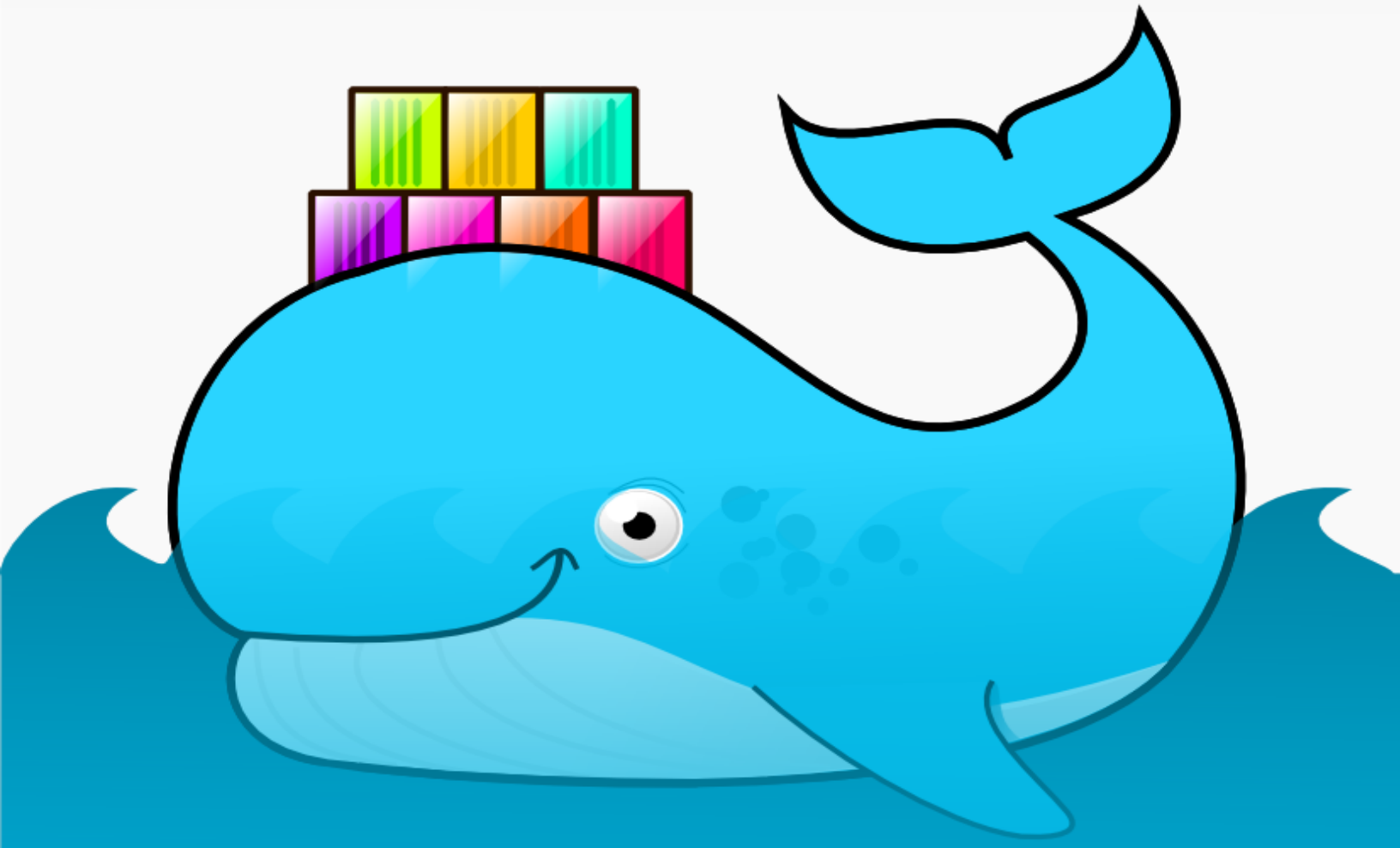
Rafael Gomes
Luis Armando Bianchin



Docker
Para Desenvolvedores

Docker para desenvolvedores

Rafael Gomes

Esse livro está à venda em http://leanpub.com/dockerparadesenvolvedores

Essa versão foi publicada em 2016-07-18

Tweet Sobre Esse Livro!

Por favor ajude Rafael Gomes a divulgar esse livro no Twitter!

A hashtag sugerida para esse livro é #docker-para-desenvolvedores.

Descubra o que as outras pessoas estão falando sobre esse livro clicando nesse link para buscar a hashtag no Twitter:

https://twitter.com/search?q=#docker-para-desenvolvedores

Conteúdo

Como ler esse livro

Breve

Propósito desse livro

Esse livro tem como objetivo explicar de formas simples e direta como os desenvolvedores podem usar Docker.

Esse livro **não** é sobre se aprofundar na infraestrutura do Docker!

Esse livro é sobre apresentar basicamente a utilização do Docker e se aprofundar nas melhores práticas de uso.

Usaremos o 12factor como espinha dorsal para demonstrar as melhores práticas de construção da sua aplicação usando Docker.

Agradecimentos

Breve

Introdução

Essa parte do livro é direcionado a todos que não tem conhecimento básico do Docker. Caso você já sabe usar, não precisa ler essa parte. Por outro lado, mesmo que você já saiba usar aqui teremos a explicação de muitos recursos que você usa e como eles funcionam de verdade.

Mesmo que você já use o Docker, ler essa parte do livro em algum momento da sua vida será importante para saber de forma mais consistente o que acontece com cada comando executado.

Por que usar Docker?

Docker tem sido um assunto bem comentado ultimamente, muitos artigos foram escrito geralmente tratando sobre como usá-lo, ferramentas auxiliares, integrações e afins, mas muitas pessoas ainda se fazem a questão mais básica quando se trata da possibilidade de utilizar qualquer nova tecnologia: "Por que devo usar isso?" ou seria "O que isso tem a me oferecer diferente do que já tenho hoje?"

É normal que ainda duvidem do potencial do Docker, alguns até acham que se trata de um hype[1], mas nesse capítulo pretendo demonstrar alguns bons motivos para se utilizar Docker.

Vale a pena frisar que o Docker não é uma "bala de prata", ou seja, ele não se propõe a resolver todos problemas, muito menos ser a solução única para as mais variadas situações.

Segue abaixo alguns bons motivos para se utilizar Docker:

1 – Ambientes semelhantes

Uma vez que sua aplicação seja transformada em uma imagem Docker, ela pode ser instanciada como container em qualquer ambiente que desejar, ou seja, isso significa que você poderá utilizar a sua aplicação no notebook do desenvolvedor da mesma forma que ela seria executada no servidor de produção.

A imagem Docker aceita parâmetros durante o inicio do container, isso indica que uma mesma imagem pode se comportar de formas diferentes entre os distintos ambientes, ou seja, esse container

[1] http://techfree.com.br/2015/06/sera-que-esse-modelo-de-containers-e-um-hype/

pode conectar a seu banco de dados local para testes, usando as credenciais e base de dados de teste, mas quanto o container, criado a partir da mesma imagem, receber parâmetros do ambiente de produção ele acessará a base de dados em uma infraestrutura mais robusta, com suas respectivas credenciais e base de dados de produção, por exemplo.

As imagens Docker podem ser consideradas como implantações atômicas, ou seja, isso é o que proporciona maior previsibilidade comparado outras ferramentas como Puppet, Chef, Ansible, etc. Impactando positivamente na analise de erros, assim como na confiabilidade do processo de entrega contínua[2], que se baseia fortemente na criação de um único artefato que migra entre ambientes. No caso do Docker o artefato seria a própria imagem com todas as dependências requeridas para executar o seu código, seja ele compilado ou dinâmico.

2 – Aplicação como pacote completo

Utilizando as imagens Docker é possível empacotar toda sua aplicação e suas dependências, dessa forma facilitando sua distribuição, pois não será mais necessário enviar uma extensa documentação explicando como configurar a infraestrutura necessária para permitir sua execução, basta disponibilizar a imagem em um repositório e liberar o acesso para o usuário da mesma e ele baixará o pacote, que será executado sem nenhum problema.

A atualização também é positivamente afetada, pois a estrutura de camadas[3] do Docker viabiliza que em caso de mudança, apenas a parte modificada seja transferida e assim o ambiente pode ser alterado de forma mais rápida e simples. O usuário só precisaria executar apenas um comando para atualizar sua imagem da aplicação, que seria refletida no container em execução apenas no momento desejado. As imagens Docker podem utilizar tags e assim viabilizar o armazenamento de múltiplas versões da mesma aplicação, isso quer dizer que em caso de problema na atualização, o plano de retorno seria basicamente utilizar a imagem com a tag anterior.

3 – Padronização e replicação

Como as imagens Docker são construídas através de arquivos de definição[4], é possível garantir que um determinado padrão seja seguido e assim aumentar confiança em sua replicação, ou seja, basta que as imagens sigam as melhores práticas[5] de construção para que seja viável escalarmos[6] a nossa estrutura rapidamente.

Caso a equipe receba um nova pessoa para trabalhar no desenvolvimento, essa poderá receber o ambiente de trabalho em alguns comandos. Esse processo durará o tempo do download das imagens a se utilizar, assim como os arquivos de definições da orquestração das mesmas, ou seja, isso quer dizer que no início de uma nova pessoa no processo de desenvolvimento da aplicação ela poderá

[2] https://www.thoughtworks.com/continuous-delivery
[3] http://techfree.com.br/2015/12/entendendo-armazenamentos-de-dados-no-docker/
[4] https://docs.docker.com/engine/reference/builder/
[5] https://docs.docker.com/engine/userguide/eng-image/dockerfile_best-practices/
[6] https://pt.wikipedia.org/wiki/Escalabilidade

reproduzir rapidamente o ambiente em sua estação e assim desenvolver códigos seguindo o padrão da equipe.

Na necessidade de se testar uma nova versão de uma determinada parte da sua solução, usando imagens Docker normalmente basta a mudança de um ou mais parâmetros de um arquivo de definição para se iniciar um ambiente modificado com a versão requerida para a avaliação, ou seja, criar e modificar sua infraestrutura ficou bem mais fácil e rápido.

4 – Idioma comum entre Infraestrutura e desenvolvimento

A sintaxe usada para parametrizar as imagens e ambientes Docker pode ser considerada um idioma comum entre áreas que costumeiramente não dialogavam bem, ou seja, agora é possível ambos setores apresentarem propostas e contra propostas com base em um documento em comum.

A infraestrutura requerida estará presente no código do desenvolvedor e a área de infraestrutura poderá analisar esse documento, sugerindo mudanças para adequação de padrões do seu setor ou não. Tudo isso em comentários e aceitação de *merge* ou *pull request* do sistema de controle de versão de códigos.

5 – Comunidade

Assim como hoje é possível acessar o github[7] ou gitlab[8] a procura de exemplos de código, usando o repositório de imagens do Docker[9] é possível conseguir alguns bons modelos de infraestrutura de aplicações, assim como serviços prontos para integrações complexas.

Um exemplo é o nginx[10] como proxy reverso e mysql[11] como banco de dados. Caso sua aplicação necessite desses dois recursos, você não precisa perder tempo instalando e configurando totalmente esses serviços, basta utilizar as imagens do repositório, configurando parâmetros mínimos para adequação com seu ambiente. Normalmente as imagens oficiais seguem as boas práticas de uso dos serviços oferecidos.

Utilizar essas imagens não significa ficar "refém" da configuração trazida com elas, pois é possível enviar sua própria configuração para esses ambientes e evitar apenas o trabalho da sua instalação básica.

Dúvidas

Algumas pessoas enviaram dúvidas relacionadas as vantagens que explicitei nesse texto, sendo assim ao invés de respondê-las pontualmente, resolvi publicar as perguntas e minhas respectivas respostas aqui.

[7] http://github.com/
[8] https://about.gitlab.com/
[9] http://hub.docker.com/
[10] https://hub.docker.com/_/nginx/
[11] https://hub.docker.com/_/mysql/

Qual a diferença entre imagem Docker e definições criadas por ferramenta de automação de infraestrutura?

Como exemplo de ferramentas de automação de infraestrutura temos o Puppet[12], Ansible[13] e Chef[14]. Todas elas podem garantir ambientes parecidos, uma vez que faz parte do seu papel manter uma determinada configuração no ativo desejado.

A diferença entre a solução Docker e gerência de configuração pode parecer bem tênue, pois ambas podem suportar a configuração necessária de toda infraestrutura que uma aplicação demanda para ser implantada, mas acho que uma das distinções mais relevante está no fato da imagem ser uma abstração completa e não requerer qualquer tratamento para lidar com as mais variadas distribuições GNU/Linux existentes, pois a imagem Docker carrega em si uma copia completa dos arquivos de uma distribuição enxuta.

Carregar em si essa cópia de uma distribuição GNU/Linux normalmente não é um problema para o Docker, pois utilizando o modelo de camadas do Docker ele economizará bastante recurso reutilizando as camadas de base. Leia esse artigo[15] para entender um pouco mais de armazenamento do Docker.

Outra vantagem da imagem em relação a gerência de configuração é o fato de que utilizando a imagem é possível disponibilizar o pacote completo da aplicação em um repositório e esse "produto final" ser utilizado facilmente sem necessidade de configuração completa. Apenas um arquivo de configuração e apenas um comando normalmente são o suficiente para iniciar uma aplicação criada como imagem Docker.

Ainda sobre o processo de imagem Docker como produto no repositório, isso pode ser usado também no processo de atualização da aplicação, como já explicamos nesse capítulo.

O uso da imagem base no Docker de uma determinada distribuição não é a mesma coisa que criar uma definição de gerência de configuração para uma distribuição?

Não! A diferença está na perspectiva do hospedeiro, pois no caso do Docker não importa qual distribuição GNU/Linux utilizada no host, pois há uma parte da imagem que carrega todos os arquivos de uma mini distribuição, que será o suficiente para suportar tudo que a aplicação precisa, ou seja, caso seu Docker host seja Fedora e o fato de sua aplicação em questão precisa de arquivos do Debian, não se preocupe, pois a imagem em questão trará arquivos Debian para suportar seu ambiente e como já foi dito anteriormente, isso normalmente não chega a impactar negativamente no consumo de espaço em disco.

[12] https://puppetlabs.com/

[13] https://www.ansible.com/

[14] https://www.chef.io/chef/

[15] http://techfree.com.br/2015/12/entendendo-armazenamentos-de-dados-no-docker/

Quer dizer então que agora eu, desenvolvedor, preciso me preocupar com tudo da Infraestrutura?

Não! Quando citamos que é possível o desenvolvedor especificar a infraestrutura, estamos falando que é aquela camada mais próxima da aplicação e não toda arquitetura necessária (Sistema operacional básico, regras de firewall, rotas de rede e etc).

A ideia do Docker é que os assuntos relevantes e diretamente ligados a aplicação possam ser configurados pelo desenvolvedor, mas isso não o obriga a realizar essa atividade, é uma possibilidade que agrada muitos desenvolvedores, mas caso não seja esse sua situação, pode ficar tranquilo que outra equipe tratará dessa parte. Apenas o seu processo de implantação será um pouco mais lento.

Muitas pessoas falam de Docker para micro serviços, é possível usar o Docker para aplicações monolíticas?

Sim! Porém em alguns casos será necessário fazer pequenas modificações em sua aplicação, para que ela possa usufruir das facilidades do Docker. Um exemplo comum é o log, que normalmente a aplicação envia para determinado arquivo de log, ou seja, no modelo Docker as aplicações que estão nos containers não devem tentar escrever ou gerir arquivos de logs. Ao invés disso, cada processo em execução escreve seu próprio fluxo de evento, sem buffer, para o stdout[16], pois o Docker tem drivers específicos para tratar o log enviado dessa forma. Essa parte de melhores práticas de gerenciado de logs será detalhada em capítulos posteriores.

Em alguns momento você perceberá que o uso do Docker para sua aplicação demanda muito esforço, nesses casos normalmente o problema está mais em como sua aplicação trabalha do que na configuração do Docker. Esteja atento.

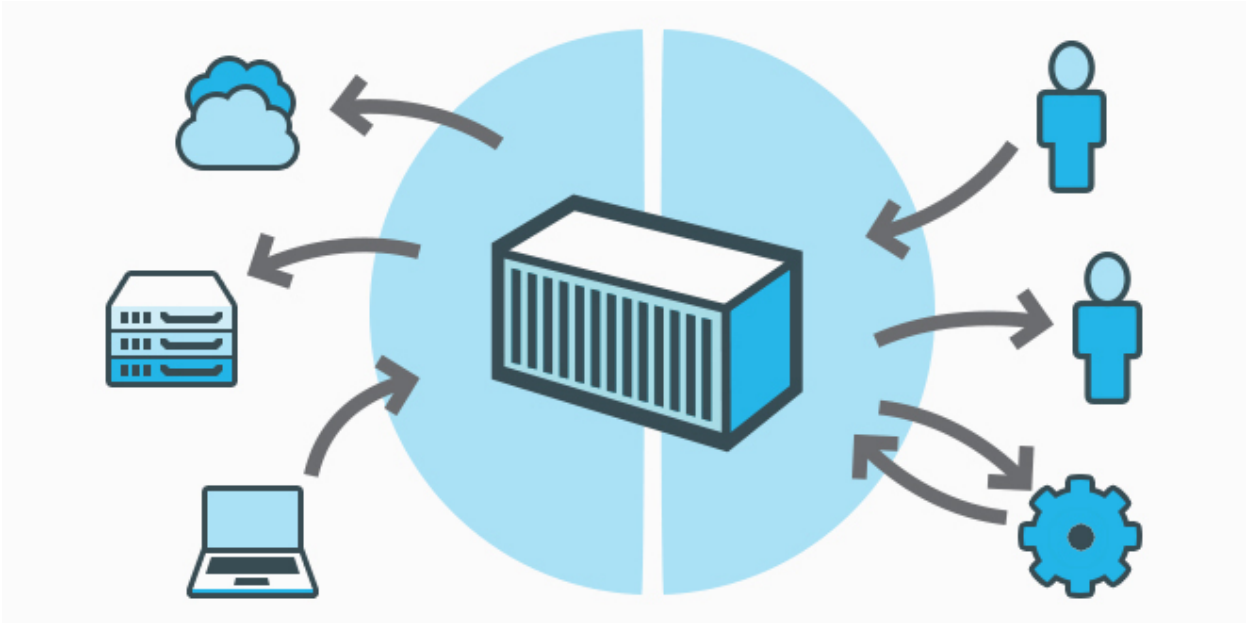
Você tem mais dúvidas e/ou bons motivos para utilizar Docker? Comente aqui[17].

[16] https://pt.wikipedia.org/wiki/Fluxos_padr%C3%A3o

[17] http://techfree.com.br/2016/03/porque-usar-docker/

O que é Docker

De forma bem resumida podemos dizer que o Docker é uma plataforma aberta criada com objetivo de facilitar o desenvolvimento, implantação e execução de aplicações em ambientes isolados. Ela foi desenhada especialmente para disponibilizar sua aplicação da forma mais rápida possível.



Usando o Docker você pode gerenciar facilmente a infraestrutura da sua aplicação, ou seja, isso agilizará o processo de criação, manutenção e modificação do seu serviço.

Todo processo é realizado sem a necessidade de qualquer acesso privilegiado a sua infraestrutura corporativa, ou seja, a equipe responsável pela aplicação poderá participar da especificação do ambiente junto com a equipe responsável pelos servidores.

O Docker viabilizou uma "linguagem" comum entre desenvolvedores e administradores de servidores. Esse novo "idioma" é utilizado para construir arquivos com as definições da infraestrutura necessária, como a aplicação será disposta nesse ambiente, em qual porta fornecerá seu serviço, quais dados de volumes externos serão requisitados e outras possíveis necessidades.

O Docker disponibiliza também uma nuvem pública para compartilhamento de ambientes prontos, que podem ser utilizados para viabilizar customizações para ambientes específicos, ou seja, é possível obter uma imagem pronta do apache e configurar os módulos específicos necessários para sua aplicação e assim criar seu próprio ambiente customizado. Tudo isso com poucas linhas de descrição.

O Docker utiliza o modelo de container para "empacotar" sua aplicação, que após ser transformada em uma imagem Docker, poderá ser reproduzida em plataforma de qualquer porte, ou seja, caso

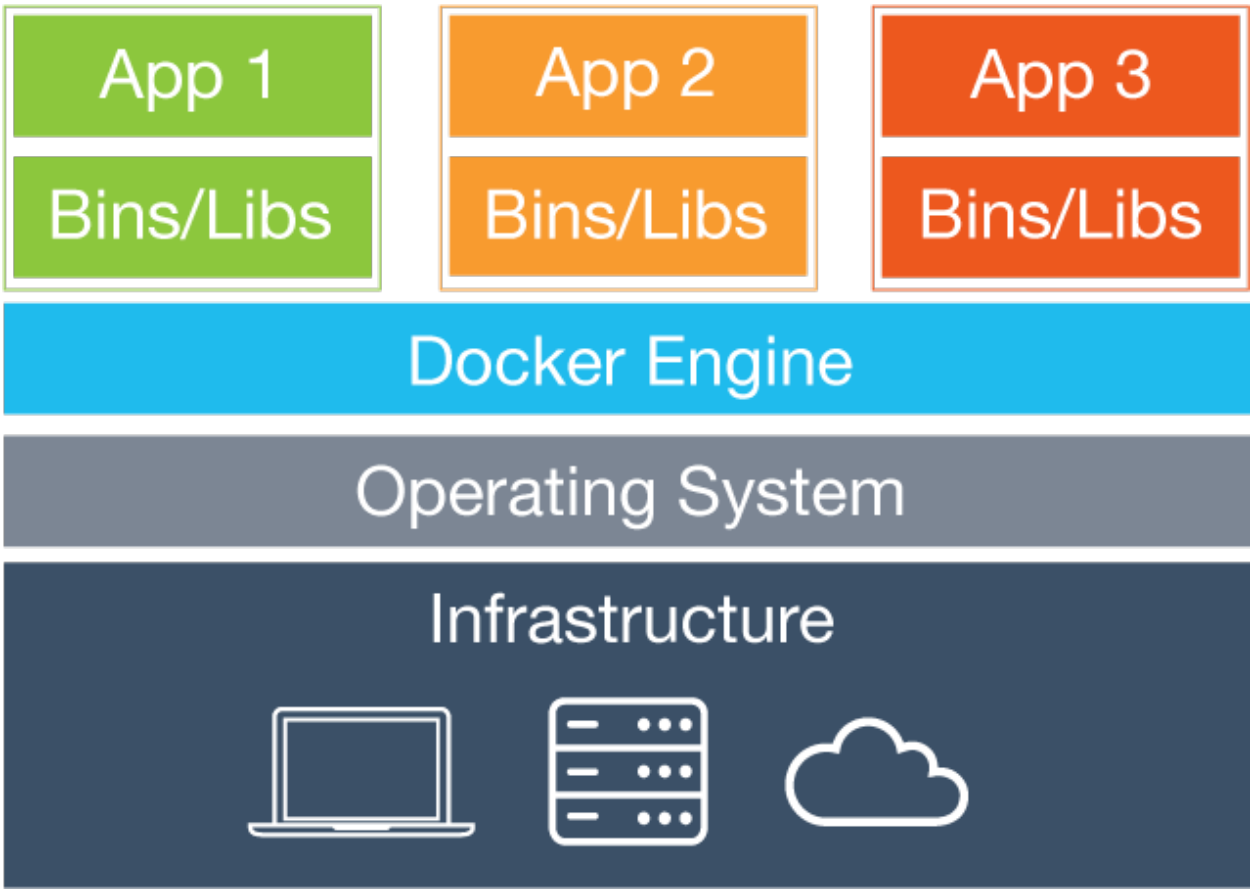
sua aplicação funcione sem falhas em seu notebook, ela funcionará também no servidor ou no mainframe. Construa uma vez, execute onde quiser.

