

Containers e Docker

Capítulo 1

- História dos containers ... e Docker
- Nosso primeiro container
- Containers em background
- Reiniciando e "atachando" (to Attach) em um container
- Entendendo Imagens Docker
- Construir imagens interativamente



História dos containers ... e Docker

[Anterior](#) | [Indice](#) | [Proxima](#)

História dos containers ... e Docker

Primeiras experiências

- [IBM VM/370 \(1972\)](#)
- [Linux VServers \(2001\)](#)
- [Solaris Containers \(2004\)](#)
- [FreeBSD jails \(1999-2000\)](#)

De fato, containers já estão na área por um *longo período*.

(Veja [esse excelente blob do Serge Hallyn](#) para mais detalhes históricos.)

Containers = mais fácil que VMs

- Eu não posso falar pelo Heroku, mas os containers foram (uma das) armas secretas da dotCloud
- dotCloud estava operando um PaaS, usando um engine próprio de containers.
- Esse engine era baseado no OpenVZ (e depois, LXC) e AUFS.
- Em 2008, começou como um simples script Python.
- Em 2012, tinha múltiplos (~10) componentes Python.
- No final de 2012, dotCloud refatorou esse engine.
- O codinome para esse projeto foi "Docker."

Primeira versão pública do Docker

- Em Março de 2013 na PyCon, Santa Clara:
"Docker" é mostrado ao público pela primeira vez.
- Lançado como uma licença open source.
- Teve reações e feedbacks muito positivos!
- O time dotCloud progressivamente chaveu para o desenvolvimento do Docker.
- No mesmo ano, dotCloud mudou o nome para Docker.
- Em 2014, o PaaS foi vendido.

Docker early days (2013-2016)

Primeiros usuários do docker

- PAAS (Flynn, Dokku, Tsuru, Deis...)
- Usuários de PAAS
- CI platforms
- developers, developers, developers, developers

Docker se torna um padrão de fato

- Docker atinge a versão 1.0 milestone.
- Plataformas existentes como Mesos e Cloud Foundry adicionam suporte ao Docker.
- Padronização na OCI (Open Containers Initiative).
- Outros engines de containers são desenvolvidos.
- Criação da CNCF (Cloud Native Computing Foundation).

Docker se torna uma plataforma


- O engine de container é agora conhecido como "Docker Engine."
- Outras ferramentas foram adicionadas:
 - Docker Compose ("Fig")
 - Docker Machine
 - Docker Swarm
 - Kitematic
 - Docker Cloud ("Tutum")
 - Docker Datacenter
 - etc.
- Docker Inc. lança ofertas comerciais.



Nosso primeiro container

[Anterior](#) | [Indice](#) | [Proxima](#)

Nosso primeiro container

 Colorful plastic tubs

Objetivo

Ao final dessa lição, você terá:

- Visto o docker em ação.
- Levantado seu primeiro container.

Hello World

Apenas rode o comando abaixo:

```
$ docker run busybox echo hello world  
hello world
```

Este foi seu primeiro container!

- Nós usamos umas das mais simples imagens disponíveis: busybox.
- busybox é usual em sistemas embarcados (telefones, roteadores...)
- Nós rodamos um simples processo e fizemos echo de hello world.

Um container mais útil

Vamos rodar um container mais legal:

```
$ docker run -it ubuntu  
root@04c0bb0a6c07:/#
```

- Esse é um novo container.
- Ele roda um sistema ubuntu simples e sem frescuras.
- `-it` é uma abreviatura para `-i -t`.
 - `-i` diz ao Docker para nos contactar com o stdin do container.
 - `-t` diz ao Docker que nós queremos um pseudo terminal..

Façamos alguma coisa no container

Tente rodar figlet em nosso container.

```
root@04c0bb0a6c07:/# figlet hello  
bash: figlet: command not found
```

Certo, nós precisamos instalar o figlet.

