuar as mudanças necessárias na infraestrutura do sistema. São como comandos executados no shell do ambiente, igual ao modelo por commit, mas nesse caso foi efetuado automaticamente e, é completamente rastreável, já que esse Dockerfile será armazenado no sistema de controle de versão.

**COPY** é usado para copiar arquivos da estação onde está executando a construção para dentro da imagem. Usamos um arquivo de teste apenas para exemplificar essa possibilidade, mas essa instrução é muito utilizada para enviar arquivos de configuração de ambiente e códigos para serem executados em serviços de aplicação.

**CMD** para informar qual comando será executado por padrão, caso nenhum seja informado na inicialização de um container a partir dessa imagem. No exemplo, colocamos o comando bash, se essa imagem for usada para iniciar um container e não informamos o comando, ele executará o bash.

Após construir seu Dockerfile basta executar o [comando](https://docs.docker.com/engine/reference/commandline/build/) abaixo:

docker image build -t meuubuntu:nginx\_auto .

Tal comando tem a opção **“-t”**, serve para informar o nome da imagem a ser criada. No caso, será **meuubuntu:nginx\_auto** e o **“.”** ao final, informa qual contexto deve ser usado nessa construção de imagem. Todos os arquivos da pasta atual serão enviados para o serviço do docker e apenas eles podem ser usados para manipulações do Dockerfile (exemplo do uso do COPY).

### A ordem importa

É importante atentar que o arquivo **Dockerfile** é uma sequência de instruções lidas do topo à base e cada linha é executada por vez. Se alguma instrução depender de outra instrução, essa dependência deve ser descrita mais acima no documento.

O resultado de cada instrução do arquivo é armazenado em cache local. Caso o **Dockerfile** não seja modificado na próxima criação da imagem (**build**), o processo não demorará, pois tudo estará no cache. Se houver alterações, apenas a instrução modificada e as posteriores serão executadas novamente.

A sugestão para melhor aproveitar o cache do **Dockerfile** é sempre manter as instruções frequentemente alteradas mais próximas da base do documento. Vale lembrar de atender também as dependências entre instruções.

Um exemplo para deixar mais claro:

FROM ubuntu:16.04

RUN apt-get update

RUN apt-get install nginx

RUN apt-get install php5

COPY arquivo\_teste /tmp/arquivo\_teste

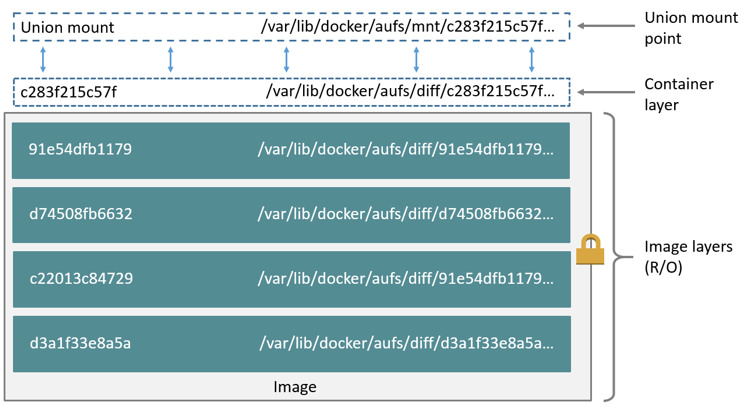
CMD bash

Caso modifiquemos a terceira linha do arquivo e, ao invés de instalar o nginx, mudarmos para apache2, a instrução que faz o update no apt não será executada novamente, e sim a instalação do apache2, pois acabou de entrar no arquivo, assim como o php5 e a cópia do arquivo, pois todos eles são subsequentes a linha modificada.

Como podemos perceber, de posse do arquivo **Dockerfile**, é possível ter a exata noção de quais mudanças foram efetuadas na imagem e, assim, registrar as modificações no sistema de controle de versão.

# Entendendo armazenamento no Docker

Para entender como o docker gerencia seus volumes, primeiro precisamos explicar como funciona ao menos um [backend](http://searchdatacenter.techtarget.com/definition/back-end) de armazenamento do Docker. Faremos aqui com o AUFS, que foi o primeiro e ainda é padrão em boa parte das instalações do Docker.



