

Containers com Docker

Do desenvolvimento à produção



© Casa do Código

Todos os direitos reservados e protegidos pela Lei nº9.610, de 10/02/1998.

Nenhuma parte deste livro poderá ser reproduzida, nem transmitida, sem autorização prévia por escrito da editora, sejam quais forem os meios: fotográficos, eletrônicos, mecânicos, gravação ou quaisquer outros.

Edição

Adriano Almeida

Vivian Matsui

Revisão

Bianca Hubert

Vivian Matsui

[2017]

Casa do Código

Livros para o programador

Rua Vergueiro, 3185 - 8º andar

04101-300 – Vila Mariana – São Paulo – SP – Brasil

www.casadocodigo.com.br

SOBRE O GRUPO CAELUM

Este livro possui a curadoria da Casa do Código e foi estruturado e criado com todo o carinho para que você possa aprender algo novo e acrescentar conhecimentos ao seu portfólio e à sua carreira.

A Casa do Código faz parte do Grupo Caelum, um grupo focado na educação e ensino de tecnologia, design e negócios.

Se você gosta de aprender, convidamos você a conhecer a Alura (www.alura.com.br), que é o braço de cursos online do Grupo. Acesse o site deles e veja as centenas de cursos disponíveis para você fazer da sua casa também, no seu computador. Muitos instrutores da Alura são também autores aqui da Casa do Código.

O mesmo vale para os cursos da Caelum (www.caelum.com.br), que é o lado de cursos presenciais, onde você pode aprender junto dos instrutores em tempo real e usando toda a infraestrutura fornecida pela empresa. Veja também as opções disponíveis lá.

AGRADECIMENTOS

Dedico esta obra à minha esposa Mychelle. Obrigado por compreender a minha ausência quando necessário, e pelo apoio em todos os momentos. Também aos meus pais, pelo apoio e por nunca deixarem de me incentivar.

Agradeço à Casa do Código por esta oportunidade de produzir mais um trabalho. Especialmente ao Paulo Silveira e Adriano Almeida, pelos ensinamentos e opiniões de muito valor sobre o conteúdo e organização do livro.

Agradecimentos especiais aos amigos Diego Plentz, Samuel Flores e Ricardo Henrique, por me ajudarem durante as intermináveis revisões.

Por fim, agradeço a Deus por mais esta oportunidade.

PREFÁCIO

In March 2013, during PyCon US in Santa Clara, we did the first public demo of Docker. We had no idea that we were about to start the *container revolution*.

What is this so-called container revolution? Is it really a revolution, or is it being blown out of proportion? After all, containers have been around for a very long time; why are we only now talking about a revolution?

I would like to answer those questions with a short story. More than ten years ago, I discovered a project called Xen. You might have heard of it. In fact, if you have used Amazon EC2, you have almost certainly used it.

Xen is a hypervisor. It lets you run multiple virtual machines on a single physical machines. In 2004, Xen had a lot of limitations: it ran only on Linux, and it required custom kernels.

Xen could do something amazing: it was able to do "live migration," meaning that you could move a running virtual machine to another physical computer without stopping (or even rebooting) that virtual machine.

I thought that this was very impressive. More importantly, Xen virtual machines could start in a few seconds, and their performance was very good. It was hard to tell the difference between a physical machine and a virtual machine. Those virtual machines were almost as good as the real ones, but they were much easier to install, to move around, to debug — and they were

cheaper, because we could put many virtual machines on a single server.

This convinced me that virtual machines could be the basis for better, cheaper hosting, and I started my own company to sell VM-based servers.

The technology was good, but many people were skeptical, and still wanted *real servers*. Keep in mind that this was a few years before Amazon Web Services launched EC2.

Ten years later, the sentiment has changed. Of course, there are still applications that cannot run in "the cloud" for various reasons; but the majority of new deployments now involve IAAS or PAAS at some point.

What is the reason for that change? Is it this quasi-magical live migration feature? No. Is it because virtual machines are cheaper than physical ones? No. The real reason is automation. We can now write programs and scripts to create servers, deploy our applications on them, and easily scale them up or down. In our repositories, we have "infrastructure as code" files, describing complex architectures composed of tens, hundreds, thousands of servers; and when we change those files, machines are provisioned and destroyed accordingly. This is the revolution that was made possible by virtualization.

With containers, we will see a similar revolution. Many things that used to be impossible or impractical are becoming not only possible, but easy. Continuous testing and integration; immutable infrastructure; blue/green and canary deployments; golden images... All those techniques (and more) are now available to the

majority of developers, instead of being the luxury or privilege of bigger and modern organizations like Amazon, Google, or Netflix.

Today, when I hear someone say "containers won't work for us," I hear the same voice that said "virtual machines won't work for us" ten years ago. Don't be that voice. I'm not asking that you embrace containers for every use and purpose. But in this book, you will see many ways in which Docker and containers can improve application development and deployment. You are sure to find at least one that will make your job and your life easier!

— Jérôme Petazzoni

Tradução

Em março de 2013, durante a PyCon US em Santa Clara, nós fizemos a primeira demonstração pública do Docker. Não tínhamos ideia de que estávamos prestes a começar a "revolução do container".

O que seria essa "revolução do container"? Seria mesmo uma revolução, ou um exagero? Afinal, *containers* têm estado conosco por muito tempo, então, por que só agora estamos falando sobre uma revolução?

Eu gostaria de responder essas questões com uma pequena história. Mais de 10 anos atrás, descobri um projeto chamado Xen. Você provavelmente já ouviu falar sobre ele. Na verdade, se você já usou a Amazon EC2, você com certeza o usou.

Xen é um *hypervisor*. Ele permite que você rode múltiplas máquinas virtuais em apenas uma única máquina física. Em 2004, o Xen tinha várias limitações: só funcionava no Linux, e requeria

kernels personalizados.

Xen podia fazer algo incrível: ele era capaz de fazer *live migration* (migração em tempo real), ou seja, você podia mover uma máquina virtual funcionando para outra máquina física, sem parar (ou mesmo reinicializar) a máquina virtual.

Eu achei isso bastante impressionante. Mais importante ainda, máquinas virtuais Xen podiam iniciar em alguns segundos, e suas performances eram ótimas. Foi difícil dizer a diferença entre uma máquina física e uma máquina virtual. Essas máquinas virtuais eram quase tão boas quando as reais, mas elas eram muito mais fáceis de serem instaladas, movidas de lugar, de debuggar — e elas eram mais baratas, porque nós podíamos colocar várias máquinas virtuais em um único servidor.

Isso me convenceu de que as máquinas virtuais poderiam ser a base para uma hospedagem melhor e mais barata, e eu comecei minha própria companhia para vender servidores baseados em máquinas virtuais.

A tecnologia era boa, porém muitas pessoas eram céticas e queriam os servidores de verdade. Lembre-se de que isso foi poucos anos antes da Amazon Web Services lançar a EC2.

Dez anos depois, o sentimento havia mudado. Claro que ainda havia aplicações que não podiam ser rodadas "na nuvem" por diversos motivos; mas a maioria dos novos *deploys* agora envolviam IAAS (*Infrastructure as a Service*) ou PAAS (*Platform as a Service*) em algum ponto.

Qual foi a razão para essa mudança? Seria essa feature mágica de migração em tempo real? Não. Seria porque máquinas virtuais

são mais baratas que as físicas? Não. A verdadeira razão é a automatização. Agora, nós podemos escrever programas e scripts para criar servidores, implantar nossas aplicações neles, e escalá-las verticalmente de forma fácil. Em nossos repositórios, nós temos arquivos *infrastructure as code* (infraestrutura como código), descrevendo arquiteturas complexas compostas por dezenas, centenas, milhares de servidores; e quando nós mudamos esses arquivos, as máquinas são provisionadas e destruídas da forma correta. Essa é a revolução que a virtualização tornou possível.

Com containers, nós vamos ver uma revolução similar. Muitas coisas que eram consideradas impossíveis ou impraticáveis estão se tornando não só possíveis, como fáceis. Teste e integração contínua; infraestrutura imutável; *blue/green* e *canary releases*; *golden images*... Todas essas técnicas (e mais) estão agora disponíveis para a maioria dos desenvolvedores, em vez de serem um luxo ou um privilégio das maiores e modernas organizações como Amazon, Google, ou Netflix.

Hoje, quando eu ouço alguém dizer "containers não vão trabalhar para nós", eu ouço a mesma voz que diziam "máquinas virtuais não vão trabalhar para nós" há dez anos. Não seja essa voz. Não estou pedindo para você abraçar os containers para todos os usos e propósitos. Porém, neste livro, você verá muitas formas nas quais o Docker e os containers podem melhorar o desenvolvimento e *deploy* de sua aplicação. Com certeza você achará pelo menos uma que fará o seu trabalho e sua vida mais fáceis!