Instalando um pacote em nosso container

Nós queremos figlet, então vamos instalar:

```
root@04c0bb0a6c07:/# apt-get update
...
Fetched 1514 kB in 14s (103 kB/s)
Reading package lists... Done
root@04c0bb0a6c07:/# apt-get install figlet
Reading package lists... Done
...
```

Um minuto depois, figlet está instalado!

Tentando rodar nosso recente programa instalado

O programa figlet requer um argumento:

```
root@04c0bb0a6c07:/# figlet hello
```

```
  _  _  _  _  _  _  _  _  _  _  _  _  _  _  _  _  _  _  _  _  _  _  
  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  
  |  '  _  \  /  _  \  /  |  |  /  _  \  |  
  |  |  |  |  |  _  /  |  |  (  )  |  
  |  |  |  |  \  _  |  |  |  \  _  /
```

Beautiful! 🥰

Host e containers são coisas independentes

- Nós rodamos o container ubuntu numa máquina Linux.
- Eles tem pacotes diferentes e independentes..
- Instalar alguma coisa no host não expõem ao container.
- E vice-versa.
- Mesmo se ambos o container o host tiverem o mesmo sistema operacional!
- Nós podemos rodar *qualquer container on qualquer host*.

Onde está nosso container?

- Nosso container agora está num estado de *stopped* .
- Ele continua existindo no disco, mas todos os recursos computacionais estão congelados.
- Ainda vamos ver como trazer o container de volta.

Subindo um outro container

O que acontece se subirmos outro container, e tentarmos rodar o comando figlet novamente?

```
$ docker run -it ubuntu  
root@b13c164401fb:/# figlet  
bash: figlet: command not found
```


- Nós subimos uma nova instância de container_.
- A imagem básica do Ubuntu foi usada, e o aplicativo figlet não está lá.



Containers em background

[Anterior](#) | [Indice](#) | [Proxima](#)

Containers em background

 Background containers

Objetivos

Nossos primeiros containers foram *interativos*.

Agora nós vamos ver como:

- Rodar um container não interativo.
- Rodar um container em background.
- Listar os containers que estão rodando.
- Verificar os logs de um container.
- Para um container.
- Listar os containers que estão parados.

Um container não interativo

Vamos rodar um container customizado não interativo.

Esse container somente mostra a data/hora a cada segundo.

```
$ docker run jpetazzo/clock  
Fri Feb 20 00:28:53 UTC 2015  
Fri Feb 20 00:28:54 UTC 2015  
Fri Feb 20 00:28:55 UTC 2015  
...
```

- Esse container vai rodar para sempre.
- Para parar, precione ^C.
- Docker fez download da imagem jpetazzo/clock automaticamente.
- Essa é uma imagem de usuário criada por jpetazzo.
- Nós vamos ver mais sobre imagens de usuário (e outros tipos de imagem) depois.

Rodando um container em background

Container podem subir em background, através do uso da flag `-d` (daemon mode):

```
$ docker run -d jpetazzo/clock  
47d677dcfba4277c6cc68fcaa51f932b544cab1a187c853b7d0caf4e8debe5ad
```

- Agora nós não vemos a saída (output) do container.
- Mas não se preocupe: Docker coleta toda a saída e loga!
- Agora o Docker nos dá o ID do container.

Listando os container que estão rodando

Como nós podemos verificar quais os containers estão rodando?

O comando `docker ps`, assim como o comando do UNIX `ps`, lista os processos rodando.

```
$ docker ps
CONTAINER ID   IMAGE                ...   CREATED          STATUS          ...
47d677dcfba4   jpetazzo/clock      ...   2 minutes ago   Up 2 minutes    ...
```

Docker nos diz:

- O ID truncado do nosso container.
- A imagem usada para subir o container.
- Que nosso container está rodando (Up) por uma porção de minutos.
- Outras informações (COMMAND, PORTS, NAMES) que serão explicadas mais tarde.