## Como funciona um backend do Docker (Ex.: AUFS)

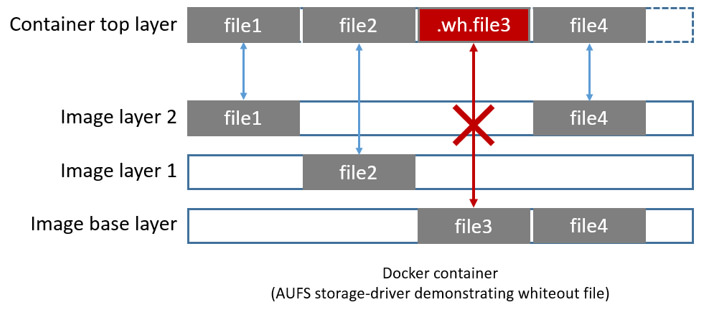
Backend de armazenamento é a parte da solução do Docker que cuida do gerenciamento dos dados. No Docker temos várias possibilidades de backend de armazenamento, mas nesse texto falaremos apenas do que implementa o AUFS.

[AUFS](https://en.wikipedia.org/wiki/Aufs) é um unification filesystem. É responsável por gerenciar múltiplos diretórios, empilhá-los uns sobre os outros e fornecer uma única e unificada visão, como se todos juntos fossem apenas um diretório.

Esse único diretório é utilizado para apresentar o container e funciona como se fosse um único sistema de arquivos comum. Cada diretório usado na pilha corresponde a uma camada. E, é dessa forma que o Docker as unifica e proporciona a reutilização entre containeres. Pois, o mesmo diretório correspondente à imagem pode ser montado em várias pilhas de vários containeres.

Com exceção da pasta (camada) correspondente ao container, todas as outras são montadas com permissão de somente leitura, caso contrário as mudanças de um container poderiam interferir em outro. O que, de fato, é totalmente contra os princípios do Linux Container.

Caso seja necessário modificar um arquivo nas camadas (pastas) referentes às imagens, se utiliza a tecnologia [Copy-on-write](https://en.wikipedia.org/wiki/Copy-on-write) (CoW), responsável por copiar o arquivo para a pasta (camada) do container e fazer todas as modificações nesse nível. Dessa forma, o arquivo original da camada inferior é sobreposto nessa pilha, ou seja, o container em questão sempre verá apenas os arquivos das camadas mais altas.



No caso da remoção, o arquivo da camada superior é marcado como whiteout file, viabilizando a visualização do arquivo de camadas inferiores.

## Problema com performance

O Docker tira proveito da tecnologia Copy-on-write (CoW) do AUFS para permitir o compartilhamento de imagem e minimizar o uso de espaço em disco. AUFS funciona no nível de arquivo. Isto significa que todas as operações AUFS CoW copiarão arquivos inteiros, mesmo que, apenas pequena parte do arquivo esteja sendo modificada. Esse comportamento pode ter impacto notável no desempenho do container, especialmente se os arquivos copiados são grandes e estão localizados abaixo de várias camadas de imagens. Nesse caso o procedimento copy-on-write dedicará muito tempo para uma cópia interna.

## Volume como solução para performance

Ao utilizar volumes, o Docker monta essa pasta (camada) no nível imediatamente inferior ao do container, o que permite o acesso rápido de todo dado armazenado nessa camada (pasta), resolvendo o problema de performance.

O volume também resolve questões de persistência de dados, pois as informações armazenadas na camada (pasta) do container são perdidas ao remover o container, ou seja, ao utilizar volumes temos maior garantia no armazenamento desses dados.

### Usando volumes

#### Mapeamento de pasta específica do host

Nesse modelo o usuário escolhe uma pasta específica do host (Ex.: /var/lib/container1) e a mapeia em uma pasta interna do container (Ex.: /var). O que é escrito na pasta /var do container é escrito também na pasta /var/lib/container1 do host.

Segue o exemplo de comando usado para esse modelo de mapeamento:

docker container run -v /var/lib/container1:/var ubuntu

Esse modelo não é portável. Necessita que o host tenha uma pasta específica para que o container funcione adequadamente.