Sumário

1 Introdução	1
1.1 O que é Docker?	3
1.2 O que diferencia um container de uma máquina virtual?	4
1.3 O que são Namespaces?	5
1.4 Para que serve o Cgroups?	6
1.5 O que é Union file systems?	6
2 Explorando o Docker	8
2.1 Instalação	8
2.2 Hello, Docker!	9
2.3 Um pouco de interatividade	10
2.4 Controlando containers	13
2.5 Destruindo imagens e containers	16
3 Construindo minha primeira imagem	18
3.1 Como funciona a construção de uma imagem?	18
3.2 Explorando o Dockerfile	20
3.3 Mapeando portas	22
3.4 Copiando arquivos	24

3.5 Definindo o diretório de trabalho	27
3.6 Inicializando serviços	29
3.7 Tratando logs	31
3.8 Exportação e importação de containers	32
4 Trabalhando com volumes	34
4.1 Gerenciando os dados	35
4.2 Utilizando volumes no Dockerfile	36
4.3 Backup e restore de volumes	39
5 Configurações de rede	44
5.1 Comunicação entre containers	44
5.2 Alterando a configuração default	46
5.3 Comunicação de containers no mesmo host	48
5.4 Comunicação de containers em hosts diferentes	50
5.5 Comunicação com automação	52
6 Docker Hub	55
6.1 Buscando e puxando imagens	55
6.2 Enviando imagens para um repositório	56
7 Trabalhando com Docker	58
7.1 Preparando o ambiente	58
7.2 Entendendo como funciona	61
7.3 Executando em produção	63
7.4 Um pouco de integração contínua	67
7.5 Limitando o uso de recursos	70
7.6 Orquestrando containers	72
8 Explorando um pouco mais	83

8.1 Docker Remote API	83
8.2 TDD para Docker	87
8.3 Docker no desktop	91
9 Um pouco sobre CoreOS Linux	95
9.1 Principais diferenças	95
9.2 ETCD	96
9.3 Fleet	98
9.4 Criando um cluster	99
9.5 Criando um serviço	103
9.6 Registrando um serviço	105
9.7 Inicializando o cluster	106
10 O que estudar além?	109
10.1 A alternativa ao Docker Compose, o azk	109
10.2 Docker Machine	109
10.3 Docker Swarm	111
10.4 Alpine Linux	114
10.5 Dúvidas?	115

INTRODUÇÃO

Você finalizou um longo processo de desenvolvimento de um software. Agora, é necessário configurar e organizar a infraestrutura para fazer o deploy de sua aplicação e pôr em produção.

De modo geral, você segue um roteiro com vários passos para realizar essa tarefa: instalação e configuração dos servidores, ajustes na aplicação para ser disponibilizada por um *web server*, estabelecer comunicação com o banco de dados, e outras dezenas de configurações até que a aplicação seja publicada e comece a funcionar.

É muito grande o tempo gasto na etapa de configuração. Mesmo utilizando automatizadores como Puppet, Chef ou Ansible, você ainda enfrenta alguns problemas para manter a infraestrutura.

Agora, imagine que a sua aplicação está separada em várias camadas – por exemplo, front-end, back-end, banco de dados, workers etc. –, e você precisa fazer a aplicação funcionar em diferentes ambientes – desenvolvimento, produção ou algum servidor customizado. Tudo isso sai fora de controle, pois você acaba instalando a mesma aplicação de diferentes formas em vários

ambientes.

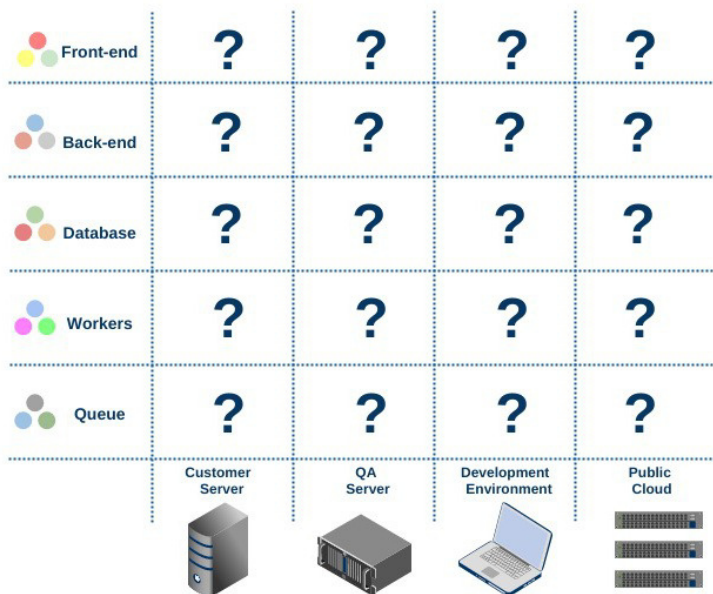


Figura 1.1: Matrix from hell

Além dos problemas citados, ainda temos o alto custo se cada camada estiver alocada em um servidor. Esses e outros problemas podem ser resolvidos com Docker, uma vez que podemos isolar e manter a mesma configuração de cada serviço, ou camada da nossa aplicação, em diferentes ambientes:

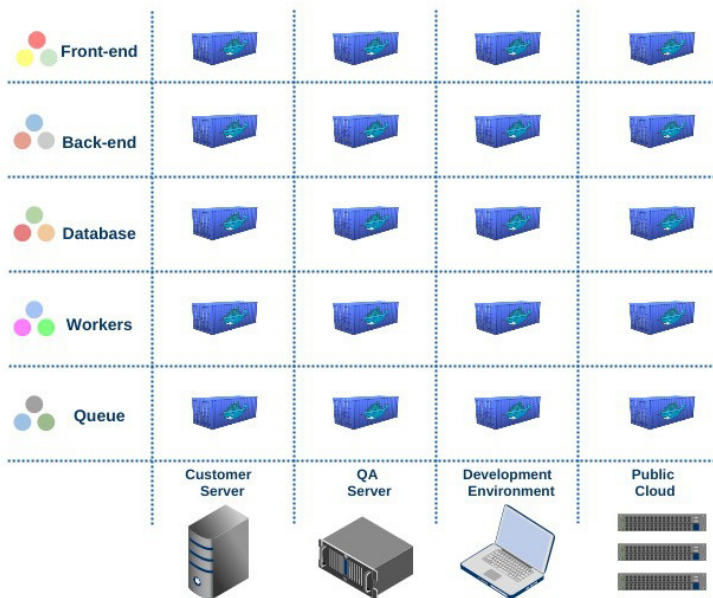


Figura 1.2: Resolvendo o problema com Docker

1.1 O QUE É DOCKER?

Imagine que o servidor é um navio cargueiro e que cada *container* leva várias mercadorias. Docker é uma ferramenta para criar e manter containers, ou seja, ele é responsável por armazenar vários serviços de forma isolada do *SO host*, como: web server, banco de dados, aplicação, memcached etc. O seu back-end é baseado em LXC (*Linux Containers*).

LXC funciona isolando processos do sistema operacional host. É uma espécie de virtualização bastante leve, pois não faz uso de emulação ou suporte a hardware, apenas proporciona a execução de vários sistemas Linux de forma isolada – daí vem a palavra *container*. Além disso, ele utiliza o mesmo *Linux Kernel* do

servidor host, o que torna tudo muito rápido.

Em outras palavras, o Docker é uma alternativa para virtualização completa e leve se comparada aos *hypervisors* KVM, Xen e VMware ESXi.

1.2 O QUE DIFERENCIA UM CONTAINER DE UMA MÁQUINA VIRTUAL?

Para entender melhor a diferença entre virtualização e containers, vamos lembrar primeiro os tipos de virtualização.

Em resumo, temos os seguintes tipos:

- Bare Metal
- Hosted

No *Bare Metal*, o software que proporciona a virtualização é instalado diretamente sobre o hardware, por exemplo, *Xen*, *VMware ESXi* e *Hyper-V*. Esse tipo de virtualização proporciona um isolamento maior e, ao mesmo tempo, uma sobrecarga, pois cada máquina virtual que é criada executará seu próprio *kernel* e instância do sistema operacional. Já o tipo *hosted*, o software que proporciona a virtualização é executado sobre um sistema operacional, por exemplo, o *VirtualBox*.

A virtualização por containers, proposta pelo LXC, ocorre de forma menos isolada, pois compartilha algumas partes do kernel do host, fazendo com que a sobrecarga seja menor.

Na figura a seguir, veja a comparação entre máquinas virtuais e Docker containers:

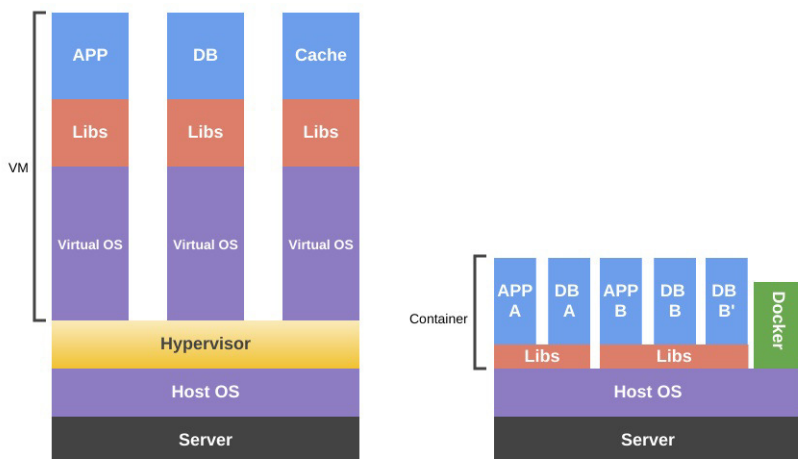


Figura 1.3: Diferença entre Docker e máquinas virtuais

O Linux Kernel possui um recurso chamado de *Cgroups* (*control groups*), que é usado para limitar e isolar o uso de **CPU**, **memória**, **disco**, **rede** etc. Um outro recurso são os *Namespaces*, responsáveis por isolar grupos de processos, de modo que eles não enxerguem os processos de outros grupos, em outros containers ou no sistema host.

