Tabela de conteúdos

Sumário

Organização	1.1
Propósito desse livro	1.2
Agradecimentos	1.3
Introdução	1.4
Por que usar Docker?	1.5
O que é Docker	1.6
Instalação	1.7
Comandos básicos	1.8
Criando sua própria imagem no Docker	1.9
Entendendo armazenamento no Docker	1.10
Entendendo a rede no Docker	1.11
Utilizando docker em múltiplos ambientes	1.12
Gerenciando múltiplos containers docker com Docker Compose	1.13
Como usar Docker sem GNU/Linux	1.14
Transformando sua aplicação em container	1.15
Base de código	1.16
Dependência	1.17
Configurações	1.18
Serviços de Apoio	1.19
Construa, lance, execute	1.20
Processos	1.21
Vínculo de portas	1.22
Concorrência	1.23
Descartabilidade	1.24
Paridade entre desenvolvimento/produção	1.25
Logs	1.26
Processos de administração	1.27
Dicas	1.28
Apêndice	1.29
Container ou máquina virtual?	1.30

Como ler esse livro

Esse material foi dividido em duas grandes partes. A primeira trata das questões mais básicas do Docker. É exatamente o mínimo necessário que um desenvolvedor precisa saber para utilizar essa tecnologia com propriedade, ou seja, ciente do que exatamente acontece ao executar cada comando.

Nessa primeira parte tentaremos não abordar questões de "baixo nível" do Docker, pois são de maior apelo para a equipe responsável pela infraestrutura.

Caso você não saiba nada sobre Docker, aconselhamos **muito** a leitura dessa primeira parte, pois assim conseguirá aproveitar a segunda parte, focada na construção de uma aplicação web no Docker seguindo as melhores práticas, sem pausas. Neste livro, adotamos as práticas do 12factor.

O **12factor** será detalhado no início da segunda parte, mas podemos adiantar que o consideramos os "12 mandamentos para aplicações web no Docker", ou seja, uma vez que sua aplicação siga todas as boas práticas apresentadas neste documento, você possivelmente estará usando todo potencial que o Docker tem a lhe proporcionar.

Essa segunda parte é dividida por cada boa prática do **12factor**. Dessa forma, apresentamos um código de exemplo no primeiro capítulo, que será evoluído ao longo do desenvolvimento do livro. A ideia é que você possa exercitar com um código de verdade e, assim, assimilar o conteúdo de forma prática. Também organizamos alguns apêndices com assuntos extras importantes, mas que não se encaixaram nos capítulos.

Propósito desse livro

Tem como objetivo explicar, de forma simples e direta, como os desenvolvedores podem usar o Docker.

Não trata sobre a infraestrutura do Docker!

Apresenta a utilização do Docker e aprofunda-se nas melhores práticas de uso.

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

Agradecimentos

Meu primeiro agradecimento vai para a pessoa que me deu a chance de estar aqui e poder escrever esse livro: minha mãe. A famosa Cigana, ou Dona Arlete, pessoa maravilhosa, que pra mim é um exemplo de ser humano.

Quero agradecer também a minha segunda mãe, Dona Maria, que tanto cuidou de mim quando eu era criança, enquanto Dona Arlete tomava conta dos outros dois filhos e um sobrinho. Me sinto sortudo por ter duas, enquanto muitos não tem ao menos uma mãe.

Aproveito para agradecer a pessoa que me apresentou o Docker, Robinho, também conhecido como Robson Peixoto. Em uma conversa informal no evento Linguágil, em Salvador (BA), ele me falou: "Estude Docker!" E, aqui estou eu terminando um livro que transformou a minha vida. Obrigado de verdade Robinho!

Obrigado a Luís Armando Bianchin, que começou como autor junto comigo, mas depois por força do destino acabou não podendo continuar. Fica aqui meu agradecimento, pois foi com seu feedback constante que pude continuar a fazer o livro.

Obrigado a Paulo Caroli que tanto me incentivou a escrever o livro e indicou a plataforma Leanpub pra fazer isso. Se não fosse ele, o livro não teria saído tão rápido.

Obrigada a fantástica Emma Pinheiro, pela belíssima capa.

Quero agradecer muito as pessoas incríveis do Raul Hacker Club que tanto me incentivaram em todo esse tempo.

Quero agradecer a mãe do meu filho, Eriane Soares, hoje minha grande amiga que tanto me incentivou a escrever o livro enquanto morávamos juntos!

Como todo produto de conhecimento aberto, esse livro não seria possível sem ajuda dessa vibrante comunidade Docker Brasil. Para alguns membros darei destaque pelo empenho ao ler diversos capítulos inúmeras vezes e dedicar seu precioso tempo sugerindo melhorias:

- Gjuniioor gjuniioor@protonmail.ch
- Marco Antonio Martins Junior - Escreveu os capítulos "Posso rodar aplicações GUI" e "Comandos úteis".
- Jorge Flávio Costa
- Glesio Paiva
- Bruno Emanuel Silva
- George Moura
- Felipe de Morais
- Waldemar Neto
- Igor Garcia
- Diogo Fernandes

É bem possível que eu tenha esquecido o nome de pessoas aqui, mas a medida que for resgatando meus logs, vou atualizar.

Introdução

Essa parte do livro é direcionada a quem não tem conhecimento básico do Docker. Caso você já saiba usar, fique à vontade para pular. Entretanto, mesmo que você já conheça o Docker, apresentamos explicações sobre vários recursos disponíveis e como funcionam.

Mesmo que você já use o Docker, ler essa parte do livro, em algum momento da sua vida, pode ser 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 escritos, geralmente tratando sobre como usá-lo, ferramentas auxiliares, integrações e afins, mas muitas pessoas ainda fazem a pergunta 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. Mas nesse capítulo pretendemos demonstrar alguns bons motivos para se utilizar Docker.

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

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. Isso significa que poderá utilizar sua aplicação no notebook do desenvolvedor da mesma forma que seria executada no servidor de produção.

A imagem Docker aceita parâmetros durante o início do container, isso indica que a mesma imagem pode se comportar de formas diferentes entre distintos ambientes. Esse container pode conectar-se a seu banco de dados local para testes, usando as credenciais e base de dados de teste. Mas quando o container, criado a partir da mesma imagem, receber parâmetros do ambiente de produção, 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 - o que proporciona maior previsibilidade comparado a outras ferramentas como Puppet, Chef, Ansible, etc - impactando positivamente na análise de erros, assim como na confiabilidade do processo de entrega contínua, 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 seu código, seja ele compilado ou dinâmico.

2 – Aplicação como pacote completo

Utilizando as imagens Docker é possível empacotar toda sua aplicação e dependências, facilitando a distribuição, pois não será mais necessário enviar uma extensa documentação explicando como configurar a infraestrutura necessária para permitir a execução, basta disponibilizar a imagem em repositório e liberar o acesso para o usuário e, ele mesmo pode baixar o pacote, que será executado sem problemas.

A atualização também é positivamente afetada, pois a estrutura de camadas 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 precisa executar apenas um comando para atualizar a imagem da aplicação, que será 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 significa que em caso de problema na atualização, o plano de retorno será basicamente utilizar a imagem com a tag anterior.

3 – Padronização e replicação

Como as imagens Docker são construídas através de arquivos de definição, é possível garantir que determinado padrão seja seguido, aumentando a confiança na replicação. Basta que as imagens sigam as melhores práticas de construção para que seja viável escalarmos a estrutura rapidamente.

Caso a equipe receba uma nova pessoa para trabalhar no desenvolvimento, essa poderá receber o ambiente de trabalho com alguns comandos. Esse processo durará o tempo do download das imagens a serem utilizadas, assim como os arquivos de definições da orquestração das mesmas. Isso facilita a introdução de um novo membro no processo de desenvolvimento da aplicação, que poderá reproduzir rapidamente o ambiente em sua estação e assim desenvolver códigos seguindo o padrão da equipe.

Na necessidade de se testar nova versão de determinada parte da solução, usando imagens Docker, normalmente basta a mudança de um ou mais parâmetros do arquivo de definição para iniciar um ambiente modificado com a versão requerida para avaliação. Ou seja: criar e modificar a 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. Agora é possível ambos os setores apresentarem propostas e contra propostas com base em um documento comum.

A infraestrutura requerida estará presente no código do desenvolvedor e a área de infraestrutura poderá analisar o 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 é possível acessar o github ou gitlab à procura de exemplos de código, usando o repositório de imagens do Docker é possível conseguir bons modelos de infraestrutura de aplicações ou serviços prontos para integrações complexas.

Um exemplo é o nginx como proxy reverso e mysql como banco de dados. Caso a 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 o 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 os ambientes e evitar apenas o trabalho da instalação básica.

Dúvidas

Algumas pessoas enviaram dúvidas relacionadas às vantagens que explicitamos nesse texto. Assim, ao invés de respondê-las pontualmente, resolvemos publicar as perguntas e as respectivas respostas aqui.

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, Ansible e Chef. Elas podem garantir ambientes parecidos, uma vez que faz parte do seu papel manter 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 achamos que uma das distinções mais relevante está no seguinte fato: a imagem é uma abstração completa e não requer qualquer tratamento para lidar com as mais variadas distribuições GNU/Linux existentes, já que a imagem Docker carrega em si uma cópia completa dos arquivos de uma distribuição enxuta.

Carregar em si a cópia de uma distribuição GNU/Linux não costuma ser problema para o Docker, pois utilizando o modelo de camadas, economiza bastante recurso, reutilizando as camadas de base. Leia esse artigo para entender mais sobre armazenamento do Docker.

Outra vantagem da imagem em relação a gerência de configuração é 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 um comando custumam ser suficientes para iniciar uma aplicação criada como imagem Docker.

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

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

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

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 da camada mais próxima da aplicação e não de toda a 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. Isso não o obriga a realizar essa atividade. É uma possibilidade que agrada muitos desenvolvedores, mas caso não seja a situação, pode ficar tranquilo, outra equipe tratará dessa parte. Apenas o processo de implantação será pouco mais lento.

Muitas pessoas falam de Docker para micro serviços. É possível usar o Docker para aplicações monolíticas?

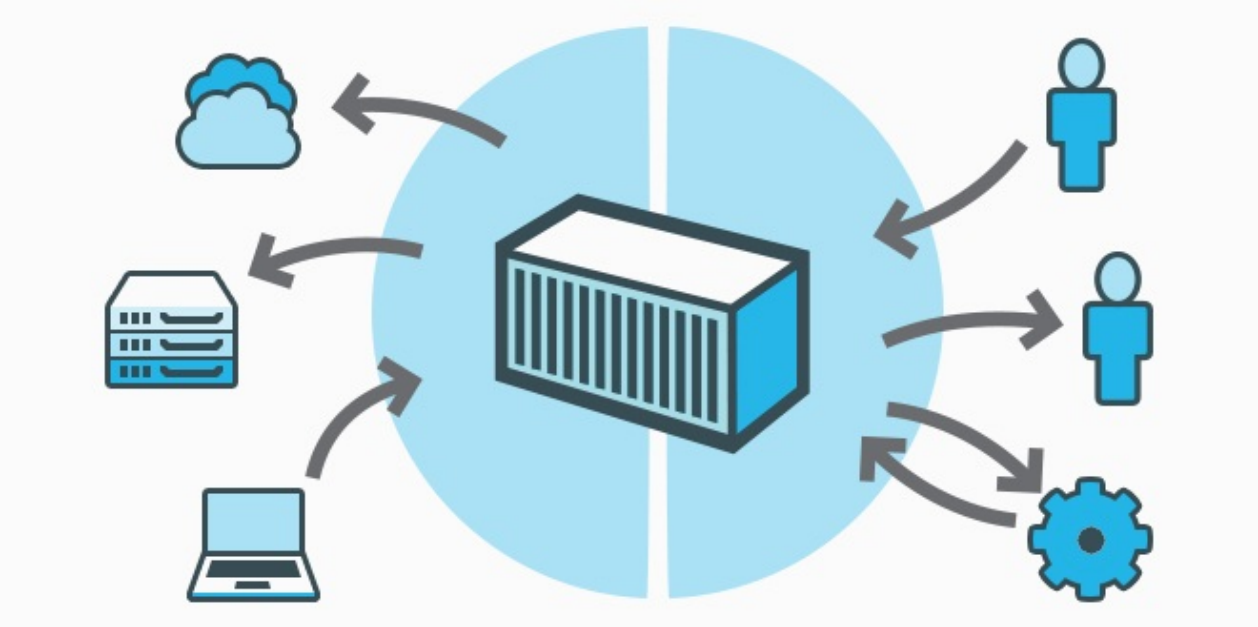
Sim! Porém, em alguns casos, é necessário pequenas modificações na aplicação, para que ela possa usufruir das facilidades do Docker. Um exemplo comum é o log que, normalmente, a aplicação envia para determinado arquivo, ou seja, no modelo Docker as aplicações que estão nos containers não devem tentar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

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

Tem mais dúvidas e/ou bons motivos para utilizar Docker? Comente aqui.

O que é Docker?

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



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

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

O Docker viabilizou uma "linguagem" comum entre desenvolvedores e administradores de servidores. Essa nova "linguagem" é utilizada para construir arquivos com as definições da infraestrutura necessária e 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 também disponibiliza uma nuvem pública para compartilhamento de ambientes prontos, que podem ser utilizados para viabilizar customizações para ambientes específicos. É possível obter uma imagem pronta do apache e configurar os módulos específicos necessários para a aplicação e, assim, criar seu próprio ambiente customizado. Tudo com poucas linhas de descrição.

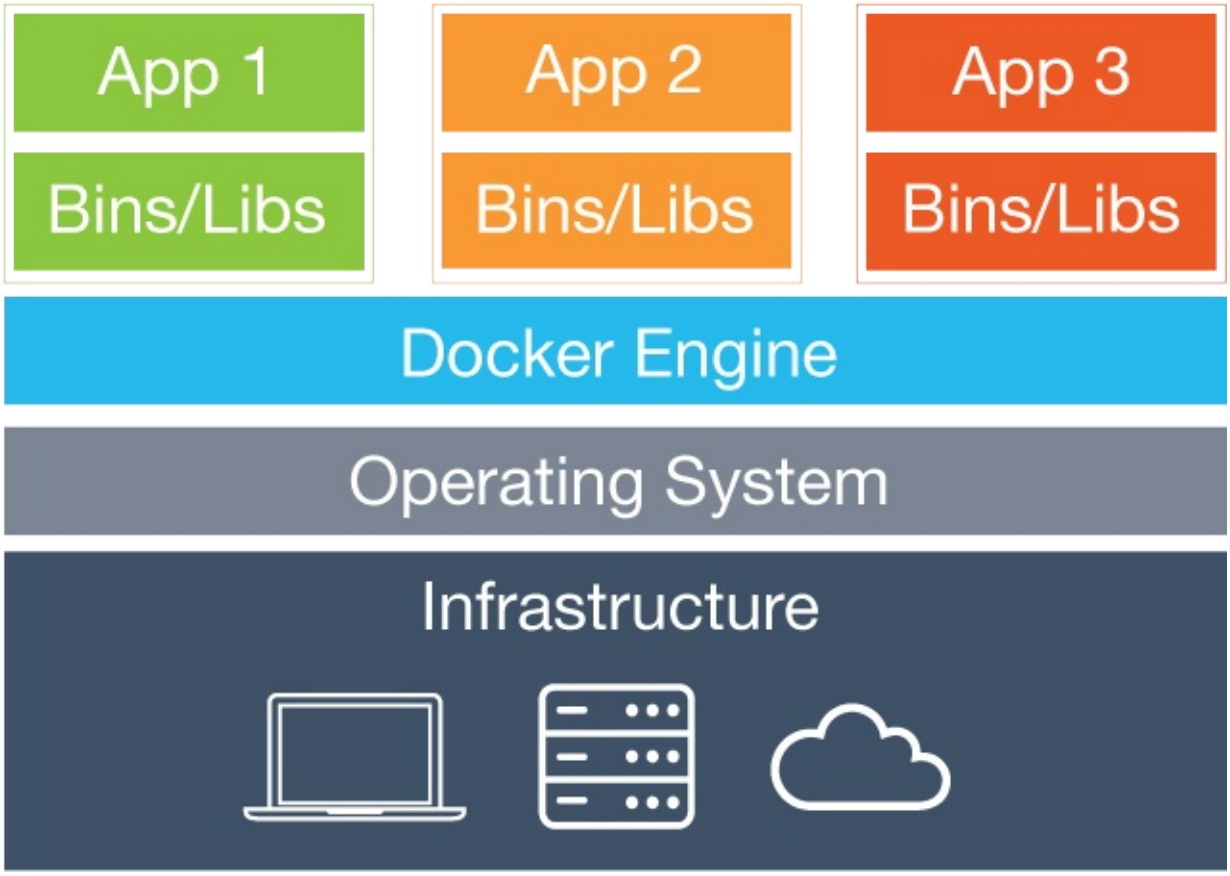
O Docker utiliza o modelo de container para “empacotar” a aplicação que, após ser transformada em imagem Docker, pode ser reproduzida em plataforma de qualquer porte; ou seja, caso a aplicação funcione sem falhas em seu notebook, funcionará também no servidor ou no mainframe. Construa uma vez, execute onde quiser.