Subindo mais containers

Vamos subir mais 2 containers.

```
$ docker run -d jpetazzo/clock  
57ad9bdfc06bb4407c47220cf59ce21585dce9a1298d7a67488359aeaea8ae2a
```

```
$ docker run -d jpetazzo/clock  
068cc994ffd0190bbe025ba74e4c0771a5d8f14734af772ddee8dc1aaf20567d
```

Verifique que o comando `docker ps` corretamente mostra 3 containeres.

Vendo somente o último container levantado

Quando multiplos containers estão rodando, pode ser util ver somente o último container levantado.

Esse comportamento pode ser atingido com o uso da flag `-l` ("Last") :

```
$ docker ps -l
CONTAINER ID   IMAGE                ...   CREATED           STATUS             ...
068cc994ffd0   jpetazzo/clock      ...   2 minutes ago     Up 2 minutes       ...
```

Visualizando somente o ID do container

Muitos comandos do docker funcionam com o ID como argumento: `docker stop`, `docker rm...`

Se você precisar listar somente o ID do container (sem as outras colunas e sem o header), você pode usar a flag `-q` ("Quiet", "Quick"):

```
$ docker ps -q  
068cc994ffd0  
57ad9bdfc06b  
47d677dcfba4
```

Combinando as flags

Nós podemos combinar `-l` e `-q` para ver somente o ID do último container levantado:

```
$ docker ps -lq  
068cc994ffd0
```

A primeira vista, pode parecer que esse comando pode servir para um script.

Entretanto, se precisamos subir um container e pegar o ID de uma forma mais confiável, é melhor usar `docker run -d`, que nós veremos em breve.

(Usar `docker ps -lq` esteja sujeito a "race conditions": o que aconteceria se alguém mais, outro programa ou script, iniciasse um outro container justamente antes de executarmos o `docker ps -lq`?)

Visualizando os logs de um container

Simplesmente...

```
$ docker logs 068  
Fri Feb 20 00:39:52 UTC 2015  
Fri Feb 20 00:39:53 UTC 2015  
...
```

- Especificamos o *prefixo* do ID do container.
- Podemos, certamente, passar o ID completo.
- O comando logs vai exibir o log *completo* do container.
(Algumas vezes, pode ser muito. Vejamos como resolver isso.)

Visualizando somente o final do log

Para evitar ser bombardeado com dezenas de páginas de saída, nós podemos usar a opção `--tail`:

```
$ docker logs --tail 3 068  
Fri Feb 20 00:55:35 UTC 2015  
Fri Feb 20 00:55:36 UTC 2015  
Fri Feb 20 00:55:37 UTC 2015
```

- O argumento é o número de linhas que serão exibidas.

Seguir os logs em tempo real

Exatamente como o comando padrão UNIX `tail -f`, nós podemos seguir o log dos nossos containers:

```
$ docker logs --tail 1 --follow 068
Fri Feb 20 00:57:12 UTC 2015
Fri Feb 20 00:57:13 UTC 2015
^C
```

- O comando acima irá exibir a última linha do log.
- Então, vai continuar a exibir as linhas em tempo real.
- Use `^C` para sair.

Parar nosso container

Existem duas maneiras que nosso container pode ser parado ou desanexado.

- Pare-o usando `docker kill`.
- Pare-o usando `docker stop`.

O primeiro para o container imediatamente, usando um `SIGNAL KILL`.

O segundo é mais suave. Ele manda um `SIGNAL TERM` e depois de 10 segundos, se o container não parou, ele envia o `SIGNAL KILL`.

Lembrando: o `SIGNAL KILL` não pode ser interceptado, e matará o container forçadamente.

Parando nossos containers

Vamos parar nossos containers:

```
$ docker stop 47d6  
47d6
```

Levará 10 segundos:

- Docker envia o SIGNAL TERM;
- O container não reage ao sinal (é somente um shellscript sem tratamento de sinais);
- 10 segundos depois, o container ainda está rodando e o docker envia o KILL signal;
- E o container foi terminado.