Resumindo, containers são mais leves, já que não precisam de um ambiente virtual completo, pois o kernel do host proporciona total gerenciamento de memória, I/O, CPU etc. Isso significa que o processo total de inicialização pode levar poucos segundos.

1.3 O QUE SÃO NAMESPACES?

Como vimos na seção anterior, o Docker tira proveito do recurso de *Namespaces* para prover um espaço de trabalho isolado para os containers. Sendo assim, quando um container é criado,

automaticamente um conjunto de namespaces também é criado para ele.

Namespaces cria uma camada de isolamento para grupos de processos. No caso do Docker, são estes:

- pid – isolamento de processos (PID);
- net – controle de interfaces de rede;
- ipc – controle dos recursos de IPC (*InterProcess Communication*);
- mnt – gestão de pontos de montagem;
- uts – UTS (*Unix Timesharing System*) isolar recursos do kernel;

1.4 PARA QUE SERVE O CGROUPS?

O segredo para executar aplicações de forma isolada é liberar apenas os recursos que você deseja. O *Cgroups* permite que o Docker compartilhe os recursos de hardware existentes no host com os containers e, caso seja necessário, pode definir limites de uso e algumas restrições.

Por exemplo, veremos como limitar o uso de memória para um container específico.

1.5 O QUE É UNION FILE SYSTEMS?

Union file systems (ou UnionFS) são sistemas de arquivos que funcionam por meio da criação de camadas. Eles são leves e muito rápidos.

O Docker utiliza sistemas de arquivos em camadas para a

construção de imagens que serão usadas na criação de containers.

Todos os códigos-fontes do livro podem ser encontrados em:

<https://github.com/infoslack/docker-exemplos>.

Você também pode participar ativamente do grupo de discussão sobre o livro, em

<http://forum.casadocodigo.com.br/>.

Caso você deseje submeter alguma errata ou sugestão, acesse

<http://erratas.casadocodigo.com.br>

Boa leitura!

EXPLORANDO O DOCKER

Por ser baseado em LXC, o Docker funciona somente no Linux. Para que o Docker funcione em outros sistemas operacionais, é necessário ter uma máquina virtual com Linux. Na seção a seguir, veremos algumas opções para instalação.

Quando o livro foi escrito o Docker encontrava-se na versão 1.3 e os exemplos foram atualizados para funcionar em versões posteriores.

2.1 INSTALAÇÃO

Instalando no Linux

No Ubuntu, para instalar e começar a utilizar o Docker é muito simples. Vamos precisar adicionar apenas o repositório oficial para instalar o pacote, para isso antes é necessário incluir a chave gpg:

```
$ sudo apt-key adv --keyserver hkp://p80.pool.sks-keyservers.net:  
80 --recv-keys 58118E89F3A912897C070ADB76221572C52609D
```

Em seguida, podemos incluir o repositório oficial do Docker ao nosso gerenciador de pacotes – pois queremos instalar a última versão estável – e atualizar a nossa lista de pacotes para executar a instalação. Com o seu editor de textos preferido, edite o arquivo

/etc/apt/sources.list.d/docker.list e insira o repositório correspondente a versão do sistema operacional em uso, neste exemplo Ubuntu 14.04 :

```
$ echo "deb https://apt.dockerproject.org/repo ubuntu-trusty main" > /etc/apt/sources.list.d/docker.list
```

Para finalizar, atualize os pacotes e instale o docker:

```
$ sudo apt-get update
$ sudo apt-get install docker-engine
```

Por padrão o docker faz parte do grupo de superusuários ou seja ele necessita do `root` . Caso você não queira fazer uso do comando `sudo` com frequência, podemos criar um grupo chamado `docker` e adicionar o nosso usuário padrão, por exemplo:

```
$ sudo usermod -aG docker infoslack
```

Pronto, o Docker está instalado e agora podemos executar o nosso primeiro teste.

2.2 HELLO, DOCKER!

Para o nosso primeiro exemplo, vamos criar um container e solicitar para que ele execute algum comando:

```
$ sudo docker run ubuntu /bin/echo Hello, Docker!
Unable to find image 'ubuntu' locally
Pulling repository ubuntu
c4ff7513909d: Download complete
511136ea3c5a: Download complete
1c9383292a8f: Download complete
9942dd43ff21: Download complete
d92c3c92fa73: Download complete
0ea0d582fd90: Download complete
```

```
cc58e55aa5a5: Download complete
Hello, Docker!
$
```

Vamos entender por partes o que aconteceu. Primeiro, o Docker fez o download de uma imagem base, no caso do repositório oficial do Ubuntu, na última versão estável 14.04 LTS . Assim, ele instanciou um novo container, configurou a interface de rede e escolheu um IP dinâmico para ele. Em seguida, selecionou um sistema de arquivos para esse novo container, e executou o comando `/bin/echo` graças ao `run` , que é uma opção utilizada pelo Docker para enviar comandos ao container. No final da execução, o Docker capturou a saída do comando que injetamos.

Não se preocupe se você não entendeu todos os detalhes neste momento. Veremos com calma todas as opções oferecidas.

2.3 UM POUCO DE INTERATIVIDADE

O Docker oferece um leque de opções que o torna bastante interativo com os containers. Por exemplo, podemos verificar a existência de containers em execução com o comando `docker ps` :

```
$ sudo docker ps
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS   NAMES
```

Como não possuímos nenhum container em execução ainda, a saída do comando foi apenas o cabeçalho. Para o próximo exemplo, vamos criar um container e acessar o seu *shell* interativo:

```
$ sudo docker run -i -t ubuntu /bin/bash
root@bc5fdb751dd1:/#
```

Desta vez, utilizamos outras opções: `-i` e `-t` em conjunto com a `run`. O parâmetro `-i` diz ao Docker que queremos ter interatividade com o container, e o `-t` que queremos nos *linkar* ao terminal do container. Em seguida, informamos o nome da imagem usada, no caso Ubuntu, e passamos o comando `/bin/bash` como argumento.

Agora que temos acesso ao shell do container, vamos conferir algumas informações para entender melhor o que está acontecendo. Por exemplo, ao verificarmos o arquivo `/etc/lsb-release`, podemos ter certeza de que este shell está sendo executado em um sistema diferente:

```
root@bc5fdb751dd1:/# cat /etc/lsb-release
DISTRIB_ID=Ubuntu
DISTRIB_RELEASE=14.04
DISTRIB_CODENAME=trusty
DISTRIB_DESCRIPTION="Ubuntu 14.04.1 LTS"
root@bc5fdb751dd1:/#
```

Ou ainda, podemos conferir que o container que estamos acessando possui um IP diferente da faixa utilizada pelo host:

```
root@bc5fdb751dd1:/# ifconfig eth0| grep 'inet addr:'
    inet addr:172.17.0.4 Bcast:0.0.0.0 Mask:255.255.0.0
root@bc5fdb751dd1:/# exit
$
```

Explore mais um pouco antes de sair do container. Navegue em seus diretórios e confira alguns arquivos. Eu espero!

...

Agora que você já fez um *tour*, podemos continuar!

Ao sair do container, ele será colocado em estado de pausa pelo Docker. Vamos conferir usando o complemento `-a` ao parâmetro

ps ; isso significa que queremos listar todos os containers que foram inicializados ou pausados. Para melhor visualização, formatei de uma forma diferente:

```
$ sudo docker ps -a
CONTAINER ID   = bc5fdb751dd1
IMAGE          = ubuntu:latest
COMMAND        = "/bin/bash"
CREATED        = 13 minutes ago
STATUS         = Exited (0) 4 minutes ago
PORTS          =
NAMES          = happy_fermi
```

Note o status que informa quando o container foi finalizado. Entretanto, o seu processo ainda existe e está pausado.