Os containers são isolados a nível de disco, memória, processamento e rede. Essa separação permite 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 carrega todos os arquivos necessários (configuração, biblioteca e afins) para execução completamente isolada.

Outro ponto interessante no Docker é a velocidade para viabilizar o ambiente desejado; 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, 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 do processo, o Docker usa a funcionalidade do kernel, denominada de namespaces, que cria ambientes isolados entre containers: os processos de uma aplicação em execução não terão acesso aos recursos de outra. A menos que 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 do kernel, 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.

Instalação

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

Nesse ecossistema temos os seguintes softwares:

- **Docker Engine:** É o software base de toda solução. É tanto o daemon responsável pelos containers como o cliente usado para enviar comandos para o daemon.
- **Docker Compose:** É a ferramenta responsável pela definição e execução de múltiplos containers com base em arquivo de definição.
- **Docker Machine:** é a ferramenta que possibilita criar e manter ambientes docker em máquinas virtuais, ambientes de nuvem e até mesmo em máquina física.

Não citamos o Swarm e outras ferramentas por não estarem alinhados com o objetivo desse livro: introdução para desenvolvedores.

Instalando no GNU/Linux

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

Docker engine no GNU/Linux

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

```
su - root
```

ou no caso da utilização de sudo

```
sudo su - root
```

Execute o comando abaixo:

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

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

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

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

Tratamento de possíveis problemas

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

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

Instalando Docker compose com pip

O pip) é um gerenciador de pacotes Python e, como o docker-compose, é escrito nessa linguagem, é possível instalá-lo da seguinte forma:

```
pip install docker-compose
```

Tratamento de possíveis problemas

Caso não tenha instalado o comando **pip** em seu computador, normalmente ele pode ser instalado usando seu sistema de gerenciamento de pacote com o nome **python-pip** ou semelhante.

Docker machine no GNU/Linux

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

```
su - root
```

ou no caso da utilização de sudo

```
sudo su - root
```

Execute o comando abaixo:

```
$ curl -L https://github.com/docker/machine/releases/download/v0.10.0/docker-machine-`uname -s`-`uname -m` > /usr/loc
al/bin/docker-machine && \
chmod +x /usr/local/bin/docker-machine
```

Para testar, execute o comando abaixo:

```
docker-machine version
```

Obs.: O exemplo anterior utiliza a versão mais recente no momento desta publicação. Verifique se há alguma versão atualizada consultando a documentação oficial.

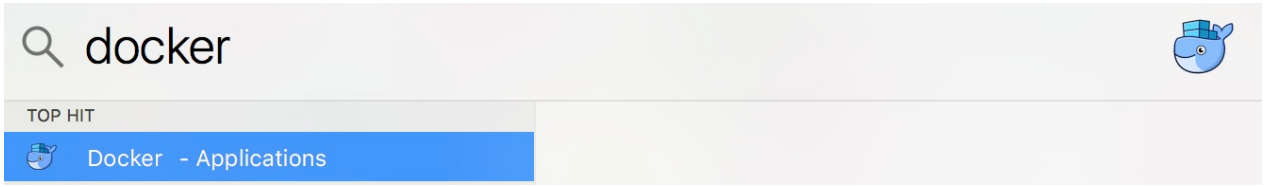
Instalando no MacOS

A instalação das ferramentas do Ecossistema Docker no MacOS pode ser realizada através de um único grande pacote chamado **Docker for Mac**.

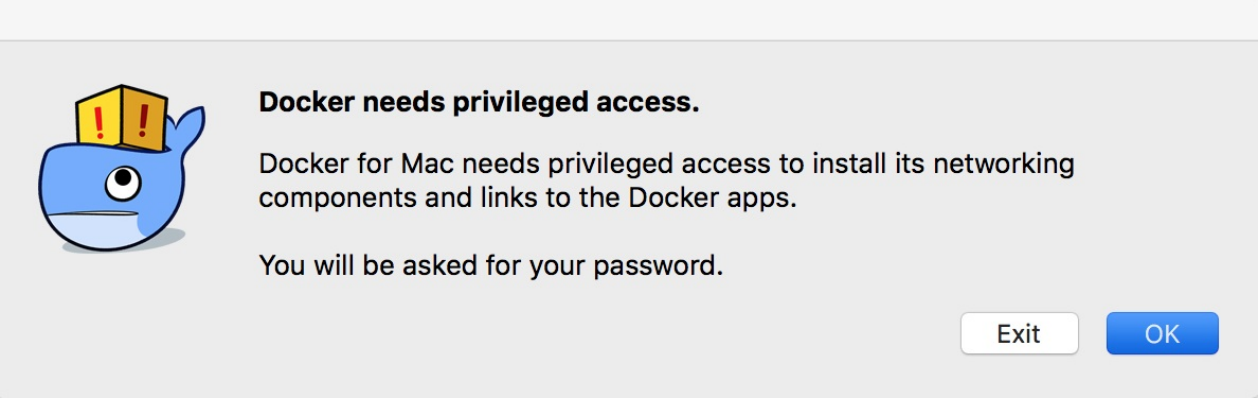
Você pode instalar via brew cask com o comando abaixo:

```
brew cask install docker
```

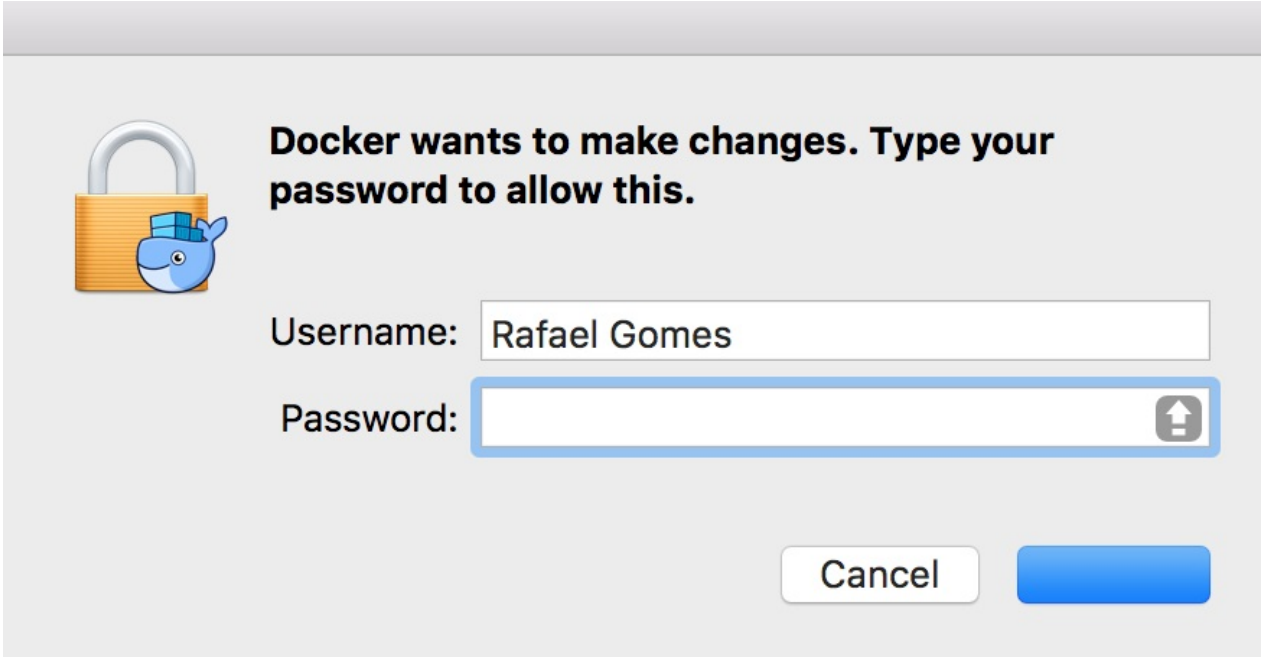
Para efetuar a configuração inicial, você deve executar o aplicativo Docker:



Na tela seguinte selecione a opção **Ok**.



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



Para testar, abra um terminal e execute o comando abaixo:

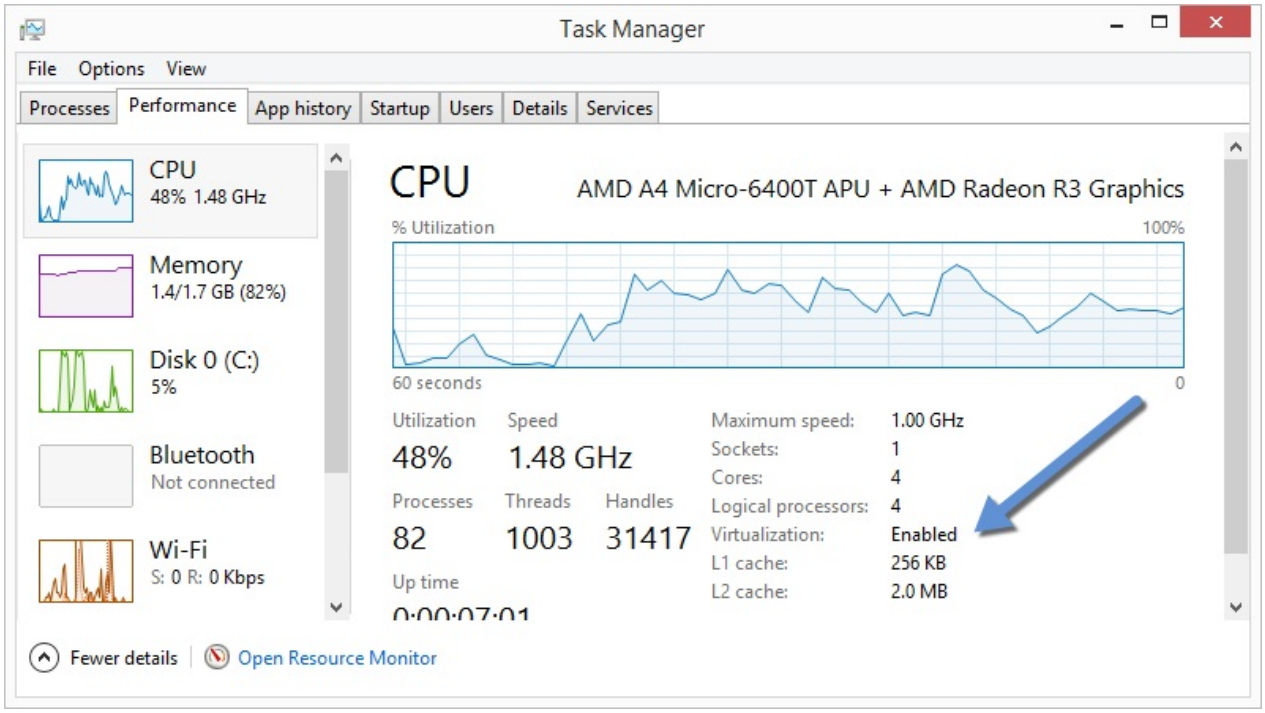
```
docker container run hello-world
```

Instalando no Windows

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

O **Docker Toolbox** funciona apenas em versões 64bit do Windows e soemnte para as versões superiores ao Windows 7.

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

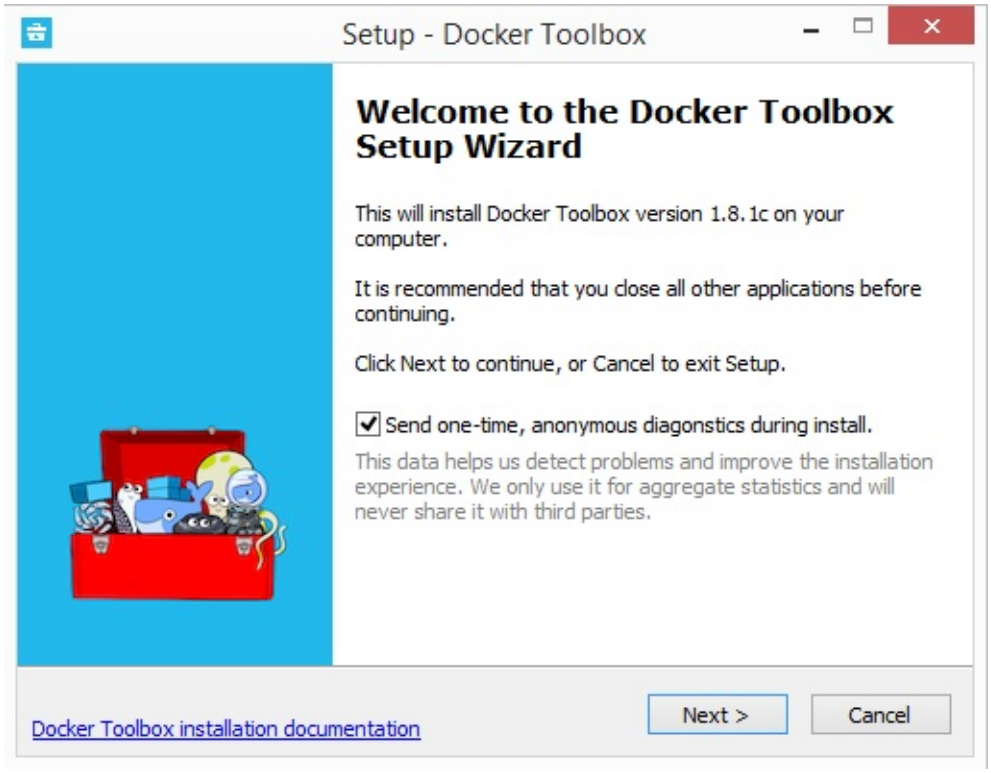


Para verificar o suporte a virtualização do Windows 7, utilize esse link para maiores informações.

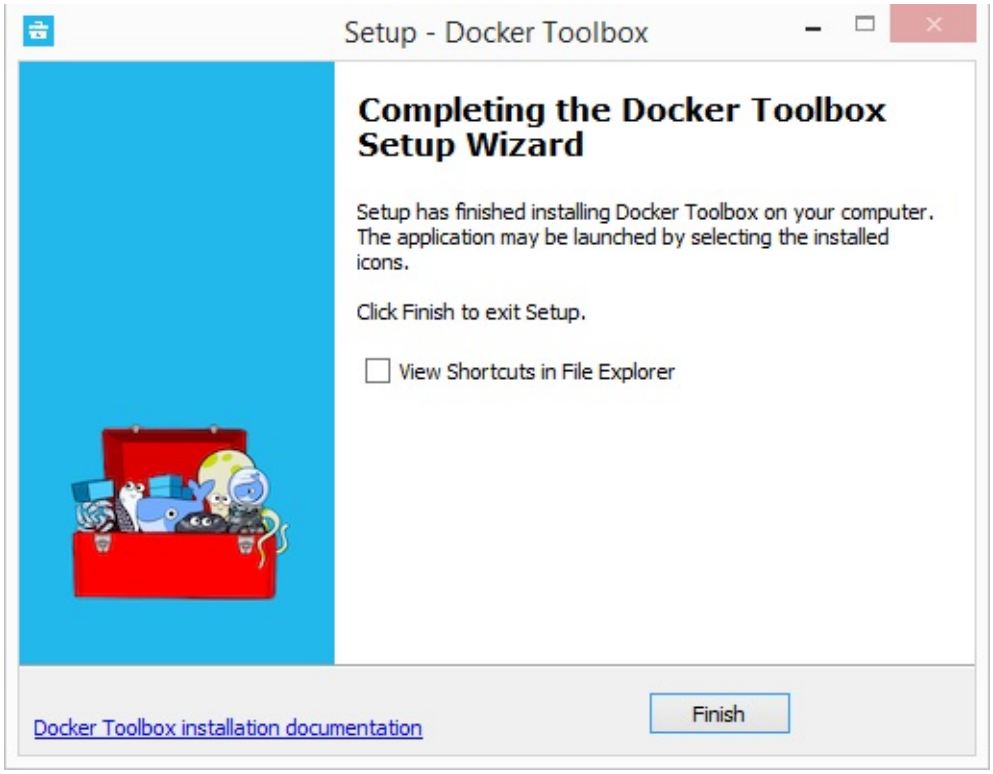
Instalando o Docker Toolbox

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

Após duplo clique no instalador, verá essa tela:



Apenas clique em **Next**.



Por fim, clique em **Finish**.

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

Nesse novo terminal execute o seguinte comando para teste:

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

Comandos básicos

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

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

Executando um container

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

```
docker image list
```

As imagens retornadas estão presentes no seu **Docker host** e não demandam qualquer download da nuvem pública do Docker, a menos que deseje atualizá-la. Para atualizar a imagem basta executar o comando abaixo:

```
docker image pull python
```

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

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

```
docker image inspect python
```

O comando inspect é responsável por informar todos os dados referentes à imagem.