Parando os container remanescentes

Vamos ser menos pacientes com os outros 2 containers:

```
$ docker kill 068 57ad  
068  
57ad
```

Vamos verificar que nossos containers realmente estão parados:

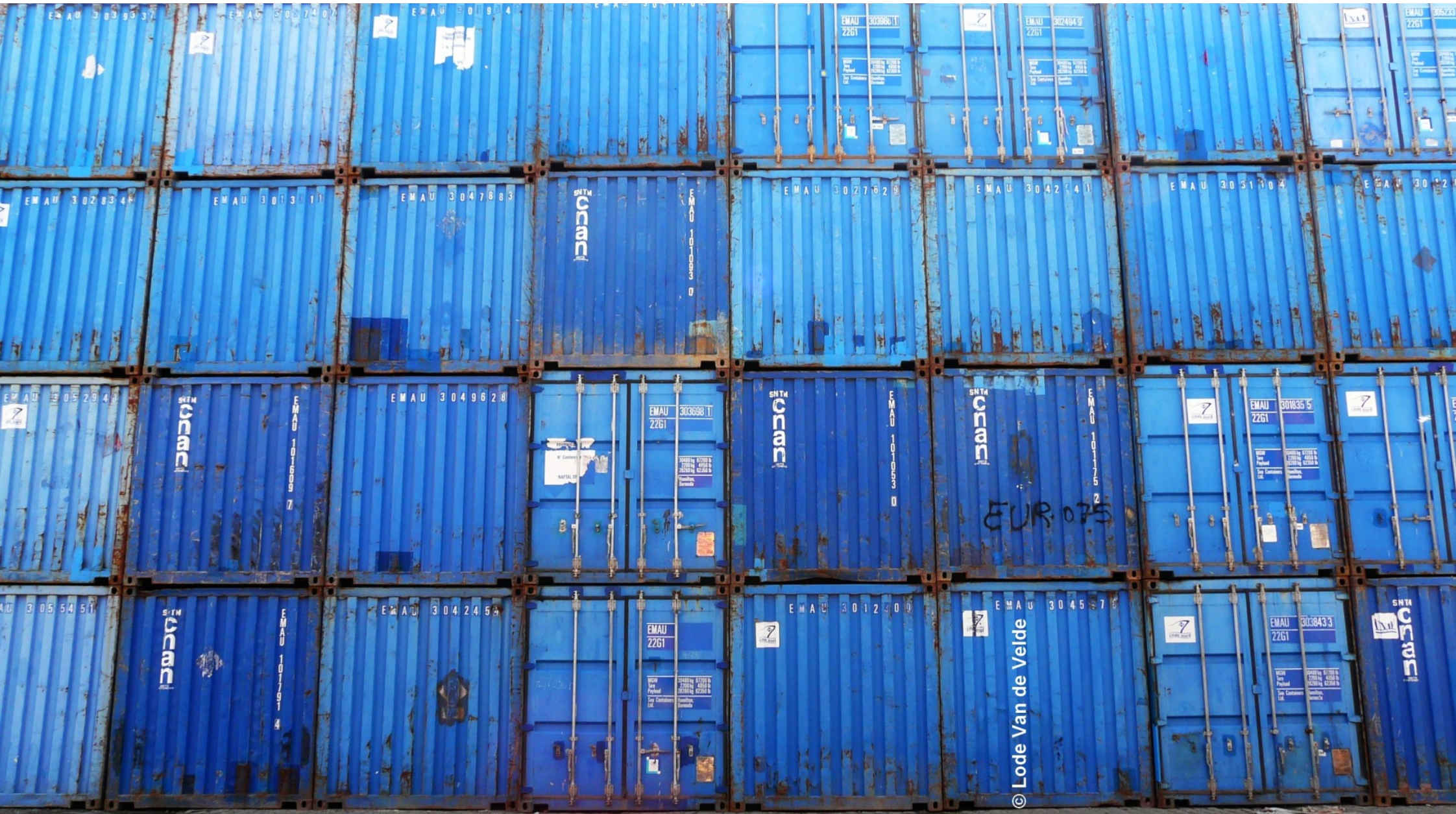
```
$ docker ps
```

Lista containers parados

We can also see stopped containers, with the `-a` (`--all`) option.

```
$ docker ps -a
```