#### Mapeamento via container de dados

Nesse modelo é criado um container e, dentro desse, é nomeado um volume a ser consumido por outros contêineres. Dessa forma não é preciso criar uma pasta específica no host para persistir dados. Essa pasta é criada automaticamente dentro da pasta raiz do Docker daemon. Porém, você não precisa se preocupar com essa pasta, pois toda a referência será feita para o container detentor do volume e não para a pasta.

Segue um exemplo de uso do modelo de mapeamento:

docker create -v /dbdata --name dbdata postgres /bin/true

No comando acima, criamos um container de dados, onde a pasta /dbdata pode ser consumida por outros containeres, ou seja, o conteúdo da pasta /dbtada poderá ser visualizado e/ou editado por outros containeres.

Para consumir esse volume do container basta utilizar o comando:

docker container run -d --volumes-from dbdata --name db2 postgres

Agora o container db2 tem uma pasta /dbdata que é a mesma do container dbdata, tornando esse modelo completamente portável.

Uma desvantagem é a necessidade de manter um container apenas para isso, pois em alguns ambientes os containeres são removidos com certa regularidade e, dessa forma, é necessário ter cuidado com containeres especiais. O que, de certa forma, é um problema adicional de gerenciamento.

#### Mapeamento de volumes

Na versão 1.9 do Docker foi acrescentada a possibilidade de criar volumes isolados de containeres. Agora é possível criar um volume portável, sem a necessidade de associá-lo a um container especial.

Segue um exemplo de uso do modelo de mapeamento:

docker volume create --name dbdata

No comando acima, o docker criou um volume que pode ser consumido por qualquer container.

A associação do volume ao container acontece de forma parecida à praticada no mapeamento de pasta do host, pois nesse caso você precisa associar o volume a uma pasta dentro do container, como podemos ver abaixo:

docker container run -d -v dbdata:/var/lib/data postgres

Esse modelo é o mais indicado desde o lançamento, pois proporciona portabilidade. Não é removido facilmente quando o container é deletado e ainda é bastante fácil de gerenciar.

# Entendendo a rede no Docker

O que o docker chama de rede, na verdade é uma abstração criada para facilitar o gerenciamento da comunicação de dados entre containers e os nós externos ao ambiente docker.

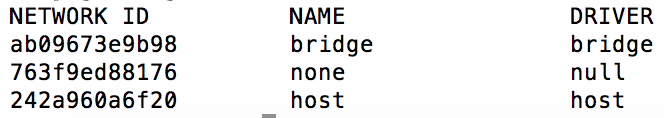
Não confunda a rede do docker com a já conhecida rede utilizada para agrupar os endereços IP (ex: 192.168.10.0/24). Sendo assim, sempre que mencionarmos esse segundo tipo de rede, usaremos “rede IP“.

## Redes padrões do Docker

O docker é disponibilizado com três redes por padrão. Essas redes oferecem configurações específicas para gerenciamento do tráfego de dados. Para visualizar essas interfaces, basta utilizar o comando abaixo:

**docker network ls**

O retorno será:



### Bridge

Cada container iniciado no docker é associado a uma rede específica. Essa é a rede padrão para qualquer container, a menos que associemos, explicitamente, outra rede a ele. A rede confere ao container uma interface que faz bridge com a interface docker0 do docker host. Essa interface recebe, automaticamente, o próximo endereço disponível na rede IP 172.17.0.0/16.

Todos os containers que estão nessa rede poderão se comunicar via protocolo TCP/IP. Se você souber qual endereço IP do container deseja conectar, é possível enviar tráfego para ele. Afinal, estão todos na mesma rede IP (172.17.0.0/16).

Um detalhe a se observar: como os IPs são cedidos automaticamente, não é tarefa trivial descobrir qual IP do container de destino. Para ajudar nessa localização, o docker disponibiliza, na inicialização de um container, a opção “–link“.