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

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

Os parâmetros mais utilizados na execução do container são:

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

Segue um exemplo simples no seguinte comando:

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

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

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

Mapeamento de volumes

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

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

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

Mapeamento de portas

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

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

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

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

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

Gerenciamento dos recursos

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

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

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

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

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

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

Usaremos como exemplo o peso **512**:

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

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

- O processo com peso **1024** usará 50% do tempo de processamento
- Os dois processos com peso **512** usarão 25% do tempo de processamento, cada.

Verificando a lista de containers

Para visualizar a lista de containers de um determinado **Docker host** utilizamos o comando docker ps.

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

```
docker container list <parâmetros>
```

Os parâmetros mais utilizados na execução do container são:

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

Gerenciamento de containers

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

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

Um exemplo de uso:

```
docker container stop meu_python
```

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

Caso deseje reiniciar o container que foi desligado e não iniciar um novo, basta executar o comando docker start:

```
docker container start meu_python
```

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

Criando sua própria imagem no Docker

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

Qual a diferença entre Imagem e Container?

Traçando um paralelo com o conceito de orientação a objeto, a **imagem** é a classe e o **container** o objeto. A imagem é a abstração da infraestrutura em estado somente leitura, de onde será instanciado o container.

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

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

Anatomia da imagem

As imagens podem ser oficiais ou não oficiais.

Imagens oficiais e não oficiais

As imagens oficiais do docker são aquelas sem usuários em seus nomes. A imagem **“ubuntu:16.04”** é oficial, por outro lado, a imagem “nuagebec/ubuntu” não é oficial. Essa segunda imagem é mantida pelo usuário nuagebec, que mantém outras imagens não oficiais.

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

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

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

Nome da imagem

O nome de uma imagem oficial é composto por duas partes. A primeira, a documentação chama de **“repositório”** e, a segunda, é chamada de **“tag”**. No caso da imagem **“ubuntu:14.04”**, **ubuntu** é o repositório e **14.04** é a tag.

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

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

Execute o comando abaixo para visualizar todas as imagens que se encontram localmente na sua estação, nesse momento:

```
docker image list
```

Como criar imagens

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

Criando imagens com commit

É possível criar imagens executando o comando commit, relacionado a um container. Esse comando usa o status atual do container escolhido e cria a imagem com base nele.

Vamos ao exemplo. Primeiro criamos um container qualquer:

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

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

```
apt-get update
apt-get install nginx -y
exit
```

Paramos o container com o comando abaixo:

```
docker container stop containercriado
```

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

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

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

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

```
docker image list
```

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

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

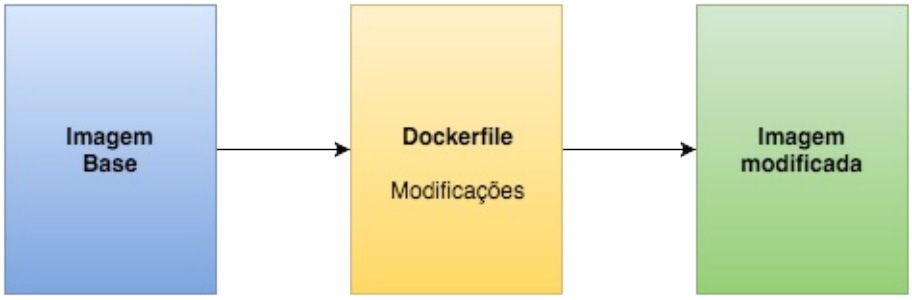
Se quiser validar, pode executar o mesmo comando na imagem oficial do ubuntu:

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

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

Criando imagens com Dockerfile

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



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

Voltemos ao exemplo da instalação do nginx no ubuntu 16.04.

Primeiro crie um arquivo qualquer para um teste futuro:

```
touch arquivo_teste
```

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

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

No arquivo acima, utilizamos quatro instruções:

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

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

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

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

Após construir seu Dockerfile basta executar o comando abaixo:

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

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

A ordem importa

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

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

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

Um exemplo para deixar mais claro:

```
FROM ubuntu:16.04
RUN apt-get update
RUN apt-get install nginx
RUN apt-get install php5
COPY arquivo_teste /tmp/arquivo_teste
CMD bash
```

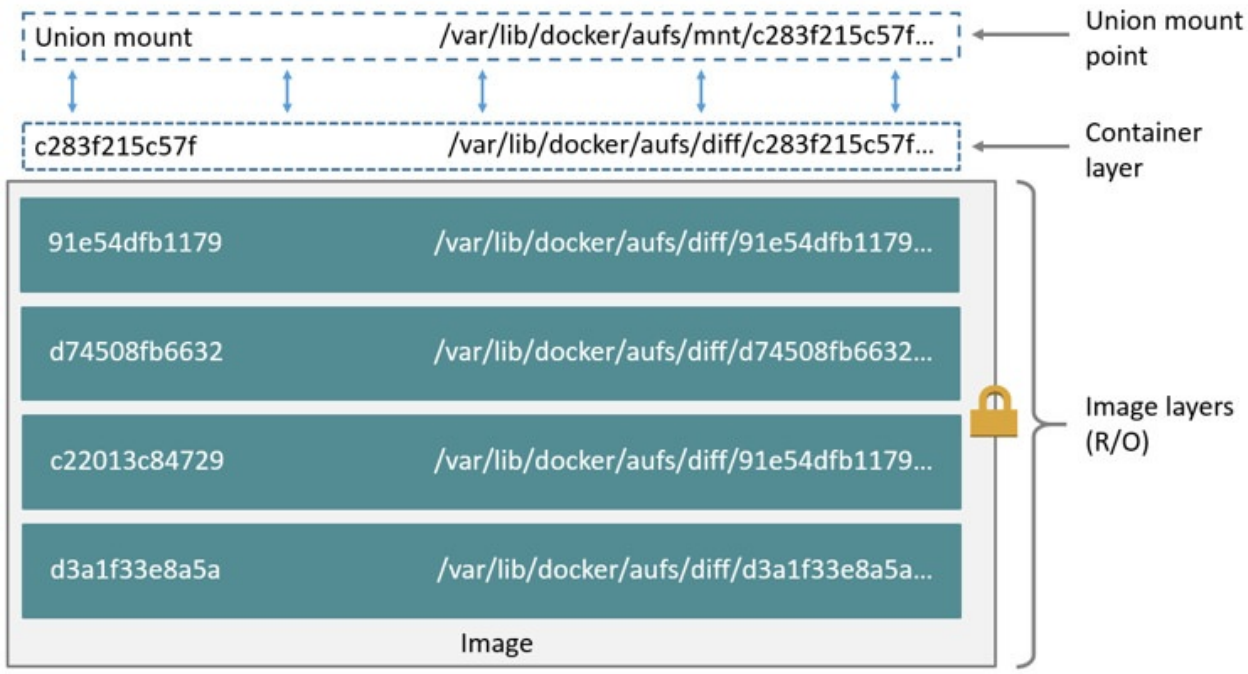
Caso modifiquemos a terceira linha do arquivo e, ao invés de instalar o nginx, mudarmos para apache2, a instrução que faz o update no apt não será executada novamente, e sim a instalação do apache2, pois acabou de entrar no arquivo, assim como o php5 e a cópia do arquivo, pois todos eles são subsequentes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

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

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 de armazenamento do Docker. Faremos aqui com o AUFS, que foi o primeiro e ainda é padrão em boa parte das instalações do Docker.



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

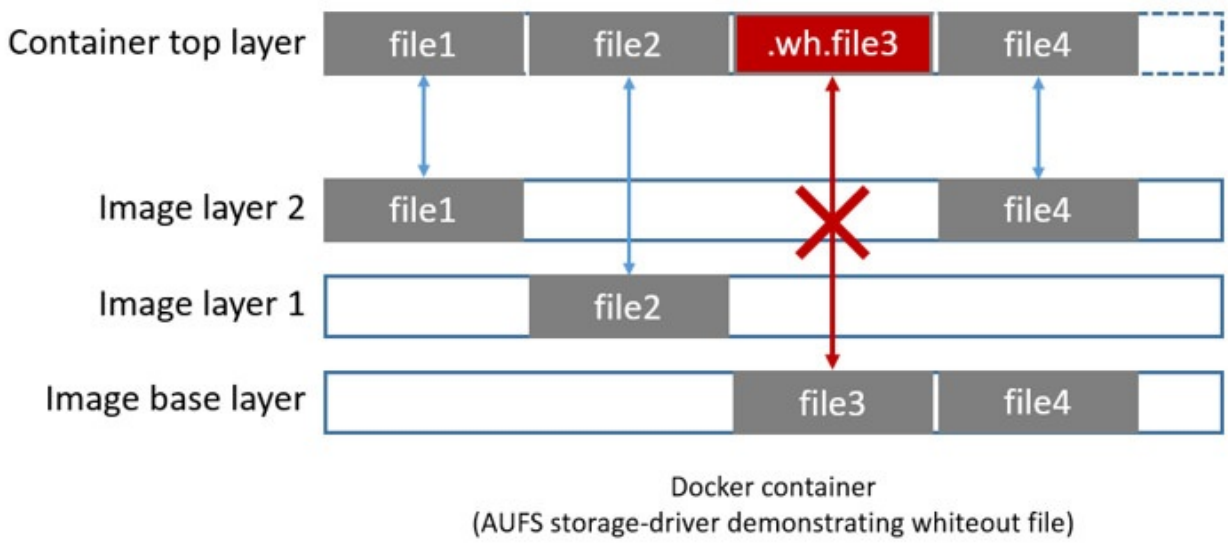
Backend de armazenamento é a parte da solução do Docker que cuida do gerenciamento dos dados. No Docker temos várias possibilidades de backend de armazenamento, mas nesse texto falaremos apenas do que implementa o AUFS.

AUFS é um unification filesystem. É responsável por gerenciar múltiplos diretórios, empilhá-los uns sobre os outros e fornecer uma única e unificada visão, como se todos juntos fossem apenas um diretório.

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

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

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



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

Problema com performance

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

Volume como solução para performance

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

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

Usando volumes

Mapeamento de pasta específica do host

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

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

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

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

Mapeamento via container de dados

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

Segue um exemplo de uso do modelo de mapeamento:

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

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

Para consumir esse volume do container basta utilizar o comando:

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

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

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

Mapeamento de volumes

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

Segue um exemplo de uso do modelo de mapeamento:

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

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

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

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

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

Entendendo a rede no Docker

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

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

Redes padrões do Docker

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

docker network ls

O retorno será:

Bridge

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

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

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

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

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

```
docker container run -d --name bd -e MYSQL_ROOT_PASSWORD=minhasenha mysql

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

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

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

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

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

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

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

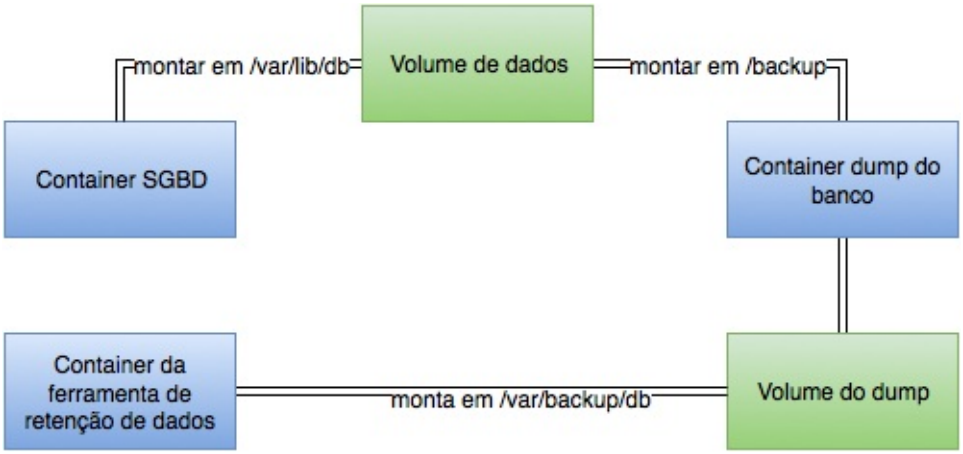
Os containers configurados para essa rede terão a possibilidade de tráfego externo utilizando as rotas das redes IP definidas no docker host. Caso o docker host tenha acesso a internet, automaticamente, os containers em questão também terão.

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

None

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

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



Em caso de dúvida sobre utilização de volumes no docker visite esse artigo e entenda mais sobre armazenamento do docker.

Host

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

Redes definidas pelo usuário

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

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

Bridge

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

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

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

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

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

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

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

Agora verificamos a rede:

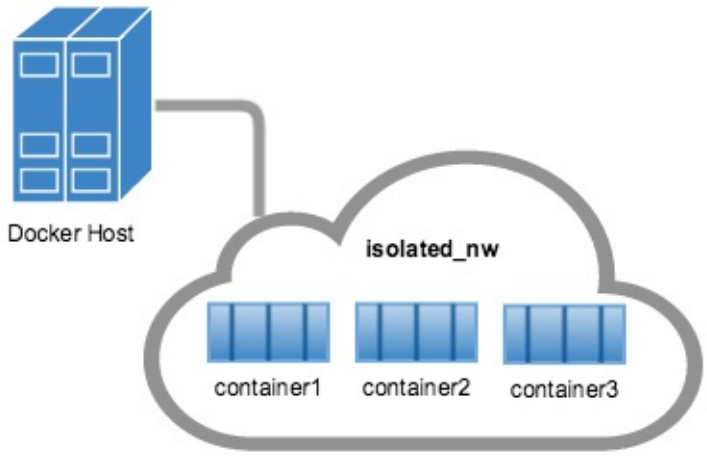
```
docker network list
```

O resultado deve ser:

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

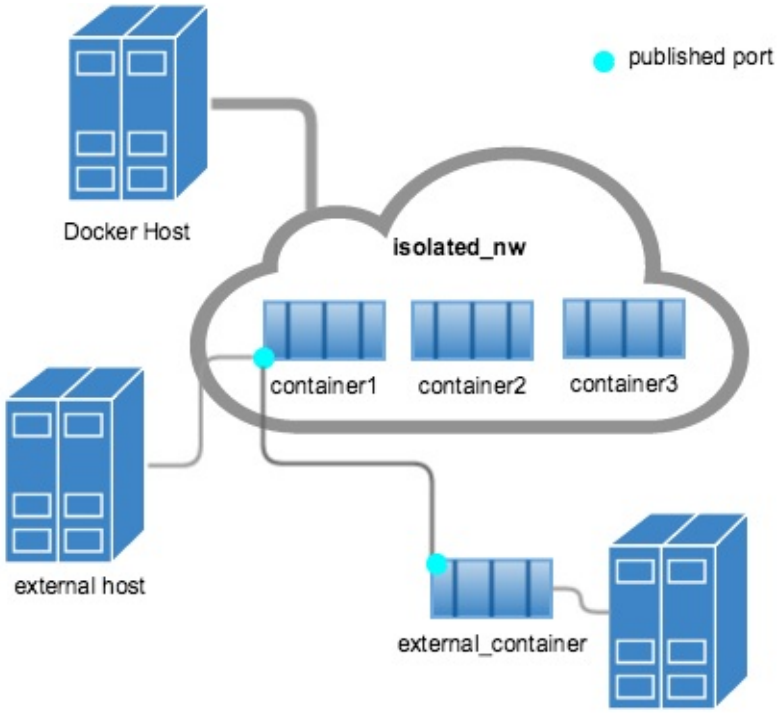
Agora iniciamos um container na rede isolated_nw:

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



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

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



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

```
docker network inspect isolated_nw
```

O resultado deve ser:

```
[
    {
        "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” é possível verificar quais containers fazem parte dessa rede. Todos os containers que estiverem na mesma rede poderão se comunicar utilizando apenas seus respectivos nomes. Como podemos ver no exemplo acima, caso um container novo acesse a rede isolated_nw, ele poderá acessar o container amazing_noyce utilizando apenas seu nome.

Overlay

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

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

Utilizando redes no docker compose

O assunto merece um artigo exclusivo. Então, apenas informaremos um link interessante para referência sobre o assunto.