Os containers são isolados a nível de disco, memória, processamento e rede. Essa separação permite uma grande flexibilidade, onde ambientes distintos podem coexistir no mesmo host, sem causar qualquer problema. Vale salientar que o overhead nesse processo é o mínimo necessário, pois cada container normalmente carrega apenas um processo, que é aquele responsável pela entrega do serviço desejado, em todo caso esse container também carregam todos os arquivos necessários (configuração, biblioteca e afins) para sua execução completamente isolada.

Outro ponto interessante no Docker é sua velocidade para viabilizar o ambiente desejado, pois como é basicamente o início de um processo e não um sistema operacional inteiro, o tempo de disponibilização é normalmente medido em segundos.

Virtualização a nível do sistema operacional

O modelo de isolamento utilizado no Docker é a virtualização a nível do sistema operacional, que é um método de virtualização onde o kernel do sistema operacional permite que múltiplos processos sejam executados isoladamente no mesmo host. Esses processos isolados em execução são denominados no Docker de container.



Para criar o isolamento necessário desse processo, o Docker usa a funcionalidade do kernel denominada de namespaces[18], que cria ambientes isolados entre containers, ou seja, os processos de uma aplicação em execução não terão acesso aos recursos de outra, a não ser que isso seja expressamente liberado na configuração de cada ambiente.

Para evitar a exaustão dos recursos da máquina por apenas um ambiente isolado, o Docker usa a funcionalidade cgroups[19] do kernel, que é responsável por criar limites de uso do hardware a disposição. Com isso é possível coexistir no mesmo host diferentes containers, sem que um afete diretamente o outro por uso exagerado dos recursos compartilhados.

[18] http://man7.org/linux/man-pages/man7/namespaces.7.html
[19] https://en.wikipedia.org/wiki/Cgroups

Instalação

O Docker há algum tempo já deixou de ser apenas um software para virar um conjunto deles. Um ecosistema.

Nesse ecosistema temos os seguintes softwares:

- **Docker Engine**: É o software base de toda solução. É tanto o daemon que responsável pelo os containers como também é o cliente usado para enviar comandos pro daemon.
- **Docker Compose**: É o 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 vamos citar o Swarm[20] e outras ferramentas por não estarem alinhados com o objetivo desse livro, que é ser introdutório para os desenvolvedores.

Instalando no GNU/Linux

Será explicado a instalação da forma mais genérica possível, ou seja, 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 muito simples. Acesse o seu terminal preferido do GNU/Linux e se torne usuário root:

```
1 su - root
```

ou no caso da utilização de sudo

```
1 sudo su - root
```

Agora execute o comando abaixo:

[20] https://docs.docker.com/swarm/overview/

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

Aconselho fortemente que leia o script que está sendo executado no seu sistema operacional. Acesse esse link[21] e analise o código assim que tiver tempo para fazê-lo.

Esse procedimento demorará um pouco. Após terminar teste executando o comando abaixo:

```
docker run hello-world
```

Tratamento de possíveis problemas

Se o acesso a internet da sua máquina passar por um controle de tráfego (aquele que bloqueia o acesso a determinadas páginas) você poderá encontrar problemas no passo do **apt-key**, sendo assim caso passe por 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 se torne usuário root:

```
su - root
```

ou no caso da utilização de sudo

```
sudo su - root
```

Agora 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
```

Instalando Docker compose com pip

O pip[22] é um gerenciador de pacotes Python, e como o docker-compose é escrito nessa linguagem, é possível instalá-lo desse jeito:

[21] https://get.docker.com/

[22] https://en.wikipedia.org/wiki/Pip_(package_manager)

```
pip install docker-compose
```

Docker machine no GNU/Linux

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

```
su - root
```

ou no caso da utilização de sudo

```
sudo su - root
```

Agora execute o comando abaixo:

```
$ curl -L https://github.com/docker/machine/releases/download/v0.6.0/docker-mach\
ine-`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
```

Instalando no MacOS

A instalação das ferramentas do Ecosistema Docker no MacOS será realizada através de um único grande pacote, que se chama **Docker Toolbox**.

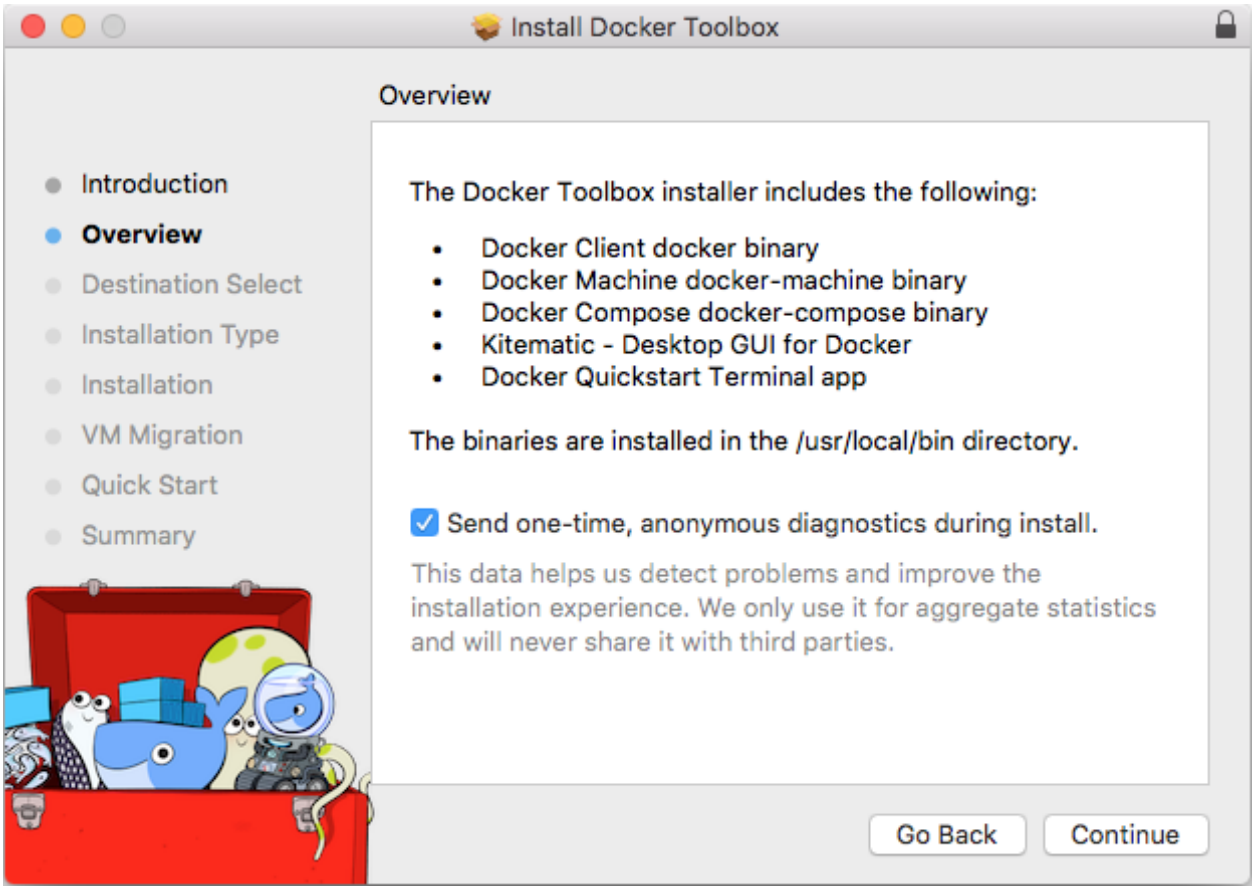
Você pode instalar via brew cask com o comando abaixo:

```
brew cask install dockertoolbox
```

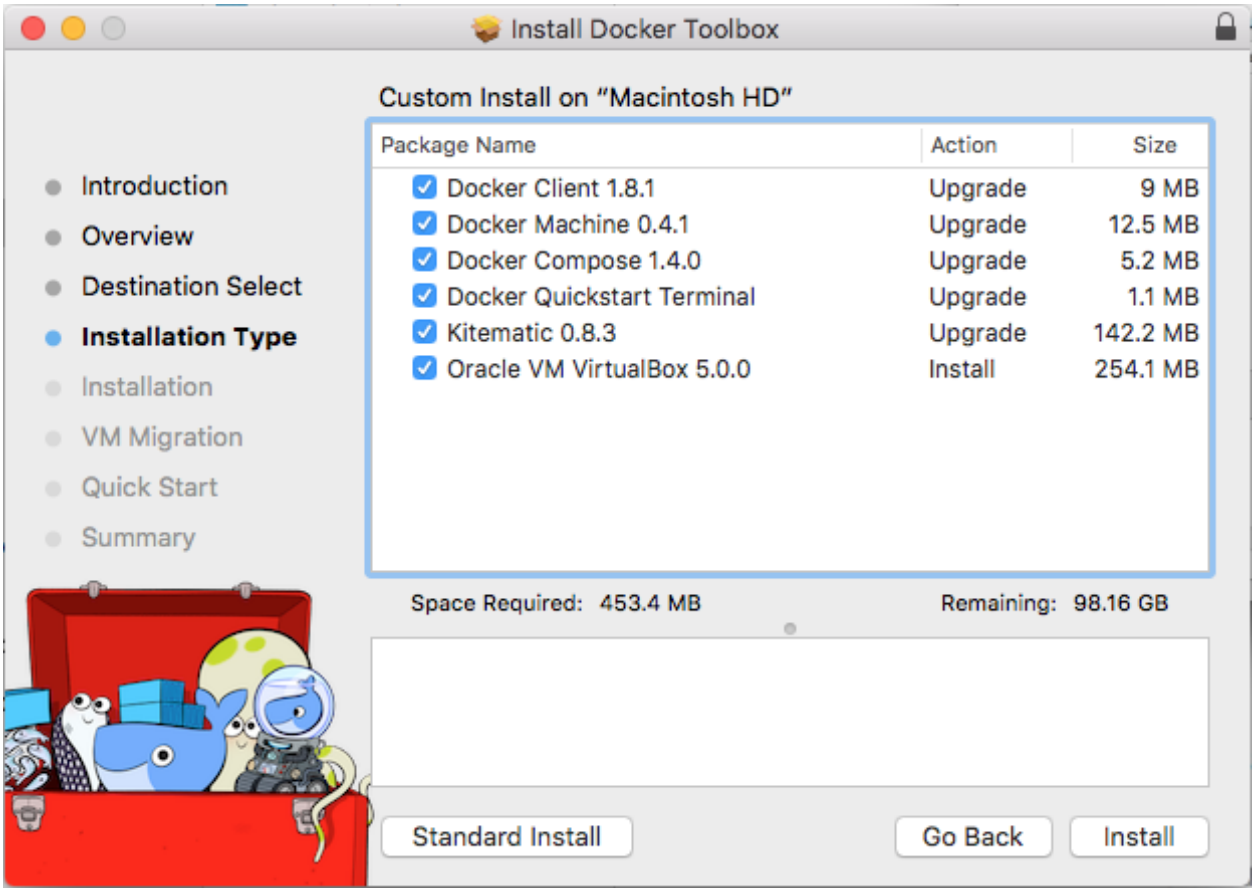
Você pode instalar manualmente acessando a página de download[23] do **Docker toolbox** e baixando o instalador correspondente ao MacOS.

Após duplo clique no instalador, verá essa tela:

[23] https://www.docker.com/products/docker-toolbox



Apenas clique em **Continue**.



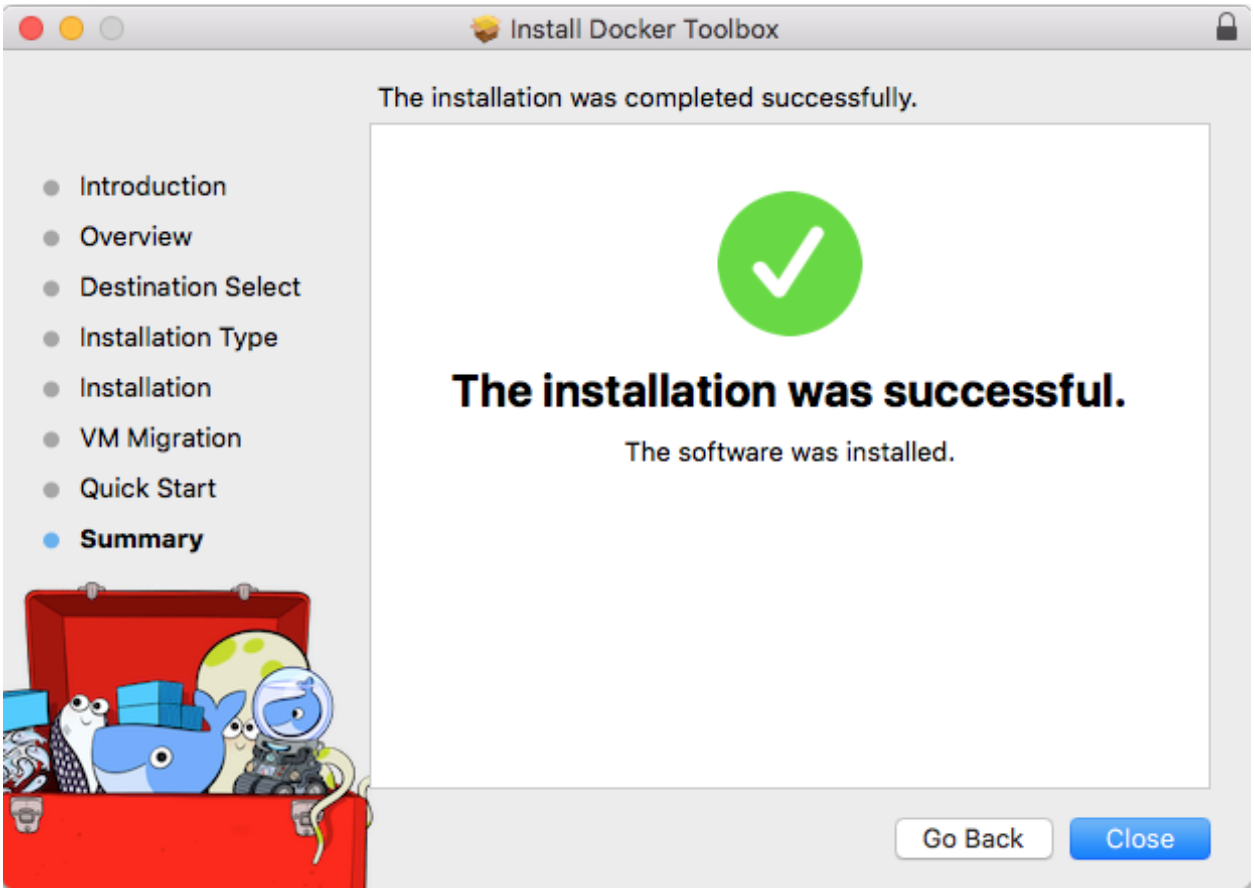
Marque todas as opções e clique **Install**.

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

Na próxima tela será apenas 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 nesse 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:

```
1 docker run hello-world
```

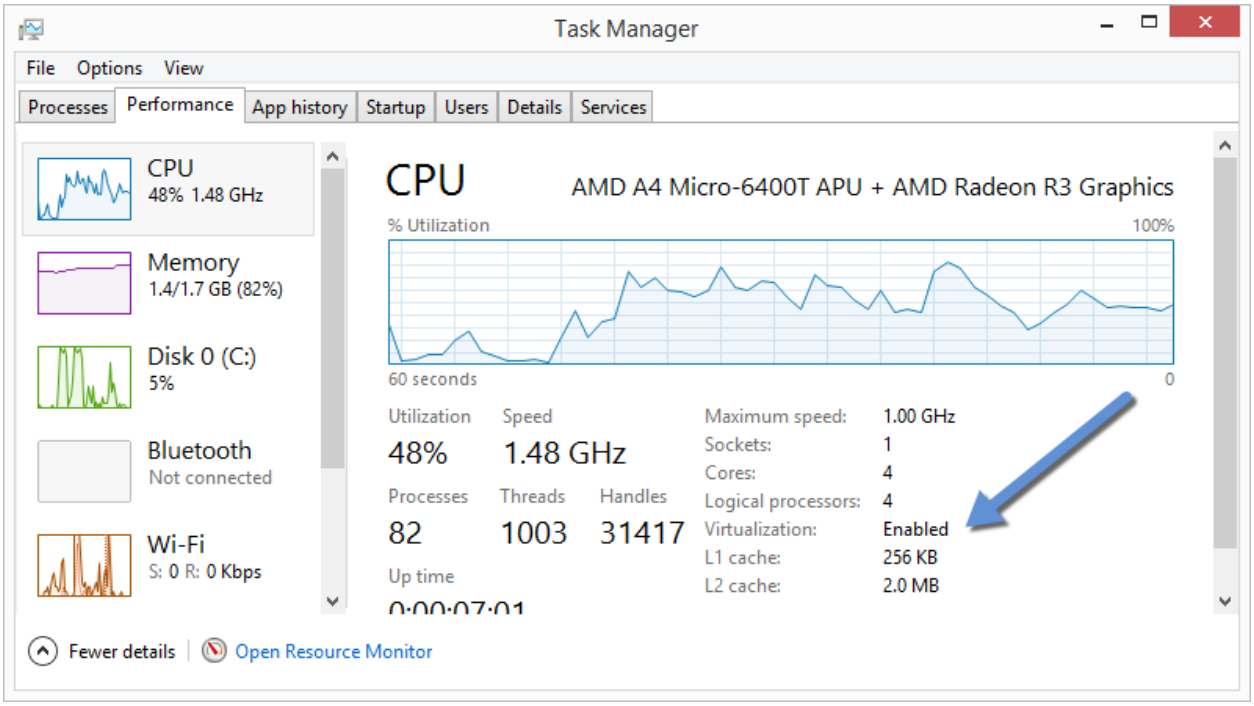
Instalando no Windows

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

O **Docker Toolbox** funcionará apenas em versões 64bit[24] do Windows e apenas as versões superiores ao Windows 7.

É importante atentar também que é necessário que o suporte a virtualização esteja habilitado. Na versões 8 do Windows é possível verificar através do **Task Manager**, na aba **Performance**, clicando em **CPU** é possível visualizar a janela abaixo:

[24] https://support.microsoft.com/en-us/kb/827218



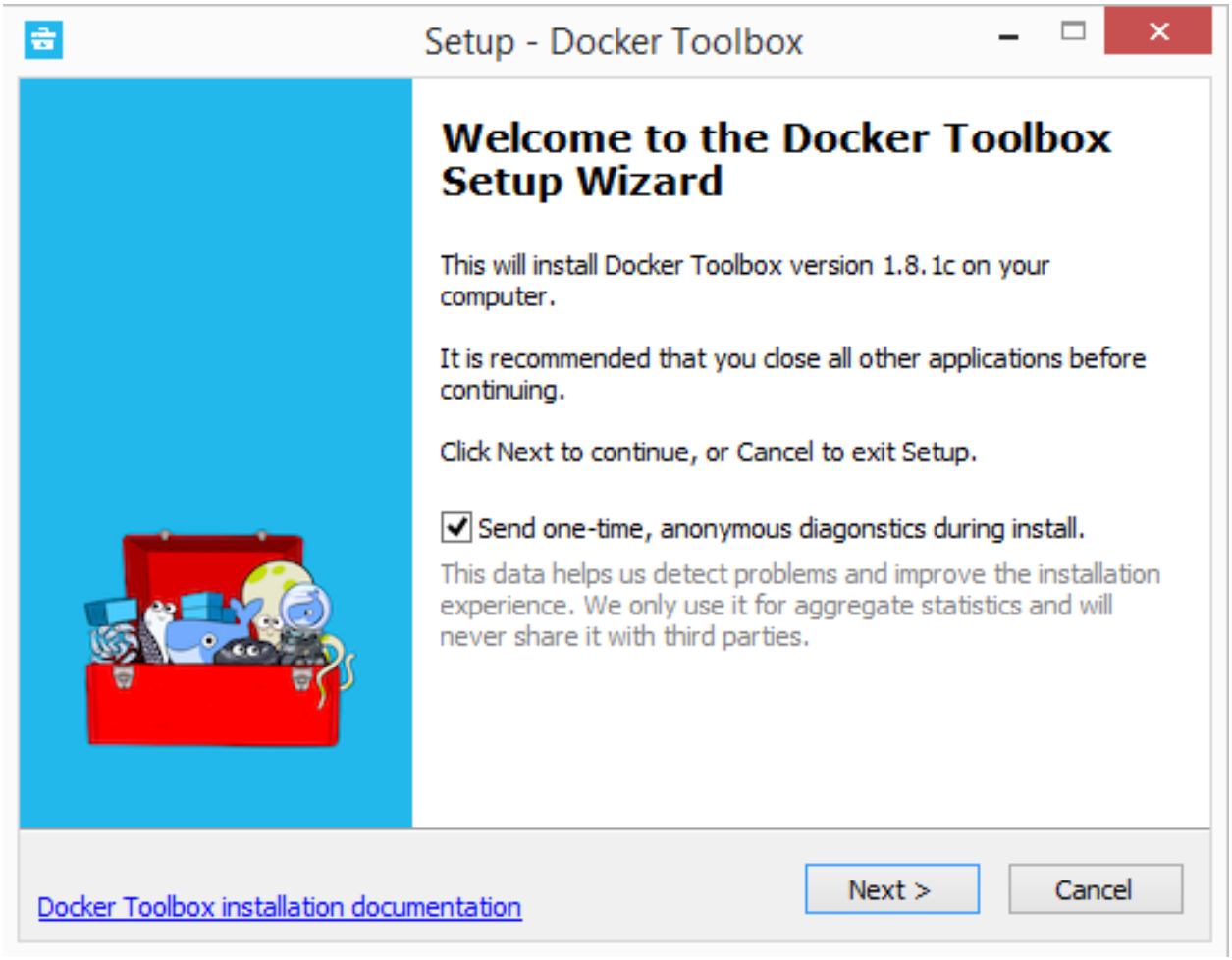
Para verificar o suporte a virtualização do Windows 7, utilize esse link[25] para maiores informações.

Instalando o Docker Toolbox

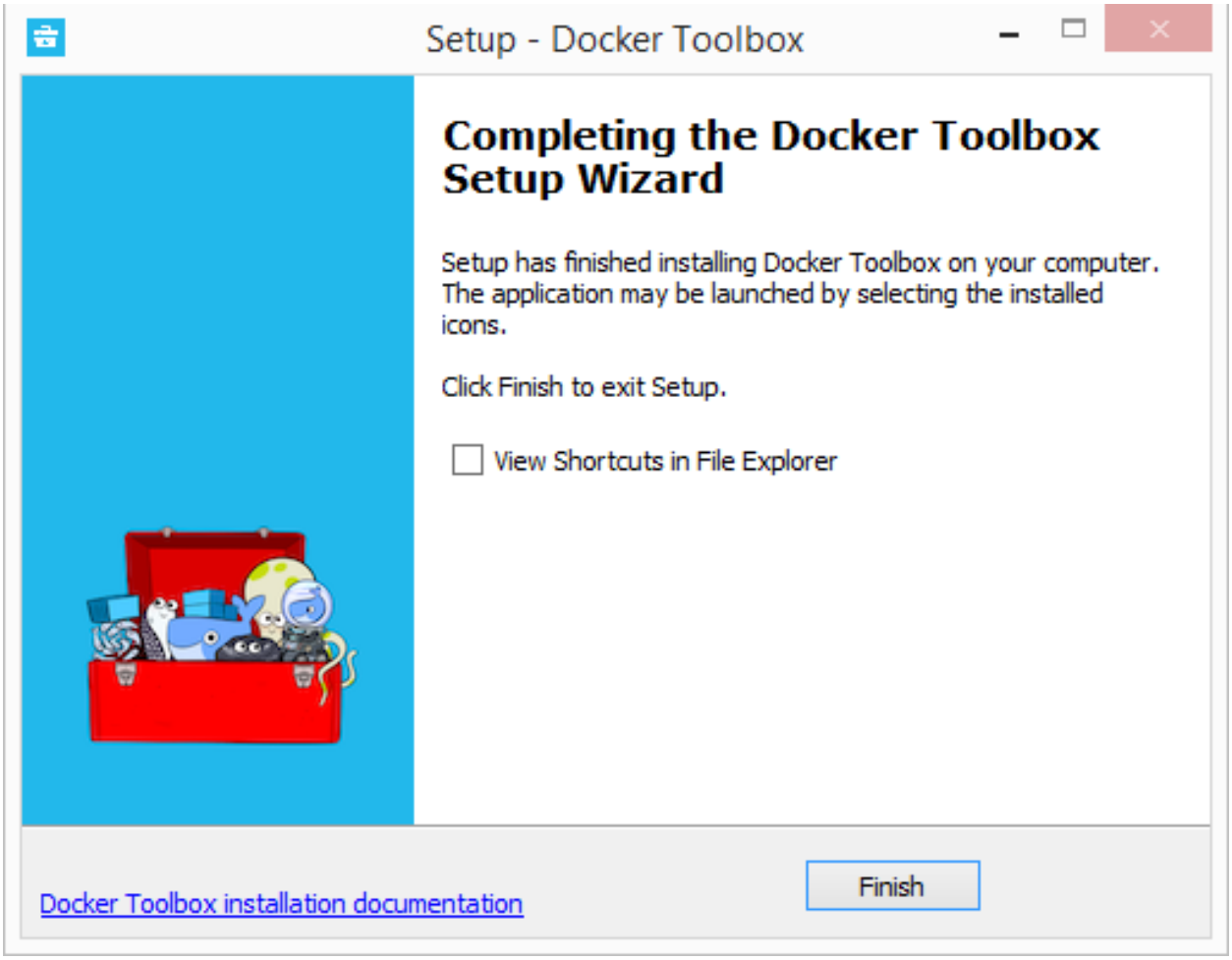
Acesse a página de download[26] do **Docker toolbox** e baixe o instalador correspondente ao Windows.

Após duplo clique no instalador, verá essa tela:

[25] http://www.microsoft.com/en-us/download/details.aspx?id=592
[26] https://www.docker.com/products/docker-toolbox



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:

```
1 docker 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 eles tem capítulos específicos para seu detalhamento.

Executando um container

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

```
1  docker images
```

As imagens retornadas estão presentes no seu **Docker host** e não demandarão nenhum download da nuvem pública do Docker[27], a não ser que você deseje atualizá-la, caso exista o que possa ser atualizado. Para atulizar a imagem basta executar o comando abaixo:

```
1  docker pull python
```

Usei a imagem chamada **python** como um 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 utilizar o comando abaixo:

```
1  docker inspect python
```

O comando inspect[28] é responsável por informar todos os dados referentes a essa imagem.

Agora que temos a imagem atualizada e inspecionada, podemos iniciar o container, mas antes de simplesmente copiarmos a colarmos o comandos, vamos entender como ele realmente funciona.

[27] hub.docker.com

[28] https://docs.docker.com/engine/reference/commandline/inspect/

```
docker 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 continer após finalização (**Não funciona com -d**) –name | Nomear o container -v | Mapeamento de volume -p | Mapemento de porta -m | Limitar o uso de memória RAM -c | Balancear o uso de CPU

Segue um exemplo simples no seguinte comando:

```
docker run -it --rm --name meu_python python bash
```

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

Vale lembra que caso o **CMD** não seja especificado no comando **docker run** será utilizado o valor padrão definido no **Dockerfile** da imagem utilizada em questão, que no nosso caso é **python** e seu comando padrão seria executar 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 ele será montado dentro do container.

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

O uso de armazenamento será melhor explicado em capítulos futuros, sendo assim não vamos detalhar o uso desse parâmetro.

Mapeamento de portas

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

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

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

```
1 docker run -it --rm -p 80:8080 python
```

Com o comando acima teríamos a porta **80** acessível no **Docker host** que repassaria todas conexões para a porta **8080** dentro do **container**, ou seja, não seria possível acessar a porta **8080** no endereço IP do **Docker host**, pois essa porta estaria 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, pois são os mais utilizados.

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

```
1 docker 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, que nesse caso é o **1024**.

Usaremos como exemplo o peso **512**:

```
1 docker 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 com o peso **512** isso quer dizer que 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 list de containers de um determinado **Docker host** utilizamos o comando docker ps[29].

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

[29] https://docs.docker.com/engine/reference/commandline/ps/

```
docker ps <parâmetros>
```

Os parâmetros mais utilizados na execução do container são:

Parâmetro | Explicação
———–|———– -a | Lista todos os containers, inclusive os desligados -l | Lista os ultimos 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 sua utilização com novos comandos.

Caso deseje desligar o container basta utilizar o comando docker stop[30]. Ele recebe como argumento o **ID** ou **nome** do container. Ambos dados podem ser obtidos com o **docker ps**, que foi explicado no tópico anterior.

Um exemplo de uso seria:

```
docker 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 ele 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[31]:

```
docker start meu_python
```

> Vale ressaltar que a ideia dos containers é serem descartáveis, ou seja, caso você esteja usando o **mesmo** container por muito tempo sem descartá-lo, você provavelmente estará usando o Docker incorretamente. O Docker **não** é uma máquina, é um processo em execução e como todo processo ele deve ser descartado para que outro possa tomar seu lugar na reinicialização do mesmo.

[30] https://docs.docker.com/engine/reference/commandline/stop/
[31] https://docs.docker.com/engine/reference/commandline/start/

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 diferença de Imagem e Container?

Fazendo um paralelo com o conceito de orientação a objeto[32], a **imagem** é a classe e o **container** o objeto, ou seja, a imagem é a abstração da infraestrutura em um estado somente leitura, que é 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, ou seja, caso deseje um comportamento diferente será necessário customizar a imagem.

Anatomia de uma imagem

As imagens podem ser oficial ou não oficial.

Imagens oficiais e não oficiais

As imagens oficiais da docker são aquelas que não tem usuário em seu nome, ou seja, a imagem "**ubuntu:16.04**" é oficial, por outro lado a imagem "nuagebec/ubuntu"[33] não é oficial. Essa segunda imagem é mantida pelo usuário nuagebec[34], que mantém muitas outras imagens não oficiais.

As imagens oficiais são mantidas pela empresa docker e são disponibilizadas[35] na nuvem docker.

O objetivo das imagens oficiais é prover um ambiente básico (ex. debian, alpine, ruby, python) que serve de ponto de partida para criação de imagens pelos usuários, que é o que 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.

[32] https://pt.wikipedia.org/wiki/Orienta%C3%A7%C3%A3o_a_objetos
[33] https://hub.docker.com/r/nuagebec/ubuntu/
[34] https://hub.docker.com/u/nuagebec/
[35] https://hub.docker.com/explore/

Nome da imagem

O nome de uma imagem oficial é composta de duas partes. A primeira é o que a documentação[36] chama de "**repositório**" e a segunda é a "**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 de conjunto de imagens, ou seja, não confunda com o local de armazenamento das imagens, que detalharemos mais a frente.

Para o docker a "**tag**" é uma abstração para criar uma 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 esse comando[37] abaixo para visualizar todas as imagens que se encontram localmente na sua estação nesse momento:

```
1 docker images
```

Como se cria 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[38] relacionado a um container, ou seja, esse comando pegará o status atual desse container escolhido e criará uma imagem com base nele.

Vamos ao exemplo. Primeiro vamos criar um container qualquer:

```
1 docker run -it --name containercriado ubuntu:16.04 bash
```

Agora que estamos no bash do container, vamos instalar o nginx nele:

```
1 apt-get update
2 apt-get install nginx -y
3 exit
```

Vamos parar o container com o comando abaixo:

[36] https://docs.docker.com/engine/userguide/containers/dockerimages/
[37] https://docs.docker.com/engine/reference/commandline/images/
[38] https://docs.docker.com/engine/reference/commandline/commit/

```
docker stop containercriado
```

Agora vamos efetuar o **commit** desse **container** em uma **imagem**:

```
docker commit containercriado meuubuntu:nginx
```

No exemplo do comando acima, **containercriado** é o nome do container que criamos e modificamos nos passos anteriores e o nome **meuubuntu:nginx** é a imagem resultante do **commit**, ou seja, o estado do **containercriado** será 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 um pacote **nginx** instalado.

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

```
docker images
```

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

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

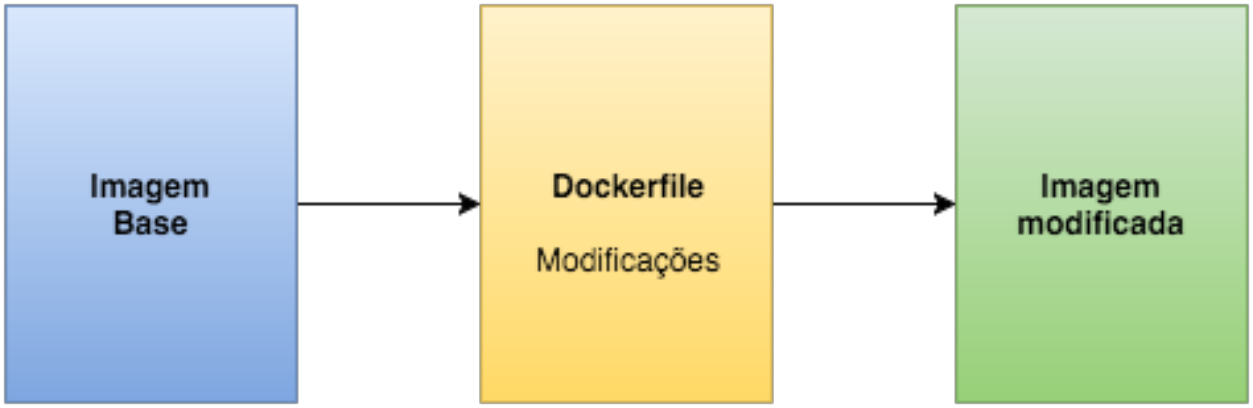
Se quiser validar, pode executar o mesmo comando na imagem oficial do ubuntu:

```
docker 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 pudemos verificar, o processo de modificação da imagem é completamente manual e apresenta certa dificuldade para rastrear as mudanças que foram 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 apresentado uma lista de instruções que serão aplicadas em uma determinada imagem para que outra seja gerada com base nessas 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, ou seja, nesse modelo temos total rastreabilidade sobre o que será modificado essa nova imagem.

Vamos ao mesmo exemplo da instalação do nginx no ubuntu 16.04.

Primeiro crie um arquivo qualquer para um teste futuro:

```
1 touch arquivo_teste
```

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

```
1 FROM ubuntu:16.04
2 RUN apt-get update && apt-get install nginx -y
3 COPY arquivo_teste /tmp/arquivo_teste
4 CMD bash
```

No arquivo acima eu utilizei quatro instruções[39]:

FROM é usado para informar qual imagem usarei como base, que nesse caso foi a **ubuntu:16.04**.

RUN é usado 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 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.

[39] https://docs.docker.com/engine/reference/builder/

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

Após construir seu Dockerfile basta executar o comando[40] abaixo:

```
1 docker build -t meuubuntu:nginx_auto .
```

O comando acima tem a opção "**-t**" que serve para informar o nome da imagem que será criada, que no nosso caso será **meuubuntu:nginx_auto** e o "." no final informa qual o contexto que deve ser usado nessa construção de imagem, ou seja, todos os arquivos da sua 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 que é lido do topo até sua base e cada linha é executada por vez, ou seja, caso alguma instrução dependa de uma outra instrução, essa dependência deve vir mais acima no documento.

O resultado de cada instrução desse arquivo é armazenado em um cache local, ou seja, caso o **Dockerfile** não seja modificado na próxima criação da imagem (**build**) o processo não demorará, pois tudo estará no cache, mas caso algo seja modificado apenas a instrução modificada e as posteriores serão executadas novamente.

A sugestão pra aproveitar bem o cache do **Dockerfile** é sempre manter instruções que mudem com mais frequência mais próximas da base do documento. Vale lembrar de atender também as dependências entre instruções.

Vamos utilizar um exemplo pra deixar mais claro:

```
1 FROM ubuntu:16.04
2 RUN apt-get update
3 RUN apt-get install nginx
4 RUN apt-get install php5
5 COPY arquivo_teste /tmp/arquivo_teste
6 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, mas a instalação do apache2 será instalado, pois acabou de entrar no arquivo, assim como o php5 e a cópia do arquivo, pois todos eles são subsequentes a linha modificada.

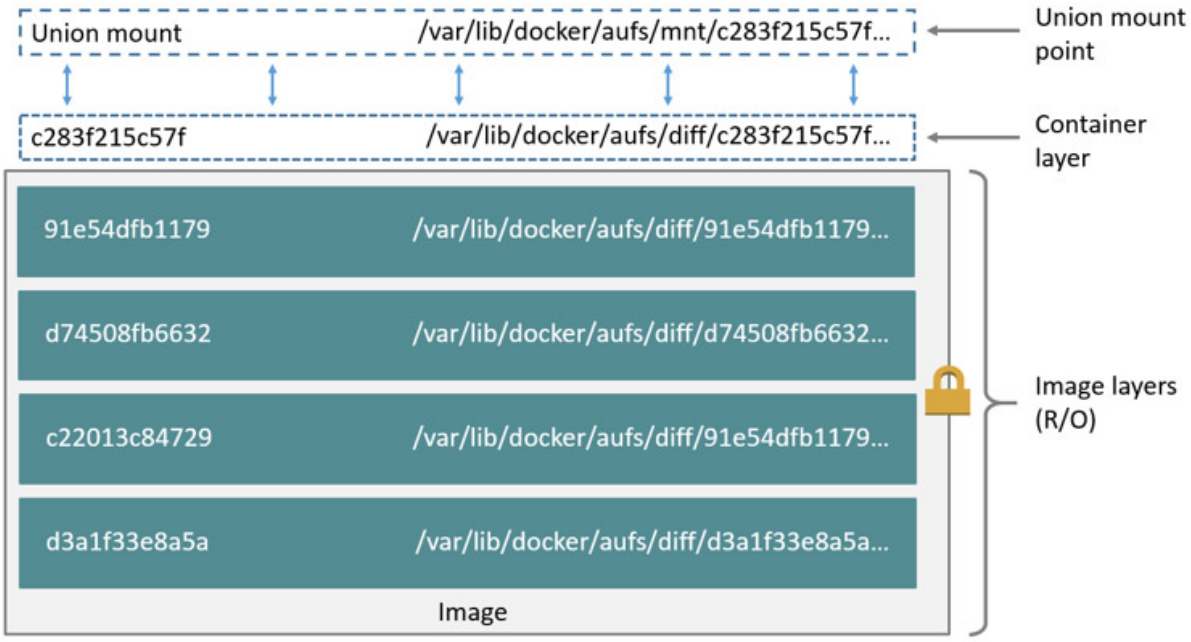
[40] https://docs.docker.com/engine/reference/commandline/build/

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

Enviando sua imagem para nuvem

Entendendo armazenamento no Docker

Para entender como o docker gerencia seus volumes, primeiro precisamos explicar como funciona ao menos um backend[41] de armazenamento do Docker. Faremos aqui com o AUFS, que foi o primeiro e ainda é um 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[42] é um unification filesystem. Isso quer dizer que ele é responsável por gerenciar múltiplos diretórios, empilha-os uns sobre os outros e fornece uma única e unificada visão. Como se todos juntos fossem apenas um diretório apenas.

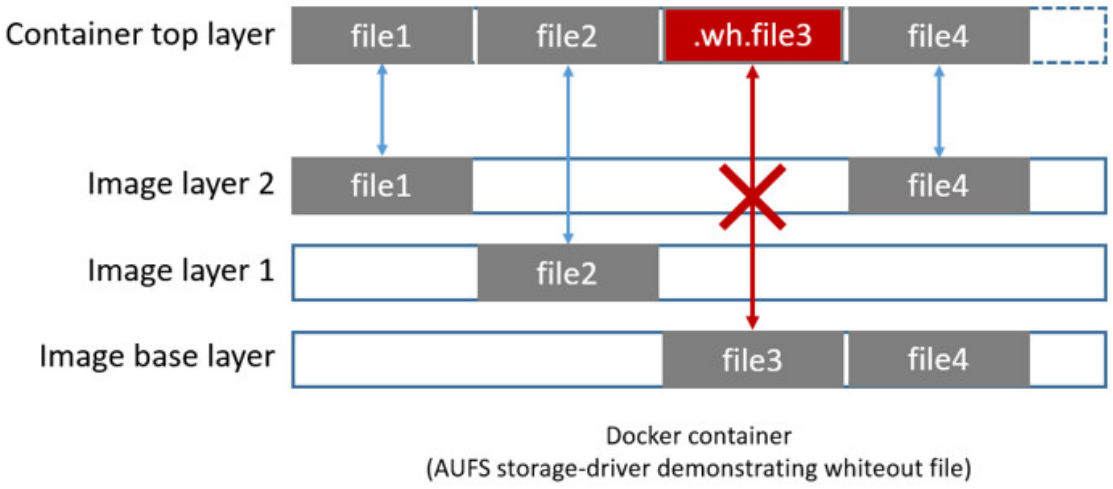
Esse único diretório é o utilizado para apresentar ao container, e funciona como se fosse um único sistema de arquivo comum. Cada diretório usado nessa pilha é correspondente a uma camada, e é

[41] http://searchdatacenter.techtarget.com/definition/back-end
[42] https://en.wikipedia.org/wiki/Aufs

dessa forma que o Docker unifica as camadas e proporciona a reutilização entre containeres, pois o mesmo diretório correspondente a uma 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, pois caso contrário as mudanças de um container poderia interferir em um outro, o que de fato é totalmente contra os princípios do Linux Container.

Caso seja necessário modificar um arquivo que esteja nas camadas (pastas) referentes a imagens é utilizado a tecnologia Copy-on-write[43] (CoW), que é responsável por copiar o arquivo necessário 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.



Removendo um arquivo

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

Problema com performance

Como o Docker aproveita 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 uma pequena parte do arquivo está sendo modificado. Esse comportamento pode ter um impacto notável no desempenho do container, especialmente se os arquivos que estão sendo copiados são grandes e estão localizados abaixo de um monte de camadas de imagem, ou seja, nesse caso o procedimento copy-on-write dedicará muito tempo para uma cópia interna.

[43] https://en.wikipedia.org/wiki/Copy-on-write

Volume como solução para performance

Ao utilizar volumes o Docker montará essa pasta (camada) no nível imediatamente inferior ao do container, o que nesse caso viabilizaria que todo dado armazenado nessa camada (pasta) fosse acessível rapidamente, ou seja, 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 uma 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 mapea ela com uma pasta interna do container (Ex. /var), ou seja, dessa forma tudo que é escrita na pasta /var do container é escrita também na pasta /var/lib/container1 do host.

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

```
1  docker run -v /var/lib/container1:/var ubuntu
```

Esse modelo não é portável, pois necessita que o host tenha uma pasta específica para que o container funciona adequadamente.

Mapeamento via container de dados

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

Segue abaixo um exemplo do uso desse modelo de mapeamento:

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

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

Para consumir esse volume do container, basta utilizar esse comando:

```
1 docker run -d --volumes-from dbdata --name db2 postgres
```

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

Uma desvantagem desse modelo é a necessidade de se manter um container apenas pra isso, pois em alguns ambiente os containeres são removidos com certa regularidade e dessa forma é necessário ter cuidado com esses containeres especiais. O que de uma certa forma é um problema adicional de gerenciamento.

Mapeamento de volumes

Na versão 1.9 do Docker foi acrescentado a possibilidade de se criar volumes isolados de containeres, ou seja, agora é possível criar um volume portável, sem a necessidade de associá-lo a um container especial.

Segue abaixo um exemplo do uso desse modelo de mapeamento:

```
1 docker volume create --name dbdata
```

Como podemos ver 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 a praticada no mapeamento de pasta do host, pois nesse caso você precisa associar o volume a uma pasta dentro do seu container, como podemos ver abaixo:

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

Esse modelo é o mais indicado desde seu lançamento, pois ele proporciona portabilidade, não é removido facilmente quando o container é deletado e ainda assim é bastante fácil sua gestão.

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), dessa forma sempre que eu precisar mencionar esse segundo tipo de rede, citarei sempre como “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:

```
NETWORK ID          NAME                DRIVER
ab09673e9b98        bridge              bridge
763f9ed88176        none                null
242a960a6f20        host                host
```

Bridge

Cada container iniciado no docker é associado a uma rede especifica, e essa é a rede padrão para qualquer container que não foi explicitamente especificado.

Ela confere ao container uma interface que faz bridge com a interface docker0 do docker host. Essa interface receberá 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, ou seja, caso você saiba qual endereço IP do container que se deseja conectar, é possível enviar tráfego pra ele, pois a final de contas estão todos na mesma rede IP (172.17.0.0/16).

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

> Vale ressaltar que “–link” é uma opção já defasada e seu uso é desaconselhado, explicarei esta funcionalidade apenas pra efeito de entendimento do legado. Essa função foi substituída por um DNS embutido do docker, que 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, ou seja, caso você inicie um container a partir da imagem docker do mysql com nome "bd" e em seguida inicie um 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 containers:

```
1 docker run -d --name bd -e MYSQL_ROOT_PASSWORD=minhasenha mysql
2
3 docker run -d -p 80:80 --name app --link db tutum/apache-php
```

Após executar os comandos, o container que tem o nome "app" poderia 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 um comando dentro de um container já existente. Para tal usaremos o nome do container como parâmetro do comando abaixo:

```
1 docker exec -it app ping db
```

O comando acima será responsável por executar o comando "ping db" dentro do container "app", ou seja, o container "app" enviará pacotes icmp, que é normalmente usado para testar conectividade entre dois hosts, para o endereço "db". Esse 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 pegou 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 uma certa confusão, pois ela 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 foi informado no parágrafo anterior, ele apenas facilita a tradução de nomes para o IP dinâmico obtido na inicialização.