**Vale ressaltar que “–link” é uma opção defasada e seu uso desaconselhado. Explicaremos esta funcionalidade apenas para efeito de entendimento do legado. Essa função foi substituída por um DNS embutido no docker e, não funciona para redes padrões do docker, apenas disponível para redes criadas pelo usuário.**

A opção “–link” é responsável por associar o IP do container de destino ao seu nome. Caso você inicie um container a partir da imagem docker do mysql com nome “bd”, em seguida inicie outro com nome “app” a partir da imagem tutum/apache-php, você deseja que esse último container possa conectar no mysql usando o nome do container “bd”, basta iniciar da seguinte forma ambos os containers:

docker container run -d --name bd -e MYSQL\_ROOT\_PASSWORD=minhasenha mysql

docker container run -d -p 80:80 --name app --link db tutum/apache-php

Após executar os comandos, o container com o nome “app” poderá conectar no container do mysql usando o nome “bd”, ou seja, toda vez que ele tentar acessar o nome “bd” ele será automaticamente resolvido para o IP da rede IP 172.17.0.0/16 que o container do mysql obteve na sua inicialização.

Pra testar, utilizaremos a funcionalidade exec para rodar o comando dentro de um container já existente. Para tal, usaremos o nome do container como parâmetro do comando abaixo:

docker container exec -it app ping db

A ação será responsável por executar o comando “ping db” dentro do container “app”, ou seja, o container “app” enviará pacotes icmp, normalmente usado para testar conectividade entre dois hosts, para o endereço “db”. O nome “db” é traduzido para o IP que o container, iniciado a partir da imagem do mysql, obteve ao iniciar.

**Exemplo:** O container “db” iniciou primeiro e obteve o IP 172.17.0.2. O container “app” iniciou em seguida e recebeu o IP 172.17.0.3. Quando o container “app” executar o comando “ping db”, na verdade, ele enviará pacotes icmp para o endereço 172.17.0.2.

**Atenção: O nome da opção “–link” causa certa confusão, pois não cria link de rede IP entre os containers, uma vez que a comunicação entre eles já é possível, mesmo sem a opção link ser configurada. Como esclarecido no parágrafo anterior, apenas facilita a tradução de nomes para o IP dinâmico obtido na inicialização.**

Os containers configurados para essa rede terão a possibilidade de tráfego externo utilizando as rotas das redes IP definidas no docker host. Caso o docker host tenha acesso a internet, automaticamente, os containers em questão também terão.

Nessa rede é possível expor portas dos containers para todos os ativos com acesso ao docker host.

### None

Essa rede tem como objetivo isolar o container para comunicações externas. A rede não recebe qualquer interface para comunicação externa. A única interface de rede IP será a localhost.

Essa rede, normalmente, é utilizada para containers que manipulam apenas arquivos, sem necessidade de enviá-los via rede para outro local. (Ex.: container de backup utiliza os volumes de container de banco de dados para realizar o dump e, será usado no processo de retenção dos dados).

Exemplo de uso da rede none

Em caso de dúvida sobre utilização de volumes no docker visite [esse artigo](http://techfree.com.br/2015/12/entendendo-armazenamentos-de-dados-no-docker/) e entenda mais sobre armazenamento do docker.

### Host

Essa rede tem como objetivo entregar para o container todas as interfaces existentes no docker host. De certa forma, pode agilizar a entrega dos pacotes, uma vez que não há bridge no caminho das mensagens. Mas normalmente esse overhead é mínimo e o uso de uma brigde pode ser importante para segurança e gerencia do seu tráfego.

## Redes definidas pelo usuário

O docker possibilita que o usuário crie redes. Essas redes são associadas ao elemento que o docker chama de driver de rede.

Cada rede criada por usuário deve estar associada a um determinado driver. E, caso você não crie seu próprio driver, deve escolher entre os drivers disponibilizados pelo docker:

### Bridge

Essa é o driver de rede mais simples de utilizar, pois demanda pouca configuração. A rede criada por usuário utilizando o driver bridge assemelha-se bastante à rede padrão do docker denominada “bridge”.

**Mais um ponto que merece atenção: o docker tem uma rede padrão chamada “bridge” que utiliza um driver também chamado de “bridge“. Talvez, por conta disso, a confusão só aumente. Mas é importante deixar claro que são distintas.**

As redes criadas pelo usuário com o driver bridge tem todas as funcionalidades descritas na rede padrão, chamada bridge. Porém, com funcionalidades adicionais.