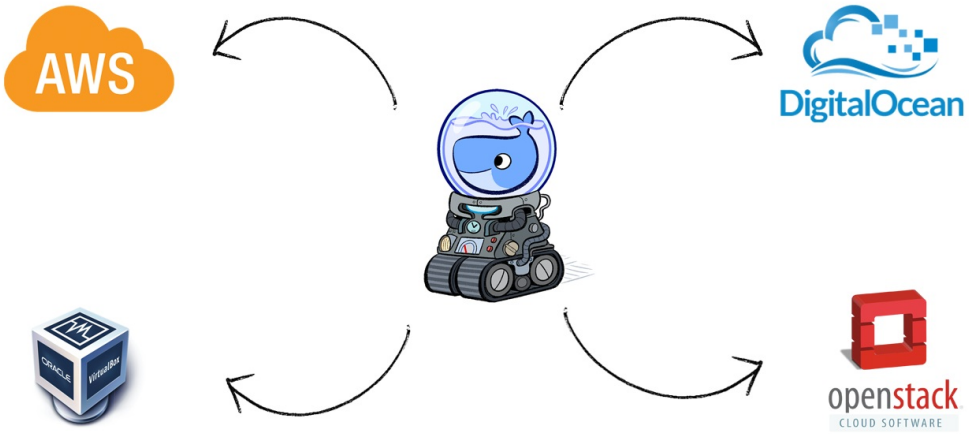
Conclusão

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

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

Utilizando Docker em múltiplos ambientes

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

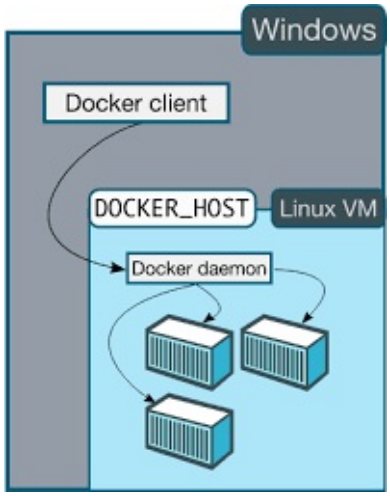


Docker machine é a ferramenta usada para essa gerência distribuída, permite a instalação e gerência de docker hosts de forma fácil e direta.

Essa ferramenta é muito usada por usuários de sistema operacional “não linux”, como demonstraremos ainda, mas sua função não limita-se a esse fim, pois também é bastante usada para provisionar e gerenciar infraestrutura Docker na nuvem, tal como AWS, Digital Ocean e Openstack.

Como funciona

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



Como demonstra a imagem acima, a utilização do Docker divide-se em dois serviços: o que roda em modo daemon, em background, chamado de **Docker Host**, responsável pela viabilização dos containers no kernel Linux; e, o cliente, que chamaremos de **Docker client**, responsável por receber comandos do usuário e traduzir em gerência do **Docker Host**.

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

O **Docker machine** possibilita utilizar diversos ambientes distintos apenas modificando a configuração do cliente para o **Docker host** desejado: basicamente modificar algumas variáveis de ambiente. Segue exemplo:

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

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

Criando ambiente

O Docker machine serve principalmente para criar ambientes, futuramente geridos por ele na troca automatizada de contexto de configuração, através da mudança de variáveis de ambiente, como explicado 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.

Máquina virtual

Para esse exemplo, usaremos o driver mais utilizado, o virtualbox, ou seja, precisamos de um virtualbox 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=

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

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

Como teste utilizamos o seguinte comando:

```
docker-machine create --driver=virtualbox --virtualbox-disk-size 30000 teste-virtualbox
```

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

Para validar se o processo aconteceu como esperado, basta utilizar o seguinte comando:

```
docker-machine ls
```

O comando acima é responsável por listar todos os ambientes que podem ser usados a partir da estação cliente.

Pra mudar de cliente basta utilizar o comando:

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

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

```
docker-machine ls
```

Inicie um container de teste pra testar o novo ambiente

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

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

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

Caso deseje desligar o ambiente, utilize o comando:

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

Caso deseje iniciar o ambiente, utilize o comando:

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

Caso deseje remover o ambiente, utilize o comando:

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

Tratamento de problema conhecido: caso esteja utilizando Docker-machine no MacOS e por algum motivo a estação hiberne quando o ambiente virtualbox tenha iniciado, é possível que, no retorno da hibernação, o Docker host apresente problemas na comunicação com a internet. Orientamos a, sempre que passar por problemas de conectividade no Docker host com driver virtualbox, desligue o ambiente e reinicie como medida de contorno.

Nuvem

Para esse exemplo usamos o driver da nuvem mais utilizada, AWS. Para tanto, precisamos de uma conta na AWS para que esse driver 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.

Quando criamos um ambiente utilizando o comando **docker-machine create**, o mesmo é traduzido para AWS na criação uma instância EC2 e, em seguida é instalado todos os softwares necessários, automaticamente, no novo ambiente.

Os parâmetros mais utilizados na criação desse ambiente são:

Parâmetro	Explicação
--amazonec2-region	Informa qual região da AWS é 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. Precisa ter sido criada previamente.
--amazonec2-security-group	Informa qual security group é utilizado nessa instância EC2. Precisa ter sido criado previamente
--amazonec2-use-private-address	Será criada uma interface com IP privado, pois por default, só especifica uma interface com IP público
--amazonec2-vpc-id	Informa qual o ID do VPC desejado para essa instância EC2. Precisa ter sido criado previamente.

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

```
docker-machine create --driver amazonec2 --amazonec2-zone a --amazonec2-subnet-id subnet-5d3dc191 --amazonec2-securit
y-group docker-host --amazonec2-use-private-address --amazonec2-vpc-id vpc-c1d33dc7 teste-aws
```

Após executar o comando, basta esperar finalizar, é normal demorar um pouco.

Para testar o sucesso da ação, execute o comando abaixo:

```
docker-machine ls
```

Verifique se o ambiente chamado teste-aws existe na lista, caso positivo, utilize o comando abaixo para mudar o ambiente:

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

Inicie um container de teste pra verificar o novo ambiente

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

Caso deseje desligar o ambiente, utilize o comando:

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

Caso deseje iniciar o ambiente, utilize o comando:

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

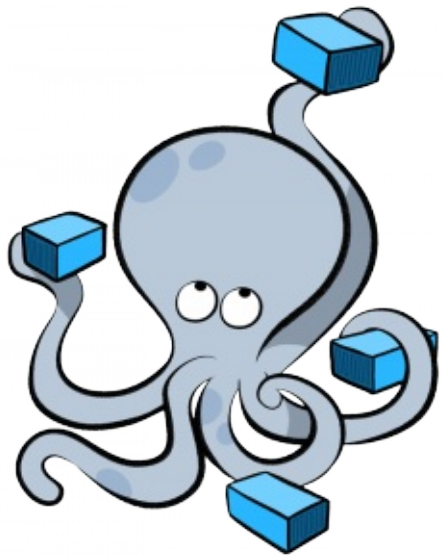
Caso deseje remover o ambiente, utilize o comando:

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

Após removido do local, automaticamente removerá a instância EC2, 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.

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. Esse arquivo por padrão tem como nome docker-compose.yml.

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:

```
version: '2'
services:
  web:
    build: .
      context: ./dir
      dockerfile: Dockerfile-alternate
      args:
        versao: 1
    ports:
      - "5000:5000"
  redis:
    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.

```
version: '2'
```

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

```
version: '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.

```
version: '2'
services:
  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 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. Ele também abre um novo bloco de código para parametrizar o funcionamento dessa construção da imagem.

```
version: '2'
services:
  web:
    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.

```
    build: .
      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" 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**.

```
    build: .
      context: ./dir
      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" 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.

```
    build: .
      context: ./dir
      dockerfile: Dockerfile-alternate
      args:
        versao: 1
```

Voltando dois níveis de indentação (quatro espaços a menos em relação a linha anterior) temos a definição **ports**, que seria o equivalente ao parâmetro "-p" do comando docker container run. 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*.

```
  web:
    build: .
    ...
    ports:
      - "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**.

```
  redis:
    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.

Executando o docker compose

Após entender e criar seu próprio **arquivo de definição** precisamos saber como gerenciá-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.

Como usar Docker sem GNU/Linux

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

Esse texto é para pessoas que já sabem sobre Docker, mas ainda não sabem como o Docker pode ser utilizado a partir de uma estação “não linux”.

Como já dissemos, o Docker utiliza recursos específicos do sistema operacional hospedeiro. Hoje temos suporte para os sistemas operacionais Windows e GNU/Linux. Significa que não é possível iniciar containers Docker em estação MacOS, por exemplo.

Mas não se preocupe, caso você não utilize GNU/Linux, ou Windows, como sistema operacional, ainda é possível fazer uso dessa tecnologia, sem, necessariamente, executá-la em seu computador.

Vale salientar que containers e imagens Docker criados no Windows, não funcionarão em um GNU/Linux, por conta da dependência do sistema operacional mencionado anteriormente.

É possível utilizar o Docker no MacOS e Windows a partir de duas maneiras:

- Toolbox
- Docker For Mac/Windows

Por ser mais complexa, e assim demandar de um contexto maior, trataremos nesse capítulo apenas sobre a instalação e configuração do Docker Toolbox. Essa solução, é na verdade, uma abstração para instalação de todo ambiente necessário para uso do Docker a partir de uma estação MacOS ou Windows.

A instalação é simples: tanto no Windows, como no MacOS, basta baixar o instalador correspondente nesse site e executá-lo seguindo os passos descritos nas telas.

Os softwares instalados na estação - MacOS ou Windows - a partir do pacote Docker Toolbox são:

- Virtualbox
- Docker machine
- Docker client
- Docker compose
- Kitematic

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, abordaremos apenas máquina virtual com virtualbox.

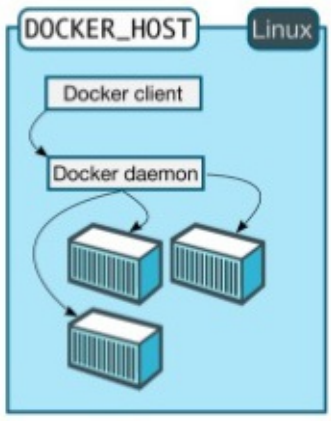
Após instalar o Docker Toolbox é muito simples criar um ambiente Docker com máquina virtual usando o Docker Machine.

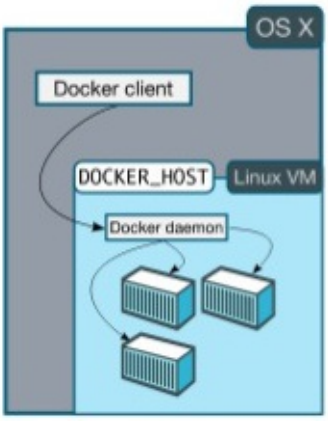
Primeiro verificamos se não existem máquinas virtuais com Docker instaladas em seu ambiente:

```
docker-machine ls
```

O comando acima mostra apenas ambientes criados e mantidos por seu Docker Machine. É possível que, após instalar o Docker Toolbox, você não encontre máquina alguma criada. Nesses casos, utilizamos o comando abaixo para criar a máquina:

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





O comando cria um ambiente denominado “default”. Na verdade é uma máquina virtual (“Linux VM” que aparece na imagem) criada no virtualbox. Com o comando abaixo é possível visualizar a máquina criada:

```
docker-machine ls
```

O retorno deve ser 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:

```
docker-machine stop default
```

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

```
docker-machine start default
```

O comando “start” é responsável apenas por iniciar a máquina. É necessário fazer com que os aplicativos de controle do Docker, instalados na estação, possam se conectar à 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 facilita o trabalho de aplicar todas as variáveis corretamente:

```
docker-machine env default
```

O resultado desse comando no MacOS é:

```
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 vemos, informa o que pode ser feito para configurar todas as variáveis. Você pode copiar as quatros primeiras linhas, que começam com “export”, e colar no terminal ou, pegar apenas a última linha sem o “#” do início e executar na linha de comando:

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

Agora os aplicativos de controle (Docker e Docker-Compose) estão aptos a utilizar o Docker Host a partir da conexão feita no serviço do IP 192.168.99.100 - máquina criada com o comando “docker-machine create” mencionados anteriormente.

Para testar, listamos os containers em execução nesse Docker Host com o comando:

```
docker ps
```

Executado na linha de comando do MacOS ou Windows, esse cliente do Docker se conecta à máquina virtual que, aqui chamamos de “Linux VM”, e solicita a lista de containers em execução no Docker Host remoto.

Iniciamos um container com o comando abaixo:

```
docker container run -itd alpine sh
```

Agora, verificamos novamente, a lista de containers em execução:

```
docker ps
```

Podemos ver que o container criado a partir da imagem “alpine” está em execução. Vale salientar que esse processo é executado no Docker Host, na máquina criada dentro do virtualbox que, nesse exemplo, tem o ip 192.168.99.100.

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

```
docker-machine ip
```

Caso o container exponha alguma porta para o Docker Host, seja via parâmetro “-p” do comando “docker container run -p porta_host:porta_container” ou via parâmetro “ports” do docker-compose.yml, vale lembrar que o IP para acessar o serviço exposto é o endereço IP do Docker Host que, no exemplo, é “192.168.99.100”.

Nesse momento, você deve estar se perguntando: como é possível mapear uma pasta da estação “não-linux” para dentro de um container? Aqui entra um novo artífício do Docker para contornar esse problema.

Toda máquina criada com o driver “virtualbox”, automaticamente, cria um mapeamento do tipo “pastas compartilhadas do virtualbox” da pasta de usuários para a raiz do Docker Host.

Para visualizar esse mapeamento, acessamos 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 é um sistema de arquivo usado pelo virtualbox para montar volumes compartilhados da estação usada para instalar o virtualbox. Ou seja, utilizando o 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 pasta /Users/SeuUsuario do MacOS, será acessível na pasta /Users/SeuUsuario da máquina GNU/Linux que atua como Docker Host no exemplo apresentado. Caso efetue a montagem da pasta /Users/SeuUsuario/MeuCodigo para dentro do container, o dado a ser montado é o mesmo da estação e nada precisa ser feito para replicar esse código para dentro do Docker Host.

Vamos testar. Crie um arquivo dentro da pasta de usuário:

```
touch teste
```

Iniciamos um container e mapeamos a pasta atual dentro dele:

```
docker container 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.

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

```
docker container exec teste ls /tmp/teste
```

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

Agora acesse o Docker Host com o comando abaixo, e verifique se o arquivo teste se encontra na pasta de usuário:

```
docker-machine ssh default
```

Tudo pode ser feito automaticamente? Claro que sim!

Agora que já sabe como fazer manualmente, se precisar instalar o Docker Toolbox em uma máquina nova e não lembrar os comandos para criar a nova máquina ou, simplesmente como aprontar o ambiente para uso, basta executar o programa “Docker Quickstart Terminal”. Ele fará o trabalho automaticamente. Caso não exista máquina criada, ele cria uma chamada “default”. Caso a máquina já tenha sido criada, automaticamente configura suas variáveis de ambiente e deixa apto para utilizar o Docker Host remoto a partir dos aplicativos de controle (Docker e Docker-Compose).

Transformando sua aplicação em container

Evoluímos continuamente para entregar aplicações cada vez melhores, em menor tempo, replicáveis e escaláveis. Entretanto, os esforços e aprendizados para atingir esse nível de maturidade muitas vezes não são simples de serem alcançados.

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 depender apenas 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 escreveram o 12Factor app, baseado na 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 (stateless) 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 com imagens Docker baseados no 12factor app. A ideia é demonstrar as melhores práticas para realizar a criação de infraestrutura para suportar, empacotar e disponibilizar a aplicação com alto nível de maturidade e agilidade.

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

Como aplicação exemplo, teremos um serviço HTTP, escrito em python, que exibe quantas vezes foi acessada. Essa informação é armazenada através de contador numa instância Redis.

Agora vamos às boas práticas!

Base de código

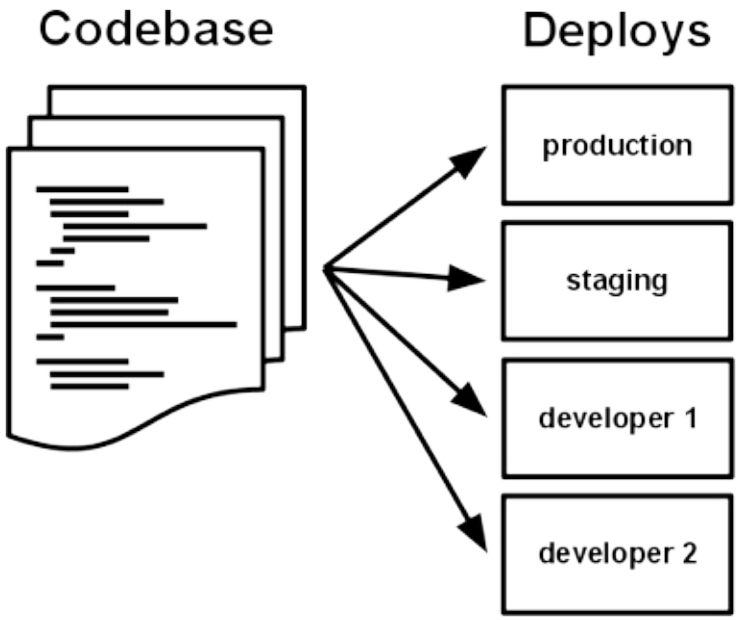
Com o objetivo de facilitar o controle das mudanças de código, viabilizando a rastreabilidade das alterações, essa boa prática indica que cada aplicação deve ter apenas uma base de código e, partindo dessa, ser implantada em distintos ambientes. Vale salientar que essa prática também é parte das práticas de Continuous Integration (CI). Tradicionalmente, a maioria dos sistemas de integração contínua tem, como ponto de partida, uma base de código que é construída e, posteriormente, implantada em desenvolvimento, teste e produção.