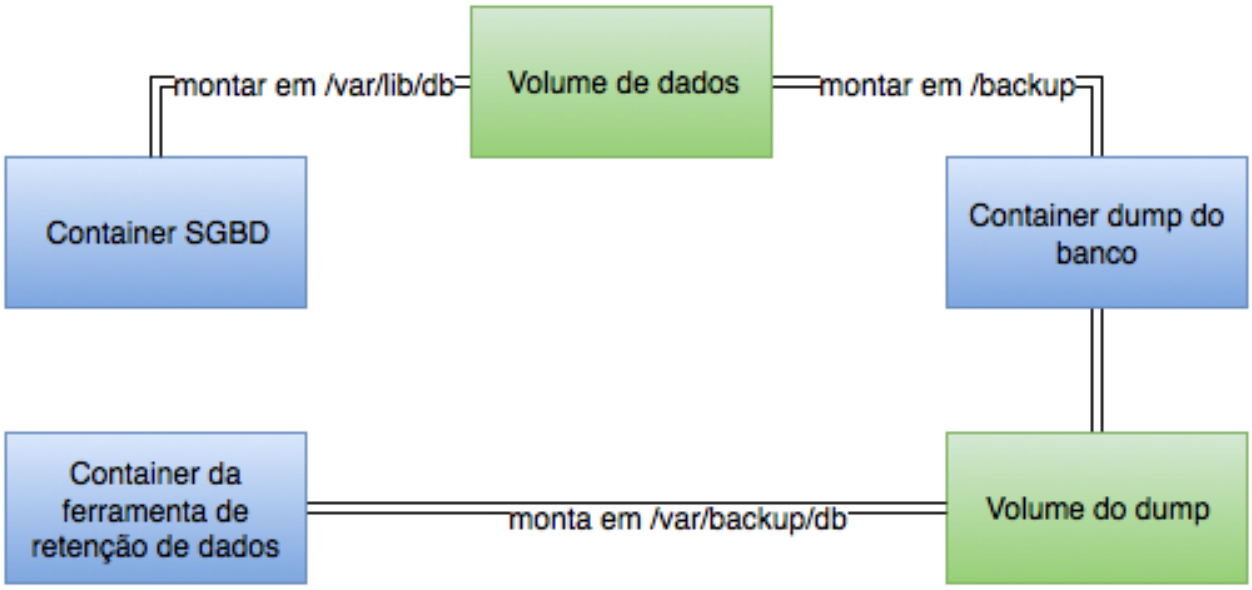
Os containers configurados para essa rede terão a possibilidade de trafego externo, utilizando as rotas das redes IP definidas no docker host, ou seja, caso o docker host tenha acesso a internet, automaticamente os containers em questão também terão.

Nessa rede é possível expor portas dos containers para todos os ativos que conseguem acessar o docker host.

None

Essa rede tem como objetivo isolar o container para comunicações externas, ou seja, ela não receberá nenhuma interface para comunicação externa. A sua única interface de rede IP será a localhost.

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



Exemplo de uso da rede none

Em caso de duvida sobre utilização de volumes no docker. Visite esse artigo[44] e entenda um pouco 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 isso 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 sejam criadas redes pelo usuário, essas redes são associadas ao que docker chama de driver de rede.

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

[44] http://techfree.com.br/2015/12/entendendo-armazenamentos-de-dados-no-docker/

Bridge

Essa é o driver de rede mais simples de utilizar, pois não demanda muita configuração. A rede criada por usuário utilizando o driver bridge se assemelha bastante a rede padrão do docker com o nome "bridge".

> Novamente 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 coisas distintas.

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

Uma dessas funcionalidades é que essa rede criada pelo usuário não precisará mais utilizar a opção legada "–link", pois toda rede criada pelo usuário com o driver bridge poderá utilizar o DNS interno do Docker, que associará automaticamente todos os nomes de container dessa rede para seus respectivos IPs da rede IP correspondente.

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

Para exemplificar a utilização de uma rede criada por usuário, primeiro vamos criar a rede chamada isolated_nw com o driver bridge:

```
1 docker network create --driver bridge isolated_nw
```

Agora vamos verificar a rede que acabamos de criar:

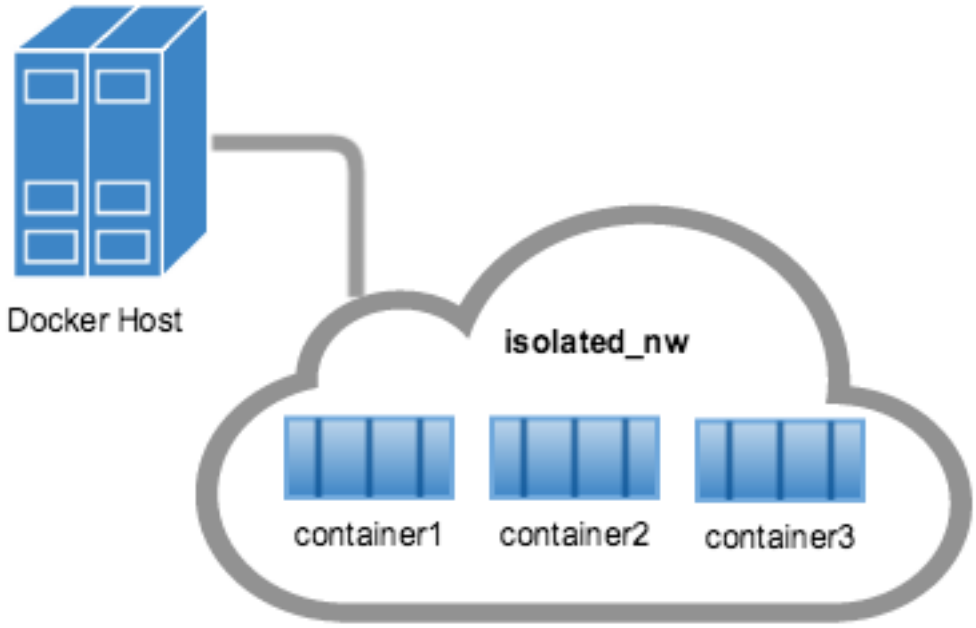
```
1 docker network ls
```

O resultado será:

```
NETWORK ID          NAME                DRIVER
ab09673e9b98        bridge              bridge
9a49dee25aa9        isolated_nw         bridge
763f9ed88176        none                null
242a960a6f20        host                host
```

Agora vamos iniciar um container na rede, chamada isolated_nw, que acabamos de criar:

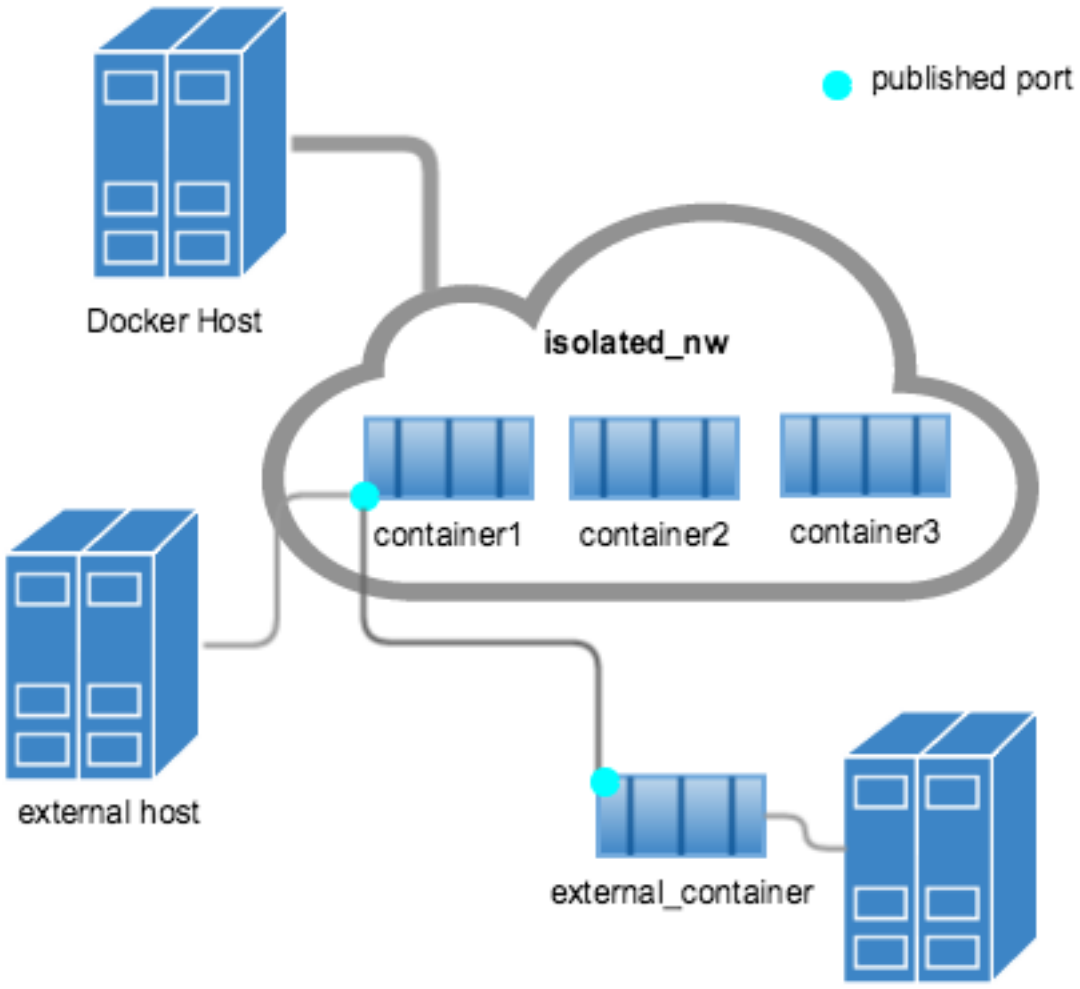
```
1 docker run -itd --net isolated_nw alpine sh
```



Rede isolada

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

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



Rede isolada publicando portas

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

```
1 docker network inspect isolated_nw
```

O resultado será:

```
[
    {
        "Name": "isolated_nw",
        "Id": "9a49dee25aa984beb923d41aab887459f059b47e71c558a8ccc38edeb4e12c7f",
        "Scope": "local",
        "Driver": "bridge",
        "IPAM": {
            "Driver": "default",
            "Options": {},
            "Config": [
                {
                    "Subnet": "172.18.0.0/16",
                    "Gateway": "172.18.0.1/16"
                }
            ]
        },
        "Containers": {
            "3b6b476a77c14249bed8344f5f75b47543c2d65f0ab399235d8b5a887ac33a5f": {
                "Name": "amazing_noyce",
                "EndpointID": "b92bf296fb368b686320470a8feb0d7398a4a33358646983831160dfcc06b9db",
                "MacAddress": "02:42:ac:12:00:02",
                "IPv4Address": "172.18.0.2/16",
                "IPv6Address": ""
            }
        },
        "Options": {}
    }
]
```


Dentro da sessão “Containers” será possível verificar quais containers fazem parte dessa rede, ou seja, todos os containers que estiverem na mesma rede poderão se comunicar tranquilamente utilizando apenas os 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

Esse driver permite comunicação entre hosts docker, ou seja, com sua utilização os containers de um determinado host docker poderão acessar nativamente containers de um outro ambiente docker.

Esse driver demanda uma configuração mais complexa, sendo assim tratarei do seu detalhamento em uma outra oportunidade.

Utilizando redes no docker compose

Esse assunto merece um artigo inteiro pra isso, sendo assim, no momento, vou apenas informar um link interessante[45] de referência sobre esse assunto.

Conclusão

Podemos perceber que a utilização de redes definidas por usuário torna obsoleta a utilização da opção “–link”, assim como viabiliza um novo serviço de DNS interno do docker, que facilita a vida

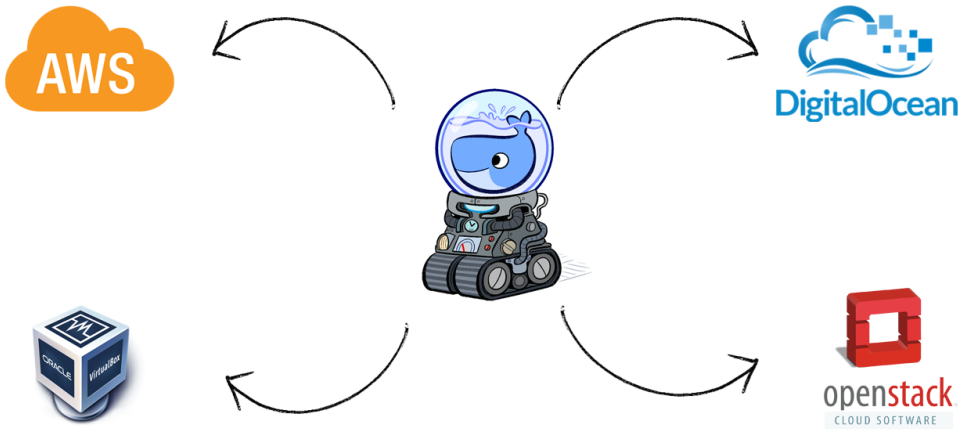
[45] https://docs.docker.com/compose/networking/

de 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.

Utilizando docker em múltiplos ambientes

Docker host é o nome do ativo responsável por gerenciar ambientes Docker, nesse capítulo mostraremos como é possível criar e gerenciá-los em infraestruturas distintas, tal como máquina virtual, nuvem e até mesmo máquina física.



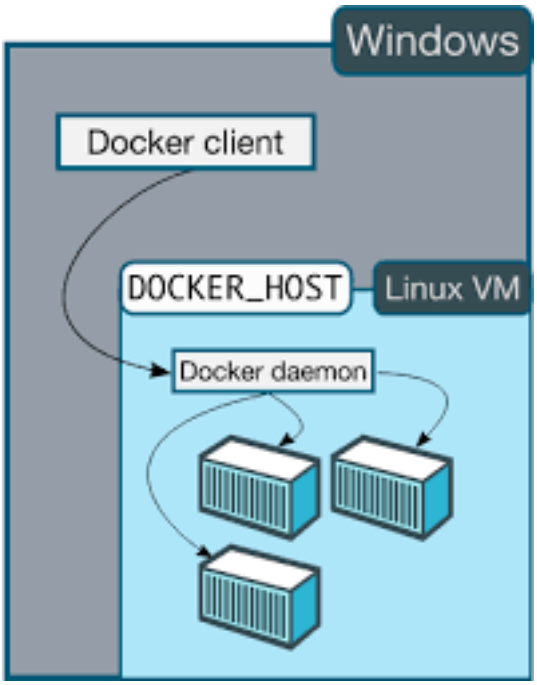
Docker machine[46] é a ferramenta usada para essa gerência distribuída, que lhe permite a instalação e gerência de docker hosts de forma fácil e direta.

Essa ferramenta é muito usada por usuários de sistema operacional "não linux", como será demonstrada ainda nesse livro, mas sua função não se limita esse fim, pois ela também é bastante usada para provisionar e gerenciar infraestrutura Docker na nuvem, tal como AWS, Digital Ocean e Openstack.

[46] https://docs.docker.com/machine/

Como funciona

Antes de explicar como utilizar o docker machine, precisamos reforçar o conhecimento sobre a arquitetura do Docker.



Como podemos ver na imagem acima, a utilização do Docker se divide em dois serviços, o que roda em modo daemon, em background, que é chamado de **Docker Host**, esse serviço é responsável pela viabilização dos containers no kernel Linux. Temos também o cliente, que chamaremos de **Docker client**, esse serviço é responsável por receber comandos do usuário e traduzir em gerência do **Docker Host**.

Cada **Docker client** é configurado para se conectar a um determinado **Docker host** e nesse momento o **Docker machine** entra em ação, pois ele viabiliza a automatização da escolha de configuração de acesso do Docker client a distintos **Docker host**.

O **Docker machine** possibilita que você utilize diversos ambientes distintos apenas modificando a configuração de seu cliente para o **Docker host** desejado, que é basicamente modificar algumas variáveis de ambiente. Segue abaixo um exemplo:

```
1 export DOCKER_TLS_VERIFY="1"
2 export DOCKER_HOST="tcp://192.168.99.100:2376"
3 export DOCKER_CERT_PATH="/Users/gomex/.docker/machine/machines/default"
4 export DOCKER_MACHINE_NAME="default"
```

Modificando essas quatro variáveis o seu Docker client poderá utilizar um ambiente diferente rapidamente e sem precisar reiniciar nenhum serviço.

Criando ambiente

O Docker machine serve principalmente para criar ambientes, que serão futuramente geridos por ele em sua troca automatizada de contexto de configuração, através da mudança de variáveis de ambiente, como foi explicada anteriormente.

Para criar o ambiente é necessário verificar se a infraestrutura que deseja criar tem algum driver com suporte a esse processo. Segue a lista de drivers disponíveis[47].

Máquina virtual

Para esse exemplo usaremos o driver mais utilizado, que é o do virtualbox[48], ou seja, precisamos de um virtualbox[49] instalado na nossa estação para que esse driver funcione adequadamente.

Antes de criar o ambiente vamos entender como funciona o comando de criação do docker machine:

docker-machine create –driver=<nome do driver> <nome do ambiente>

Para o driver **virtualbox** temos alguns parâmetros que podem ser utilizadas:

Parâmetro	Explicação
–virtualbox-memory	Especifica a quantidade de memória RAM que esse ambiente poderá utilizar. O valor padrão é 1024MB. (Sempre em MB)
–virtualbox-cpu-count	Especifica a quantidade de núcleos de CPU que esse ambiente poderá utilizar. O valor padrão é 1
–virtualbox-disk-size	Especifica o tamanho do disco que esse ambiente poderá utilizar. O valor padrão é 20000MB (Sempre em MB)

[47] https://docs.docker.com/machine/drivers/
[48] https://docs.docker.com/machine/drivers/virtualbox/
[49] https://www.virtualbox.org/

Como teste vamos utilizar o seguinte comando:

```
1 docker-machine create --driver=virtualbox --virtualbox-disk-size 30000 teste-vir\
2 tualbox
```

O resultado desse comando é a criação de uma máquina virtual no virtualbox, essa máquina terá 30GB de espaço em disco, 1 núcleo e 1GB de memória RAM.

Para validar que todo processo aconteceu tranquilamente, basta utilizar o seguinte comando:

```
1 docker-machine ls
```

O comando acima é responsável por listar todos os ambiente que podem ser usados a partir de sua estação cliente.

Pra mudar de cliente basta utilizar o comando abaixo:

```
1 eval $(docker-machine env teste-virtualbox)
```

Executando o comando **ls** será possível verificar qual ambiente está ativo:

```
1 docker-machine ls
```

Inicie um container de teste pra testar o novo ambiente

```
1 docker run hello-world
```

Caso deseje mudar pra outro ambiente basta digitar o comando abaixo usando o nome do ambiente desejado:

```
1 eval $(docker-machine env <ambiente>)
```

Caso deseje desligar seu ambiente, utilize o comando abaixo:

```
1 docker-machine stop teste-virtualbox
```

Caso deseje iniciar seu ambiente, utilize o comando abaixo:

```
docker-machine start teste-virtualbox
```

Caso deseje remover seu ambiente, utilize o comando abaixo:

```
docker-machine rm teste-virtualbox
```

Tratamento de problema conhecido: Caso esteja utilizando docker-machine no MacOS e por algum motivo sua estação hiberne quando o ambiente virtualbox esteja iniciado, é possível que no retorno da hibernação esse Docker host apresente problemas em sua comunicação com a internet. Dessa forma oriento a sempre que passar problemas de conectividade no seu Docker host com driver virtualbox, desligue seu ambiente e reinicie como medida de contorno.

Nuvem

Para esse exemplo usaremos o driver da nuvem mais utilizada, que é AWS[50], ou seja, precisamos de uma conta na AWS para que esse driver[51] funcione adequadamente.

É necessário que suas credenciais estejam no arquivo ~/.aws/credentials da seguinte forma:

```
[default]
aws_access_key_id = AKID1234567890
aws_secret_access_key = MY-SECRET-KEY
```

Caso não deseje colocar essas informações em arquivo, você pode especificar via variáveis de ambiente:

```
export AWS_ACCESS_KEY_ID=AKID1234567890
export AWS_SECRET_ACCESS_KEY=MY-SECRET-KEY
```

Você pode encontrar mais informações sobre credencial AWS nesse artigo[52].

Quando criado um ambiente utilizando o comando **docker-machine create**, isso é traduzido para AWS na criação uma instância EC2[53] e em seguida é instalado todos os softwares necessários automaticamente nesse novo ambiente.

Os parâmetros mais utilizados na criação desse ambiente são:

[50] http://aws.amazon.com/

[51] https://docs.docker.com/machine/drivers/aws/

[52] http://blogs.aws.amazon.com/security/post/Tx3D6U6WSFGOK2H/A-New-and-Standardized-Way-to-Manage-Credentials-in-the-AWS-SDKs

[53] https://aws.amazon.com/ec2/

Parâmetro	Explicação
–amazonec2-region	Informa qual região da AWS será utilizada para hospedar seu ambiente. O valor padrão é us-east-1.
–amazonec2-zone	É a letra que representa a zona utilizada. O valor padrão é “a”
–amazonec2-subnet-id	Informa qual a sub-rede utilizada nessa instância EC2. Ela precisa ter sido criada previamente.
–amazonec2-security-group	Informa qual o security group será utilizado nessa instância EC2. Ele precisa ter sido criado previamente
–amazonec2-use-private-address	Será criado uma interface com IP privado, pois por default ele só especifica uma com IP público
–amazonec2-vpc-id	Informa qual o ID do VPC desejado para essa instância EC2. Ela precisa ter sido criado previamente.

Como exemplo, usarei o seguinte comando de criação do ambiente:

```
1 docker-machine create --driver amazonec2 --amazonec2-zone a --amazonec2-subnet-i\
2 d subnet-5d3dc191 --amazonec2-security-group docker-host --amazonec2-use-private\
3 -address --amazonec2-vpc-id vpc-c1d33dc7 teste-aws
```

Após executar o comando, basta esperar finalizar, pois demora um pouco.

Para testar o sucesso do comando, execute o comando abaixo:

```
1 docker-machine ls
```

Verifique se o ambiente chamado teste-aws existe em sua lista, caso positivo utiliza o comando abaixo para mudar o ambiente:

```
1 eval $(docker-machine env teste-aws)
```

Inicie um container de teste pra testar o novo ambiente

```
1 docker run hello-world
```

Caso deseje desligar seu ambiente, utilize o comando abaixo:

```
1 docker-machine stop teste-aws
```

Caso deseje iniciar seu ambiente, utilize o comando abaixo:

```
1 docker-machine start teste-aws
```

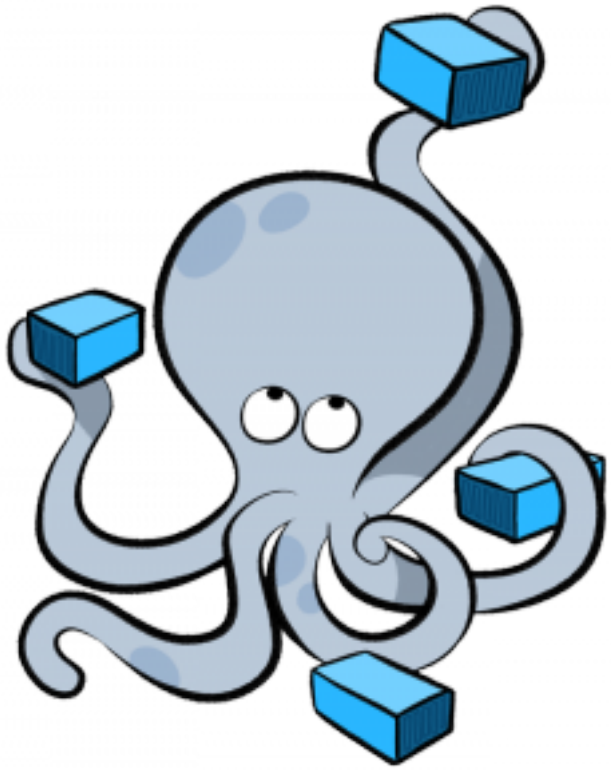
Caso deseje remover seu ambiente, utilize o comando abaixo:

```
1 docker-machine rm teste-aws
```

Após removido localmente, ele automaticamente removerá a instância EC2 que foi provisionada na AWS.