CONTAINER ID	IMAGE	...	CREATED	STATUS
068cc994ffd0	jpetazzo/clock	...	21 min. ago	Exited (137) 3 min. ago
57ad9bdfc06b	jpetazzo/clock	...	21 min. ago	Exited (137) 3 min. ago
47d677dcfba4	jpetazzo/clock	...	23 min. ago	Exited (137) 3 min. ago
5c1dfd4d81f1	jpetazzo/clock	...	40 min. ago	Exited (0) 40 min. ago
b13c164401fb	ubuntu	...	55 min. ago	Exited (130) 53 min. ago



Reiniciando e "atachando" (to Attach) em um container

[Anterior](#) | [Indice](#) | [Proxima](#)

Reiniciando e "atachando" (to Attach) em um container

Nós iniciamos containers em foreground, e em background.

Agora vamos ver como:

- Colocar um container em background.
- "Atachar" (Attach) em um container rodando em background e trazê-lo para foreground.
- Reiniciar um container que foi "parado".

Background e foreground

A distinção entre container em foreground e background é arbitrária.

Pelo ponto de vista do Docker, não existe diferença.

Os container rodam da mesma forma, tanto um cliente attached ou não.

É sempre possível realizar um detach de um container, e fazer o reattach novamente.

Analogia: fazer o attach em um container é como plugar um teclado e um monitor a um servidor físico.

Detaching de um container (Linux/macOS)

- Se você subiu um container `_interativo` (com a opção `-i t`), você pode fazer o detach.
- Simplesmente `^P^Q`.
- (Nunca com `^C`, pois aí é enviado um SIGINT para o container.)

O que mesmo significa `-i t` ?

- `-t` "crie um terminal."
- `-i` "conecte a entrada padrão nesse terminal.."

E no windows ? (Win PowerShell e cmd.exe)

- O Docker para windows é diferente, pois não tem shell.
- ^P^Q não funciona.
- ^C irá fazer o detach, ao invés de parar o container.
- Se é utilizado o Subsystem para Linux, aí é como se fosse Linux/macOS.

Detaching from non-interactive containers

- **Warning:** Se o container subiu sem `-it`...
 - Você não conseguirá fazer o `^P^Q`.
 - Se você fizer o `^C`, o container vai parar.

Saída do container

- Na prática, use `docker attach` se a intenção é enviar input para o container.
- Se você quer somente ver a saída, use o `docker logs`.

```
$ docker logs --tail 1 --follow <containerID>
```

Reiniciando um container

Quando um container sai (para), ele fica no estado "stopped".

Então ele pode ser reiniciado com o comando `start`.

```
$ docker start <yourContainerID>
```

O container será reiniciado com as mesmas opções originalmente utilizadas para lançá-lo inicialmente.

Você pode fazer o reattach normalmente:

```
$ docker attach <yourContainerID>
```

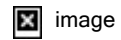
Use o comando `docker ps -a` para identificar o IDs anteriores dos container `jpetazzo/clock`, e tente esses comandos.



Entendendo Imagens Docker

[Anterior](#) | [Indice](#) | [Proxima](#)

Entendendo Imagens Docker



Objetivo

Nessa seção, vamos estudar:

- O que é uma imagem.
- O que é uma camada.
- Os vários namespaces das imagens.
- Como procurar e fazer download de imagens.
- Tag de imagens e como utilizar.