Uma outra opção que utilizaremos bastante é a `-q` , que retorna apenas o ID do container:

```
$ sudo docker ps -qa
bc5fdb751dd1
```

A partir da versão 1.5, o Docker possui a feature `stats` , que informa, em tempo de execução, detalhes sobre o nível de consumo de recursos na máquina host, feito pelos containers. Veja em código:

```
$ sudo docker stats bc5fdb751dd1
```

CONTAINER	CPU %	MEM USAGE/LIMIT	MEM %	NET I/O
bc5fdb751dd1	0.00%	5.098 MiB/7.686 GiB	0.06%	648 B/648 B

No primeiro exemplo, quando criamos o container, junto a ele geramos uma imagem. Imagens podem ser listadas com a opção `images` :

```
$ sudo docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	VIRTUAL SIZE
ubuntu	latest	eca7633ed783	37 hours ago	192.7 MB

Voltando à listagem de containers, removeremos o container criado com a opção `rm` – isso mesmo, igual ao comando do Linux! Junto ao comando, é necessário informar o nome ou ID:

```
$ sudo docker ps -qa
bc5fdb751dd1
$ sudo docker rm bc5fdb751dd1
bc5fdb751dd1
$ sudo docker ps -a
CONTAINER ID   IMAGE     COMMAND                  CREATED        STATUS        PORTS      NAMES
```

2.4 CONTROLANDO CONTAINERS

Ainda utilizando a imagem que foi construída no primeiro exemplo, veremos agora como vamos controlar o funcionamento dos containers. Para isso, vamos criar um novo container e fazer a instalação do web server Nginx:

```
$ sudo docker run -it --name ex_nginx ubuntu
root@943b6186d4f8:/#
```

Perceba o uso da opção `--name`, onde foi informado um apelido para o novo container criado. Agora que estamos no shell da nova instância, vamos atualizar os pacotes e, em seguida, instalar o Nginx:

```
root@943b6186d4f8:/# apt-get update
root@943b6186d4f8:/# apt-get install -y nginx
root@943b6186d4f8:/# nginx -v
nginx version: nginx/1.4.6 (Ubuntu)
root@943b6186d4f8:/# exit
```

É importante saber que toda e qualquer alteração realizada dentro de um container é volátil. Ou seja, se o finalizarmos, ao iniciar novamente essa instalação do Nginx, a alteração não vai permanecer.

Para tornar as alterações permanentes, é necessário *commitar* o container – sim, o Docker possui um recurso para versionamento igual ao Git. Sempre que um commit for feito, estaremos salvando o estado atual do container na imagem utilizada.

Para isso, basta usarmos o ID ou o apelido do container:

```
$ sudo docker commit f637c4df67a1 ubuntu/nginx  
78f4a58b0f314f5f81c2e986d61190fbcc01a9e378dcbebf7c6c16786d0a270c
```

Ao executar o *commit*, estamos criando uma nova imagem baseada na inicial. A nomenclatura utilizada `ubuntu/nginx` é o nome da nova imagem. Verifique com o `docker images` :

```
docker images  
REPOSITORY      TAG         IMAGE ID      CREATED        VIRTUAL SIZE  
ubuntu/nginx    latest     78f4a58b0f31  55 seconds ago 231.2 MB  
ubuntu          latest     eca7633ed783  38 hours ago   192.7 MB
```

Com a nova imagem gerada contendo o Nginx instalado, agora vamos explorar um pouco mais. É possível criar containers autodestrutivos com o uso da opção `--rm` . Isso significa que, ao finalizar a execução do container, ele será excluído automaticamente. Para exemplificar, vamos criar uma nova instância com os seguintes argumentos:

```
$ sudo docker run -it --rm -p 8080:80 ubuntu/nginx /bin/bash  
root@e98f73a63965:/# service nginx start  
root@e98f73a63965:/#
```

Observe o uso da opção `-p` . Com ela, estamos informando ao Docker que a porta `8080` no host será aberta e mapeada para a porta `80` no container, estabelecendo uma conexão entre eles. Dentro da nova instância, executei o comando `service nginx start` , para inicializar o Nginx.

Para entender melhor o `-p 8080:80`, podemos abrir o *browser* e acessar o seguinte endereço: <http://localhost:8080/>.



Figura 2.1: Nginx funcionando em um container

O acesso ao Nginx dentro do container está passando para a porta `8080` no host local.

De volta à nossa instância, vamos finalizar saindo do container, e ver o que o parâmetro `--rm` pode fazer:

```
root@e98f73a63965:/# exit
$ sudo docker ps -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
--------------	-------	---------	---------	--------	-------	-------

O container só existiu enquanto estávamos dentro dele. Outra opção bastante usada é o `-d`, pois envia toda a execução para *background*. Para testar, execute o código:

```
# docker run -d -p 8080:80 ubuntu/nginx /usr/sbin/nginx -g
"daemon off;"
574911d47ca2fc8da14e59843b0e319ca8899215f7ee3b9a1a41f708a72b7495
```

O processo do novo container foi enviado para background, e

o Docker retornou o ID da nova instância.

Podemos conferir se o container está em execução, ao executarmos o comando `ps -q` :

```
# docker ps -q
574911d47ca2
```

Ou simplesmente acessando no browser o endereço <http://localhost:8080/>.

Já temos uma instância em plano de fundo com o Nginx funcionando. Agora, vamos manipular o seu funcionamento utilizando as opções `stop` e `start` . Para isso, basta indicar o nome do container ou o seu ID:

```
$ sudo docker stop 574911d47ca2
574911d47ca2
$ sudo docker ps -q
$
$ sudo docker start 574911d47ca2
574911d47ca2
$ sudo docker ps -q
574911d47ca2
```

Note que, ao executar o `stop` , quando verificamos se o processo que faz referência ao container existe, nada é retornado. Isso acontece, pois o processo está em pausa.

2.5 DESTRUINDO IMAGENS E CONTAINERS

Já vimos como destruir containers utilizando a opção `rm` . Para destruir imagens, usamos o `rmi` , ao indicarmos o ID da imagem que queremos apagar:

```
$ sudo docker rmi 78f4a58b0f31
...
```

É possível apagar todos os containers e imagens de uma só vez. Para isso, basta um pouco de shell script:

```
```bash
$ sudo docker rm $(docker ps -qa)
574911d47ca2
```

O mesmo serve para apagar imagens:

```
$ sudo docker rmi $(docker images -q)
Untagged: ubuntu/nginx:latest
Deleted:
78f4a58b0f314f5f81c2e986d61190fbcc01a9e378dcbebf7c6c16786d0a270c
Untagged: ubuntu:latest
Deleted:
eca7633ed7835b71aa4712247b62fa5ed6ec37ce4aecde66ee80f646b3806c90
Deleted:
88fba6f3d2d898898cdfef1fcf03b19d357bef3fa307726c7aabea1ce08940bcd
Deleted:
67983a9b1599fc4e4d1050dd32194865e43536f63c554f5404b7309a9f1479f4
Deleted:
49bb1c57a82cb0c1784806374c050637de62f4265d0f1325e2c7a776ae73d1dc
Deleted:
c5fcd5669fa5f71149944169bc8bb242348b16d0597aec52cfb23d917df6972e
Deleted:
d497ad3926c8997e1e0de74cdd5285489bb2c4acd6db15292e04bbab07047cd0
Deleted:
511136ea3c5a64f264b78b5433614aec563103b4d4702f3ba7d4d2698e22c158
```

No próximo capítulo, veremos como podemos criar e gerenciar nossas imagens de forma programada.

# CONSTRUINDO MINHA PRIMEIRA IMAGEM

Antes de começar a executar comandos e partir para o ataque criando a sua primeira imagem, é preciso conhecer sobre o *Dockerfile*. Ele é um recurso feito para automatizar o processo de execução de tarefas no Docker.

Com este recurso, podemos descrever o que queremos inserir em nossa imagem. Desta forma, quando geramos um *build*, o Docker cria um *snapshot* com toda a instalação que elaboramos no Dockerfile.

## 3.1 COMO FUNCIONA A CONSTRUÇÃO DE UMA IMAGEM?

Antes de prosseguir, vamos entender como o Docker trabalha durante a etapa de construção de uma imagem. Quando criamos um container, o Docker monta o *rootfs* em modo *read-only*, igual ao processo de *boot* tradicional em qualquer Linux.

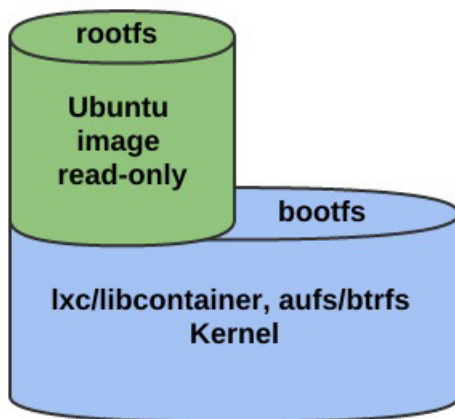


Figura 3.1: Layers durante criação de container

O *rootfs* (ou *root file system*) inclui a estrutura típica de diretórios ( `/dev` , `/proc` , `/bin` , `/etc` , `/lib` , `/usr` e `/tmp` ), além de todos os arquivos de configuração, binários e bibliotecas necessários, para o funcionamento de programas.

Em vez de alterar o sistema de arquivos para o modo *read-write*, o Docker tira proveito do *unionfs*, criando uma camada com permissões de *read-write* sobre a camada de *read-only*.



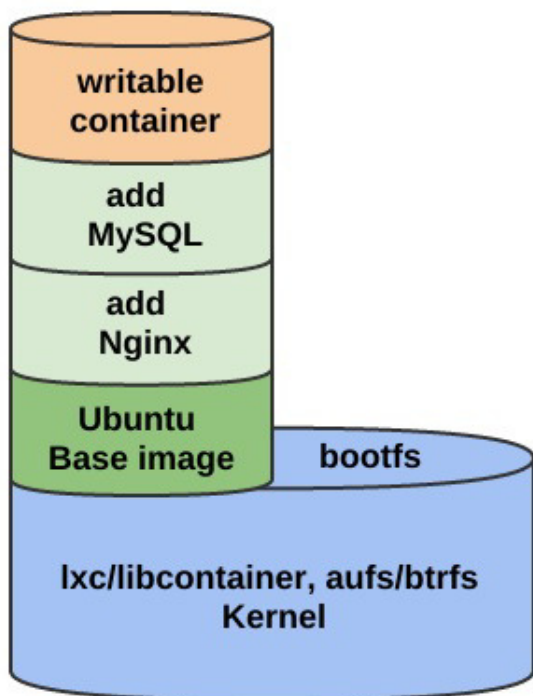


Figura 3.2: Modos de read-write e read-only na construção da imagem

Desta forma, a camada do topo é criada em modo *read-write* quando inicializamos um container. Se um programa for instalado, ele não vai existir na imagem base. Em vez disso, será criada uma nova camada sobre a base, contendo a instalação do programa.

Agora que ficou clara a forma como o Docker trabalha para construir uma imagem, podemos prosseguir.

## 3.2 EXPLORANDO O DOCKERFILE

O Dockerfile é um arquivo que aceita rotinas em shell script para serem executadas. Veja a seguir como ficaria o processo de instalação do Nginx, que fizemos no capítulo anterior, em um script Dockerfile:

```
FROM ubuntu

MAINTAINER Daniel Romero <infoslack@gmail.com>

RUN apt-get update

RUN apt-get install -y nginx
```

Vamos por partes:

- FROM – estamos escolhendo a imagem base para criar o container;
- MAINTAINER – especifica o nome de quem vai manter a imagem;
- RUN – permite a execução de um comando no container.

Note a simplicidade! Com apenas 4 linhas no nosso script, podemos gerar a nova imagem e instalar o Nginx.

Vamos continuar o processo e gerar a nossa primeira imagem com a opção `build` :

```
$ sudo docker build -t nginx .
```

O ponto no final indica o *path* do script de Dockerfile para gerar a imagem, supondo que você está executando o comando no mesmo diretório onde o Dockerfile se encontra.

Vamos conferir se a imagem realmente foi gerada da seguinte forma:

```
$ sudo docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	VIRTUAL SIZE
nginx	latest	37bb1a8d7ea6	58 seconds ago	231.2 MB
ubuntu	latest	eca7633ed783	39 hours ago	192.7 MB

O próximo passo é testarmos a imagem que foi gerada criando um container, e dizer que queremos utilizar a porta 8080 no nosso host, assim como fizemos no capítulo anterior.

```
$ sudo docker run -d -p 8080:80 nginx /usr/sbin/nginx -g
"daemon off;"
f885136907edca5215f9f2515be67da0b832d96616540f58b4ed1092f75f0e96
```

Mais uma vez, teste entrando em <http://localhost:8080/>, para garantir que está tudo funcionando.

Se preferir, teste direto no console utilizando o `curl`, para acessar a URL enviando um *request*. Assim, só precisamos observar o *status code*:

```
$ curl -IL http://localhost:8080
HTTP/1.1 200 OK
Server: nginx/1.4.6 (Ubuntu)
Date: Wed, 22 Oct 2014 22:09:49 GMT
Content-Type: text/html
Content-Length: 612
Last-Modified: Tue, 04 Mar 2014 11:46:45 GMT
Connection: keep-alive
ETag: "5315bd25-264"
Accept-Ranges: bytes
```

## 3.3 MAPEANDO PORTAS

O comando para execução de um container com Nginx está um pouco grande. Vamos reduzir a instrução de inicialização inserindo a opção `EXPOSE` no nosso Dockerfile. Além disso, faremos uso de um atalho para mapeamento de portas, o parâmetro `-P`.

No Dockerfile, adicionamos a instrução `EXPOSE` :

```
FROM ubuntu

MAINTAINER Daniel Romero <infoslack@gmail.com>

RUN apt-get update

RUN apt-get install -y nginx

EXPOSE 80
```

Em seguida, nós atualizamos a imagem executando o `build` novamente:

```
$ sudo docker build -t nginx .
Sending build context to Docker daemon 2.56 kB
Sending build context to Docker daemon
Step 0 : FROM ubuntu
----> eca7633ed783
Step 1 : MAINTAINER Daniel Romero <infoslack@gmail.com>
----> f6fe0fa267d2
Step 2 : RUN apt-get update
----> Using cache
----> 2ed51e9c4e64
Step 3 : RUN apt-get install -y nginx
----> Using cache
----> 37bb1a8d7ea6
Step 4 : EXPOSE 80
----> Running in 94b7bd18ac37
----> 3568a7119034
Removing intermediate container 94b7bd18ac37
Successfully built 3568a7119034
```

Note que o Docker apenas atualizou a imagem que já existia, acrescentando a instrução para expor a porta `80` sempre que um novo container for criado.

Agora, podemos testar criando uma nova instância:

```
$ sudo docker run -d -P nginx /usr/sbin/nginx -g "daemon off;"
0be882bb7472d85dfe1020a63683f855a033d037c0b2481a1f972158e868d533
```

Ao criar um container passando a instrução `-P` , estamos permitindo que o Docker faça o mapeamento de qualquer porta utilizada no container, para alguma outra porta no host. Conferiremos isso com o `docker ps` , ou usando o seguinte atalho:

```
$ sudo docker port 5df8942eeac5
80/tcp -> 0.0.0.0:49153
```

Ou podemos utilizar especificando a porta do container:

```
$ sudo docker port 5df8942eeac5 80
0.0.0.0:49153
```

Sempre que um container for criado desta forma, o retorno da instrução `-P` será uma porta aleatória criada no host. Se testarmos o acesso ao Nginx na porta criada, teremos o seguinte retorno:

```
$ curl -IL http://localhost:49153/
HTTP/1.1 200 OK
Server: nginx/1.4.6 (Ubuntu)
```

## 3.4 COPIANDO ARQUIVOS

Temos um pequeno problema ao utilizar a porta `8080` , pois o Nginx está configurado para utilizar a porta `80` . Podemos criar um container acessando o seu shell, atualizar a sua configuração, sair e commitar a imagem.

Imagine se você precisar fazer várias alterações nos arquivos de configuração do web server, a produtividade pode ser comprometida. Para resolver isso, vamos utilizar a opção `ADD` no Dockerfile. Desta forma, vamos referenciar o arquivo que queremos copiar e o local de destino para a imagem durante o

processo de *build*.

```
FROM ubuntu
```

```
MAINTAINER Daniel Romero <infoslack@gmail.com>
```

```
RUN apt-get update
```

```
RUN apt-get install -y nginx
```

```
ADD exemplo /etc/nginx/sites-enabled/default
```

```
EXPOSE 8080
```

Com a instrução `ADD`, o arquivo chamado `exemplo` será copiado para o diretório `/etc/nginx/sites-enabled`, e será chamado de `default`. O arquivo `exemplo` deve existir no mesmo contexto do `Dockerfile`. O seu conteúdo é bem simples e foi alterado apenas para o Nginx utilizar a nova porta, no nosso caso, a `8080`.

```
server {
 listen 8080 default_server;
 server_name localhost;

 root /usr/share/nginx/html;
 index index.html index.htm;
}
```

Agora, podemos executar o nosso processo de `build` e gerar a nova imagem, e o arquivo será copiado para as configurações do Nginx:

```
$ sudo docker build -t nginx .
Sending build context to Docker daemon 3.584 kB
Sending build context to Docker daemon
Step 0 : FROM ubuntu
----> eca7633ed783
Step 1 : MAINTAINER Daniel Romero <infoslack@gmail.com>
----> Using cache
----> f6fe0fa267d2
```

```
Step 2 : RUN apt-get update
---> Using cache
---> 2ed51e9c4e64
Step 3 : RUN apt-get install -y nginx
---> Using cache
---> 37bb1a8d7ea6
Step 4 : ADD exemplo /etc/nginx/sites-enabled/default
---> 3eb7bb07b396
Removing intermediate container 3d5b9c346517
Step 5 : EXPOSE 8080
---> Running in de3478bbb1ff
---> f12246bf988d
Removing intermediate container de3478bbb1ff
Successfully built f12246bf988d
```

Observe as etapas, estão na ordem em que colocamos no Dockerfile. Então, vamos criar um novo container, sem informar instruções de portas, e testar para ver se está tudo certo:

```
$ sudo docker run -d nginx /usr/sbin/nginx -g "daemon off;"
$ curl -IL http://localhost:8080
curl: (7) Failed to connect to localhost port 8080: Connection
refused
```

Como resposta, recebemos um erro em nosso teste. Será que esquecemos alguma coisa? Não, não esquecemos de nada.

Quando utilizamos o `-p` ou o `-P`, estamos mapeando uma porta interna do container para uma porta em nosso host local. Agora verificaremos o funcionamento do Nginx acessando o diretamente o IP do container.

Para capturar informações do container, podemos fazer uso da opção `inspect`, que retorna informações *low-level* de um container, ou de uma imagem.

Veremos em um capítulo posterior mais detalhes sobre a API do Docker. Por enquanto, vamos apenas analisar o retorno do

seguinte comando:

```
$ sudo docker inspect a6b4fe21aece
```

Como o retorno do comando é muito grande, você pode consultá-lo em <http://bit.ly/docker-inspect>.