Gerenciando múltiplos containers docker com Docker Compose

Esse artigo tem como objetivo explicar de forma detalhada, e com exemplos, como funciona o processo de gerenciamento de múltiplos containers Docker, pois a medida que sua confiança em utilizar Docker aumenta, sua necessidade de utilizar um maior número de containers ao mesmo tempo cresce na mesma proporção, e seguir a boa prática de manter apenas um serviço por container comumente resulta em alguma demanda extra.



Geralmente com o aumento do número de containers em execução, fica evidente a necessidade de um melhor gerenciamento da sua comunicação, pois é ideal que os serviços consigam trocar dados entre os containers quando necessário, ou seja, você precisa lidar com a **rede** desse novo ambiente.

Imagine o trabalho que seria executar algumas dezenas de containers manualmente na linha de comando, um por um e todos seus parâmetros necessários, suas configurações de rede entre containers, volumes e afins. Pode parar de imaginar, pois isso não será mais necessário. Para atender essa demanda de gerenciamento de múltiplos containers a solução é o Docker Compose[54].

[54] https://docs.docker.com/compose/overview/

Docker compose é uma ferramenta para definição e execução de múltiplos containers Docker. Com ela é possível configurar todos os parâmetros necessários para executar cada container a partir de um **arquivo de definição**. Dentro desse arquivo, definimos cada container como **serviço**, ou seja, sempre que esse texto citar **serviço** de agora em diante, imagine que é a definição que será usada para iniciar um **container**, tal como portas expostas, variáveis de ambiente e afins.

Com o Docker Compose podemos também especificar quais **volumes** e **rede** serão criados para serem utilizados nos parâmetros dos **serviços**, ou seja, isso quer dizer que não preciso criá-los manualmente para que os **serviços** utilizem recursos adicionais de **rede** e **volume**.

O **arquivo de definição** do Docker Compose é o local onde é especificado todo o ambiente (**rede**, **volume** e **serviços**), ele é escrito seguindo o formato YAML[55]. Esse arquivo por padrão tem como nome docker-compose.yml[56].

Anatomia do docker-compose.yml

O padrão YAML utiliza a indentação como separador dos blocos de códigos das definições, por conta disso o uso da indentação é um fator muito importante, ou seja, caso não a utilize corretamente, o docker-compose falhará em sua execução.

Cada linha desse arquivo pode ser definida com uma chave valor ou uma lista. Vamos aos exemplos pra ficar mais claro a explicação:

```
1  version: '2'
2  services:
3    web:
4      build: .
5        context: ./dir
6        dockerfile: Dockerfile-alternate
7        args:
8          versao: 1
9      ports:
10       - "5000:5000"
11   redis:
12     image: redis
```

No arquivo acima temos a primeira linha que define a versão do **docker-compose.yml**, que no nosso caso usaremos a versão mais atual do momento, caso tenha interesse em saber a diferença entre as versões possíveis, veja esse link[57].

[55] https://en.wikipedia.org/wiki/YAML
[56] https://docs.docker.com/compose/compose-file/
[57] https://docs.docker.com/compose/compose-file/#versioning

```
1 version: '2'
```

No mesmo nível de indentação temos **services**, que define o início do bloco de **serviços** que serão definidos logo abaixo.

```
1 version: '2'
2 services:
```

No segundo nível de indentação (aqui feito com dois espaços) temos o nome do primeiro **serviço** desse arquivo, que recebe o nome de **web**. Ele abre o bloco de definições do **serviço**, ou seja, a partir do próximo nível de indentação, tudo que for definido faz parte desse serviço.

```
1 version: '2'
2 services:
3   web:
```

No próximo nível de indentação (feito novamente com mais dois espaços) temos a primeira definição do **serviço web**, que nesse caso é o build[58] que informa que esse serviço será criado não a partir de uma imagem pronta, mas que será necessário construir sua imagem antes de sua execução. Seria o equivalente ao comando docker build[59]. Ele também abre um novo bloco de código para parametrizar o funcionamento dessa construção da imagem.

```
1 version: '2'
2 services:
3   web:
4     build: .
```

No próximo nível de indentação (feito novamente com mais dois espaços) temos um parâmetro do **build**, que nesse caso é o **context**. Ele é responsável por informar qual contexto de arquivos será usado para construir a imagem em questão, ou seja, apenas arquivos existentes dentro dessa pasta poderão ser usados na construção da imagem. O contexto escolhido foi o "**./dir**", ou seja, isso indica que uma pasta chamada **dir**, que se encontra no mesmo nível de sistema de arquivo do **docker-compose.yml** ou do lugar onde esse comando será executado, será usada como contexto da criação dessa imagem. Quando logo após da chave um valor é fornecido, isso indica que nenhum bloco de código será aberto.

[58] https://docs.docker.com/compose/reference/build/

[59] https://docs.docker.com/engine/reference/commandline/build/

```
1       build: .
2         context: ./dir
```

No mesmo nível de indentação da definição **context**, ou seja, ainda dentro do bloco de definição do **build**, temos o **dockerfile**, ele indica o nome do arquivo que será usado para construção da imagem em questão. Seria o equivalente ao parâmetro "-f"[60] do comando **docker build**. Caso essa definição não existisse, o **docker-compose** procuraria por padrão por um arquivo chamado **Dockerfile** dentro da pasta informada no **context**.

```
1       build: .
2         context: ./dir
3         dockerfile: Dockerfile-alternate
```

No mesmo nível de indentação da definição **dockerfile**, ou seja, ainda dentro do bloco de definição do **build**, temos o **args**, ele define os argumentos que serão usados pelo **Dockerfile**, seria o equivalente ao parâmetro "–build-args"[61] do comando **docker build**. Como não foi informado o seu valor na mesma linha, fica evidente que ela abre um novo bloco de código.

No próximo nível de indentação (feito novamente com mais dois espaços) temos a chave "**versao**" e o valor "**1**", ou seja, como essa definição faz parte do bloco de código **args**, essa chave valor é o único argumento que será passado para o **Dockerfile**, ou seja, o arquivo **Dockerfile** em questão deverá estar preparado para receber esse argumento ou ele se perderá na construção da imagem.

```
1       build: .
2         context: ./dir
3         dockerfile: Dockerfile-alternate
4         args:
5           versao: 1
```

Voltando dois nível de indentação (quatro espaços a menos em relação a linha anterior) temos a definição **ports**, que seria o equivalente ao parâmetro "-p"[62] do comando docker run[63]. Ele define qual porta do container será exposta no **Docker host**. Que no nosso caso será a porta *5000 do container, com a 5000 do **Docker host**.

[60] https://docs.docker.com/engine/reference/commandline/build/#specify-dockerfile-f
[61] https://docs.docker.com/engine/reference/commandline/build/#set-build-time-variables-build-arg
[62] https://docs.docker.com/engine/reference/commandline/run/#publish-or-expose-port-p-expose
[63] https://docs.docker.com/engine/reference/commandline/run/

```
1   web:
2     build: .
3     ...
4     ports:
5       - "5000:5000"
```

Voltando um nível de indentação (dois espaços a menos em relação a linha anterior) saímos do bloco de código do serviço **web**, isso indica que nenhuma definição informada nessa linha será aplicada a esse serviço, ou seja, precisamos iniciar um bloco de código de um serviço novo, que no nosso caso será com nome de **redis**.

```
1   redis:
2     image: redis
```

No próximo nível de indentação (feito novamente com mais dois espaços) temos a primeira definição do serviço **redis**, que nesse caso é o **image** que é responsável por informar qual imagem será usada para iniciar esse container. Essa imagem será obtida do repositório configurado no **Docker host**, que por padrão é o hub.docker.com[64].

Executando o docker compose

Após entender e criar seu próprio **arquivo de definição** precisamos saber como gerencia-lo e para isso utilizaremos o binário docker-compose, que entre várias opções de uso temos as seguintes mais comuns:

- **build** : Usada para construir todas as imagens dos **serviços** que estão descritos com a definição **build** em seu bloco de código.
- **up** : Iniciar todos os **serviços** que estão no arquivo **docker-compose.yml**
- **stop** : Parar todos os **serviços** que estão no arquivo **docker-compose.yml**
- **ps** : Listar todos os **serviços** que foram iniciados a partir do arquivo **docker-compose.yml**

Para outras opções visite sua documentação[65].

[64] https://hub.docker.com/
[65] https://docs.docker.com/compose/reference/

Como usar Docker sem GNU/Linux

Esse artigo tem como objetivo explicar de forma detalhada, e com exemplos, como funciona o uso de docker em estações MacOS e Windows.

Docker Toolbox

Esse texto tem como público alvo pessoas que já sabem um pouco sobre docker, mas ainda não sabiam como o docker pode ser utilizado a partir de uma estação "não linux".

Como já explicado anteriormente nesse livro, o docker utiliza recursos específicos do kernel hospedeiro e o GNU/Linux é o único sistema operacional que suporta o docker de forma estável. Isso quer dizer que não é possível iniciar containers docker em uma estação MacOS e Windows, por exemplo.

Não precisa ficar preocupado, caso você não utilize GNU/Linux como sistema operacional, ainda será possível fazer uso dessa tecnologia, sem necessariamente executá-la em seu computador.

Um dos produtos da suíte docker é o docker toolbox[66]. Essa solução na verdade é uma abstração para instalação de todo ambiente necessário para fazer uso do docker a partir de uma estação MacOS ou Windows.

Para instalar é muito simples, tanto no Windows, como no MacOS, basta baixar o instalador correspondente nesse site[67] e executá-lo seguindo dos passos descritos nas telas.

Os softwares instalados na sua estação (MacOS ou Windows) a partir do pacote docker toolbox são:

- Virtualbox[68]
- Docker machine[69]
- Docker client[70]
- Docker compose[71]
- Kitematic[72]

Docker machine é a ferramenta que possibilita criar e manter ambientes docker em máquinas virtuais, ambientes de nuvem e até mesmo em máquina física, mas nesse tópico vamos nos manter apenas em máquina virtual com virtualbox.

Após instalar o docker toolbox, é muito simples criar um ambiente docker com máquina virtual usando o docker machine.

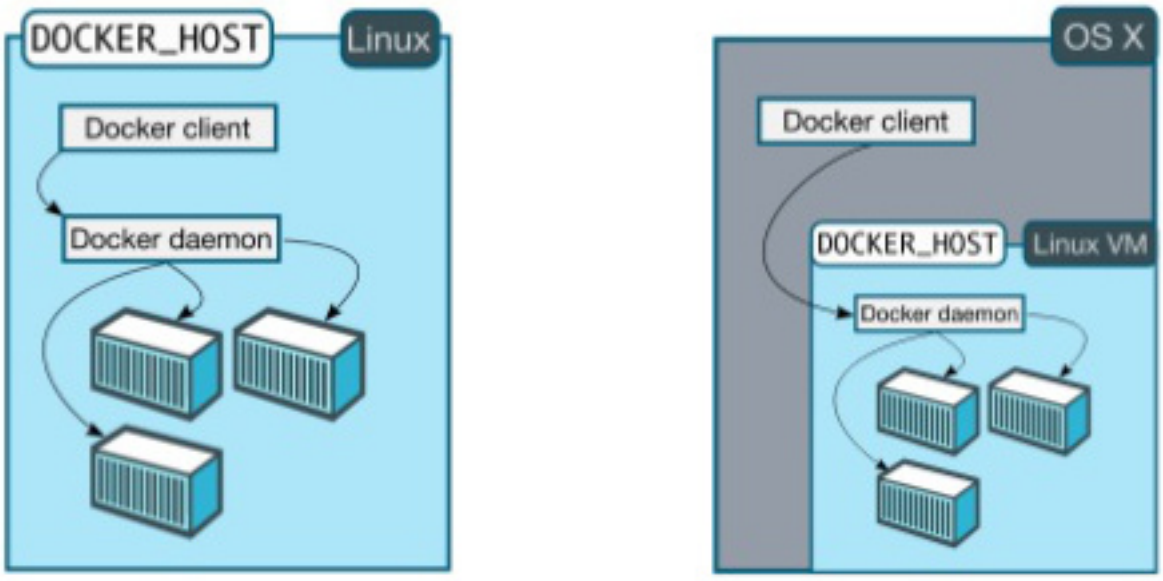
Primeiro vamos verificar se não existem máquinas virtuais com docker instalado em seu ambiente:

```
1 docker-machine ls
```

O comando acima mostrará apenas ambientes criados e mantidos por seu docker-machine, ou seja, é possível que após instalar seu docker toolbox você não verá máquina alguma criada, sendo assim vamos utilizar o comando abaixo para criar uma máquina:

```
1 docker-machine create --driver virtualbox default
```

[66] https://www.docker.com/products/docker-toolbox
[67] https://www.docker.com/products/docker-toolbox
[68] https://www.virtualbox.org/
[69] https://docs.docker.com/machine/overview/
[70] https://docs.docker.com/
[71] https://docs.docker.com/compose/overview/
[72] https://docs.docker.com/kitematic/userguide/



Arquitetura do Docker toolbox

O comando acima criou um ambiente que se chama "default", que na verdade é uma máquina virtual ("Linux VM" que aparece na imagem) criada no virtualbox. Com o comando abaixo será possível visualizar a máquina que acabou de criar:

```
1 docker-machine ls
```

O retorno será algo parecido com isto:

```
NAME      ACTIVE   DRIVER       STATE     URL                         SWARM   DOCKER    ERRORS
default   -        virtualbox   Running   tcp://192.168.99.100:2376           v1.10.1
```

Uma máquina virtual foi criada, dentro dela temos um sistema operacional GNU/Linux com docker host instalado. Esse serviço docker está escutando na porta TCP 2376 do endereço 192.168.99.100. Essa interface utiliza uma rede específica entre seu computador e as máquinas do virtualbox.

Para desligar a máquina virtual basta executar o comando abaixo:

```
1 docker-machine stop default
```

Para iniciar novamente a máquina, basta executar o comando abaixo:

```
1 docker-machine start default
```

O comando "start" é responsável apenas por iniciar a máquina, agora é necessário fazer com que os aplicativos de controle do docker, que foram instalados na sua estação, possam se conectar a máquina virtual criada no virtualbox com o comando "docker-machine create".

Os aplicativos de controle (docker e docker-compose) fazem uso de variáveis de ambiente para configurar qual docker host será utilizado. O comando abaixo facilitará esse trabalho de aplicar todas as variáveis corretamente:

```
1 docker-machine env default
```

O resultado desse comando no MacOS será:

```
export DOCKER_TLS_VERIFY="1"
export DOCKER_HOST="tcp://192.168.99.100:2376"
export DOCKER_CERT_PATH="/Users/rgomes/.docker/machine/machines/default"
export DOCKER_MACHINE_NAME="default"
# Run this command to configure your shell:
# eval $(docker-machine env default)
```

Como podemos ver, ele informa o que pode ser feito para configurar todas as suas variáveis. Você pode copiar as quatros primeiras linhas, que começam com "export", e colar no seu terminal ou pegar apenas a última linha sem o "#" no começo e executar na linha de comando:

```
1 eval $(docker-machine env default)
```

Agora os seus aplicativos de controle (docker e docker-compose) estarão aptos a utilizar o docker host a partir da conexão feita no serviço do ip 192.168.99.100, que na verdade é a máquina criada com o comando "docker-machine create" mencionados anteriormente nesse capítulo.

Para testar vamos listar os containers que temos em execução nesse docker host com o comando abaixo:

```
1 docker ps
```

O comando acima é executado na linha de comando do MacOS ou Windows e esse cliente do docker se conectará na máquina virtual, que aqui chamamos de "Linux VM", e solicitará a lista de containers em execução nesse docker host remoto.

Vamos iniciar um container com o comando abaixo:

```
1 docker run -itd alpine sh
```

Agora vamos verificar novamente a lista de containers em execução:

```
1 docker ps
```

Agora poderemos ver que o container criado a partir da imagem "alpine" está em execução. Vale salientar que esse processo está sendo executado no docker host, na máquina criada dentro do virtualbox, que aqui nesse exemplo tem o ip 192.168.99.100.

Para verificar o endereço IP da sua máquina, basta executar o comando abaixo

```
docker-machine ip
```

Caso seu container exponha alguma porta para o docker host, seja via parâmetro "-p" do comando "docker run -p porta_host:porta_container" ou via parâmetro "ports" do docker-compose.yml, vale lembrar que o IP que você deve usar para acessar o serviço exposto é endereço IP do docker host, que no nosso exemplo é "192.168.99.100".

Nesse momento você deve estar se perguntando como é possível mapear uma pasta da sua estação "não-linux" para dentro de um container. Aqui entra um novo artífício do docker para contornar esse problema.

Toda máquia criada com o driver "virtualbox" automaticamente cria um mapeamento do tipo "pastas compartilhadas do virtualbox" da sua pasta de usuários para a raiz do docker host.

Para visualizar esse mapeamento vamos acessar a máquina virtual que acabamos de criar nos passos anteriores:

```
docker-machine ssh default
```

No console da máquina GNU/Linux digite os seguintes comandos:

```
sudo su
mount | grep vboxsf
```

O vboxsf[73] é um sistema de arquivo usado pelo virtualbox para montar volumes que são compartilhados da estação que foi usada para instalar o virtualbox, ou seja, utilizando esse recurso de pasta compartilhada é possível montar a pasta /Users do MacOS na pasta /Users da máquina virtual do docker host.

Todo conteúdo existente na sua pasta /Users/SeuUsuario do seu MacOS será acessível na pasta /Users/SeuUsuario da máquina GNU/Linux que atua como docker host nesse exemplo apresentado, ou seja, caso você efetue a montagem da pasta /Users/SeuUsuario/MeuCodigo para dentro do container, o dado a ser montado é o mesmo da sua estação e nada precisa ser feito para replicar esse código para dentro do docker host.

Vamos efetuar um teste. Crie um arquivo dentro da sua pasta de usuário:

```
touch teste
```

Vamos iniciar um container e mapear nossa pasta atual dentro dele:

[73] https://help.ubuntu.com/community/VirtualBox/SharedFolders

```
1 docker run -itd -v "$PWD:/tmp" --name teste alpine sh
```

No comando acima iniciamos um container que será nomeado como “teste” e terá mapeado a pasta atual (a variável PWD indica o endereço atual no MacOS) na pasta /tmp dentro do container.

Vamos verificar se o arquivo que acabamos de criar está dentro do container:

```
1 docker exec teste ls /tmp/teste
```

A linha acima executou o comando “ls /tmp/teste” dentro do container nomeado de “teste” que foi criado no passo anterior.

Agora acesse o docker host novamente com o comando abaixo, e por fim verificar se o arquivo teste se encontra na pasta de usuário.:

```
1 docker-machine ssh default
```

Tudo pode ser feito automaticamente? Claro que sim!

Agora que você já sabe como fazer tudo manualmente, se precisar instalar o docker toolbox em uma máquina nova e não lembrar quais comandos precisa executar para criar uma máquina nova ou simplesmente aprontar seu ambiente para uso, basta executar o programa “Docker quickstart terminal” e ele fará todo trabalho automaticamente pra ti, ou seja, caso não exista nenhuma máquina criada, ele criará uma chamada “default”, caso a máquina já tenha sido criada, ele automaticamente configura suas variáveis de ambiente e lhe deixar apto pra utilizar o docker host remoto apartir de seus aplicativos de controle (docker e docker-compose).

Transformando sua aplicação em container

Estamos evoluindo continuamente para entregar aplicações cada vez melhores, em menor tempo, replicáveis e escaláveis. Porém os esforços e aprendizados para atingir esse nível de maturidade muitas vezes não são simples de se alcançar.

Atualmente notamos o surgimento de várias opções de plataformas para facilitar a implantação, configuração e escalabilidade das aplicações que desenvolvemos. Porém, para melhorar nossa maturidade não podemos apenas depender da plataforma, precisamos construir nossa aplicação seguindo boas práticas.

Visando sugerir uma série de boas práticas comuns a aplicações web modernas, alguns desenvolvedores do Heroku[74] escreveram o 12Factor app[75], baseado em uma larga experiência em desenvolvimento desse tipo de aplicação.

"The Twelve-Factor app" (12factor) é um manifesto com uma série de boas práticas para construção de software utilizando formatos declarativos de automação, maximizando portabilidade e minimizando divergências entre ambientes de execução, permitindo a implantação em plataformas de nuvem modernas e facilitando a escalabilidade. Assim, aplicações são construídas sem manter estado (staless) e conectadas a qualquer combinação de serviços de infraestrutura para retenção de dados (Banco de dados, fila, memória cache e afins).

Nesse capítulo falaremos sobre criação de aplicações como imagems docker com base no 12factor app, ou seja, a ideia é demonstrar as melhores práticas de como se realizar a criação de uma infraestrutura para suportar, empacotar e disponibilizar a sua aplicação com alto nível de maturidade e agilidade.

[74] https://www.heroku.com/
[75] http://12factor.net/pt_br/

O uso do 12factor com Docker é uma combinação perfeita, pois muitos dos recursos do Docker são melhores aproveitados caso sua aplicação tenha sido pensada para tal. Dessa forma, nesse texto daremos uma ideia de como aproveitar todo potencial da sua solução.

Como exemplo, usaremos duas aplicações como modelo. Uma usando linguágem dinâmica (Python) e outra compilada (Java).

Como exemplo teremos um modelo de aplicação simples. Através de um serviço HTTP, ela exibe quantas vezes foi acessada. Essa informação é armazenada através de um contador numa instância Redis.

Agora vamos às boas práticas!

Base de código

Com o objetivo de facilitar o controle das mudanças de código, viabilizando a rastreabilidade das alterações, essa boa prática indica que cada aplicação deve ter apenas uma base de código e, a partir do mesmo, uma aplicação pode ser implantada em distintos ambientes, mas a base precisa ser a mesma. Vale salientar que essa boa prática é necessária caso pretenda praticar o Continuous Integration (CI[76]), ou seja, o mesmo código deverá ser consumido pelo processo de integração contínua e posteriormente implantado em desenvolvimento, teste e produção, mudando apenas parâmetros específicos ao ambiente em questão.

Para essa explicação, criei um repositório[77] de exemplo.