Para essa explicação, usamos o sistema de controle de versão Git e o serviço de hospedagem Github. Criamos e disponibilizamos um repositório de exemplo.

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

O Docker tem possibilidade de utilizar variável de ambiente para parametrização da infraestrutura. Sendo assim, a mesma aplicação terá comportamento distinto com base no valor das variáveis de ambiente.

Aqui usamos o Docker Compose para realizar a composição de diferentes serviços pertinentes para a aplicação em tempo de execução. Desse modo, devemos definir a configuração desses distintos serviços e a forma como se comunicam.



Posteriormente, mais precisamente na terceira boa prática, chamada **Configuração**, desse compêndio de sugestões, trataremos com mais detalhes sobre parametrização da aplicação. Por ora, apenas aplicamos 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, criamos o arquivo docker-compose.yml:

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

Podemos notar que o serviço "redis" é utilizado a partir da imagem oficial "redis", sem modificação. E o serviço web é gerado a partir da construção de uma imagem Docker.

Para a contrução da imagem Docker do serviço web, criamos o seguinte Dockerfile, usando como base a imagem oficial do python 2.7:

```
FROM python:2.7
COPY requirements.txt requirements.txt
RUN pip install -r requirements.txt
ADD . /code
WORKDIR /code
CMD python app.py
```

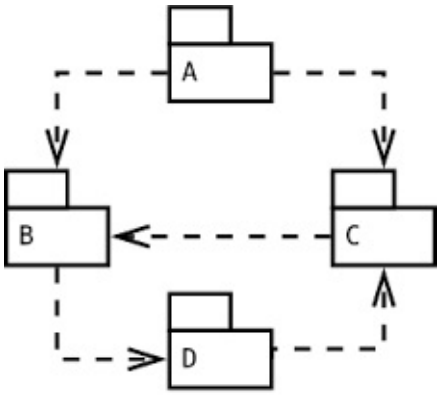
De posse de todos os arquivos na mesma pasta, iniciamos o ambiente com o seguinte comando:

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

Como podemos perceber no exemplo desse capítulo, a variável de ambiente `ENV_APP` define qual volume é usado para persistir os dados que são enviados pela aplicação web. Ou seja, com base na mudança dessa opção, teremos o serviço rodando com comportamento diferente, mas sempre a partir do mesmo código. Dessa forma, segue o conceito da primeira boa prática.

Dependência

Seguindo a lista do modelo 12factor, logo após a base de código que tratamos nesse artigo, temos a **“Dependência”** como segunda boa prática.



Essa boa prática sugere a declaração de todas as dependências necessárias para executar o código. Você não deve assumir que algum componente já está previamente instalado no ativo responsável por hospedar a 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 dá suporte à aplicação.

Automatizar o processo de instalação de dependência é o grande segredo do sucesso para atender essa boa prática. Caso a instalação da infraestrutura não seja, suficientemente, automática para viabilizar a inicialização sem erros, o atendimento da boa prática fica prejudicado.

Esses procedimentos, automatizados, colaboram com a manutenção da integridade do processo, pois, o nome dos pacotes de dependências e respectivas versões, estão especificados no arquivo localizado no mesmo repositório do código que, por sua vez, é rastreado em um sistema de controle de versão. Com isso, podemos concluir que nada é modificado sem o devido registro.

O Docker se encaixa perfeitamente na boa prática. É possível entregar um perfil mínimo de infraestrutura para a aplicação. Por sua vez, a declaração explícita das dependências, para que a aplicação funcione no ambiente, faz-se necessária.

A aplicação do exemplo, escrita em Python, como verificamos em parte do código abaixo, necessita de duas bibliotecas para funcionar corretamente:

```
from flask import Flask
from redis import Redis
```

Essas duas dependências estão especificadas no arquivo requirements.txt e, esse arquivo é usado como 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, junto ao arquivo requirements.txt, na criação da imagem, como demonstrado no Dockerfile da boa prática anterior (codebase):

```
FROM python:2.7
ADD requirements.txt requirements.txt
RUN pip install -r requirements.txt
ADD . /code
WORKDIR /code
CMD python app.py
```

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

```
flask==0.11.1
redis==2.10.5
```

É importante salientar a necessidade de especificar as versões de cada dependência, pois, como no modelo de contêiner, as imagens podem ser construídas a qualquer momento. É importante saber qual versão específica a aplicação precisa. Caso contrário, podemos encontrar problemas com compatibilidade se uma das dependências atualizar e não permaneer compatível com a composição completa das outras dependências e a aplicação que a utiliza.

Para acessar o código descrito aqui, baixe o repositório e acesse a pasta **“factor2“**.

Outro resultado positivo do uso da boa prática é a simplificação da utilização do código por outro desenvolvedor. Um novo programador pode verificar, nos arquivos de dependências, quais os pré-requisitos para a aplicação executar, assim como executar o ambiente sem necessidade de seguir a extensa documentação que, raramente, é atualizada.

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

Configurações

Seguindo a lista do modelo 12factor, **“Configurações”** é terceira boa prática.

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

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

O objetivo da boa prática é viabilizar a configuração da aplicação sem a necessidade de modificar o código. Já que, o comportamento da aplicação varia de acordo com o ambiente onde é executada, as configurações devem considerar o ambiente.

Seguem alguns exemplo:

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

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

Como demonstramos na boa prática codebase, usamos uma variável de ambiente para modificar o volume que usaremos no redis. De certa forma, já estamos seguindo a boa prática, mas podemos ir além e mudarmos não somente o comportamento da infraestrutura, mas sim algo inerente ao código em si.

Segue a aplicação modificada:

```
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 da prática, basta clonar esse repositório e acessar a pasta **“factor3“**.

Como podemos notar, adicionamos alguns parâmetros na configuração do endereço usado para iniciar a aplicação web que será parametrizada com base no valor da variável de ambiente **“HOST_RUN”**. E, a possibilidade de efetuar, ou não, o debug da aplicação com a variável de ambiente **“DEBUG“**.

Vale salientar: nesse caso a variável de ambiente precisa ser passada para o contêiner, não basta ter a variável no Docker Host. É preciso enviá-la para o contêiner usando o parâmetro “-e”, caso utilize o comando “docker container 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=${DEBUG}
  redis:
    image: redis:3.2.1
    volumes:
     - dados:/data
    labels:
     - 'app.environment=${ENV_APP}'
volumes:
  dados:
    external: false
```

Para executar o Docker-Compose, deveríamos fazer da seguinte maneira:

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

No comando acima, usamos as variáveis de ambiente **“HOST_RUN”** e **“DEBUG”** do Docker Host para enviar às variáveis de ambiente com os mesmos nomes dentro do contêiner que, por sua vez, é consumido pelo código Python. Caso não haja parâmetros, o contêiner assume os valores padrões estipulados no código.

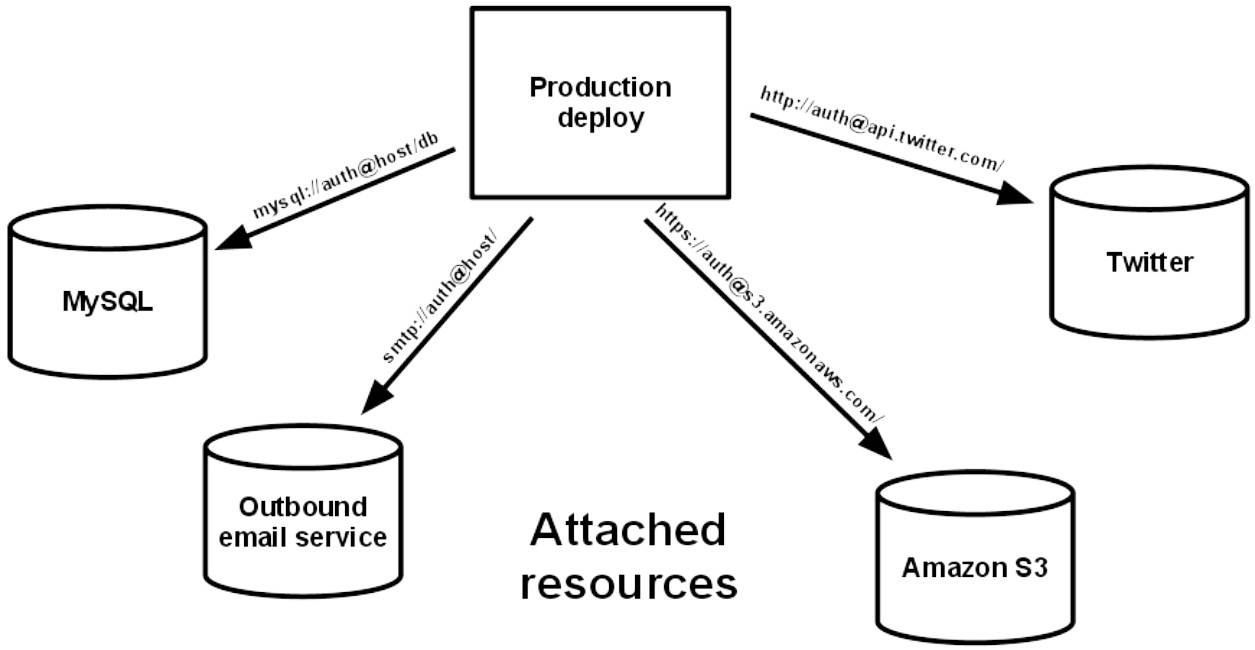
Essa boa prática é seguida com ajuda do Docker, pois o código é o mesmo e, a configuração, um anexo da solução que pode ser parametrizada de maneira distinta com base no que for configurado nas variáveis de ambiente.

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

Serviços de Apoio

Seguindo a lista do modelo 12factor, 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 o código seja demasiadamente dependente de determinada infraestrutura, a boa prática indica que você, no momento da escrita do software, não faça distinção entre serviço interno ou externo. Ou seja, o aplicativo deve estar pronto para receber parâmetros que configurem o serviço corretamente e, assim, possibilitem o consumo de aplicações necessárias à solução proposta.

A aplicação exemplo sofreu modificações para suportar a boa prática:

```
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 pode perceber no código acima, a aplicação agora recebe variáveis de ambiente para configurar o hostname e a porta do serviço Redis. Nesse caso, é possível configurar o host e a porta da Redis que deseja conectar. E isso pode e deve ser especificado no docker-compose.yml que também passou por mudança para se adequar a nova boa prática:

```
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}'
volumes:
  dados:
    external: false
```

Como observamos nos códigos já tratados, a vantagem da boa prática passa pela possibilidade de mudança de comportamento sem mudança do código. Mais uma vez é possível viabilizar que, o mesmo código construído em um momento, possa ser reutilizado de forma semelhante, tanto no notebook do desenvolvedor como no servidor de produção.

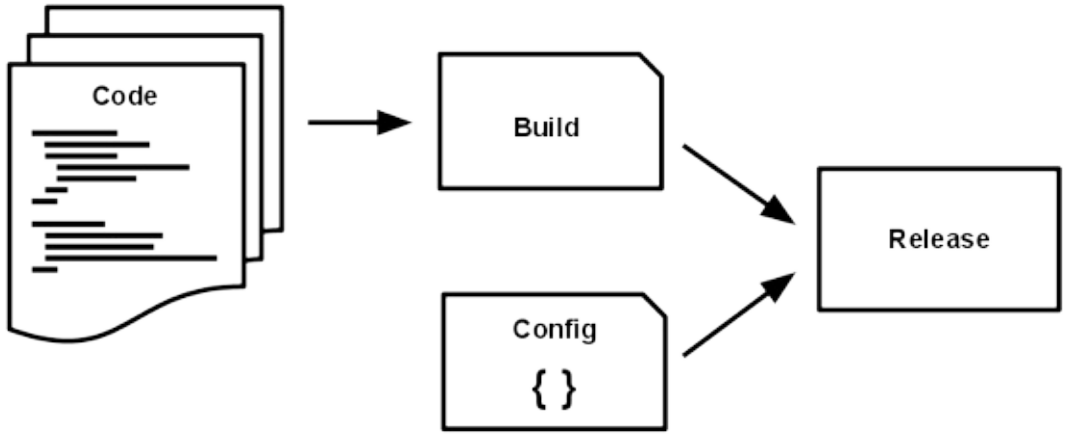
Fique atento para armazenamento de segredos dentro do docker-compose.yml, pois esse arquivo é enviado para o repositório de controle de versão e é importante pensar outra estratégia de manutenção de segredos.

Uma estratégia possível é a manutenção de variáveis de ambiente no Docker Host. Dessa forma, você precisa usar variáveis do tipo **${variavel}** dentro do docker-compose.yml para repassar a configuração ou, utilizar outro recurso mais avançado de gerenciamento de segredos.

Construa, lance, execute

O próximo item da lista do modelo 12factor, “Construa, lance, execute” é a quinta boa prática..

No processo de automatização de infraestrutura de implantação de software precisamos cuidado para que o comportamento do processo esteja dentro das expectativas e erros humanos causem baixo impacto no processo completo do desenvolvimento, do lançamento à produção.



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

- **Construa** - converter código do repositório em pacote executável. Nesse processo se obtém as dependências, compila-se o binário e os ativos do código.
- **Lance** - pacote produzido na fase **construir** é combinado com a configuração. O resultado é o ambiente completo, configurado e pronto para ser colocado em **execução**.
- **Execute** (também conhecido como “runtime”) - inicia a **execução** do **lançamento** (aplicação + configuração daquele ambiente), com base nas configurações específicas do ambiente requerido.

A boa prática indica que a 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 de **Construa**. Mudanças da configuração não necessitam nova **construção**, sendo necessário, apenas, as etapas de **lançar** e **executar**.

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

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

Os artefatos produzidos devem ter um identificador de **lançamento** único. Pode ser o timestamp (como 2011-04-06-20:32:17) ou um número incremental (como v100). Com o artefato único é possível garantir o uso de versão antiga, seja para plano de retorno ou, até mesmo, para comparar comportamentos após mudanças no código.

Para atendermos a boa prática precisamos construir a imagem Docker com a aplicação dentro. Ela será nosso artefato.

Teremos um script novo, aqui chamado build.sh, com o seguinte conteúdo:

```
#!/bin/bash

USER="gomex"
TIMESTAMP=$(date "+%Y.%m.%d-%H.%M")

echo "Construindo a imagem ${USER}/app:${TIMESTAMP}"
docker build -t ${USER}/app:${TIMESTAMP} .