Para continuarmos, vamos filtrar a saída usando o `grep` :

```
$ sudo docker inspect a6b4fe21aece | grep IPAddress
 "IPAddress": "172.17.0.70",
```

Aí está o IP do nosso container. Outra opção seria executar o comando `ifconfig` diretamente no container. A partir da versão 1.3 do Docker, temos a opção `exec` justamente para facilitar testes rápidos nos containers:

```
$ sudo docker exec -it a6b4fe21aece ifconfig eth0
eth0 Link encap:Ethernet HWaddr 02:42:ac:11:00:46
 inet addr:172.17.0.70 Bcast:0.0.0.0 Mask:255.255.0.0
 inet6 addr: fe80*42:acff:fe11:46/64 Scope:Link
 UP BROADCAST RUNNING MTU:1500 Metric:1
 RX packets:41 errors:0 dropped:0 overruns:0 frame:0
 TX packets:33 errors:0 dropped:0 overruns:0 carrier:0
 collisions:0 txqueuelen:1000
 RX bytes:4170 (4.1 KB) TX bytes:3569 (3.5 KB)
```

Bem, já temos o IP que está sendo utilizado, então podemos refazer o teste de *request* na porta `8080` , e verificar o funcionamento:

```
curl -IL http://172.17.0.70:8080
HTTP/1.1 200 OK
```

## 3.5 DEFININDO O DIRETÓRIO DE TRABALHO

A área de trabalho padrão do Docker é o diretório raiz `/` . Podemos alterar isso durante a criação de um container ao



usarmos a opção `-w` , ou tornando padrão usando a diretiva `WORKDIR` no Dockerfile.

Imagine que queremos utilizar um container para desenvolvimento de um projeto Ruby on Rails. O Dockerfile poderia ser adicionado dentro do diretório do projeto e teria as seguintes configurações:

```
FROM ubuntu

MAINTAINER Daniel Romero <infoslack@gmail.com>

RUN apt-get update

RUN apt-get install -y nginx

ADD exemplo /etc/nginx/sites-enabled/default

RUN echo "daemon off;" >> /etc/nginx/nginx.conf

ADD ./ /rails

WORKDIR /rails

EXPOSE 8080

CMD service nginx start
```

Com o diretório de trabalho definido em `WORKDIR` , temos antes dele o `ADD ./ /rails` , que copia os arquivos a partir do contexto no qual foi executado para o *filesystem* do container.

Lembre-se de que este arquivo Dockerfile, neste exemplo, está dentro do projeto Rails:

```
$ rails new docker_example
...
$ cp Dockerfile docker_example/
$ cd docker_example
$ sudo docker build -t nginx .
```

Com a nova imagem gerada, vamos criar um novo container:

```
$ sudo docker run -d -p 8080:8080 --name app nginx
```

Para entendermos o que de fato aconteceu nas instruções `ADD` e `WORKDIR`, que foram adicionadas ao `Dockerfile`, podemos acessar o container utilizando o `exec`:

```
$ sudo docker exec -it app bash
root@683062aa3c39:/rails# ls -la
total 80
drwxr-xr-x 12 root root 4096 Oct 29 02:12 .
drwxr-xr-x 22 root root 4096 Oct 29 02:19 ..
-rw-r--r-- 1 root root 466 Oct 29 02:04 .gitignore
-rw-r--r-- 1 root root 226 Oct 29 02:12 Dockerfile
-rw-r--r-- 1 root root 1339 Oct 29 02:04 Gemfile
-rw-r--r-- 1 root root 2782 Oct 29 02:05 Gemfile.lock
-rw-r--r-- 1 root root 478 Oct 29 02:04 README.rdoc
-rw-r--r-- 1 root root 249 Oct 29 02:04 Rakefile
drwxr-xr-x 8 root root 4096 Oct 29 02:04 app
drwxr-xr-x 2 root root 4096 Oct 29 02:05 bin
drwxr-xr-x 5 root root 4096 Oct 29 02:04 config
-rw-r--r-- 1 root root 154 Oct 29 02:04 config.ru
drwxr-xr-x 2 root root 4096 Oct 29 02:04 db
-rw-r--r-- 1 root root 127 Oct 29 02:12 exemplo
drwxr-xr-x 4 root root 4096 Oct 29 02:04 lib
drwxr-xr-x 2 root root 4096 Oct 29 02:04 log
drwxr-xr-x 2 root root 4096 Oct 29 02:04 public
drwxr-xr-x 8 root root 4096 Oct 29 02:04 test
drwxr-xr-x 3 root root 4096 Oct 29 02:04 tmp
drwxr-xr-x 3 root root 4096 Oct 29 02:04 vendor
root@683062aa3c39:/rails#
```

Basicamente agora a área de trabalho principal dentro deste container é a própria aplicação Rails. Desta forma, poderíamos usar este container como ambiente de desenvolvimento.

## 3.6 INICIALIZANDO SERVIÇOS

O comando utilizado para geração de containers ficou bem

menor. Entretanto, podemos reduzi-lo um pouco mais, informando no Dockerfile a instrução `CMD`, para que ele possa inicializar o Nginx sempre que um container novo for criado. Vamos à edição:

```
FROM ubuntu

MAINTAINER Daniel Romero <infoslack@gmail.com>

RUN apt-get update

RUN apt-get install -y nginx

ADD exemplo /etc/nginx/sites-enabled/default

RUN echo "daemon off;" >> /etc/nginx/nginx.conf

EXPOSE 8080

CMD service nginx start
```

Recrie a imagem, inicialize um container e faça o teste de *request*:

```
$ sudo docker build -t nginx .
$ sudo docker run -d nginx
$ curl -IL http://172.17.0.74:8080
HTTP/1.1 200 OK
Server: nginx/1.4.6 (Ubuntu)
```

Existe outra forma de inicializar serviços, utilizando `ENTRYPOINT`. Sempre que usamos `CMD` em seu background, ele está chamando o *bash* assim: `/bin/sh -c`. Em seguida, envia como parâmetro o comando ou instrução que especificamos.

A diferença em usar `ENTRYPOINT` é que ele chama o comando ou script diretamente, por exemplo:

```
...
EXPOSE 8080
```

```
ENTRYPOINT ["/usr/sbin/nginx"]
```

Quando utilizamos `ENTRYPOINT`, tudo o que for especificado em `CMD` será enviado como complemento para `ENTRYPOINT`:

```
...
ENTRYPOINT ["/etc/init.d/nginx"]

CMD ["start"]
```

Veremos mais detalhes sobre inicialização de serviços em outro capítulo.

## 3.7 TRATANDO LOGS

O Docker possui um recurso para visualizar os logs de saída e de erro padrão ( `stdout` e `stderr` ). Isso é interessante para verificarmos o que está acontecendo dentro de um container, sem a necessidade de conferir um determinado arquivo de log.

Para este exemplo, vamos utilizar o Dockerfile da seção anterior e redirecionar os logs do Nginx para `stdout` e `stderr`. O Dockerfile ficará assim:

```
FROM ubuntu

MAINTAINER Daniel Romero <infoslack@gmail.com>

RUN apt-get update

RUN apt-get install -y nginx

ADD exemplo /etc/nginx/sites-enabled/default

RUN ln -sf /dev/stdout /var/log/nginx/access.log
RUN ln -sf /dev/stderr /var/log/nginx/error.log

EXPOSE 80
```

```
CMD ["nginx", "-g", "daemon off;"]
```

Recrie a imagem com `build` e, em seguida, um container:

```
$ docker build -t nginx .
$ docker run -d -p 80:80 nginx
```

Faça alguns *requests* para gerar registros nos logs:

```
$ for ((i=1; i<=10; i++)); do curl -IL http://127.0.0.1; done
...
HTTP/1.1 200 OK
Server: nginx/1.4.6
```

Agora, podemos conferir os logs utilizando o comando `docker logs` e a identificação do nosso container:

```
$ docker logs 2e0841de2c17
...
127.0.0.1 - [04/01/15:23:18:16]
"HEAD / HTTP/1.1" 200 "curl/7.35.0"
127.0.0.1 - [04/01/15:23:19:53]
"HEAD / HTTP/1.1" 200 "curl/7.35.0"
127.0.0.1 - [04/01/15:23:19:53]
"HEAD / HTTP/1.1" 200 "curl/7.35.0"
127.0.0.1 - [04/01/15:23:19:53]
"HEAD / HTTP/1.1" 200 "curl/7.35.0"
...
```

Desta forma, conseguimos capturar os logs do serviço Nginx de dentro do container sem precisar abrir os arquivos de log, pois todo o log está sendo redirecionado de forma que a opção `logs` do Docker consiga acompanhar.

## 3.8 EXPORTAÇÃO E IMPORTAÇÃO DE CONTAINERS

Imagine que temos um container em execução e queremos

exportá-lo para outro host. Outra situação parecida é um container funcionando localmente, porém queremos levá-lo para um host em produção.

Podemos criar uma imagem partindo de um container que está funcionando, e gerar um arquivo `.tar` usando a opção `save`. Para criar a nova imagem, basta fazer um commit no container em execução:

```
$ sudo docker ps -q
b02af9430141
$ sudo docker commit b02af9430141 nova_imagem
9a8c0a1a72d2f9c2815e455a22be91f9cf788f769d2cca5b603baebd10ccc38a
```

De posse da nova imagem que foi gerada, faremos a exportação criando um arquivo:

```
$ sudo docker save nova_imagem > /tmp/nova_imagem.tar
```

Agora o arquivo `.tar` que foi criado pode ser enviado para o outro host, via `SCP`, por exemplo. Para fazer a importação desse arquivo, utilizamos a opção `load`:

```
$ sudo docker load < /tmp/nova_imagem.tar
```

Este fluxo pode ser usado em operações para projetos privados. Veremos outras opções no capítulo *Docker Hub*.