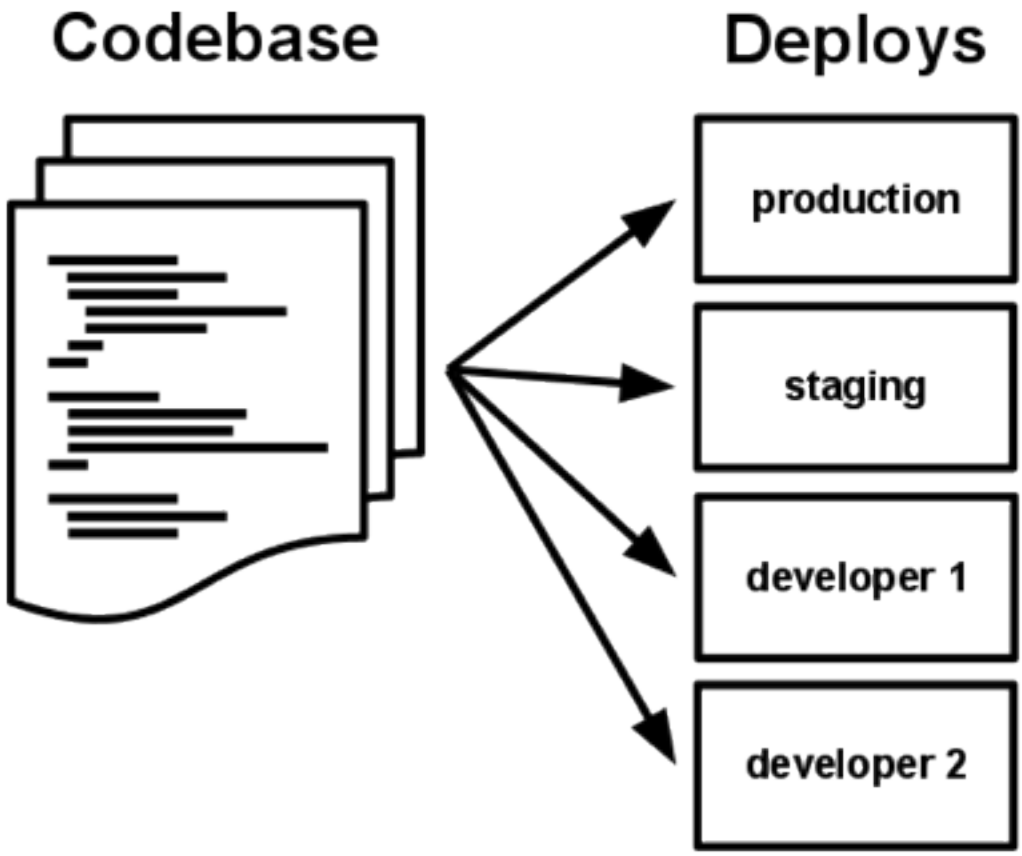
Perceba que todo nosso código está dentro desse repositório, organizado por prática em cada pasta, para facilitar a reprodução. Lembre de entrar na pasta correspondente a cada boa prática apresentada.

O Docker tem uma infraestrutura que permite a utilização de variável de ambiente para parametrização da infraestrutura, sendo assim o mesmo código terá um comportamento distinto com base no valor das variáveis de ambiente.

Aqui usaremos o Docker Compose para realizar a composição de todo ambiente, fazendo com que a configuração dos distintos serviços e sua comunicação seja facilitada.

[76] https://www.thoughtworks.com/continuous-integration

[77] https://github.com/gomex/exemplo-12factor-docker.git



Posteriormente, mais precisamente na terceira boa prática **Configuração** desse compêndio de sugestões, trataremos mais detalhadamente sobre parametrização da aplicação, por hora apenas aplicaremos opções via variável de ambiente para a arquitetura ao invés de utilizar internamente no código da aplicação.

Para configurar o ambiente de desenvolvimento para o exemplo apresentado temos esse arquivo docker-compose.yml:

```
1  version: '2'
2  services:
3    web:
4      build: .
5      ports:
6       - "5000:5000"
7      volumes:
8       - .:/code
9      labels:
10      - 'app.environment=${ENV_APP}'
11   redis:
12     image: redis
```

```
13      volumes:
14       - dados_${ENV_APP}:/data
15      labels:
16       - 'app.environment=${ENV_APP}'
```

O serviço redis será utilizado a partir da imagem oficial redis, sem modificação no momento e o serviço web será gerado a partir da construção de uma imagem Docker, que tem como base a imagem oficial do python 2.7. Criaremos nossa imagem através do seguinte Dockerfile:

```
1  FROM python:2.7
2  COPY requirements.txt requirements.txt
3  RUN pip install -r requirements.txt
4  ADD . /code
5  WORKDIR /code
6  CMD python app.py
```

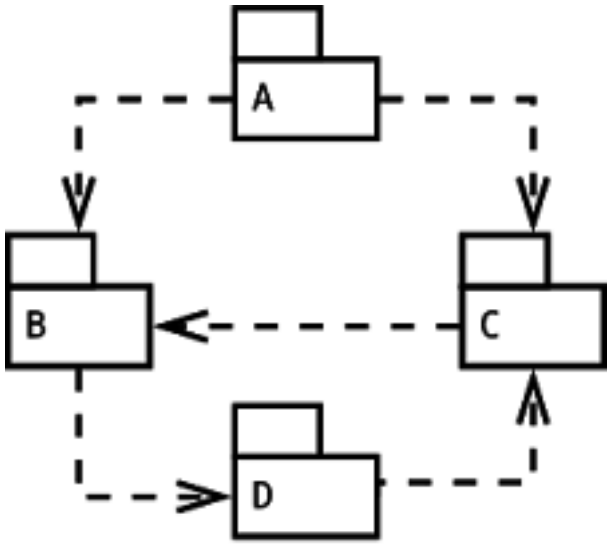
Com posse de todos os arquivos na mesma pasta, iniciaremos o ambiente com o seguinte comando:

```
1  export ENV_APP=devel ; docker-compose -p $ENV_APP up -d
```

Como podemos perceber na configuração que acabamos de construir, a variável "ENV_APP" definirá qual volume será usado para persistir os dados que serão enviados pela aplicação web, ou seja, com base na mudança dessa opção teremos o serviço rodando com um comportamento diferente, mas sempre a partir do mesmo código, dessa forma seguindo perfeitamente a ideia dessa primeira boa prática.

Dependência

Seguindo a lista do modelo 12factor[78], logo após o base de código que tratamos nesse artigo[79], agora temos "**Dependência**" como segunda boa prática.



Essa boa prática sugere a declaração de todas as dependências necessárias para executar seu código, ou seja, você nunca deve assumir que algum componente já estará previamente instalado no ativo responsável por hospedar essa aplicação.

Para viabilizar o "sonho" da portabilidade, precisamos gerenciar corretamente as dependências da aplicação em questão, isso indica que devemos também evitar a necessidade de trabalho manual na preparação da infraestrutura que suportará sua aplicação.

Automatizar o processo de instalação de dependência é o grande segredo de sucesso para atender essa boa prática, pois caso a instalação da sua infraestrutura não seja automática o suficiente para viabilizar a inicialização sem erros, o atendimento dessa boa prática estará prejudicada.

Automatizar esses procedimentos colabora com a manutenção da integridade do seu processo, pois o nome dos pacotes de dependências e suas respectivas versões estariam especificadas explicitamente em um arquivo localizado no mesmo repositório que o código, que por sua vez seria rastreado em um sistema de controle de versão. Com isso podemos concluir que nada seria modificado sem que se tenha o devido registro.

[78] http://12factor.net/

[79] http://techfree.com.br/2016/06/dockerizando-aplicacoes-base-de-codigo/

O Docker se encaixa perfeitamente nessa boa prática, pois é possível entregar um perfil mínimo de infraestrutura para essa aplicação, que por sua vez se fará necessária a declaração explícita de suas dependências para que a aplicação funcione nesse ambiente.

A aplicação que usamos como exemplo foi escrita em Python. Como podem ver na parte do código abaixo ela necessita de duas bibliotecas para funcionar corretamente:

```
1 from flask import Flask
2 from redis import Redis
```

Essas duas dependências estão especificados no arquivo requirements.txt e esse arquivo será usado como um parâmetro do pip.

"O PIP é um sistema de gerenciamento de pacotes usado para instalar e gerenciar pacotes de software escritos na linguagem de programação Python". (Wikipedia)

O comando pip é usado, juntamente com o arquivo requirements.txt, na criação da imagem como foi demonstrado no Dockerfile da boa prática anterior (codebase):

```
1 FROM python:2.7
2 ADD requirements.txt requirements.txt
3 RUN pip install -r requirements.txt
4 ADD . /code
5 WORKDIR /code
6 CMD python app.py
```

Perceba que um dos passos do Dockerfile é instalar as dependências explicitamente descritas no arquivo requirements.txt com o gerenciador de pacotes pip do Python. Veja o conteúdo do arquivo requirements.txt:

```
1 flask==0.11.1
2 redis==2.10.5
```

É importante salientar a necessidade de se especificar as versões de cada dependência, pois como no modelo de contêiner as suas imagens podem ser construídas a qualquer momento, é importante saber qual versão específica a sua aplicação precisa, caso contrário poderemos encontrar problemas com compatibilidade caso uma das dependências atualize e não esteja mais compatível com a composição completa das outras dependências e a aplicação que a está utilizando.

Para acessar o código descrito aqui, baixe o repositório[80] e acesse a pasta "**factor2**".

Um outro resultado positivo do uso dessa boa prática é a simplificação da utilização do código por outro desenvolvedor. Um novo programador pode verificar nos arquivos de dependências quais

[80] https://github.com/gomex/exemplo-12factor-docker

os pré-requisitos para sua aplicação executar, assim como executar facilmente o ambiente sem a necessidade de seguir uma extensa documentação que raramente é atualizada.

Usando o Docker é possível configurar automaticamente o necessário para rodar o código da aplicação, seguindo com isso essa boa prática perfeitamente.

Configurações

Seguindo a lista do modelo 12factor[81], temos "**Configurações**" como terceira boa prática.

Quando estamos criando um software aplicamos um determinado comportamento dentro do código e normalmente ele não é parametrizável, ou seja, para que essa aplicação se comporte de uma forma diferente será necessário mudar uma parte do código.

A necessidade de modificar o código para trocar o comportamento da aplicação inviabiliza que a mesma seja executa em sua máquina (desenvolvimento) da mesma forma que será usada para atender os usuários (produção) e com isso acabamos com toda possibilidade de portabilidade, e sem isso qual seria a vantagem de se usar contêineres, certo?

O objetivo dessa boa prática é viabilizar a configuração da aplicação sem a necessidade de modificar o código da mesma.

Como o comportamento da aplicação varia de acordo com o ambiente onde ela está executando, sendo assim, as configurações devem ser feitas baseadas nisso.

Seguem abaixo alguns exemplo:

- Configuração de banco de dados que normalmente são diferentes entre os ambientes
- Credenciais para acesso a serviços remotos (Ex. Digital Ocean ou Twitter)
- Qual nome de DNS será usado pela aplicação

Como já falamos anteriormente, quando a configuração está estaticamente explícita no código, será necessário modificar manualmente e efetuar um novo build dos binários a cada reconfiguração do sistema.

Como demonstramos na boa prática codebase, usamos uma variável de ambiente para modificar qual o volume que usaremos no redis, ou seja, de certa forma já estamos seguindo essa boa prática, mas iremos um pouco além e mudaremos não somente o comportamento da infraestrutura, mas sim algo inerente ao código em si.

Segue abaixo a aplicação modificada:

[81] http://12factor.net/

```
from flask import Flask
from redis import Redis
import os
host_run=os.environ.get('HOST_RUN', '0.0.0.0')
debug=os.environ.get('DEBUG', 'True')
app = Flask(__name__)
redis = Redis(host='redis', port=6379)
@app.route('/')
def hello():
    redis.incr('hits')
    return 'Hello World! %s times.' % redis.get('hits')
if __name__ == "__main__":
    app.run(host=host_run, debug=debug)
```

Lembrando! Para acessar o código dessa prática basta clonar esse repositório[82] e acessar a pasta "**factor3**".

Como podemos perceber, adicionamos alguns parâmetros na configuração do endereço usado para iniciar a aplicação web, que será parametrizado com base no valor da variável de ambiente "**HOST_RUN**" e a possibilidade de efetuar ou não o debug dessa aplicação com a variável de ambiente "**DEBUG**".

Vale salientar que nesse caso a variável de ambiente precisa ser passada para o contêiner, ou seja, não basta ter essa variável no docker host. Ela precisa ser enviada para o contêiner usando o parâmetro "-e" caso utilize o comando "docker run" ou a instrução "environment" no docker-compose.yml:

```
version: "2"
services:
  web:
    build: .
    ports:
     - "5000:5000"
    volumes:
     - .:/code
    labels:
     - 'app.environment=${ENV_APP}'
    environment:
     - HOST_RUN=${HOST_RUN}
     - DEBUG=${DEBUG}
  redis:
    image: redis:3.2.1
    volumes:
```

[82] https://github.com/gomex/exemplo-12factor-docker

```
17       - dados:/data
18     labels:
19       - 'app.environment=${ENV_APP}'
20 volumes:
21   dados:
22     external: false
```

Para executar o docker-compose, deveríamos fazer da seguinte maneira:

```
1 export HOST_RUN="0.0.0.0"; export DEBUG=True ; docker-compose up -d
```

Como podem perceber no comando acima ele usará as variáveis de ambiente “**HOST_RUN**” e “**DEBUG**” do docker host para enviar para as variáveis de ambiente com os mesmos nomes dentro do contêiner, que por sua vez será consumido pelo código python. Em caso de não haver parâmetros ele assume os valores padrões estipulados no código.

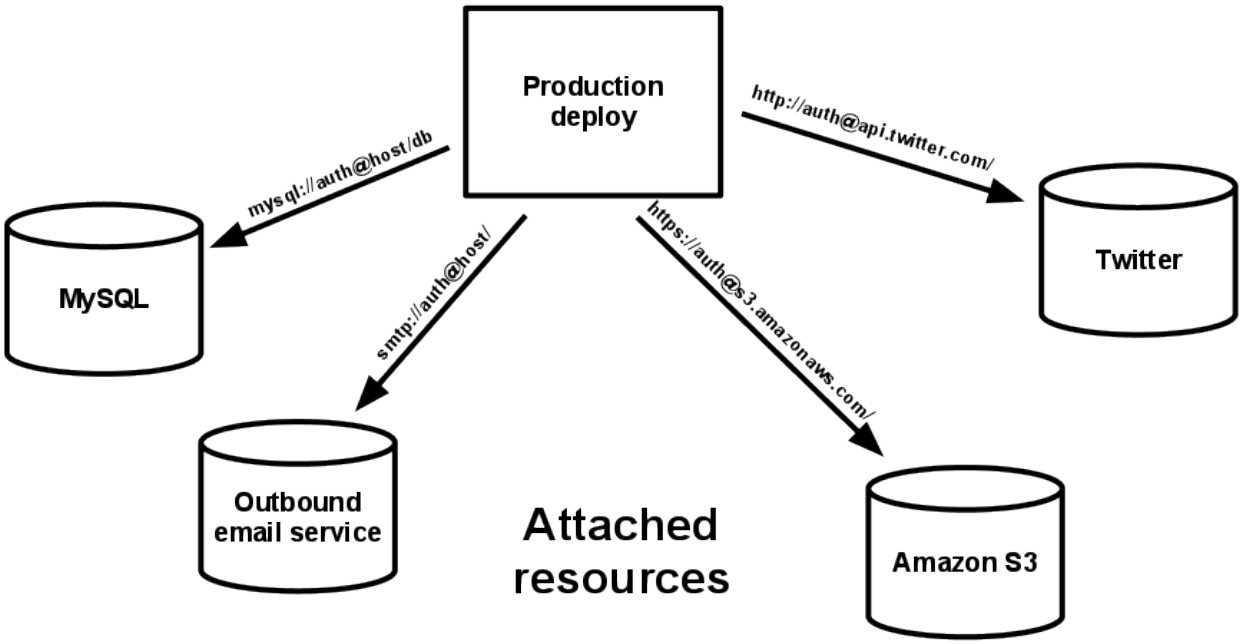
Essa boa prática é seguida com ajuda do Docker, pois o código é o mesmo e a configuração é um anexo da solução, que pode ser parametrizada de maneira distinta com base no que for configurado nas variáveis de ambiente.

Se aplicação crescer bastante, as variáveis podem ser carregadas em arquivo e parametrizadas no docker-compose.yml com a opção “env_file”.

Serviços de Apoio

Seguindo a lista do modelo 12factor[83], temos "**Serviços de Apoio**" como quarta boa prática.

Para contextualizar, serviços de apoio é qualquer aplicação que seu código consome para operar corretamente (Ex. Banco de dados, serviço de mensagens e afins).



Com objetivo de evitar que seu código seja demasiadamente dependente de uma determinada infraestrutura, essa boa prática indica que você, no momento da escrita do software, não faça distinção se o serviço é interno ou externo, ou seja, seu aplicativo deve estar pronto para receber parâmetros que farão a configuração do serviço correto e assim possibilitem o consumo de aplicações necessárias da solução proposta.

A aplicação exemplo sofreu algumas modificações para suportar essa boa prática:

[83] http://12factor.net/pt_br

```python
from flask import Flask
from redis import Redis
import os
host_run=os.environ.get('HOST_RUN', '0.0.0.0')
debug=os.environ.get('DEBUG', 'True')
host_redis=os.environ.get('HOST_REDIS', 'redis')
port_redis=os.environ.get('PORT_REDIS', '6379')
app = Flask(__name__)
redis = Redis(host=host_redis, port=port_redis)
@app.route('/')
def hello():
    redis.incr('hits')
    return 'Hello World! %s times.' % redis.get('hits')
if __name__ == "__main__":
    app.run(host=host_run, debug=True)
```

Como você pode perceber no código acima, sua aplicação agora pode receber variáveis de ambiente para configurar o hostname e porta do serviço Redis, ou seja, nesse caso é possível configurar um host e porta da redis que você deseja conectar. E tudo isso pode, e deve, ser especificado no docker-compose.yml, que também passou por uma mudança para se adequar a essa nova boa prática:

```yaml
version: "2"
services:
  web:
    build: .
    ports:
     - "5000:5000"
    volumes:
     - .:/code
    labels:
     - 'app.environment=${ENV_APP}'
    environment:
     - HOST_RUN=${HOST_RUN}
     - DEBUG=${DEBUG}
     - PORT_REDIS=6379
     - HOST_REDIS=redis
  redis:
    image: redis:3.2.1
    volumes:
     - dados:/data
    labels:
     - 'app.environment=${ENV_APP}'
```

```
22 volumes:
23   dados:
24     external: false
```

Como podemos observar nos códigos já explicados, a grande vantagem de se utilizar dessa boa prática é o fato da possibilidade de mudança de comportamento sem a necessidade de mudança do código, mais uma vez é possível viabilizar que o mesmo código que foi construído em um momento, possa ser reutilizado de forma semelhante tanto no notebook do desenvolvedor, como no servidor de produção.

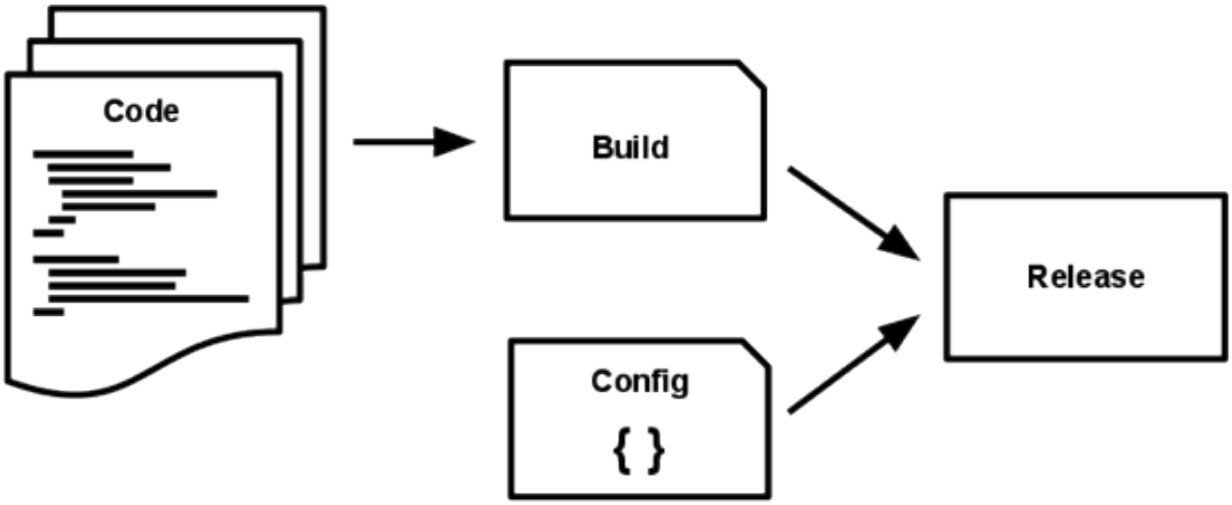
Fique atento para armazenamento de segredos dentro do docker-compose.yml, pois esse arquivo será enviado para o repositório de controle de versão, ou seja, é importante pensar em outra estratégia de manutenção de segredos.

Uma estratégia possível é a manutenção de variáveis de ambiente no docker host e dessa forma você precisaria usar variáveis do tipo **${variavel}** dentro do docker-compose.yml pra pode repassar essa configuração ou utilizar outro recurso mais avançado de gerenciamento de segredos.

Construa, lance, execute

Seguindo a lista do modelo 12factor[84], temos "Construa, lance, execute" como quinta boa prática.

Em um processo de automatização de infraestrutura de implantação de software, precisamos ter alguns cuidados para que o comportamento do processo esteja dentro das expectativas e que erros humanos causem baixo impacto no processo completo do desenvolvimento ao lançamento em produção.



Visando organizar, dividir responsabilidade e deixar o processo mais claro, o 12factor indica que o código base para ser colocado em produção deva passar por três fases:

- **Construa** é quando o código do repositório é convertido em um pacote executável. Nesse processo é onde se obtém todas as dependências, compila-se o binário e todos os ativos desse código.
- **Lance** é quando o pacote produzido pela fase de **construir** é combinado com sua configuração. O resultado é o ambiente completo, configurado e pronto para ser colocado em **execução**.
- **Execute** (também como conhecido como "runtime") é quando um **lançamento** (aplicação + configuração daquele ambiente) é colocado em **execução**, iniciado com base nas configurações especificas do seu ambiente requerido.

Essa boa prática indica que sua aplicação tenha separações explícitas nas fases de **Construa**, **Lance** e **Execute**. Assim cada mudança no código da aplicação é construída apenas uma vez na etapa

[84] http://12factor.net/pt_br/

de **Construa**. Mudanças da configuração não necessitam de uma nova **construção**, sendo apenas necessário passar pelas etapas de **lançar** e **executar**.

Dessa forma é possível criar controles e processos claros em cada etapa, ou seja, caso algo ocorra na **construção** do código, uma medida pode ser tomada e até mesmo abortado o lançamento do mesmo, para que o código em produção não seja comprometido por conta do possível erro.

Com a separação das responsabilidades é possível saber exatamente em qual etapa o problema aconteceu e atuar manualmente caso necessário.