O que é uma imagem?

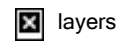
- Imagem = arquivos + metadados
- Esses arquivos formam o sistema de arquivos raiz do nosso contêiner.
- Os metadados podem indicar várias coisas, por exemplo:
 - o autor da imagem
 - o comando a ser executado no contêiner ao iniciá-lo
 - variáveis de ambiente a serem definidas
 - etc.
- As imagens são feitas de *camadas* , conceitualmente empilhadas umas sobre as outras.
- Cada camada pode adicionar, alterar e remover arquivos e / ou metadados.
- As imagens podem compartilhar camadas para otimizar o uso do disco, os tempos de transferência e o uso da memória.

Exemplo numa aplicação Web Java

Cada um dos itens a seguir corresponderá a uma camada:

- Camada base do CentOS
- Pacotes e arquivos de configuração adicionados por nossa equipe de TI local
- JRE
- Tomcat
- Dependências de nossa aplicação
- Nosso código de aplicação e ativos
- Nossa configuração de aplicativo

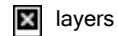
A camada read-write



Diferenças entre containers e imagens

- Uma imagem é um sistema de arquivos somente leitura.
- Um contêiner é um conjunto de processos encapsulados, executando em uma cópia de leitura e gravação desse sistema de arquivos.
- Para otimizar o tempo de inicialização do contêiner, *copy-on-write* é usado em vez de cópia regular.
- `docker run` inicia um contêiner a partir de uma determinada imagem.

Vários contêineres compartilhando a mesma imagem



Comparação com programação orientada a objetos

- As imagens são conceitualmente similares às *classes*.
- Camadas são conceitualmente semelhantes a *herança*.
- Contêineres são conceitualmente similares a *instances*.

Espera um minuto...

Se uma imagem é somente leitura, como a alteramos?

- Nós não.
- Criamos um novo contêiner a partir dessa imagem.
- Então fazemos alterações nesse contêiner.
- Quando estamos satisfeitos com essas mudanças, as transformamos em uma nova camada.
- Uma nova imagem é criada empilhando a nova camada sobre a imagem antiga.

Um problema de galinha e ovo

- A única maneira de criar uma imagem é "congelando" um contêiner.
- A única maneira de criar um contêiner é instanciando uma imagem.
- Socorro!

Criando as primeiras imagens

Há uma imagem vazia especial chamada scratch.

- Permite construir a partir do zero.

O comando `docker import` carrega um tarball no Docker.

- O tarball importado se torna uma imagem independente.
- Essa nova imagem tem uma única camada.

Nota: você provavelmente nunca precisará fazer isso sozinho.

Criando outras imagens

`docker commit`

- Salva todas as alterações feitas em um contêiner em uma nova camada.
- Cria uma nova imagem (efetivamente uma cópia do contêiner).

`docker build` **(usado 99% do tempo)**

- Executa uma sequência de compilação repetível.
- Este é o método preferido!

Explicaremos os dois métodos em um momento.

Imagens e as namespaces

Existem três namespaces:

- Imagens oficiais por exemplo. ubuntu,busybox ...
- Imagens de usuário (e organizações) por exemplo. jpetazzo/clock
- Imagens auto-hospedadas por exemplo. registry.example.com:5000/my-private/ image

Vamos explicar cada um deles.

Espaço para nome raiz

O espaço para nome raiz é para imagens oficiais.

Eles são fechados pela Docker Inc.

Eles geralmente são criados e mantidos por terceiros.

Essas imagens incluem:

- Imagens pequenas, "canivetes suíços", como o busybox.
- Imagens de distribuição para serem usadas como bases para suas compilações, como ubuntu, fedora ...
- Componentes e serviços prontos para uso, como redis, postgresql ...
- Mais de 150 neste momento!