# TRABALHANDO COM VOLUMES

Um volume pode ser um diretório localizado fora do sistema de arquivos de um container. O Docker permite especificar diretórios no container para que possam ser mapeados no sistema de arquivos do host. Com isso, temos a capacidade de manipular dados no container sem que tenham relação alguma com as informações da imagem. Um volume pode ser compartilhado entre vários containers. A figura a seguir ilustra como os volumes funcionam:

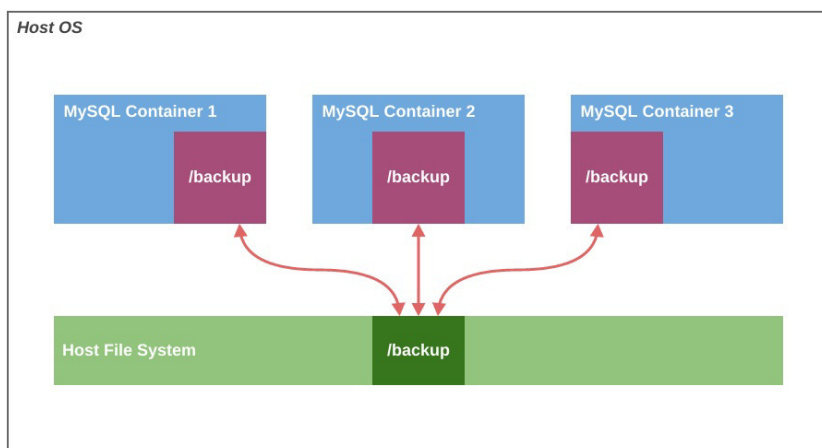


Figura 4.1: O funcionamento de Docker Volumes

## 4.1 GERENCIANDO OS DADOS

Volumes são diretórios configurados dentro de um container que fornecem o recurso de compartilhamento e persistência de dados.

Antes de continuar, precisamos entender três coisas:

- Os volumes podem ser compartilhados ou reutilizados entre containers;
- Toda alteração feita em um volume é de forma direta;
- Volumes alterados não são incluídos quando atualizamos uma imagem.

Ainda utilizando a imagem criada para o Nginx, podemos testar o uso de volumes com a opção `-v`. Com ela, é possível informar um diretório no host local que poderá ser acessado no container, funcionando de forma parecida com um mapeamento. Além disso, vamos informar o tipo de permissão que o container terá sob o diretório local, sendo: `ro` para somente leitura, e `rw` para leitura e escrita.

Utilizando a opção `-v`, veja na prática como isso funciona:

```
docker run -d -p 8080:8080 -v
/tmp/nginx:/usr/share/nginx/html:ro nginx
c88120d5efbeb0607ea861ad9f0d9a141eb4ecb18a518a096f9ad38ebe9207bd
```

O diretório `/tmp/nginx` pertence ao host local, e quero que ele seja mapeado para `/usr/share/nginx/html` dentro do container que está sendo inicializado. Além disso, estou passando a instrução `ro` no final para informar ao container que ele só poderá ler o conteúdo em `/tmp/nginx`.



Para testar o volume criado, vamos gerar um arquivo em `/tmp/nginx` e inserir algum texto. Entretanto, antes note o retorno de um *request* feito ao acessarmos o web server:

```
$ curl -IL http://localhost:8080
HTTP/1.1 403 Forbidden
Server: nginx/1.4.6 (Ubuntu)
```

Agora, criando o arquivo no host e testando novamente:

```
$ echo "It works!" > /tmp/nginx/index.html
$ curl http://localhost:8080
It works!
```

## 4.2 UTILIZANDO VOLUMES NO DOCKERFILE

Imagine que temos um container com um banco de dados em execução. Nele, é possível controlar os dados que são armazenados em disco para fazer backups e até restaurações.

Para isso, devemos separar a construção deste novo Dockerfile. Antes de começar, crie uma pasta chamada `mysql` e um novo arquivo Dockerfile com as seguintes instruções de instalação:

```
FROM ubuntu
MAINTAINER Daniel Romero <infoslack@gmail.com>

ENV DEBIAN_FRONTEND noninteractive
RUN apt-get update -qq && apt-get install -y mysql-server-5.5

ADD my.cnf /etc/mysql/conf.d/my.cnf
RUN chmod 664 /etc/mysql/conf.d/my.cnf

ADD run /usr/local/bin/run
RUN chmod +x /usr/local/bin/run

VOLUME ["/var/lib/mysql"]

EXPOSE 3306
```

CMD ["/usr/local/bin/run"]

Não se preocupe com os arquivos `run` e `my.cnf` ; eles estão sendo adicionados ao Dockerfile apenas para tornar a instalação e execução do MySQL mais simples.

Bem, aqui temos duas novidades. A primeira é o uso da opção `ENV` para declarar uma variável ambiente, que neste caso será utilizada no processo de instalação do MySQL, a fim de evitar telas interativas que são exibidas durante a instalação para cadastrar usuários e configurar o banco de dados.

E a segunda é que estamos informando que o diretório `/var/lib/mysql` será um `VOLUME` ; assim, poderemos manipular os arquivos de dados salvos pelo MySQL.

Agora, vamos gerar uma nova imagem chamada `mysql` , e inicializar um container:

```
$ sudo docker build -t mysql .
$ sudo docker run -d -p 3306:3306 -e MYSQL_ROOT_PASSWORD=xpto1234
mysql
```

A variável `MYSQL_ROOT_PASSWORD` existe no script de execução `run` , que foi copiado quando geramos a imagem. A opção `-e` serve para designar valores a variáveis ambiente – neste caso, o valor que ela recebe é a senha de `root` para o acesso ao MySQL.

Para testar se o container está funcionando perfeitamente, podemos acessar o MySQL partindo do host local:

```
$ mysql -h 127.0.0.1 -u root -p
Enter password:
Welcome to the MySQL monitor. Commands end with ; or \g.
```

```
Your MySQL connection id is 1
Server version: 5.5.40-0ubuntu0.14.04.1-log (Ubuntu)
```

```
Copyright (c) 2000, 2014, Oracle, Monty Program Ab and others.
```

```
Type 'help;' or '\h' for help.
Type '\c' to clear the current input statement.
```

```
mysql>
```

Agora, com o container em execução e o teste feito, vamos focar no uso de volumes.

Por padrão, os dados armazenados pelo MySQL residem em `/var/lib/mysql`, justamente onde inicializamos o nosso volume. Assim, é possível criarmos um novo container que terá acesso a esses dados. Para isso, podemos utilizar a opção `--volumes-from`, que recebe como argumento o ID ou nome do container a cujos dados queremos ter acesso.

No exemplo a seguir, usaremos uma imagem mínima chamada `busybox` para ter acesso aos dados do container que está em execução com MySQL:

```
$ sudo docker run -i -t --volumes-from e1c9c12c71ed busybox
Unable to find image 'busybox' locally
busybox:latest: The image you are pulling has been verified

df7546f9f060: Pull complete
e433a6c5b276: Pull complete
e72ac664f4f0: Pull complete
511136ea3c5a: Already exists
Status: Downloaded newer image for busybox:latest
/ #
```

O Docker inicializou o novo container utilizando a imagem `busybox`, e mapeou o volume do container `mysql` para a nova instância que foi criada. Isso pode ser visto ao acessarmos o

diretório do `mysql` no novo container:

```
/ # ls -la /var/lib/mysql/
total 28692
drwx----- 4 102 105 4096 Oct 25 18:00 .
drwxrwxr-x 4 root root 4096 Oct 25 18:01 ..
-rw-r--r-- 1 102 105 0 Oct 24 01:01 debian-5.5.flag
-rw-rw---- 1 102 105 5242880 Oct 25 18:00 ib_logfile0
-rw-rw---- 1 102 105 5242880 Oct 24 01:01 ib_logfile1
-rw-rw---- 1 102 105 18874550 Oct 25 18:00 ibdata1
drwx----- 2 102 105 4096 Oct 24 01:01 mysql
drwx----- 2 102 105 4096 Oct 25 18:00 performance_schema
/ #
```

Temos acesso aos dados que estão sendo gravados em disco pelo `mysql`. O mesmo poderia ser feito com arquivos de logs, por exemplo. Além disso, a opção `--volumes-from` pode ser usada para replicar esse volume em múltiplos containers.

É interessante saber que se você remover o container que monta o volume inicial – ou seja, o do `mysql` –, os posteriores que foram inicializados com a opção `--volumes-from` não serão excluídos.

Para remover o volume do disco, você pode utilizar a opção `-v` combinada com o `rm`, como `docker rm -v volume_name`.