Os artefatos produzidos devem sempre ter um identificador de **lançamento** único, que pode ser o timestamp (como 2011-04-06-20:32:17) ou um número incremental (como v100).

Com o uso de artefato único é possível garantir o uso de uma versão antiga, seja para um plano de retorno ou até mesmo para comparar comportamentos após mudanças no código.

Para atendermos essa boa prática precisamos primeiro construir a imagem Docker com a aplicação dentro, ou seja, ela será nosso artefato.

Teremos um script novo, que aqui chamaremos de build.sh, nesse arquivo teremos o seguinte conteúdo:

```
1  #!/bin/bash
2
3  USER="gomex"
4  TIMESTAMP=$(date "+%Y.%m.%d-%H.%M")
5
6  echo "Construindo a imagem ${USER}/app:${TIMESTAMP}"
7  docker build -t ${USER}/app:${TIMESTAMP} .
8
9  echo "Marcando a tag latest também"
10 docker tag ${USER}/app:${TIMESTAMP} ${USER}/app:latest
11
12 echo "Enviando a imagem para nuvem docker"
13 docker push ${USER}/app:${TIMESTAMP}
14 docker push ${USER}/app:latest
```

Como podem ver no script acima, além de construir a imagem ele envia a mesma para o repositório[85] de imagem do docker.

Lembre-se que o código acima e todos os outros dessa boa prática estão nesse repositório[86] na pasta "**factor5**".

O envio da imagem para o repositório é parte importante da boa prática em questão, pois isso faz com que o processo seja isolado, ou seja, caso a imagem não fosse enviada para um repositório,

[85] http://hub.docker.com/

[86] https://github.com/gomex/exemplo-12factor-docker

ela estaria apenas no servidor que executou o processo de **construção** da imagem, sendo assim a próxima etapa precisaria necessariamente ser executada no mesmo servidor, pois ela precisará da imagem disponível.

No modelo proposto a imagem estando no repositório central, ela estará disponível para ser baixada no servidor caso ela não exista localmente. Caso você utilize uma ferramenta de pipeline é importante você ao invés de utilizar a data para tornar o artefato único, utilize variáveis do seu produto para garantir que a imagem que será consumida na etapa de Executar seja a mesma construída no Lançar. Ex. no GoCD temos as variáveis **GO_PIPELINE_NAME** e **GO_PIPELINE_COUNTER** que podem ser usadas em conjunto para garantir isso.

Com a geração da imagem podemos garantir que a etapa **Construir** foi atendida perfeitamente, pois agora temos um artefato construído e pronto para ser reunido a sua configuração.

A etapa de **Lançamento** é o arquivo docker-compose.yml em si, pois o mesmo deve receber as configurações devidas para o ambiente que se deseja colocar a aplicação em questão, sendo assim o arquivo docker-compose.yml muda um pouco e deixará de fazer **construção** da imagem, pois agora ele será utilizado apenas para **Lançamento** e **Execução** (posteriormente):

```
version: "2"
services:
  web:
    image: gomex/app:latest
    ports:
     - "5000:5000"
    volumes:
     - .:/code
    labels:
     - 'app.environment=${ENV_APP}'
    environment:
     - HOST_RUN=${HOST_RUN}
     - DEBUG=${DEBUG}
     - PORT_REDIS=6379
     - HOST_REDIS=redis
  redis:
    image: redis:3.2.1
    volumes:
     - dados:/data
    labels:
     - 'app.environment=${ENV_APP}'
volumes:
  dados:
    external: false
```

No exemplo **docker-compose.yml** acima usamos a tag latest para garantir que ele usará sempre a ultima imagem **construída** nesse processo, mas como falei anteriormente, caso utilize alguma ferramenta de entrega contínua (Ex. GoCD) faça uso das suas variáveis para garantir o uso da imagem criada naquela execução específica do pipeline.

Dessa forma, **lançamento** e **execução** sempre utilizarão o mesmo artefato, a imagem Docker, construída na fase de construção..

E etapa de **execução** basicamente é executar o docker-compose com o comando abaixo:

```
1 docker-compose up -d
```

Processos

Seguindo a lista do modelo 12factor[87], temos "**Processos**" como sexta boa prática.

Com advento da automatização, e uma devida inteligência na manutenção das aplicações, hoje é esperado que a sua aplicação possa atender a picos de demandas com inicialização automática de novos processos, sem que isso afete negativamente o comportamento da mesma.

Essa boa prática indica que processos de aplicações 12factor são stateless (não armazenam estado) e share-nothing. Quaisquer dados que precise persistir deve ser armazenado em um serviço de apoio stateful(que armazena o seu estado), tipicamente uma base de dados.

O objetivo final dessa prática é não fazer distinção se a aplicação está sendo executada na máquina do desenvolvedor ou em produção, pois nesse caso o que mudaria seria a quantidade de processos iniciados para atender as suas respectivas demandas, ou seja, na máquina do desenvolvedor poderia ser apenas um e em produção um número maior.

O **12factor** indica que o espaço de memória ou sistema de arquivos do servidor pode ser usado brevemente como cache de transação única. Por exemplo, o download de um arquivo grande, operando sobre ele, e armazenando os resultados da operação no banco de dados.

Vale a pena salientar que um estado nunca deve ser armazenado entre requisições, não importando o estado do processamento da próxima requisição.

É importante salientar que ao seguir essa prática, uma aplicação nunca assume que qualquer coisa armazenada em cache de memória ou no disco estará disponível em uma futura solicitação ou job – com muitos processos de cada tipo rodando, as chances são altas de que uma futura solicitação será servida por um processo diferente, até mesmo em um servidor diferente. Mesmo quando rodando em apenas um processo, um restart (desencadeado pelo deploy de um código, mudança de configuração,

[87] http://12factor.net/pt_br

ou o ambiente de execução realocando o processo para uma localização física diferente) geralmente vai acabar com todo o estado local (por exemplo, memória e sistema de arquivos).

Algumas aplicações demandam de sessões persistentes, para armazenar informações da sessão de usuários e afins. Essas sessões são usadas em futuras requisições do mesmo visitante, ou seja, se armazenado junto ao processo isso é uma clara violação dessa boa prática. Nesse caso o mais aconselhável é usar um serviço de apoio, tal como redis, memcached ou afins para esse tipo de trabalho externo ao processo, com isso será possível que o próximo processo, independente onde ele esteja, conseguirá obter as informações atualizadas.

A aplicação que estamos trabalhando não guarda nenhum dado local, e tudo que ela precisa é armazenado no Redis. Dessa forma não precisamos fazer adequação alguma nesse código para seguir essa boa prática, como podemos ver abaixo:

```
1  from flask import Flask
2  from redis import Redis
3  import os
4  host_redis=os.environ.get('HOST_REDIS', 'redis')
5  port_redis=os.environ.get('PORT_REDIS', '6379')
6  app = Flask(__name__)
7   redis = Redis(host=host_redis, port=port_redis)
8  @app.route('/')
9  def hello():
10  redis.incr('hits')
11  return 'Hello World! %s times.' % redis.get('hits')
12 if __name__ == "__main__":
13  app.run(host="0.0.0.0", debug=True)
```

Para acessar o código dessa prática, acesse o nosso repositório[88] e acesse a pasta "**factor6**".

[88] https://github.com/gomex/exemplo-12factor-docker

Dicas para uso do Docker

Se você leu a primeira parte desse livro já sabe o básico de Docker, mas agora que pretende começar a usar com mais frequência alguns desconfortos podem surgir, pois como qualquer ferramenta, Docker tem seu próprio conjunto de boas práticas e dicas para ser efetivo.

O objetivo dessse texto é demonstrar algumas dicas para o bom uso do Docker. Isso não quer dizer que a forma que você executa hoje seja necessariamente errada.

Toda ferramenta demanda de algumas melhores práticas para tornar o seu uso mais efetivo e com menor possibilidade de problemas futuros.

Esse capítulo está separado em duas seções: Dicas para rodar (`docker run`) e boas práticas de construção de imagens (`docker build`/`Dockerfile`).

Dicas para Rodar

Lembre-se que cada comando `docker run` cria um novo container com base em uma imagem especificada e inicia um processo dentro dele a partir de um comando (`CMD` especificado no Dockerfile).

Containers descartáveis

É esperado que os containers executados possam ser descartados sem qualquer problema, sendo assim é importante utilizar containers verdadeiramente efêmeros.

Para tal utilize o argumento `--rm`. Isso faz com que todos o container, e todos seus dados, sejam removidos após o termino da execução, evitando consumir disco de maneira desnecessária.

Em geral, pode-se utilizar o comando `run` conforme o exemplo:

```
1  docker run --rm -it debian /bin/bash
```

Note aqui que `-it` significa `--interactive --tty`. Ele é usado para fixar a linha de comando com o container, assim após esse `docker run` todos comandos serão executados pelo `bash` de dentro do container. Para sair use `exit` ou pressione `Control-d`. Esses parâmetros são muito úteis para executar um container em primeiro plano.

Verifique variáveis de ambiente

Às vezes se faz necessário verificar que metadados estão definidos como variáveis de ambiente em uma imagem. Use o comando `env` para obter esta informação:

```
docker run --rm -it debian env
```

Para verificar as variáveis de ambiente passados de um container já criado:

```
docker inspect --format '{{.Config.Env}}' <container>
```

Para outros metadados, use variações do comando `docker inspect`.

Logs

Docker captura logs da saída padrão (`STDOUT`) e saída de erros (`STDERR`). Esses registros podem ser roteados para diferentes sistemas (`syslog`, `fluentd`, ...) que podem ser especificado através da configuração de driver[89] `--log-driver=VALUE` no comando `docker run`.

Quando utilizado o driver padrão `json-file` (e também `journald`), pode-se utilizar o seguinte comando para recuperar os logs:

```
docker logs -f <container_name>
```

Observe também o argumento `-f` para acompanhar as próximas mensagens de log de forma interativa. Quando quiser parar, pressione `Ctrl-c` .

Backup

Dados em containers Docker são expostos e compartilhados através de argumentos de volumes utilizados ao criar e iniciar o container. Esse volumes não seguem as regras do Union File System[90], pois os dados desse volume são persistidos mesmo quando o container é removido.

Para criar um volume em um determinado container, execute da seguinte forma:

```
docker run --rm -v /usr/share/nginx/html --name nginx_teste nginx
```

Com a execução desse container teremos um serviço Nginx que usará o volume criado para persistir seus dados, ou seja, os dados persistirão mesmo após o container seja removido.

É boa prática de administração de sistema fazer cópias de segurança (backups) periodicamente e para executar essa atividade (extrair dados), use comando a seguir:

[89] https://docs.docker.com/engine/admin/logging/overview/
[90] https://docs.docker.com/engine/reference/glossary/#union-file-system

```
docker run --rm -v /tmp:/backup --volumes-from nginx-teste busybox tar -cvf /bac\
kup/backup_nginx.tar /usr/share/nginx/html
```

Após a execução desse comando teremos um arquivo chamado `backup_nginx.tar` dentro da pasta /tmp do **Docker host**.

Para restaurar esse backup utlize:

```
docker run --rm -v /tmp:/backup --volumes-from nginx-teste busybox tar -xvf /bac\
kup/backup.tar /usr/share/nginx/html
```

Mais informação também pode ser encontrada nessa resposta[91], onde também é possível encontrar alguns *aliases* para esses dois comandos. Esse *aliases* também estão disponíveis abaixo na seção *Aliases*.

Outras fontes são:

- Documentação oficial do Docker sobre Backup, restauração ou migração de dados (em inglês)[92]
- Uma ferramenta de backup (atualmente deprecada): docker-infra/docker-backup[93]

Use docker exec para "entrar num container"

Eventualmente é necessário entrar em um container em execução afim de verificar algum problema, efetuar testes ou simplemente depurar (*debug*). Nunca instale o daemon SSH em um container docker. Use `docker exec` para entrar em um container e rodar um comando:

```
docker exec -it <nome do container em execução> bash
```

Essa funcionalidade é muito útil em desenvolvimento local e experimentações. Mas evite utilizá-lo em produção ou automatizar ferramentas em volta dele.

Verifique a documentação[94] do mesmo.

Sem espaço em disco do Docker Host

Ao executar containers e construir imagens várias vezes, o espaço em disco pode tornar-se escassos. Quando isso acontece, torna-se necessário limpar alguns containers, imagens e logs.

Uma maneira rápida de limpar containers e imagens é utilizar os seguintes comandos:

Para remover containers:

[91] http://stackoverflow.com/a/34776997/1046584
[92] https://docs.docker.com/engine/userguide/containers/dockervolumes/#backup-restore-or-migrate-data-volumes
[93] https://github.com/docker-infra/docker-backup
[94] https://docs.docker.com/engine/reference/commandline/exec/

```
docker ps -aq | xargs docker rm
```

Para remover imagens não utilizadas

```
docker images -aq -f dangling=true | xargs docker rmi
```

Dependendo do tipo de aplicação, logs também podem ser volumosos. O seu gerenciamento depende muito de qual driver[95] está sendo utilizado. No driver padrão (`json-file`) a limpeza pode ser feita através da execução do seguinte comando dentro do **Docker host**:

```
echo "" > $(docker inspect --format='{{.LogPath}}' <container_name_or_id>)
```

- A proposta de funcionalidade de limpar o histórico de logs foi na verdade rejeitada, mais informação: https://github.com/docker/compose/issues/1083[96]
- Considere especificar a opção `max-size` para o *driver* de log ao executar `docker run`: https://docs.docker.com/engine/reference/logging/overview/#json-file-options[97]

Aliases

Com alias é possível transformar comandos grandes em menores, ou seja, dessa forma teremos algumas novas opções para executar tarefas mais complexas.

Utilize esses *aliases* no seu `.zshrc` ou `.bashrc` para limpar imagens e containers, fazer backup e restauração, etc.

```
alias dockercleancontainers="docker ps -aq | xargs docker rm"
alias dockercleanimages="docker images -aq -f dangling=true | xargs docker rmi"
alias dockerclean="dockercleancontainers && dockercleanimages"
alias docker-killall="docker ps -q | xargs docker kill"

# runs docker exec in the latest container
function docker-exec-last {
  docker exec -ti $( docker ps -a -q -l) /bin/bash
}

function docker-get-ip {
  # Usage: docker-get-ip (name or sha)
  [ -n "$1" ] && docker inspect --format "{{ .NetworkSettings.IPAddress }}" $1
```

[95] https://docs.docker.com/engine/admin/logging/overview/
[96] https://github.com/docker/compose/issues/1083
[97] https://docs.docker.com/engine/reference/logging/overview/#json-file-options

```
}

function docker-get-id {
  # Usage: docker-get-id (friendly-name)
  [ -n "$1" ] && docker inspect --format "{{ .ID }}" "$1"
}

function docker-get-image {
  # Usage: docker-get-image (friendly-name)
  [ -n "$1" ] && docker inspect --format "{{ .Image }}" "$1"
}

function docker-get-state {
  # Usage: docker-get-state (friendly-name)
  [ -n "$1" ] && docker inspect --format "{{ .State.Running }}" "$1"
}

function docker-memory {
  for line in `docker ps | awk '{print $1}' | grep -v CONTAINER`; do docker ps |\
 grep $line | awk '{printf $NF" "}' && echo $(( `cat /sys/fs/cgroup/memory/docke\
r/$line*/memory.usage_in_bytes` / 1024 / 1024 ))MB ; done
}
# keeps the commmand history when running a container
function basher() {
    if [[ $1 = 'run' ]]
    then
        shift
        docker run -e HIST_FILE=/root/.bash_history -v $HOME/.bash_history:/root\
/.bash_history "$@"
    else
        docker "$@"
    fi
}
# backup files from a docker volume into /tmp/backup.tar.gz
function docker-volume-backup-compressed() {
  docker run --rm -v /tmp:/backup --volumes-from "$1" debian:jessie tar -czvf /b\
ackup/backup.tar.gz "${@:2}"
}
# restore files from /tmp/backup.tar.gz into a docker volume
function docker-volume-restore-compressed() {
  docker run --rm -v /tmp:/backup --volumes-from "$1" debian:jessie tar -xzvf /b\
ackup/backup.tar.gz "${@:2}"
```

```
56    echo "Double checking files..."
57    docker run --rm -v /tmp:/backup --volumes-from "$1" debian:jessie ls -lh "${@:\
58  2}"
59  }
60  # backup files from a docker volume into /tmp/backup.tar
61  function docker-volume-backup() {
62    docker run --rm -v /tmp:/backup --volumes-from "$1" busybox tar -cvf /backup/b\
63  ackup.tar "${@:2}"
64  }
65  # restore files from /tmp/backup.tar into a docker volume
66  function docker-volume-restore() {
67    docker run --rm -v /tmp:/backup --volumes-from "$1" busybox tar -xvf /backup/b\
68  ackup.tar "${@:2}"
69    echo "Double checking files..."
70    docker run --rm -v /tmp:/backup --volumes-from "$1" busybox ls -lh "${@:2}"
71  }
```

Fontes:

- https://zwischenzugs.wordpress.com/2015/06/14/my-favourite-docker-tip/[98]
- https://website-humblec.rhcloud.com/docker-tips-and-tricks/[99]

Boas práticas para construção de imagens

Em Docker, as imagens são tradicionalmente construídas usando um arquivo `Dockerfile`. Existem alguns bons guias sobre as melhores práticas para construir imagens docker. Recomendo dar uma olhada neles:

- Documentação oficial[100]
- Guia do projeto Atomic[101]
- Melhores práticas do Michael Crosby Parte 1[102]
- Melhores práticas do Michael Crosby Parte 2[103]

[98] https://zwischenzugs.wordpress.com/2015/06/14/my-favourite-docker-tip/
[99] https://website-humblec.rhcloud.com/docker-tips-and-tricks/
[100] https://docs.docker.com/engine/articles/dockerfile_best-practices/
[101] http://www.projectatomic.io/docs/docker-image-author-guidance/
[102] http://crosbymichael.com/dockerfile-best-practices.html
[103] http://crosbymichael.com/dockerfile-best-practices.html

Use um "linter"

Um *"linter"* é uma ferramenta que fornece dicas e avisos sobre algum código fonte. Para `Dockerfile` existem algumas opções simples, mas ainda é um espaço muito novo e que muito tem a evoluir.

Muitas opções foram discutidas aqui[104].

Desde Janeiro de 2016, o mais completo parece ser hadolint[105]. Disponível em duas versões: online e terminal. O interessante dessa ferramente é que usa o maduro Shell Check[106] para validar os comandos shell.

O básico

O container produzido pela imagem do `Dockerfile` deve ser o máximo efêmero possível. Isso significa que pode ser parado, destruído e substituído por um novo container construído com o minimo de esforço.

As vezes é comum colocar outros arquivos, como documentação, no diretório junto ao `Dockerfile`. Para melhorar a performance de contrução, exclua arquivos e diretórios criando um arquivo dockerignore[107] no mesmo diretório. Esse arquivo funciona de maneira semelhante ao `.gitignore`. Usá-lo ajuda a minimizar o contexto de contrução que é enviado docker host a cada `docker build`.

Evite adicionar pacotes e dependências extras e não necessárias a sua aplicação. Isso minimiza a complexidade, tamanho da imagem, tempo de contrução e superficie de ataque.

Minimize o número de camadas, sempre que possível agrupe vários comandos. Porém também leve em conta a volatilidade e manutenção dessas camadas.

Na maioria dos casos, rode apenas um único processo por container. Desacoplando aplicações em vários containers facilida a escalabilidade horizontal, reuso e monitoramento dos containers.

Prefira COPY ao invés de ADD

O comando `ADD` existe desde o início do Docker. É muito versátil e permite alguns truques além de simplesmente copiar arquivos do contexto de contrução, o que o torna muito mágico e difícil de entender. Ele permite baixar arquivos de urls e automaticamente extrair arquivos de formatos conhecidos (tar, gzip, bzip2, etc.).

Por outro lado, `COPY` é um comando muito mais simples para inserir arquivos e pastas do caminho de construção para dentro da imagem Docker. Assim, favoreça `COPY` a menos que tenha certeza absoluta que `ADD` é necessário. Para mais detalhes veja aqui[108].

[104] https://stackoverflow.com/questions/28182047/is-there-a-way-to-lint-the-dockerfile
[105] http://hadolint.lukasmartinelli.ch/
[106] http://www.shellcheck.net/about.html
[107] https://docs.docker.com/engine/reference/builder/
[108] https://labs.ctl.io/dockerfile-add-vs-copy/

Execute um "checksum" depois de baixar e antes de usar o arquivo

Em vez de usar `ADD` para baixar e adicionar arquivos à imagem, favoreça a utilização de curl[109] e então a verificação através de um `checksum` após o download. Isso permite garantir que o arquivo é o esperado e não poderá variar ao longo do tempo. Se o arquivo que a URL aponta mudar, o `checksum` irá mudar e a contrução da imagem irá falhar. Isso é importante pois favorece a reproducibilidade e a segurança na contrução de imagens.

Um bom exemplo para se inspirar é o Dockerfile oficial do Jenkins[110]:

```
1 ENV JENKINS_VERSION 1.625.3
2 ENV JENKINS_SHA 537d910f541c25a23499b222ccd37ca25e074a0c
3
4 RUN curl -fL http://mirrors.jenkins-ci.org/war-stable/$JENKINS_VERSION/jenkins.w\
5 ar -o /usr/share/jenkins/jenkins.war \
6   && echo "$JENKINS_SHA /usr/share/jenkins/jenkins.war" | sha1sum -c -
```

Use uma imagem de base mínima

Sempre que possível utilize imagens oficiais como base para sua imagem. Você pode usar a imagem `debian`[111], por exemplo, que é muito bem controlada e mantida mínima (por volta de 150 mb). Lembre-se também usar *tags* específicas, por exemplo `debian:jessie`.

Se mais ferramentas e dependências são necessários, olhe por imagens como `buildpack-deps`[112].

Porém, caso `debian` ainda seja muito grande, existem imagens minimalistas como `alpine`[113] ou mesmo `busybox`[114]. Evite `alpine` se DNS é necessário, existem alguns problemas a serem resolvidos[115]. Além disso, evite-o para linguagens que usam o GCC, como Ruby, Node, Python, etc, isso é porque `alpine` utiliza libc MUSL que pode produzir binários diferentes.

Evite imagens gigantes como `phusion/baseimage`[116]. Essa uma imagem é muito grande, que foge da filosofia de um processo por container e muitas das coisas que inclui não são essenciais para containers Docker, veja mais aqui[117] .

Outras fontes[118]

[109] https://curl.haxx.se/
[110] https://github.com/jenkinsci/docker/blob/83ce6f6070f1670563a00d0f61d04edd62b78f4f/Dockerfile#L36
[111] https://hub.docker.com/_/debian/
[112] https://hub.docker.com/_/buildpack-deps/
[113] https://hub.docker.com/r/gliderlabs/alpine/
[114] https://hub.docker.com/r/gliderlabs/alpine/
[115] https://github.com/gliderlabs/docker-alpine/blob/master/docs/caveats.md
[116] https://hub.docker.com/r/phusion/baseimage/
[117] https://blog.docker.com/2014/06/why-you-dont-need-to-run-sshd-in-docker/
[118] http://www.iron.io/microcontainers-tiny-portable-containers/

Use o cache de construção de camadas

Uma funcionalidade muito útil que o uso de `Dockerfile` proporciona é as reconstruções rápidas usando o cache de camadas. A fim de aproveitar esse recurso, coloque ferramentas e dependências que mudam com menos frequência no topo do `Dockerfile`.