echo "Marcando a tag latest também"
docker tag ${USER}/app:${TIMESTAMP} ${USER}/app:latest

echo "Enviando a imagem para nuvem docker"
docker push ${USER}/app:${TIMESTAMP}
docker push ${USER}/app:latest
```

Além de construir a imagem, a envia para o repositório de imagem do Docker.

Lembre-se que, o código acima e os demais da boa prática, estão no repositório na pasta **"factor5"**.

O envio da imagem para o repositório é parte importante da boa prática em questão, pois isola o processo. Caso a imagem não seja enviada para o repositório, permanece apenas no servidor que executou o processo de **construção**, sendo assim, a próxima etapa precisa, necessariamente, ser executada no mesmo servidor, pois tal etapa precisa da imagem disponível.

No modelo proposto, a imagem no repositório central fica disponível para ser baixada no servidor. Caso utilize uma ferramenta de pipeline, é importante - ao invés de utilizar a data para tornar o artefato único - usar variáveis do produto para garantir que a imagem a ser consumida na etapa Executar, seja a mesma construída na etapa Lançar. Exemplo no GoCD: variáveis **GO_PIPELINE_NAME** e **GO_PIPELINE_COUNTER** podem ser usadas em conjunto como garantia.

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 à configuração.

A etapa de **Lançamento** é o arquivo docker-compose.yml em si, pois o mesmo recebe as configurações devidas para o ambiente no qual se deseja colocar a aplicação. Sendo assim, o arquivo docker-compose.yml muda um pouco e deixa de fazer a **construção** da imagem, já que, agora, 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 busque sempre a última imagem **construída** no processo. Mas como já mencionamos, caso utilize alguma ferramenta de entrega contínua (como GoCD, por exemplo) faça uso das variáveis, para garantir a imagem criada na execução específica do pipeline.

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

A etapa de **execução**, basicamente, executa o Docker-Compose com o comando abaixo:

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

Processos

Seguindo a lista do modelo 12factor, temos **“Processos”** como sexta boa prática.

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

A boa prática indica que processos de aplicações 12factor são stateless (não armazenam estado) e share-nothing. Quaisquer dados que precisem persistir devem ser armazenados em serviço de apoio stateful (armazena o estado), normalmente é usado uma base de dados.

O objetivo final dessa prática não faz distinção se a aplicação é executada na máquina do desenvolvedor ou em produção, pois, nesse caso, o que muda é a quantidade de processos iniciados para atender as respectivas demandas. Na máquina do desenvolvedor pode 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 no banco de dados.

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

É importante salientar: ao seguir a prática, uma aplicação não assume que, qualquer item armazenado em cache de memória ou no disco, estará disponível em futura solicitação ou job – com muitos processos de cada tipo rodando, são altas as chances de futura solicitação ser servida por processo diferente, até mesmo em servidor diferente. Mesmo quando, rodando em apenas um processo, um restart (desencadeado pelo deploy de um código, mudança de configuração, ou o ambiente de execução realocando o processo para localização física diferente) geralmente vai acabar com o estado local (memória e sistema de arquivos, por exemplo).

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

A aplicação que estamos trabalhando não guarda dado local e tudo o que precisa é armazenado no Redis. Não precisamos fazer adequação alguma nesse código para seguir a boa prática, como vemos abaixo:

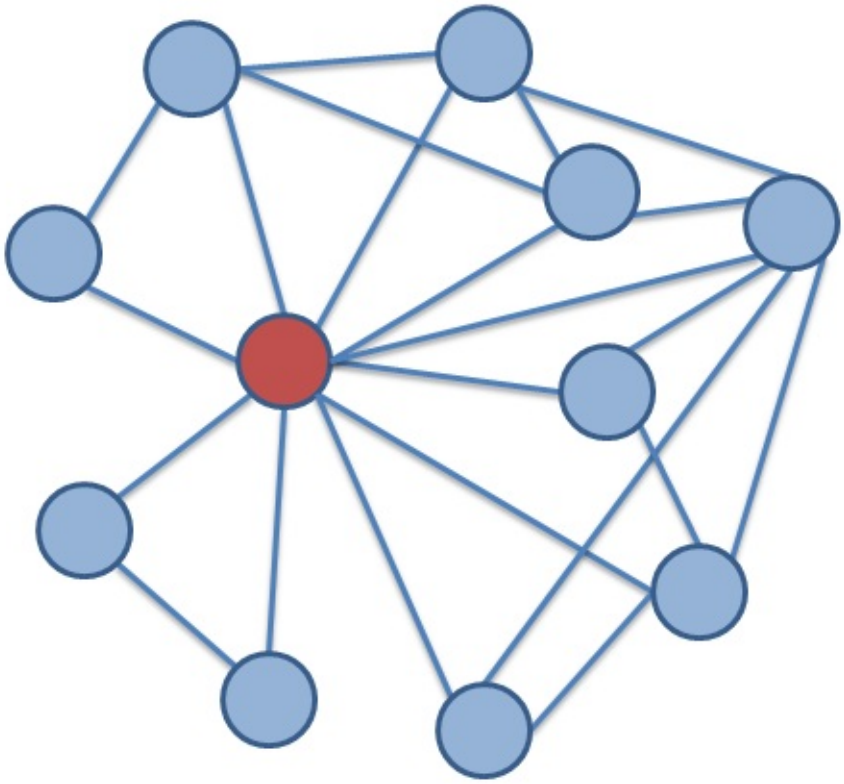
```
from flask import Flask
from redis import Redis
import os
host_redis=os.environ.get('HOST_REDIS', 'redis')
port_redis=os.environ.get('PORT_REDIS', '6379')
app = Flask(__name__)
 redis = Redis(host=host_redis, port=port_redis)
@app.route('/')
def hello():
 redis.incr('hits')
 return 'Hello World!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

Para acessar o código da prática, acesse o repositório e a pasta **“factor6“**.

Vínculo de portas

De acordo com a lista do modelo 12factor, a sétima boa prática é **“Vínculo de portas”**.

É comum encontrar aplicações executadas dentro de contêineres de servidores web, tal como Tomcat, ou Jboss, por exemplo. Normalmente, essas aplicações são implantadas dentro dos serviços para que possam ser acessadas pelos usuários externamente.



A boa prática sugere que o aplicativo em questão seja auto-contido e dependa de um servidor de aplicação, tal como Jboss, Tomcat e afins. O software deve exportar um serviço HTTP e lidar com as requisições que chegam por ele. Significa que, qualquer aplicação adicional é desnecessária para o código estar disponível à comunicação externa.

Tradicionalmente, a implantação de artefato em servidor de aplicação, tal como Tomcat e Jboss, exige a geração de um artefato e, esse, é enviado para o serviço web em questão. Mas no modelo de contêiner Docker, a idéia é que o artefato do processo de implantação seja o próprio contêiner.

O processo antigo de implantação do artefato em servidor de aplicação, normalmente, não tinha retorno rápido, o que aumentava demasiadamente o processo de implantação de um serviço, pois, cada alteração demandava enviar o artefato para o serviço de aplicação web; e, esse tinha a responsabilidade de importar, ler e executar o novo artefato.

Usando Docker, facilmente, a aplicação torna-se auto-contida. Já construímos um Dockerfile que descreve o que a aplicação precisa:

```
FROM python:2.7
ADD requirements.txt requirements.txt
RUN pip install -r requirements.txt
ADD . /code
WORKDIR /code
CMD python app.py
EXPOSE 5000
```

As dependências estão descritas no arquivo requirements.txt e os dados que devem ser persistidos são geridos por um serviço externo (serviços de apoio) à aplicação.

Outro detalhe da boa prática: a aplicação deve exportar o serviço através da vinculação a uma única porta. Como vemos no código exemplo, a porta padrão do python (5000) é iniciada, mas você pode escolher outra, se julgar necessário. Segue o recorte do código que trata do assunto:

```
if __name__ == "__main__":
  app.run(host="0.0.0.0", debug=True)
```

A porta 5000 pode ser utilizada para servir dados localmente em ambiente de desenvolvimento ou, através de proxy reverso, quando for migrada para produção, com nome de domínio adequado a aplicação em questão.

Utilizar o modelo de vinculação de portas torna o processo de atualização de aplicação mais fluído, uma vez que, na utilização de um proxy reverso inteligente, é possível adicionar novos nós gradativamente, com a nova versão, e remover os antigos à medida que as versões atualizadas são executadas em paralelo.

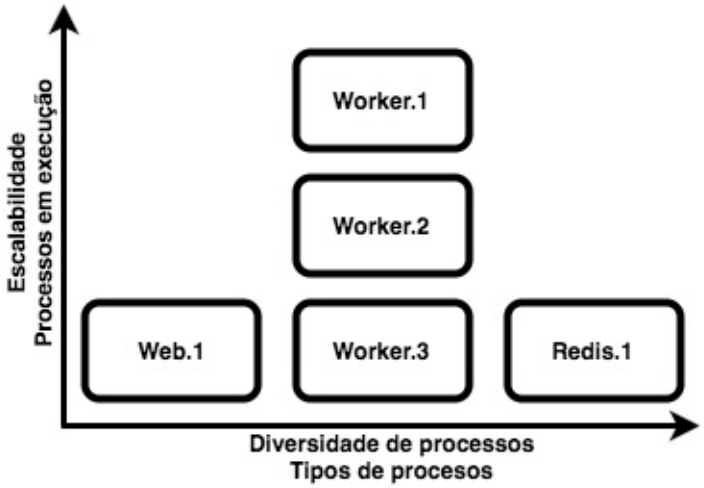
Convém salientar: mesmo que o Docker permita a utilização de mais de uma porta por contêineres, a boa prática enfatiza que você só deve utilizar uma porta vinculada por aplicação.

Concorrência

A oitava boa prática da lista do modelo 12factor, é **“Concorrência”**.

Durante o processo de desenvolvimento de uma aplicação é difícil imaginar o volume de requisição que ela terá no momento que for colocada em produção. Por outro lado, um serviço que suporte grandes volumes de uso é esperado nas soluções modernas. Nada é mais frustante que solicitar acesso a uma aplicação e ela não estar disponível. Sugere falta de cuidado e profissionalismo, na maioria dos casos.

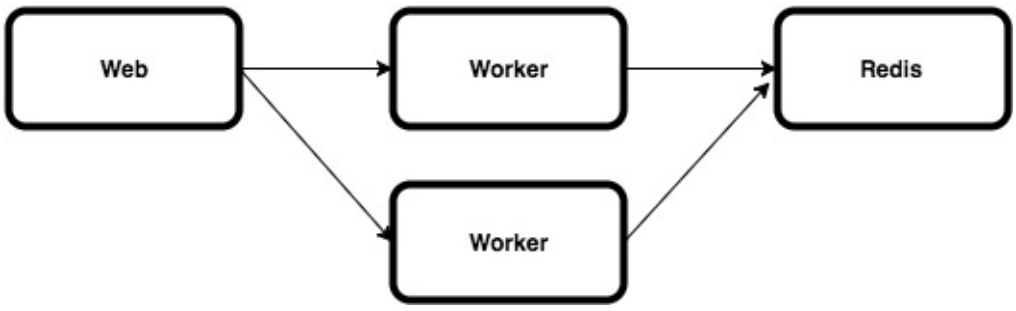
Quando a aplicação é colocada em produção, normalmente é dimensionada para determinada carga esperada, porém é importante que o serviço esteja pronto para escalar. A solução deve ser capaz de iniciar novos processo da mesma aplicação, caso necessário, sem afetar o produto. A figura abaixo, apresenta gráfico de escalabilidade de serviços.



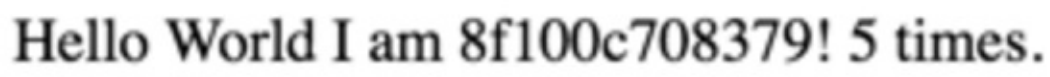
Com objetivo de evitar qualquer problema na escalabilidade do serviço, a boa prática indica que as aplicações devem suportar execuções concorrentes e, quando um processo está em execução, instanciar outro em paralelo e o serviço atendido, sem perda alguma.

Para tal, é importante dividir as tarefas corretamente. É interessante o processo se ater aos objetivos, caso seja necessário executar alguma atividade em backend e, depois retornar uma página para o navegador, é salutar que haja dois serviços tratando as duas atividades, de forma separada. O Docker torna essa tarefa mais simples, pois, nesse modelo, basta especificar um container para cada função e configurar corretamente a rede entre eles.

Para exemplificar a boa prática, usaremos a arquitetura demonstrada na figura abaixo:



O serviço web é responsável por receber a requisição e balancear entre os workers, os quais são responsáveis por processar a requisição, conectar ao redis e retornar a tela de “Hello World” informando quantas vezes foi obtida e qual nome de worker está respondendo a requisição (para ter certeza que está balanceando a carga), como podemos ver na figura abaixo:



O arquivo **docker-compose.yml** exemplifica a boa prática:

```
version: "2"
services:
  web:
    container_name: web
    build: web
    networks:
      - backend
    ports:
      - "80:80"

  worker:
    build: worker
    networks:
      backend:
        aliases:
          - apps
    expose:
      - 80
    depends_on:
      - web

  redis:
    image: redis
    networks:
      - backend

networks:
  backend:
      driver: bridge
```

Para efetuar a construção do balanceador de carga, temos o diretório web contendo arquivos Dockerfile (responsável por criar a imagem utilizada) e nginx.conf (arquivo de configuração do balanceador de carga utilizado).

Segue o conteúdo DockerFile do web:

```
FROM nginx:1.9

COPY nginx.conf /etc/nginx/nginx.conf
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

E, o conteúdo do arquivo nginx.conf:

```
user nginx;
worker_processes 2;

events {
  worker_connections 1024;
}

http {
  access_log /var/log/nginx/access.log;
  error_log /var/log/nginx/error.log;

  resolver 127.0.0.11 valid=1s;

  server {
    listen 80;
    set $alias "apps";

    location / {
      proxy_pass http://$alias;
    }
  }
}
```

No arquivo de configuração acima, foram introduzidas algumas novidades. A primeira, **“resolver 127.0.0.11“**, é o serviço DNS interno do Docker. Usando essa abordagem é possível efetuar o balanceamento de carga via nome, usando recurso interno do Docker. Para mais detalhes sobre o funcionamento do DNS interno do Docker, veja esse documento (https://docs.docker.com/engine/userguide/networking/configure-dns/) (apenas em inglês).

A segunda novidade, função set **$alias “apps”;**, responsável por especificar o nome **“apps”** usado na configuração do proxy reverso, em seguida **“proxy_pass http://$alias;“**. Vale salientar, o **“apps”** é o nome da rede especificada dentro do arquivo **docker-compose.yml**. Nesse caso, o balanceamento é feito para a rede, todo novo contêiner que entra nessa rede é automaticamente adicionado ao balanceamento de carga.

Para efetuar a construção do **worker** temos o diretório **worker** contendo os arquivos **Dockerfile** (responsável por criar a imagem utilizada), **app.py** (aplicação usada em todos os capítulos) e **requirements.txt** (descreve as dependências do app.py).

Segue abaixo o conteúdo do arquivo **app.py** modificado para a prática:

```
from flask import Flask
from redis import Redis
import os
import socket
print(socket.gethostname())
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 I am %s! %s times.' % (socket.gethostname(), redis.get('hits'))
if __name__ == "__main__":
   app.run(host="0.0.0.0", debug=True)
```

Segue o conteúdo do **requirements.txt**:

```
flask==0.11.1
redis==2.10.5
```

E por fim, o **Dockerfile** do **worker** tem o seguinte conteúdo:

```
FROM python:2.7
COPY requirements.txt requirements.txt
RUN pip install -r requirements.txt
COPY . /code
WORKDIR /code
CMD python app.py
```

No serviço **redis** não há construção da imagem, usaremos a imagem oficial para efeitos de exemplificação.

Para testar o que foi apresentado até então, realize o clone do repositório (https://github.com/gomex/exemplo-12factor-docker) e acesse a pasta **factor8**, executando o comando, abaixo, para iniciar os contêineres:

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

Acesse os contêineres através do navegador na porta 80 do endereço localhost. Atualize a página e veja que apenas um nome aparece.

Por padrão, o Docker-Compose executa apenas uma instância de cada serviço explicitado no **docker-compose.yml**. Para aumentar a quantidade de contêineres **"worker"**, de um para dois, execute o comando abaixo:

```
docker-compose scale worker=2
```

Atualize a página no navegador e veja que o nome do host alterna entre duas possibilidades, ou seja, as requisições estão sendo balanceadas para ambos contêineres.

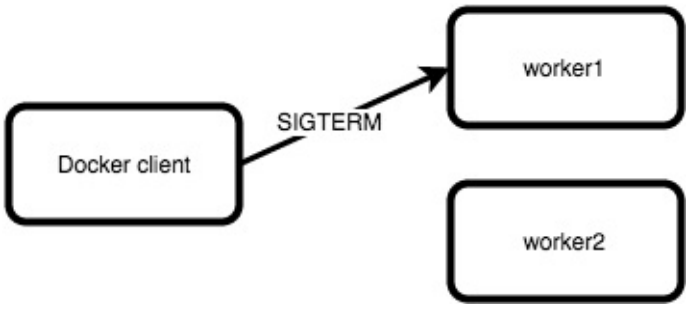
Nessa nova proposta de ambiente, o serviço **web** se encarrega de receber as requisições HTTP e fazer o balanceamento de carga. Então, o **worker** é responsável por processar as requisições, basicamente obter o nome de host, acessar o **redis** e a contagem de quantas vezes o serviço foi requisitado e, então, gerar o retorno para devolvê-lo ao serviço **web** que, por sua vez, responde ao usuário. Como podemos perceber, cada instância do ambiente tem função definida e, com isso, é mais fácil escalá-lo.

Aproveitamos para dar os créditos ao capitão Marcosnils, que nos mostrou como é possível balancear carga pelo nome da rede docker.

Descartabilidade

Na nona posição da lista do modelo 12factor, a **“Descartabilidade”**.

Quando falamos de aplicações web, espera-se que mais de um processo atenda a todo tráfego requisitado para o serviço. Porém, tão importante quanto a habilidade de iniciar novos processos, a capacidade de um processo defeituoso terminar na mesma velocidade que iniciou, pois um processo que demora para finalizar pode comprometer toda solução, uma vez que ela pode ainda estar atendemos requisições de forma defeituosa.



Em resumo, podemos dizer que aplicações web deveriam ser capazes de remover rapidamente processos defeituosos.

Com objetivo de evitar que o serviço prestado seja dependente das instâncias que o servem, a boa prática indica que as aplicações devem ser descartáveis, ou seja, desligar uma de suas instâncias não deve afetar a solução como um todo.

O Docker tem a opção de descartar automaticamente um contêiner após o uso - no **docker container run** utilize a opção **–rm**. Vale salientar que essa opção não funciona em modo **daemon (-d)**, portanto, só faz sentido utilizar no modo **interativo (-i)**.

Outro detalhe importante na boa prática é viabilizar que o código desligue “graciosamente” e reinicie sem erros. Assim, ao escutar um **SIGTERM**, o código deve terminar qualquer requisição em andamento e então desligar o processo sem problemas e de forma rápida, permitindo, também, que seja rapidamente atendido por outro processo.

Entendemos como desligamento “gracioso” uma aplicação capaz de auto finalizar sem danos à solução; ao receber sinal para desligar, imediatamente recusa novas requisições e apenas finaliza as tarefas pendentes em execução naquele momento. Implícito nesse modelo: as requisições HTTP são curtas (não mais que poucos segundos) e, nos casos de conexões longas, o cliente pode se reconectar automaticamente caso a conexão seja perdida.

A aplicação sofreu a seguinte mudança para atender a especificação:

```
from flask import Flask
from redis import Redis
from multiprocessing import Process
import signal, os

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__":
    def server_handler(signum, frame):
        print 'Signal handler called with signal', signum
        server.terminate()
        server.join()

    signal.signal(signal.SIGTERM, server_handler)

    def run_server():
        app.run(host="0.0.0.0", debug=True)

    server = Process(target=run_server)
    server.start()
```

No código acima, adicionamos tratamento para quando receber um sinal de SIGTERM, finalizar rapidamente o processo. Sem o tratamento, o código demora mais para ser desligado. Dessa forma, concluímos que a solução é descartável o suficiente. Podemos desligar e reiniciar os containers em outro Docker Host e a mudança não causará impacto na integridade dos dados.

Para fins de entendimento sobre o que trabalhamos aqui, cabe esclarecimento, de acordo com o Wikipedia, sinal é: “(...) notificação assíncrona enviada a processos com o objetivo de notificar a ocorrência de um evento.” E, o SIGTERM: “(...) nome de um sinal conhecido por um processo informático em sistemas operativos POSIX. Este é o sinal padrão enviado pelos comandos kill e killall. Ele causa o término do processo, como em SIGKILL, porém pode ser interpretado ou ignorado pelo processo. Com isso, SIGTERM realiza um encerramento mais amigável, permitindo a liberação de memória e o fechamento dos arquivos.”

Para realizar o teste do que foi apresentado até então, realize o clone do repositório (https://github.com/gomex/exemplo-12factor-docker) e acesse a pasta factor8 (isso mesmo, a número 8, vamos demonstrar a diferença pra o factor9), executando o comando abaixo para iniciar os contêineres:

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

Depois, execute o comando abaixo para finalizar os contêineres:

```
time docker-compose stop
```

Você verá que a finalização do worker demora cerca de 11 segundos, isso porque o comportamento do Docker-Compose, para finalizar, primeiro faz um **SIGTERM** e espera por 10 segundos que a aplicação finalize sozinha, caso contrário, envia um **SIGKILL** que finaliza o processo bruscamente. Esse timeout é configurável. Caso deseje modificar, basta usar o parâmetro **“-t”** ou **“–timeout“**. Veja um exemplo:

```
docker-compose stop -t 5
```

Obs: O valor informado, após o parâmetro, é considerado em segundos.

Agora, para testar o código modificado, mude para a pasta **factor9** e execute o seguinte comando:

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

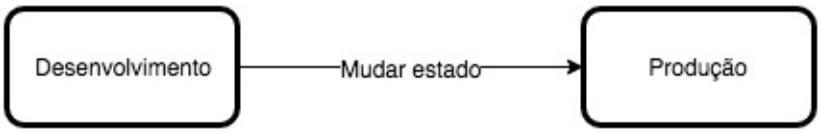
Depois, solicite o término:

```
time docker-compose stop
```

Veja que o processo worker finalizou mais rápido, pois recebeu o sinal SIGTERM. A aplicação fez o serviço de auto finalização e não precisou receber um sinal SIGKILL para ser, de fato, finalizado.

Paridade entre desenvolvimento/produção

Seguindo a lista do modelo 12factor, temos **“Paridade entre desenvolvimento/produção”** como décima boa prática.



Infelizmente, na maioria dos ambientes de trabalho com software, existe grande abismo entre desenvolvimento e produção. Não é mero acaso ou falta de sorte, existe por conta das diferenças entre as equipes de desenvolvimento e as de infraestrutura. E, de acordo com o 12factor, manifestam-se nos seguintes âmbitos:

- **Tempo**: o desenvolvedor pode trabalhar um código que demora dias, semanas ou até meses para ser transferido à produção.
- **Pessoal**: desenvolvedores escrevem código, engenheiros de operação fazem o deploy do código.
- **Ferramentas**: desenvolvedores podem usar conjuntos como Nginx, SQLite e OS X, enquanto o app em produção, usa Apache, MySQL e Linux.

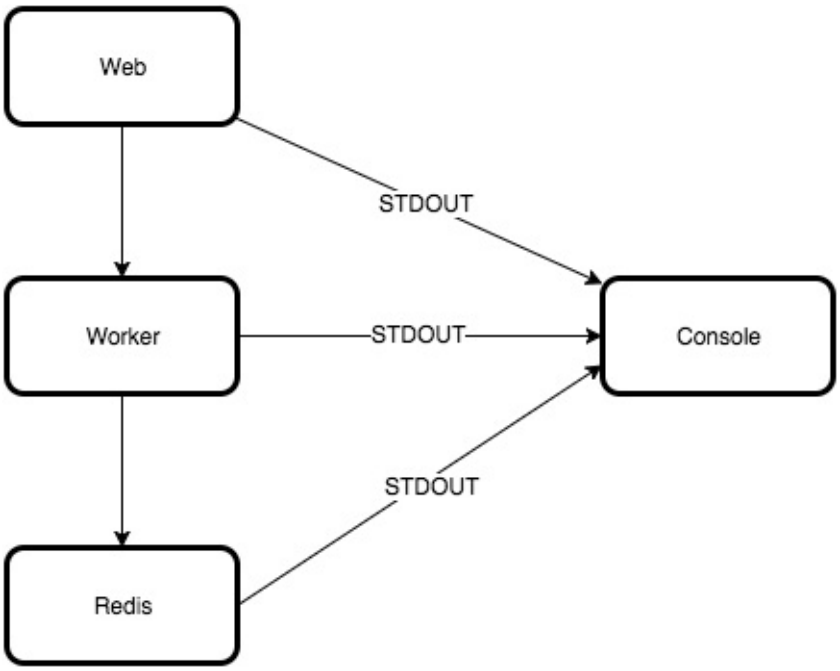
O 12factor pretende colaborar para reduzir o abismo entre as equipes e equalizar os ambientes. Com relação aos âmbitos apresentados, seguem as respectivas propostas:

- **Tempo**: desenvolvedor pode escrever código e ter o deploy concluído em horas ou até mesmo minutos depois.
- **Pessoal**: desenvolvedores que escrevem código estão proximamente envolvidos em realizar o deploy e acompanhar o comportamento na produção.
- **Ferramentas**: manter desenvolvimento e produção o mais similares possível.

A solução de contêiner tem como um dos principais objetivos colaborar com a portabilidade entre ambiente de desenvolvimento e produção. A ideia é que a imagem seja construída e apenas seu status modifique para ser posta em produção. O código atual já está pronto para esse comportamento, assim, não há muito a ser modificado para garantir a boa prática. É como um bônus pela adoção do Docker e o seguimento das outras boas práticas do 12factor.

Logs

A décima primeira boa prática, na lista do modelo 12factor, é **“Logs”**.



No desenvolvimento de código, gerar dados para efeitos de log é algo bastante consolidado. Não acreditamos que existam softwares em desenvolvimento sem essa preocupação. Porém, o uso correto do log vai além de, apenas, gerar dados.

Para efeito de contextualização, de acordo com o 12factor, o log é: “(…) fluxo de eventos agregados e ordenados por tempo coletados dos fluxos de saída de todos os processos em execução e serviços de apoio.”

Logs, normalmente, são armazenados em arquivos, com eventos por linha (pilhas de exceção podem ocupar várias linhas). Mas essa prática não é indicada, ao menos não na perspectiva da aplicação. Isso quer dizer que, a aplicação não deveria se preocupar em qual arquivo guardar os logs.

Especificar arquivos, implica informar o diretório correto desse arquivo, que, por sua vez, resulta em configuração prévia do ambiente. Isso impacta, negativamente, na portabilidade da aplicação, pois é necessário que, o ambiente que receberá a solução, siga uma série de requisitos técnicos para suportar a aplicação, enterrando, assim, a possibilidade do “Construa uma vez, rode em qualquer lugar”.

A boa prática indica que as aplicações não devem gerenciar ou rotear arquivos de log, mas devem ser depositados sem qualquer esquema de buffer na saída padrão (STDOUT). Assim, uma infraestrutura externa à aplicação - plataforma - deve gerenciar, coletar e formatar a saída dos logs para futura leitura. Isso é realmente importante quando a aplicação está rodando em várias instâncias.

Com o Docker, tal tarefa se torna fácil, pois o Docker já coleta logs da saída padrão e encaminha para algum dos vários drivers de log. O driver pode ser configurado na inicialização do container de forma a centralizar os logs no serviço remoto de logs, por exemplo syslog.

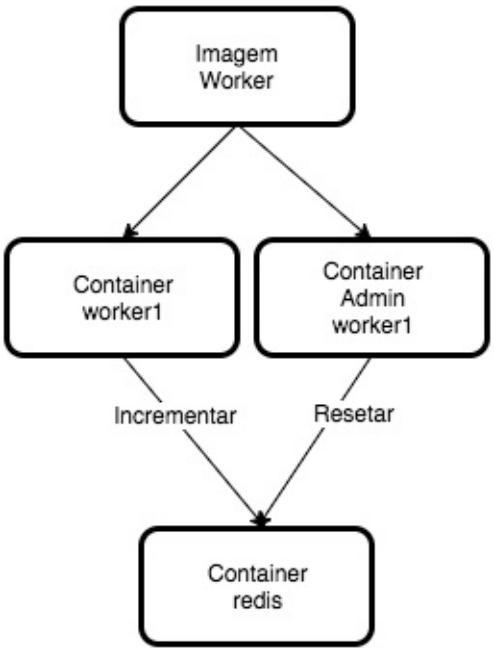
O código exemplo no repositório(https://github.com/gomex/exemplo-12factor-docker), na pasta factor11, está pronto para testar a boa prática, pois já envia todas as saídas de dados para STDOUT e você pode conferir iniciando o serviço com o comando abaixo:

```
docker-compose up
```

Depois de iniciar, acesse o navegador e verifique as requisições da aplicação que aparecem na console do Docker-Compose.

Processos de administração

Décima segunda e última boa prática da lista do modelo 12factor: **“Processos de administração”**.



Toda aplicação demanda administração. Isso quer dizer que, uma vez implantada, é possível que a aplicação precise receber determinados comandos para corrigir possíveis problemas ou simplesmente mudar de comportamento. Como exemplos temos migrações de banco de dados, execução de scripts diversos como backup e, também, execução de um console para inspeção do serviço.

A boa prática recomenda processos de administração executados em ambientes idênticos ao utilizado no código em execução, seguindo todas as práticas apresentadas até então.

Usando Docker é possível rodar os processos utilizando a mesma imagem base no ambiente de execução que desejar. Com isso, podemos nos beneficiar da comunicação entre os contêineres e do uso de volumes que forem necessários e afins.

Para exemplificar a boa prática criamos o arquivo **reset.py**:

```
from redis import Redis
import signal, os

host_redis=os.environ.get('HOST_REDIS', 'redis')
port_redis=os.environ.get('PORT_REDIS', '6379')

redis = Redis(host=host_redis, port=port_redis)

redis.set('hits', 0)
```

Responsável por reinicializar o contador de visitas do Redis, o comando é dado usando um contêiner diferente a partir da mesma imagem Docker. Primeiro iniciamos o ambiente, baixe o repositório e acesse a pasta factor12 e execute o comando:

```
docker-compose up
```

Acesse a aplicação pelo navegador. Caso esteja usando GNU/Linux ou Docker For MAC e Windows, acesse o endereço 127.0.0.1. Você verá a seguinte frase:

```
“Hello World! 1 times.”
```

Acesse a aplicação mais algumas vezes para o marcador aumentar.

Depois, execute o comando de administração a partir do serviço worker:

```
docker-compose exec worker python reset.py
```

O comando **“python reset.py”** será executado dentro de um novo contêiner, mas usando a mesma imagem de um worker regular.

Acesse novamente a aplicação e verifique se o marcador iniciou a partir de 1 novamente.

Dicas para uso do Docker

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

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

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

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

Dicas para Rodar

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

Contêiners descartáveis

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

Para tal, utilize o argumento `--rm` . Isso faz com que todos os contêiners, e seus dados, sejam removidos após o termino da execução, evitando consumir disco desnecessariamente.

Em geral, pode-se utilizar o comando `run` como no exemplo:

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

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

Verifique variáveis de ambiente

Às vezes faz-se necessário verificar qual metadados são definidos como variáveis de ambiente em uma imagem. Use o comando `env` para obter essa informação:

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

Para verificar as variáveis de ambientes passados de um contêiner 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 especificados através da configuração de driver `--log-driver=VALUE` no comando `docker container 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 contêiners Docker são expostos e compartilhados através de argumentos de volumes utilizados ao criar e iniciar o contêiner. Esses volumes não seguem as regras do Union File System, pois os dados persistem mesmo quando o contêiner é removido.

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

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

Com a execução desse container, teremos serviço Nginx que usa o volume criado para persistir seus dados; os dados persistirão mesmo após o contêiner ser removido.

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

```
docker container run --rm -v /tmp:/backup --volumes-from nginx-teste busybox tar -cvf /backup/backup_nginx.tar /usr/s
hare/nginx/html
```

Após a execução do comando, temos um arquivo `backup_nginx.tar` dentro da pasta /tmp do **Docker host**.

Para restaurar esse backup utlize:

```
docker container run --rm -v /tmp:/backup --volumes-from nginx-teste busybox tar -xvf /backup/backup.tar /usr/share/n
ginx/html
```

Mais informação também podem ser encontradas na resposta, onde é possível encontrar alguns *aliases* para esses dois comandos. Esses *aliases* também estão disponíveis abaixo, na seção *Aliases*.

Outras fontes são:

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

Use docker container exec para "entrar em um contêiner"

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

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

A funcionalidade é útil em desenvolvimento local e experimentações. Mas evite utilizar o contêiner em produção ou automatizar ferramentas em volta dele.

Verifique a documentação.

Sem espaço em disco do Docker Host

Ao executar contêiners e construir imagens várias vezes, o espaço em disco pode se tornar escasso. Quando isso acontece, é necessário limpar alguns contêiners, imagens e logs.

Uma maneira rápida de limpar contêiners e imagens é utilizar o seguinte comando:

```
docker system prune
```

Com esse comando você removerá:

- Todos os contêiners que não estão em uso no momento
- Todos os volumes que não estão em uso por ao menos um contêiner
- Todas as redes que não estão em uso por ao menos um contêiner
- Todas as imagens *dangling*

Obs: Pra não aprofundar muito no conceito baixo nível do Docker, podemos dizer que imagens *dangling* são simplesmente sem ponteiros e assim desnecessárias para o uso convencional.

Dependendo do tipo de aplicação, logs também podem ser volumosos. O gerenciamento depende muito de qual driver é utilizado. No driver padrão (`json-file`) a limpeza pode se dar 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
- Considere especificar a opção `max-size` para o *driver* de log ao executar `docker container run` : https://docs.docker.com/engine/reference/logging/overview/#json-file-options

Aliases

Com alias é possível transformar comandos grandes em menores. Temos algumas novas opções para executar tarefas mais complexas.

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

```
# runs docker container exec in the latest container
function docker-exec-last {
  docker container 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
}

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 $(( `cat /sys/fs/cgroup/memory/docker/$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 container 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 container run --rm -v /tmp:/backup --volumes-from "$1" debian:jessie tar -czvf /backup/backup.tar.gz "${@:2}"

}
# restore files from /tmp/backup.tar.gz into a docker volume
function docker-volume-restore-compressed() {
  docker container run --rm -v /tmp:/backup --volumes-from "$1" debian:jessie tar -xzvf /backup/backup.tar.gz "${@:2}"

  echo "Double checking files..."
  docker container run --rm -v /tmp:/backup --volumes-from "$1" debian:jessie ls -lh "${@:2}"
}
# backup files from a docker volume into /tmp/backup.tar
function docker-volume-backup() {
  docker container run --rm -v /tmp:/backup --volumes-from "$1" busybox tar -cvf /backup/backup.tar "${@:2}"
}
# restore files from /tmp/backup.tar into a docker volume
function docker-volume-restore() {
  docker container run --rm -v /tmp:/backup --volumes-from "$1" busybox tar -xvf /backup/backup.tar "${@:2}"
  echo "Double checking files..."
  docker container run --rm -v /tmp:/backup --volumes-from "$1" busybox ls -lh "${@:2}"
}
```

Fontes:

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

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. Recomendamos dar uma olhada:

- Documentação oficial
- Guia do projeto Atomic
- Melhores práticas do Michael Crosby Parte 1
- Melhores práticas do Michael Crosby Parte 2

Use um "linter"

"Linter" é uma ferramenta que fornece dicas e avisos sobre algum código fonte. Para `Dockerfile` existem opções simples, mas é, ainda, um espaço novo em evolução.

Muitas opções foram discutidas aqui.

Desde janeiro de 2016, o mais completo parece ser hadolint, disponível em duas versões: on-line e terminal. O interessante dessa ferramenta é que usa o maduro Shell Check para validar os comandos shell.

O básico

O contêiner produzido pela imagem do `Dockerfile` deve ser o mais efêmero possível. Significa que deve ser possível para-lo, destruí-lo e substituí-lo por um novo contêiner construído com o minimo de esforço.

É comum colocar outros arquivos, como documentação, no diretório junto ao `Dockerfile`; para melhorar a performance de construção, exclua arquivos e diretórios criando um arquivo dockerignore no mesmo diretório. Esse arquivo funciona de maneira semelhante ao `.gitignore`. Usá-lo ajuda a minimizar o contexto de construção enviado -completa com a versão correta- docker host a cada `docker build`.

Evite adicionar pacotes e dependências extras não necessárias à aplicação e minimize a complexidade, tamanho da imagem, tempo de contrução e superfície de ataque.

Minimize também 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 contêiner. Desacoplar aplicações em vários contêiners facilita a escalabilidade horizontal, reuso e monitoramento dos contêiners.

Prefira COPY ao invés de ADD

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

Por outro lado, `COPY` é um comando 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.

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 e a verificação através de um `checksum` após o download. Isso garante 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 construção da imagem falhará. Isso é importante, pois, favorece a reproducibilidade e a segurança na construção de imagens.

Bom exemplo para inspiração é o Dockerfile oficial do Jenkins:

```
ENV JENKINS_VERSION 1.625.3
ENV JENKINS_SHA 537d910f541c25a23499b222ccd37ca25e074a0c

RUN curl -fL http://mirrors.jenkins-ci.org/war-stable/$JENKINS_VERSION/jenkins.war -o /usr/share/jenkins/jenkins.war
\
  && echo "$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`, 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 necessárias, olhe por imagens como `buildpack-deps`.