Dentre uma das funcionalidades: a rede criada pelo usuário não precisa mais utilizar a opção antiga “–link”. Pois, toda rede criada pelo usuário com o driver bridge poderá utilizar o DNS interno do Docker que, associa, automaticamente, todos os nomes de containers dessa rede para seus respectivos IPs da rede IP correspondente.

Para deixar mais claro: todos os containers que estiverem utilizando a rede padrão bridge não poderão usufruir da funcionalidade de DNS interno do Docker. Caso utilize essa rede, é preciso especificar a opção legada “–link” para tradução dos nomes em endereços IPs dinamicamente alocados no docker.

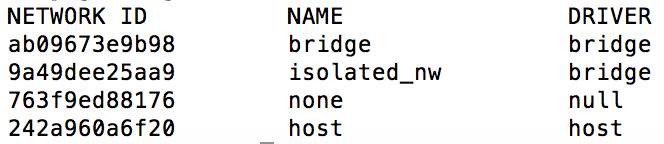
Para exemplificar a utilização de rede criada por usuário vamos criar a rede chamada isolated\_nw com o driver bridge:

docker network create --driver bridge isolated\_nw

Agora verificamos a rede:

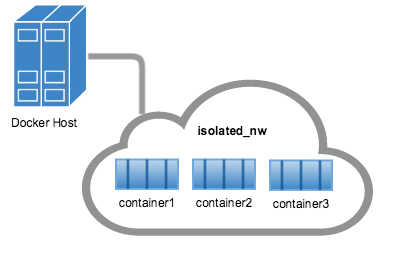
docker network list

O resultado deve ser:



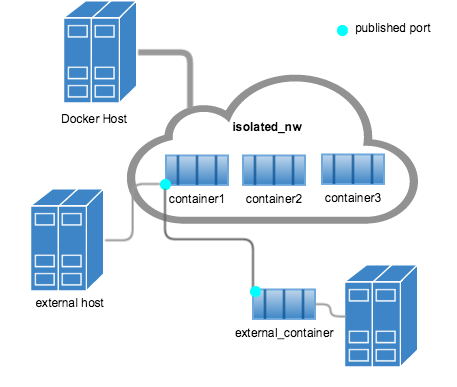
Agora iniciamos um container na rede isolated\_nw:

docker container run -itd --net isolated\_nw alpine sh



Vale salientar: um container que está em determinada rede não acessa outro container que está em outra rede. Mesmo que você conheça o IP de destino. Para que um container acesse outro container de outra rede, é necessário que a origem esteja presente nas duas redes que deseja alcançar.

Os containers que estão na rede isolated\_nw podem expor suas portas no docker host e essas portas podem ser acessadas tanto por containers externos a rede, chamada isolated\_nw, como máquinas externas com acesso ao docker host.



Para descobrir quais containers estão associados a uma determinada rede, execute o comando abaixo:

docker network inspect isolated\_nw

O resultado deve ser:



Dentro da sessão “Containers” é possível verificar quais containers fazem parte dessa rede. Todos os containers que estiverem na mesma rede poderão se comunicar utilizando apenas seus respectivos nomes. Como podemos ver no exemplo acima, caso um container novo acesse a rede isolated\_nw, ele poderá acessar o container amazing\_noyce utilizando apenas seu nome.

### Overlay

O driver overlay permite comunicação entre hosts docker, utilizando-o os containers de um determinado host docker poderão acessar, nativamente, containers de um outro ambiente docker.

Esse driver demanda configuração mais complexa, sendo assim, trataremos do detalhamento em outra oportunidade.

Utilizando redes no docker compose

O assunto merece um artigo exclusivo. Então, apenas informaremos [um link interessante](https://docs.docker.com/compose/networking/) para referência sobre o assunto.

Conclusão

Percebemos que a utilização de redes definidas por usuário torna obsoleta a utilização da opção “–link”, bem como viabiliza novo serviço de DNS interno do docker, o que facilita para quem se propõe a manter uma infraestrutura docker grande e complexa, assim como viabilizar o isolamento de rede dos seus serviços.

Conhecer e utilizar bem as tecnologias novas é uma boa prática que evita problemas futuros e facilita a construção e manutenção de projetos grandes e complexos.

# Comandos de servidores

netstat -nltp - portas abertas

ifconfig – serviço que informa os ips usados

127.0.0.1 – IP Host padrão.