Por exemplo, considere instalar as dependências de código antes de adicionar o código. No caso de NodeJS:

```
1 COPY package.json /app/
2 RUN npm install
3 COPY . /app
```

Para ler um pouco mais sobre isso, veja esse link[119].

Limpe na mesma camada

Ao usar um gerenciador de pacotes para instalar algum software, é uma boa prática limpar o cache gerado pelo gerenciador de pacotes logo após a instalação das dependências. Por exemplo, ao usar `apt-get`:

```
1 RUN apt-get update && \
2     apt-get install -y curl python-pip && \
3     pip install requests && \
4     apt-get remove -y python-pip curl && \
5     rm -rf /var/lib/apt/lists/*
```

Em geral deve-se limpar o cache do apt (gerado por `apt-get update`) através da remoção de `/var/lib/apt/lists`. Isso ajuda a manter o tamanho da imagem pequeno. Além disso, observe aqui, que `pip` e `curl` também são removidos uma vez que não são necessários para a aplicação de produção. Lembre-se que a limpeza precisa ser feito na mesma camada (comando `RUN`). Caso contrário os dados serão persistidos nessa camada, e removê-lo mais tarde não terá efeito no tamanho da imagem final.

Note que, segundo a documentação[120], as imagens oficiais de Debian e Ubuntu rodam autoamaticamente `apt-get clean`, logo a invocação explicita não é necesária.

Evite rodar `apt-get upgrade` ou `dist-upgrade`, já que várias pacotes da imagem base não vão atualizar dentro de um container desprovido de privilégios. Se há um pacote em específico a ser atualizado, simplesmete use `apt-get install -y foo` para atualizar automaticamente.

Para ler um pouco mais sobre isso, veja esse link[121] e esse outro[122].

[119] http://bitjudo.com/blog/2014/03/13/building-efficient-dockerfiles-node-dot-js/

[120] https://github.com/docker/docker/blob/03e2923e42446dbb830c654d0eec323a0b4ef02a/contrib/mkimage/debootstrap#L82-L105

[121] http://blog.replicated.com/2016/02/05/refactoring-a-dockerfile-for-image-size/

[122] https://docs.docker.com/engine/userguide/eng-image/dockerfile_best-practices/#apt-get

Use um script "wrapper" como ENTRYPOINT, as vezes

Um *script wrapper* pode ajudar ao tomar a configuração do ambiente e definir a configuração do aplicativo. Ele pode até mesmo definir configurações padrões quando não disponíveis.

Um ótimo exemplo é fornecido no artigo de Kelsey Hightower: 12 Fracturated Apps[123]:

```
#!/bin/sh
set -e
datadir=${APP_DATADIR:="/var/lib/data"}
host=${APP_HOST:="127.0.0.1"}
port=${APP_PORT:="3306"}
username=${APP_USERNAME:=""}
password=${APP_PASSWORD:=""}
database=${APP_DATABASE:=""}
cat <<EOF > /etc/config.json
{
  "datadir": "${datadir}",
  "host": "${host}",
  "port": "${port}",
  "username": "${username}",
  "password": "${password}",
  "database": "${database}"
}
EOF
mkdir -p ${APP_DATADIR}
exec "/app"
```

Note, **sempre** use `exec` em scripts shell que envolvem a aplicação. Desta forma, a aplicação pode receber sinais Unix.

Considere também usar um sistema de inicialização simples (e.g. dumb init[124]) como a `CMD` base, desta forma os sinais do Unix podem ser devidamente tratados. Leia mais sobre isso aqui[125].

Log para stdout

Aplicações dentro de Docker devem emitir logs para `stdout`. Porém algumas aplicações escrevem esses logs em arquivos. Nestes casos, a solução é criar um *symlink* do arquivo para `stdout`.

Um exemplo é o Dockerfile do nginx[126]:

[123] https://medium.com/@kelseyhightower/12-fractured-apps-1080c73d481c#.xn2cylwnk
[124] https://github.com/Yelp/dumb-init
[125] http://engineeringblog.yelp.com/2016/01/dumb-init-an-init-for-docker.html
[126] https://github.com/nginxinc/docker-nginx/blob/master/Dockerfile

```
# forward request and error logs to docker log collector
RUN ln -sf /dev/stdout /var/log/nginx/access.log
RUN ln -sf /dev/stderr /var/log/nginx/error.log
```

Para ler um pouco mais sobre isso, veja esse link[127].

Cuidado ao adicionar dados em um volume no Dockerfile

Lembre de usar a intrução `VOLUME` para expor dados de bancos de dados, configuração ou arquivos e pastas criados pelo container. Use para qualquer dados mutável e partes que são servidas ao usuário do serviço que para qual essa imagem foi criada.

Evite adicionar muitos dados para uma pasta e, em seguida, transformá-lo em um `VOLUME` apenas na inicialização do container, pois nesse momento pode tornar seu carregamento mais lento. Ao criar container, os dados serão copiados da imagem para o volume montado. Como foi dito antes, use `VOLUME` na criação da imagem.

Além disso, ainda em tempo de criação da imagem(`build`), não adicione dados para caminhos que tenham sido previamente declarados como `VOLUME`. Isso não irá funcionar, os dados não serão persistidos pois dados em volumes não são *comitados* em imagens.

Leia mais sobre isso aqui na explicação de Jérôme Petazzoni[128].

EXPOSE de portas

Docker favorece reproducibilidade e portabilidade. Imagens devem ser capazes de rodar em qualquer servidor e quantas vezes forem necessárias. Dessa forma, nunca exponha portas públicas. Porém exponha, de maneira privada, as portas padrões da sua aplicação.

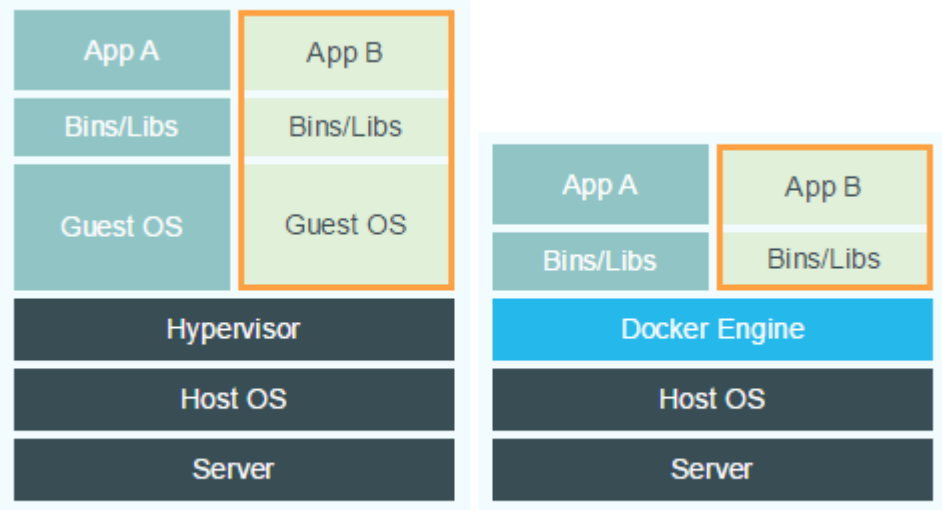
```
# mapeamento publico e privado, evite isso
EXPOSE 80:8080

# apenas privado
EXPOSE 80
```

[127] https://serverfault.com/questions/599103/make-a-docker-application-write-to-stdout
[128] https://jpetazzo.github.io/2015/01/19/dockerfile-and-data-in-volumes/

Apêndice

Container ou máquina virtual?

Após o sucesso repentino do Docker, que utiliza a virtualização com base em containers, muitas pessoas tem se questionado sobre uma possível migração do modelo de máquina virtual para containers.



Eu respondo tranquilamente: **Os dois!**

Ambos são métodos de virtualização, mas elas atuam em "camadas" distintas. Vale a pena detalhar cada solução para deixar claro que elas não são necessariamente concorrentes.

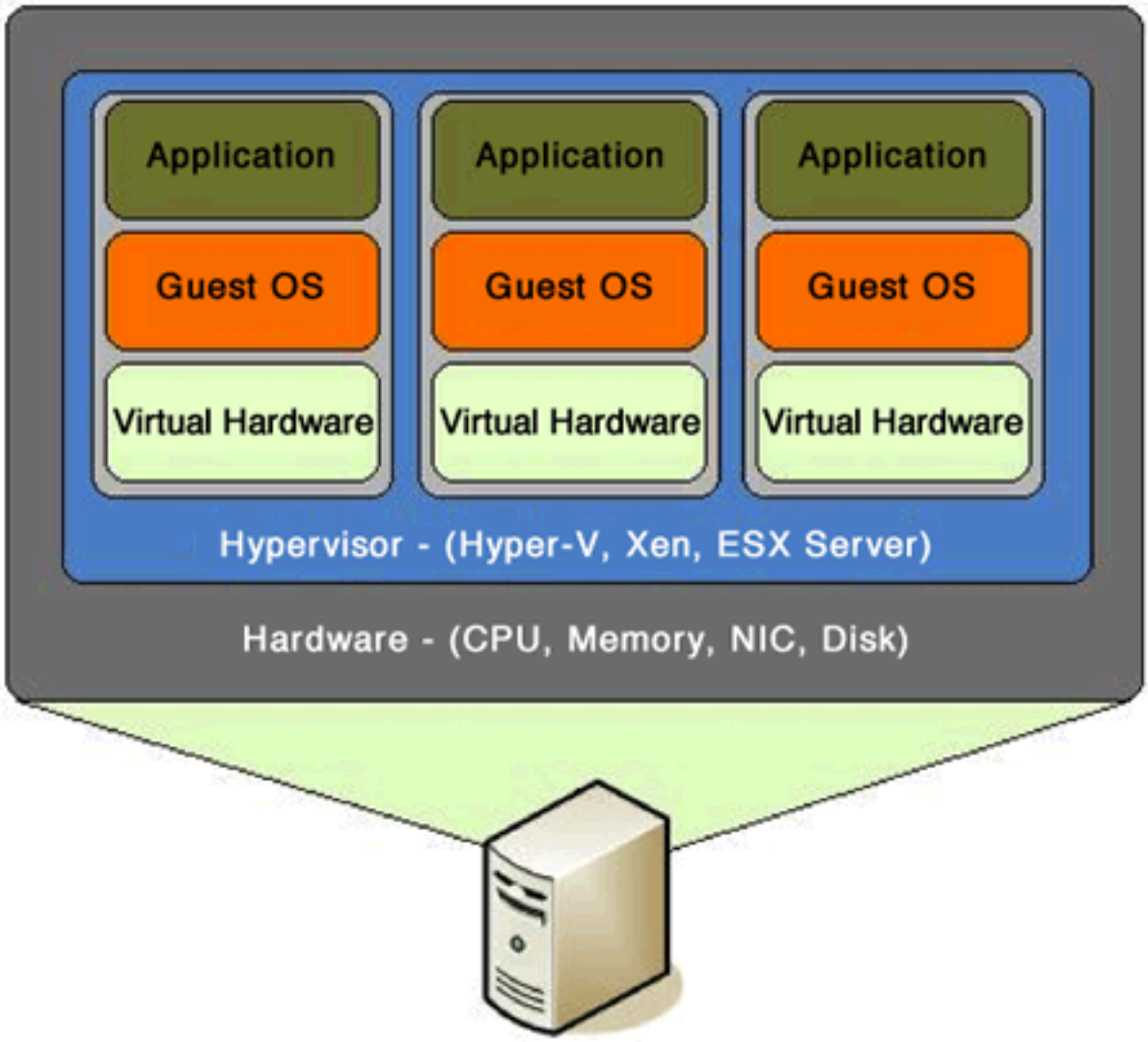
Máquina virtual

Um conceito bem antigo, oriundo dos Mainframes em meados de 1960, onde cada operador desse ambiente tinha a visão de estar acessando uma máquina dedicada, mas na verdade todo recurso do Mainframe era compartilhado para todos operadores.

O objetivo desse modelo é compartilhar os recursos físicos entre vários ambientes isolados, sendo que cada um deles tem sob sua tutela uma máquina inteira, com memória, disco, processador, rede e outros periféricos, todos entregues via abstração de virtualização.

É como se dentro de uma máquina física criasse máquinas menores e independentes entre sí. Cada máquina dessa tem seu próprio sistema operacional completo, que por sua vez interage com todos os hardwares virtuais que lhe foi entregue pelo modelo de virtualização a nível de máquina.

Vale ressaltar que o sistema operacional instalado dentro de uma máquina virtual fará interação com os hardwares virtuais e não com o hardware real.



Com a evolução desse modelo, os softwares que implementam essa solução puderam oferecer mais funcionalidades, tal como melhor interface para gerência de ambientes virtuais e alta disponibilidade utilizando vários hosts físicos.

Com as novas funcionalidades de gerência de ambientes de máquinas virtuais é possível especificar quanto recurso físico cada ambiente virtual usará e até mesmo aumentar gradualmente em caso de necessidade pontual.

Atualmente as máquinas virtuais são uma realidade para qualquer organização que necessite de

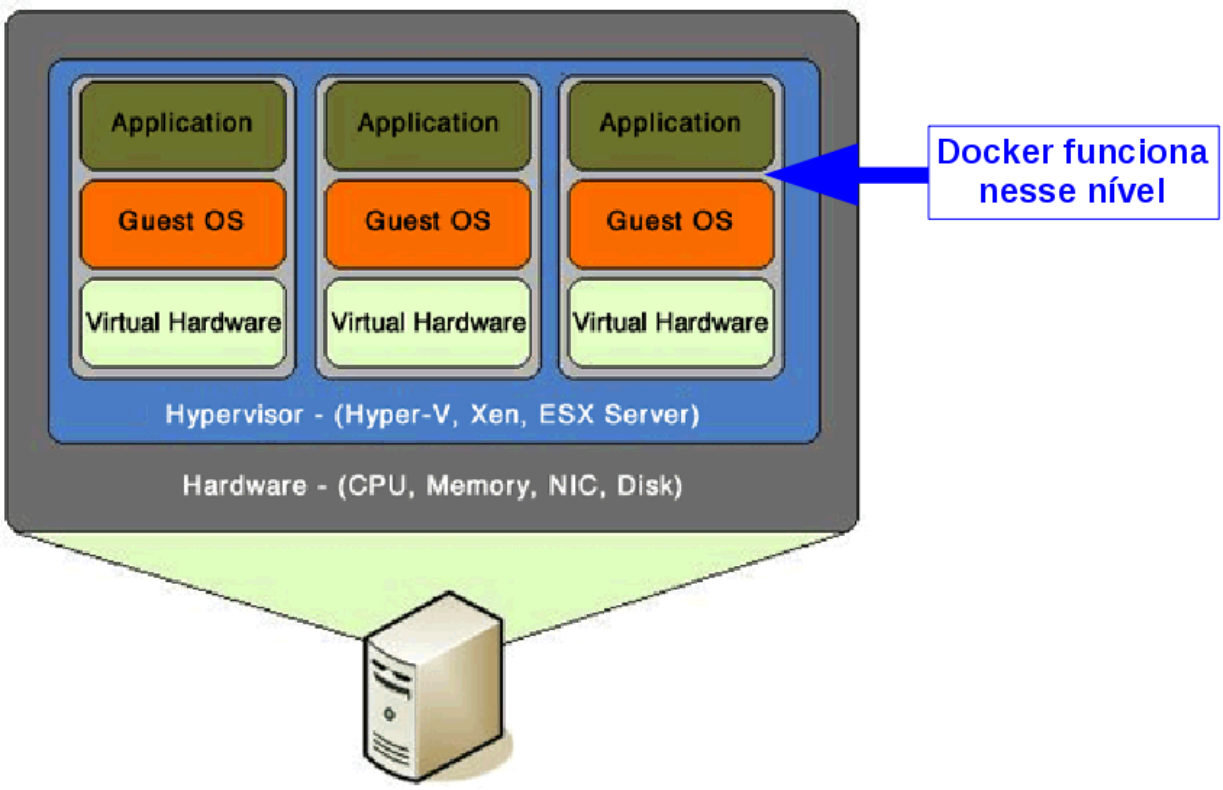
um ambiente de TI, pois facilita a gerência das máquinas físicas e seu compartilhamento entre os diversos ambientes necessários para sua infraestrutura básica.

Container

Esse modelo de virtualização está no nível de sistema operacional, ou seja, ao contrário da máquina virtual um container não tem visão de uma máquina inteira, ele é apenas um processo em execução em um kernel compartilhado entre todos os outros containers.

Ele utiliza o namespace para prover o devido isolamento de memória RAM, processamento, disco e acesso a rede, ou seja, mesmo compartilhamento o mesmo kernel, esse processo em execução tem a visão de estar usando um sistema operacional dedicado.

Esse modelo de virtualização é relativamente antigo, meados de 1982 o chroot já fazia algo que podemos considerar que era uma virtualização a nível de sistema operacional e em 2008 o LXC já fazia algo relativamente parecido ao que o Docker faz hoje. Inclusive o Docker usava o LXC no início, mas hoje já tem interface própria para acessar o namespace, cgroup e afins.



Como solução inovadora, o Docker trás diversos serviços e novas facilidades que deixam esse modelo muito mais atrativo.

A configuração de um ambiente LXC não era uma tarefa relativamente simples. Era necessário algum conhecimento técnico médio para criar e manter um ambiente com ele. Com advento do

Docker esse processo ficou bem mais simples. Basta instalar o binário, baixar as imagens e executá-las.

Outra novidade do Docker foi a criação do conceito de "imagens", que grosseiramente podemos descrever que as imagens são definições estáticas de como os containers devem ser no momento da sua inicialização. São como fotografias de um ambiente. Uma vez instanciadas, colocadas em execução, elas assumem a função de containers, ou seja, saem da abstração de definição e se transformam em processos em execução, dentro de um contexto isolado, que enxergam um sistema operacional dedicado pra sí, mas na verdade compartilham o mesmo kernel.

Junto a facilidade de uso dos containers, o Docker agregou o conceito de nuvem, que dispõe de serviço um para carregar e "baixar" imagens Dockers, ou seja, se trata de uma aplicação web que disponibiliza um repositório de ambientes prontos, onde viabilizou um nível absurdo de compartilhamento de ambientes.

Com o uso do serviço de nuvem do Docker, podemos perceber que a adoção do modelo de containers ultrapassa a questão técnica e adentra nos assuntos de processo, gerência e atualização do ambiente. Onde agora é possível compartilhar facilmente as mudanças e viabilizar uma gestão centralizada das definições de ambiente.

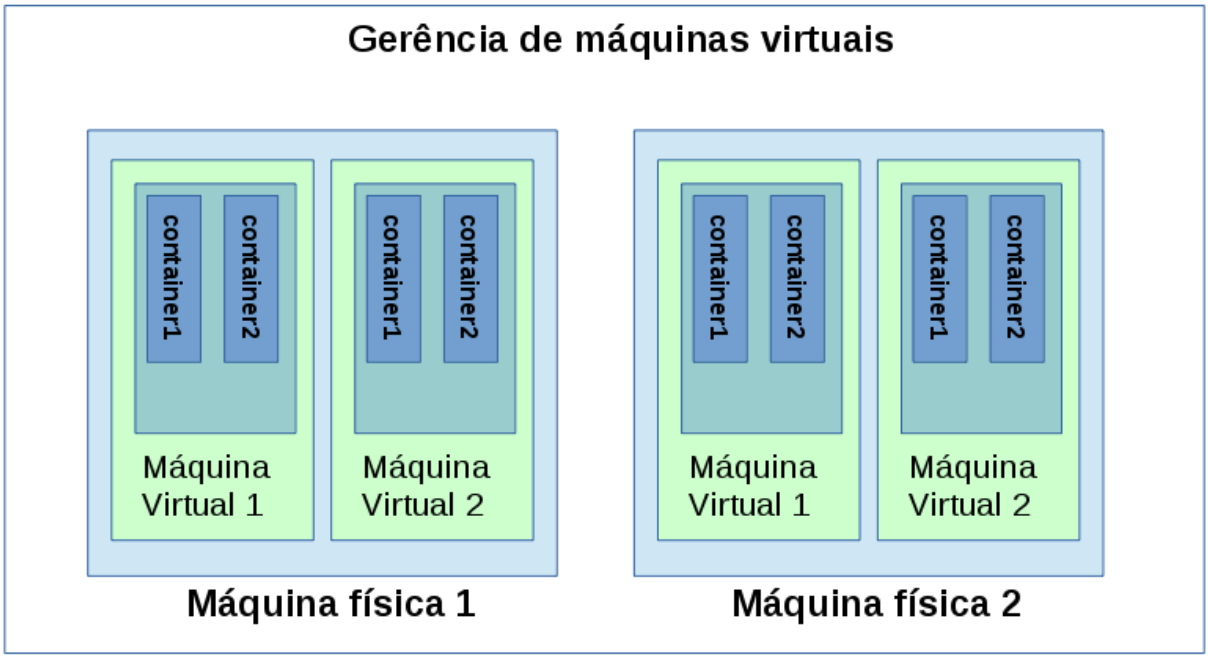
Utilizando a nuvem Docker agora é possível disponibilizar ambientes de teste mais leves. Isso permite a situação onde em plena reunião com seu chefe você pode baixar aquela solução para um problema que ele descreve e mostrar antes que ele saia da sala. Permite também que você consiga disponibilizar um padrão de melhores práticas para um determinado serviço e compartilhar com todos da sua empresa ou mundo, onde pode receber críticas e modificar com o passar do tempo.

Conclusão

Com os dados apresentados, podemos perceber que o ponto de conflito entre as soluções é baixo. Elas podem e normalmente são adotadas em conjunto. Você pode provisionar uma máquina física com um servidor de máquinas virtuais, onde serão criadas máquinas virtuais hospedes, que por sua vez terão o docker instalado em cada máquina. Nesse docker serão disponibilizados os ambientes com seus respectivos serviços segregados, cada um em um container.

Percebam que teremos vários níveis de isolamento. No primeiro temos uma máquina física que foi repartida em várias máquinas virtuais, ou seja, já temos nessa camada sistemas operacionais interagindo com hardwares virtuais distintos, como placas de rede virtuais, discos, processo e memória. Nesse ambiente teríamos apenas o sistema operacional básico instalado e o docker.

No segundo nível de isolamento, temos o Docker baixando as imagens prontas e provisionamento containers em execução, que por sua vez criam novos ambientes isolados, a nível de processamento, memória, disco e rede. Nesse caso podemos ter na mesma máquina virtual um ambiente de aplicação web e banco de dados, mas em containers diferentes e isso não seria nenhum problema de boas práticas de gerência de serviços, muito menos de segurança.



Caso esses containers sejam replicados entre as máquinas virtuais, seria possível prover alta disponibilidade sem grandes custos, ou seja, usando um balanceador externo e viabilizando o cluster dos dados persistidos pelos bancos de dados.

Toda essa facilidade com poucos comandos, recursos e conhecimento. Basta um pouco de tempo para mudar o paradigma de gerência dos ativos e paciência para lidar com os novos problemas inerentes do modelo.