Porém, caso `debian` ainda seja muito grande, existem imagens minimalistas como `alpine` ou mesmo `busybox`. Evite `alpine` se DNS é necessário, existem alguns problemas a serem resolvidos. 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`. Essa imagem é muito grande, foge da filosofia de processo por contêiner e muito do que a compõe não é essencial para contêiners Docker, veja mais aqui.

Outras fontes

Use o cache de construção de camadas

Outra funcionalidade útil que o uso de `Dockerfile` proporciona é a reconstrução rápida usando o cache de camadas. A fim de aproveitar o recurso, insira ferramentas e dependências que mudem com menos frequência no topo do `Dockerfile`.

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

```
COPY package.json /app/
RUN npm install
COPY . /app
```

Para ler mais sobre isso, veja esse link.

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`:

```
RUN apt-get update && \
    apt-get install -y curl python-pip && \
    pip install requests && \
    apt-get remove -y python-pip curl && \
    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 reduzido o tamanho da imagem. Além disso, observe que, `pip` e `curl` também são removidos uma vez que são desnecessários para a aplicação de produção. Lembre-se que a limpeza precisa ser feita na mesma camada (comando `RUN`). Caso contrário, os dados serão persistidos nessa camada e removê-los mais tarde não terá efeito no tamanho da imagem final.

Note que, segundo a documentação, as imagens oficiais de Debian e Ubuntu rodam automaticamente `apt-get clean`. Logo, a invocação explícita não é necessária.

Evite rodar `apt-get upgrade` ou `dist-upgrade`, pois, vários pacotes da imagem base não vão atualizar dentro de um contêiner desprovido de privilégios. Se há um pacote específico a ser atualizado, simplesmete use `apt-get install -y foo` para atualizá-lo automaticamente.

Para ler mais sobre o assunto, veja o link e esse outro.

Use um script "wrapper" como ENTRYPOINT, às vezes

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

Ótimo exemplo é o fornecido no artigo de Kelsey Hightower: 12 Fracturated Apps:

```
#!/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) como a `CMD` base, assim, os sinais do Unix podem ser devidamente tratados. Leia mais aqui.

Log para stdout

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

Exemplo: Dockerfile do nginx:

```
# 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 mais, veja esse link.

Cuidado ao adicionar dados em um volume no Dockerfile

Lembre-se de usar a instrução `VOLUME` para expor dados de bancos de dados, configuração ou arquivos e pastas criados pelo contêiner. Use para qualquer dado mutável e partes servidas ao usuário do serviço para qual a imagem foi criada.

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

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

Leia mais na explicação de Jérôme Petazzoni.

EXPOSE de portas

O 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 aplicação.

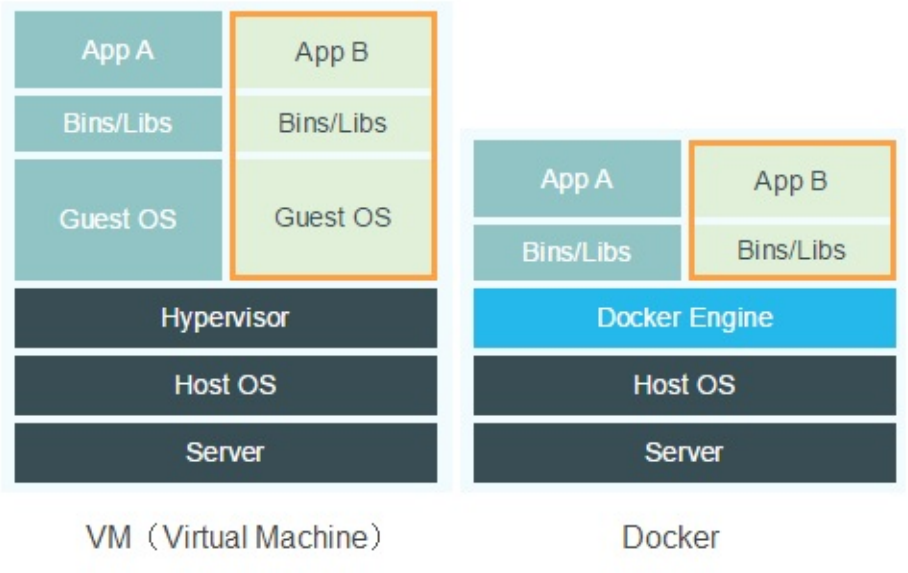
```
# mapeamento público e privado, evite
EXPOSE 80:8080

# apenas privado
EXPOSE 80
```

Apêndice

Container ou máquina virtual?

Após o sucesso repentino do Docker - virtualização com base em contêiners - muitas pessoas têm se questionado sobre possível migração do modelo de máquina virtual para contêiners.



Respondemos tranquilamente: **Os dois!**

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

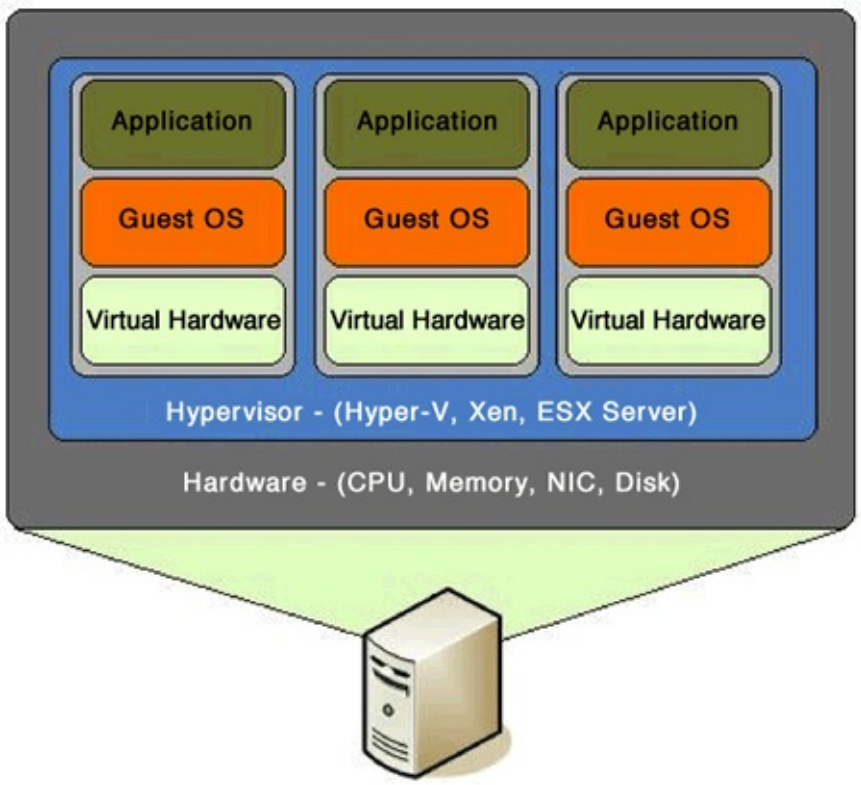
Máquina virtual

Conceito antigo, oriundo dos Mainframes em meados de 1960. Cada operador tinha a visão de estar acessando uma máquina dedicada, mas na verdade, todo recurso do Mainframe, era compartilhado para todos os operadores.

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

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

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



Com a evolução desse modelo, os softwares que implementam a solução, puderam oferecer mais funcionalidades, tais 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 usa e, até mesmo, aumentar gradualmente em caso de necessidade pontual.

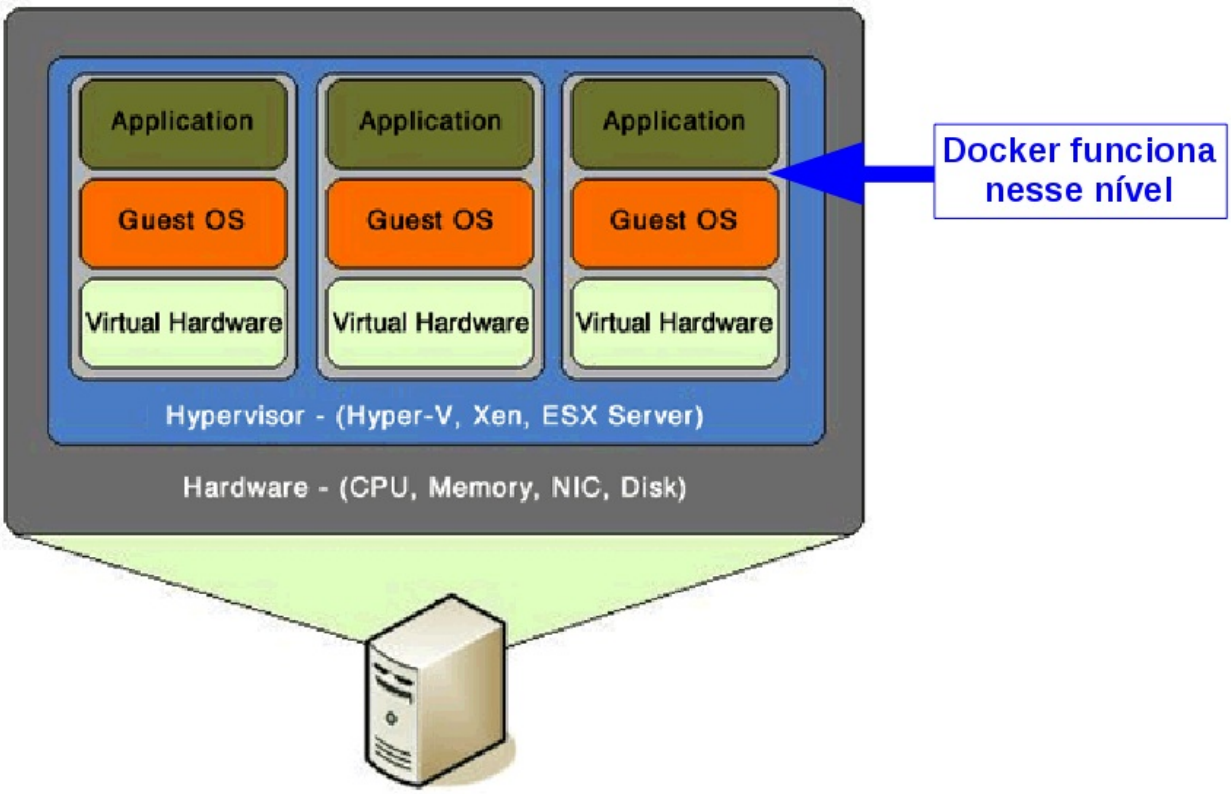
Atualmente, as máquinas virtuais são realidade para qualquer organização que necessite de ambiente de TI, pois facilita a gerência das máquinas físicas e o compartilhamento entre os diversos ambientes necessários para a infraestrutura básica.

Contêiner

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

Utiliza o namespace para prover o devido isolamento de memória RAM, processamento, disco e acesso à rede. Mesmo compartilhando o mesmo kernel, esse processo em execução tem a visão de estar usando um sistema operacional dedicado.

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



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

A configuração de ambiente LXC não era tarefa simples; era necessário algum conhecimento técnico médio para criar e manter um ambiente com ele. Com o 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”. Grosseiramente podemos descrever as imagens como definições estáticas daquilo que os contêiners devem ser na inicialização. São como fotografias do ambiente. Uma vez instanciadas e, colocadas em execução, assumem a função de contêiners; saem da abstração de definição e se transformam em processos em execução, dentro de um contexto isolado. Enxergam um sistema operacional dedicado para sí, mas na verdade compartilham o mesmo kernel.

Junto à facilidade de uso dos contêiners, o Docker agregou o conceito de nuvem, que dispõe de serviço para carregar e “baixar” imagens Docker. Trata-se de aplicação web que disponibiliza 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 contêiners ultrapassa a questão técnica e adentra 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.

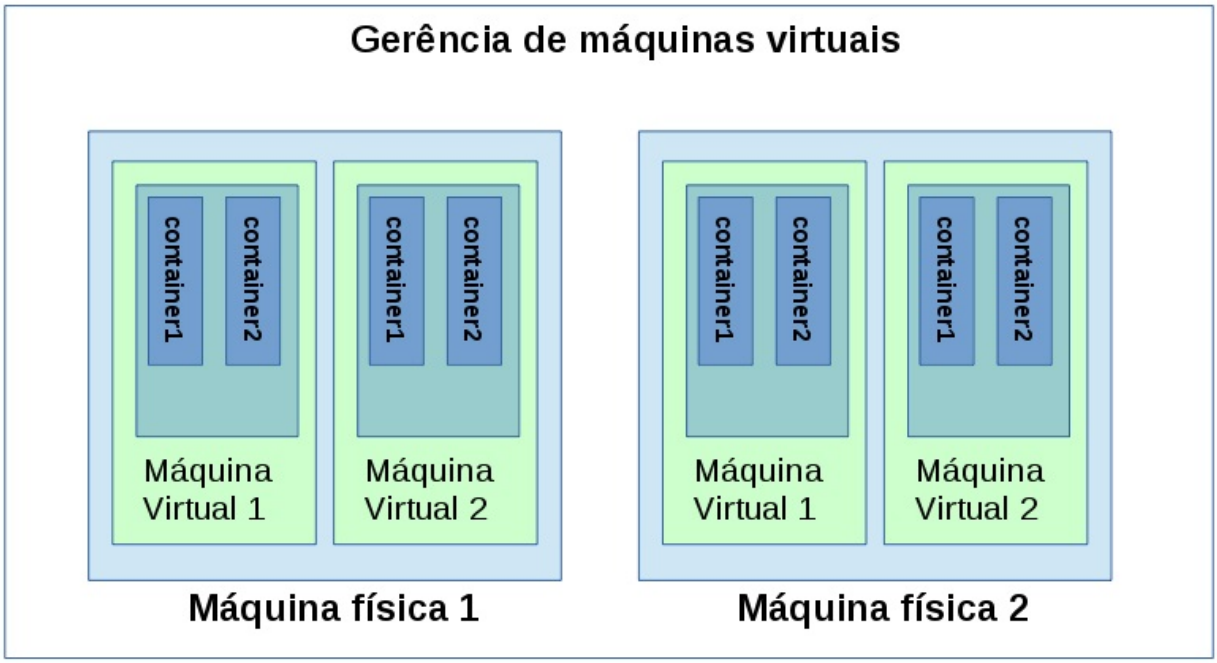
Utilizando a nuvem Docker agora, é possível disponibilizar ambientes de teste mais leves, permitindo que, em plena reunião com o chefe, você pode baixar a solução para um problema que ele descreve e mostrar antes que ele saia da sala. Permite também que você consiga disponibilizar padrão de melhores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

Conclusão

Com os dados apresentados, percebemos que o ponto de conflito entre as soluções é baixo. 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 contêiner.

Percebam que teremos vários níveis de isolamento. No primeiro, a máquina física, que foi repartida em várias máquinas virtuais, ou seja, já temos nossa camada de 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 contêiners 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 contêiners diferentes e, isso não seria nenhum problema de boas práticas de gerência de serviços, muito menos de segurança.



Caso esses contêiners 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.