Espaço de nome do usuário

O espaço de nome do usuário contém imagens para usuários e organizações do Docker Hub.

Por exemplo:

vriesman/xapi

O usuário do Docker Hub é:

dvriesman

O nome da imagem é:

xapi

Espaço para nome auto-hospedado

Esse espaço para nome contém imagens que não estão hospedadas no Docker Hub, mas em terceiros registros de terceiros.

Eles contêm o nome do host (ou endereço IP) e, opcionalmente, a porta, do servidor de registro.

Por exemplo:

`organizacao-xpto:5000/minha-api-de-mineracao`

- `organizacao-xpto:5000` é o host e a porta do registro
- `minha-api-de-mineracao` é o nome da imagem

Outros exemplos:

`quay.io/coreos/etcd`

`gcr.io/google-containers/hugo`

Como você armazena e gerencia imagens?

As imagens podem ser armazenadas:

- No seu host do Docker.
- Em um registro do Docker.

Você pode usar o cliente Docker para baixar (pull) ou carregar (push) imagens.

Para ser mais preciso: você pode usar o cliente Docker para informar ao Docker Engine empurrar e puxar imagens de e para um registro.

Mostrando as imagens correntes

Vendo quais imagens estão na sua máquina.

```
$ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
fedora	latest	ddd5c9c1d0f2	3 days ago	204.7 MB
centos	latest	d0e7f81ca65c	3 days ago	196.6 MB
ubuntu	latest	07c86167cdc4	4 days ago	188 MB
redis	latest	4f5f397d4b7c	5 days ago	177.6 MB
postgres	latest	afe2b5e1859b	5 days ago	264.5 MB
alpine	latest	70c557e50ed6	5 days ago	4.798 MB
debian	latest	f50f9524513f	6 days ago	125.1 MB
busybox	latest	3240943c9ea3	2 weeks ago	1.114 MB
training/namer	latest	902673acc741	9 months ago	289.3 MB
jpetazzo/clock	latest	12068b93616f	12 months ago	2.433 MB

Pesquisando imagens

Não podemos listar todas as imagens em um registro remoto, mas podemos procurar uma palavra-chave específica:

```
$ docker search marathon
```

NAME	DESCRIPTION	STARS	OFFICIAL	AUTOM
mesosphere/marathon	A cluster-wide init and co...	105		[OK]
mesoscloud/marathon	Marathon	31		[OK]
mesosphere/marathon-lb	Script to update haproxy b...	22		[OK]
tobilg/mongodb-marathon	A Docker image to start a ...	4		[OK]



- "Estrelas" indicam a popularidade da imagem.
- Imagens "oficiais" são aquelas no espaço de nomes raiz.
- Imagens "automatizadas" são criadas automaticamente pelo Docker Hub. (Isso significa que a receita de compilação deles está sempre disponível.)

Downloading images

Existem duas maneiras de baixar imagens.

- Explicitamente, com `docker pull`.
- Implicitamente, ao executar o `docker run` e a imagem não é encontrada localmente.

Baixando (pulling) uma image

```
$ docker pull debian:jessie
Pulling repository debian
b164861940b8: Download complete
b164861940b8: Pulling image (jessie) from debian
d1881793a057: Download complete
```

- Como visto anteriormente, as imagens são compostas de camadas.
- O Docker baixou todas as camadas necessárias.
- Neste exemplo, `:jessie` indica qual versão exata do Debian nós gostaríamos.

É uma tag de versão.

Imagem e tags

- As imagens podem ter tags.
- Tags definem versões ou variantes de imagem.
- `docker pull ubuntu` irá se referir a `ubuntu:latest`.
- A tag `:latest` é geralmente atualizada frequentemente.

Quando (não) usar tags

Não especifique tags:

- Ao fazer testes e prototipagem rápidos.
- Ao experimentar.
- Quando você quiser a versão mais recente.

Especifique tags:

- Ao gravar um procedimento em um script.
- Ao ir para a produção.
- Para garantir que a mesma versão seja usada em qualquer lugar.
- Para garantir a repetibilidade mais tarde.

Isso é semelhante ao que faríamos com `pip install`, `npm install`, etc.

Resumo da seção

Aprendemos sobre:

- Imagens e camadas.
- Os 3 espaços de nomes (namespaces) das imagens do Docker.
- Como pesquisar e baixar imagens.



Construir imagens interativamente

[Anterior](#) | [Indice](#) | [Proxima](#)

Construir imagens interativamente

Nesta seção, criaremos nossa primeira imagem de contêiner.

Será uma imagem básica de distribuição, mas iremos pré-instalar o pacote figlet.

Nós vamos:

- Crie um contêiner a partir de uma imagem base.
- Instale o software manualmente no contêiner e gire-o para uma nova imagem.
- Aprenda sobre novos comandos: `docker commit`, `docker tag` e `docker diff`.

O plano

1. Crie um contêiner (com `docker run`) usando nossa distribuição básica de escolha.
2. Execute vários comandos para instalar e configurar nosso software no contêiner.
3. (Opcionalmente) revise as alterações no contêiner com `docker diff`.
4. Transforme o container em uma nova imagem com `docker commit`.
5. (Opcionalmente) adicione tags à imagem com `docker tag`.

Configurando nosso contêiner

Inicie um contêiner Ubuntu:

```
$ docker run -it ubuntu  
raiz @ <yourContainerId>: # /
```

Execute o comando `apt-get update` para atualizar a lista de pacotes disponíveis para instalação.

Em seguida, execute o comando `apt-get install figlet` para instalar o programa em que estamos interessados.

```
root @ <yourContainerId>: # / apt-get update && apt-get install figlet  
.... SAÍDA DE COMANDOS APT-GET ....
```

Inspecione as alterações

Digite `exit` no prompt do contêiner para sair da sessão interativa.

Agora vamos executar o `docker diff` para ver a diferença entre a imagem base e nosso recipiente.

```
$ docker diff <yourContainerId>  
C / raiz  
Um /root/.bash_history  
C / tmp  
C / usr  
C / usr / bin  
Um / usr / bin / figlet  
...
```

Docker rastreia alterações no sistema de arquivos

Como explicado anteriormente:

- Uma imagem é somente leitura.
- Quando fazemos alterações, elas acontecem em uma cópia da imagem.
- O Docker pode mostrar a diferença entre a imagem e sua cópia.
- Para desempenho, o Docker usa sistemas de cópia na gravação.
(isto é, iniciar um contêiner com base em uma imagem grande não incorre em uma cópia enorme.)

Benefícios de segurança da cópia na gravação

- `docker diff` nos fornece uma maneira fácil de auditar alterações
(à Tripwire)
- Os contêineres também podem ser iniciados no modo somente leitura
(o sistema de arquivos raiz será somente leitura, mas eles ainda podem ter volumes de dados de leitura e gravação)

Confirme nossas alterações em uma nova imagem

O comando `docker commit` criará uma nova camada com essas alterações, e uma nova imagem usando essa nova camada.

```
$ docker confirmar <yourContainerId>  
<newImageId>
```

A saída do comando `docker commit` será o ID da sua imagem recém-criada.

Podemos usá-lo como argumento para `docker run`.

Como marcar imagens

Referir uma imagem pelo seu ID não é conveniente. Vamos marcá-lo.

Podemos usar o comando tag:

```
$ docker tag <newImageId> figlet
```

Mas também podemos especificar a tag como um argumento extra para commit:

```
$ docker commit figlet <containerId>
```

E execute-o usando sua tag:

```
$ docker run -it figlet
```

Nos próximos capítulos...?

Processo manual = ruim.

Processo automatizado = bom.

No próximo capítulo, aprenderemos como automatizar a construção processo escrevendo um Dockerfile.