## 4.3 BACKUP E RESTORE DE VOLUMES

Outra forma de uso para `--volumes-from` pode ser para a construção de backups. Imagine que você queira realizar um backup do diretório `/var/lib/mysql` do container para o seu host local:

```
$ sudo docker run --volumes-from e1c9c12c71ed -v \
$(pwd):/backup ubuntu tar cvf /backup/backup.tar /var/lib/mysql
$ ls -la
```

```
total 29988
drwxr-xr-x 2 root root 4096 Oct 25 15:22 ./
drwxrwxrwt 1025 root root 225280 Oct 25 15:16 ../
-rw-r--r-- 1 root root 30474240 Oct 25 15:22 backup.tar
```

Por partes: primeiro estamos criando um novo container com acesso ao volume de dados do `mysql` e, em seguida, estamos criando um mapeamento entre o nosso host local e o novo container, de forma que tudo o que for criado no diretório mapeado dentro deste novo container esteja disponível para o nosso host local. Observe a figura a seguir:

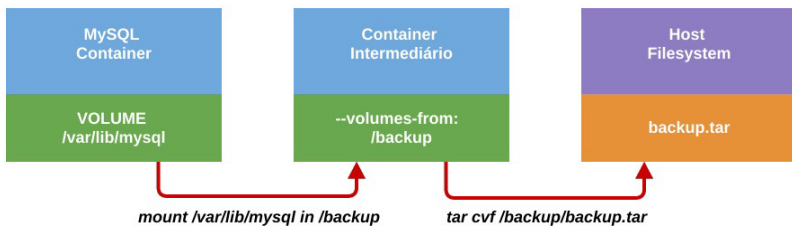


Figura 4.2: Realizando backup de volumes

O container intermediário é criado usando a imagem base do Ubuntu e, por último, executamos a compactação e cópia dos dados. Resumindo: o novo container copia os dados da instância que está com o `mysql` ativo, compacta tudo e disponibiliza para o nosso host local.

Para entender como o *restore* funciona, faremos um pequeno teste, criando um banco no `mysql` :

```
$ mysql -h 127.0.0.1 -u root -p
mysql> create database example;
Query OK, 1 row affected (0.00 sec)
mysql> use example;
Database changed
mysql> create table docker (
```

```

-> id int,
-> name varchar(200)
->);
Query OK, 0 rows affected (0.03 sec)
mysql> show tables;
+-----+
| Tables_in_example |
+-----+
| docker |
+-----+
1 row in set (0.00 sec)

```

Agora que temos uma estrutura de exemplo, se destruírmos o container que está executando o `mysql` sem salvar os dados persistidos no volume, perderemos o exemplo criado. Antes de destruir esse container, vamos executar o backup novamente:

```

$ sudo docker run --volumes-from e1c9c12c71ed -v \
$(pwd):/backup ubuntu tar cvf /backup/backup.tar /var/lib/mysql

```

Vamos remover o container do `mysql` e criar um novo, para ver o que acontece com os dados no volume `/var/lib/mysql`:

```

$ sudo docker rm -f e1c9c12c71ed

```

A opção `-f` serve para forçar a remoção do container que está em execução. Isso funciona como um atalho para evitar a instrução `stop` antes de remover.

Com o container excluído, podemos gerar um novo, acessar o `mysql` e conferir se o exemplo criado ainda existe:

```

$ sudo docker run -d -p 3306:3306 -e MYSQL_ROOT_PASSWORD=xpto1234
mysql
9bcf9f5a6a2e2f639c58636fcef9278b75b32cbf9c34253e220ad3fb1140215d
$ mysql -h 127.0.0.1 -u root -p
mysql> show databases;
+-----+
| Database |
+-----+

```

```
| information_schema |
| mysql |
| performance_schema |
+-----+
3 rows in set (0.00 sec)
```

O banco `example` não existe mais. Então, vamos restaurar o último backup realizado e ver o que acontece. Para isso, faremos o caminho inverso, informando que o container criado utilizará o diretório mapeado em nosso host local para descompactar o arquivo `backup.tar`, dentro do volume mapeado no container do `mysql`:

```
$ sudo docker run --volumes-from 9bcf9f5a6a2e -v \
$(pwd):/backup busybox tar xvf /backup/backup.tar
```

Desta vez, usamos a imagem `busybox` apenas para diferenciar; o efeito é o mesmo.

Voltando ao `mysql`, podemos verificar se o banco `example` existe:

```
$ mysql -h 127.0.0.1 -u root -p
mysql> show databases;
+-----+
| Database |
+-----+
| information_schema |
| example |
| mysql |
| performance_schema |
+-----+
4 rows in set (0.00 sec)
mysql> use example;
Database changed
mysql> show tables;
+-----+
| Tables_in_example |
+-----+
| docker |
+-----+
```

1 row in set (0.00 sec)

Tudo funciona! Podemos utilizar estas opções para automatizar backups, migrações e restaurações com qualquer ferramenta de nossa preferência.



# CONFIGURAÇÕES DE REDE

Por padrão, sempre que o Docker é inicializado, uma interface de rede virtual chamada `docker0` é criada na máquina host. Então, de forma aleatória, o Docker escolhe um endereço privado de IP e uma máscara de sub-rede, e os configura na interface virtual `docker0`.

Essa interface virtual nada mais é que uma ponte virtual que encaminha os pacotes de forma automática para quaisquer outras interfaces que estejam conectadas. Ou seja, os containers conseguem estabelecer comunicação com o host e entre si.

## 5.1 COMUNICAÇÃO ENTRE CONTAINERS

Para a comunicação entre containers funcionar, vai depender de dois fatores:

- A topologia de rede do host deve permitir a conexão das interfaces de rede dos containers, usando a ponte `docker0`;
- O `iptables` deve permitir as ligações especiais que serão feitas – normalmente, são regras de

## redirecionamento de pacotes.

Na sua configuração padrão, o Docker utiliza apenas uma interface ponte para concentrar todas as comunicações. Além disso, ele providencia as regras de *firewall* no *iptables* para prover as rotas de tráfego. Isso pode ser conferido ao inicializarmos dois containers:

```
$ sudo docker run -d -p 8080:8080 nginx
93118bcfea03c4eec377b0463e4a9364c26000ec56812b560642aed19b2101af
$ sudo docker run -d -p 3306:3306 -e MYSQL_ROOT_PASSWORD=xpto1234
mysql
208a0476edc9955fd493803ff6f656efcfbb10730b1260be4e242da88a2c7ab3
```

Agora, na máquina host, podemos verificar as regras criadas pelo Docker no *iptables* :

```
$ sudo iptables -L -n
...
Chain FORWARD (policy ACCEPT)
target prot opt source destination
ACCEPT tcp -- 0.0.0.0/0 172.17.0.9 tcp dpt:8080
ACCEPT tcp -- 0.0.0.0/0 172.17.0.8 tcp dpt:3306
```

O retorno da verificação no *iptables* exibe o envio de pacotes do host local para os IPs de destino, em suas respectivas portas. Podemos, assim, realizar alguns testes de um container para outro, utilizando a opção *exec* . Veremos a seguir a comunicação entre os dois containers criados:

```
$ sudo docker exec -it 93118bcfea03 ping 172.17.0.8
PING 172.17.0.8 (172.17.0.8) 56(84) bytes of data.
64 bytes from 172.17.0.8: icmp_seq=1 ttl=64 time=0.060 ms
64 bytes from 172.17.0.8: icmp_seq=2 ttl=64 time=0.133 ms
64 bytes from 172.17.0.8: icmp_seq=3 ttl=64 time=0.132 ms

--- 172.17.0.8 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 1999ms
rtt min/avg/max/mdev = 0.060/0.108/0.133/0.035 ms
```

Ao usarmos a opção `exec`, foi solicitado para que o container `nginx` realizasse um *ping* no IP do container `mysql`, apenas para confirmar a comunicação estabelecida entre eles. Veremos na seção *Comunicação de containers no mesmo host* o comportamento desta comunicação em uma aplicação real.

## 5.2 ALTERANDO A CONFIGURAÇÃO DEFAULT

Como vimos, por padrão o Docker inicializa e configura a interface de rede virtual `docker0`. Veremos agora como alterar essas configurações no processo da sua inicialização.

Vamos especificar uma faixa IP e uma máscara de rede de nossa preferência para serem usadas na interface `docker0`. Para isso, o parâmetro `--bip=` pode ser usado antes de inicializar o Docker. Outra opção é a de restringir um *range* de IPs privados utilizando `--fixed-cidr=`.

Verificaremos a comunicação da interface virtual `docker0` *bridge* (ponte) com as interfaces virtuais dos containers, utilizando o comando `brctl show`:

```
$ sudo brctl show
```

bridge name	bridge id	STP enabled	interfaces
docker0	8000.show56847afe9799	no	vethcca6472 vethd963455

Com o `brctl`, vemos que, cada vez que um novo container é criado, o Docker seleciona um IP disponível em *bridge*, configura o IP escolhido na interface `eth0` do container, e define todo o mapeamento.

Todo esse comportamento pode ser alterado, até mesmo o nome da *bridge* pode ser escolhido. Para alterar, vamos utilizar a opção `-b`, ou `--bridge=`. Para definir estas configurações, primeiro temos de parar o serviço do Docker e, em seguida, eliminar a interface `docker0`. Veja em código:

```
$ sudo service docker stop
$ sudo ip link set dev docker0 down
$ sudo brctl delbr docker0
```

Note que, antes de remover a interface `docker0` com o comando `brctl`, ela foi desligada. Agora, podemos criar a nossa própria *bridge*:

```
$ sudo brctl addbr bridge0
$ sudo ip addr add 192.168.13.1/24 dev bridge0
$ sudo ip link set dev bridge0 up
```

O resultado da nossa nova configuração deve ficar parecido com o seguinte no nosso host:

```
$ sudo ifconfig bridge0
bridge0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
 inet 192.168.13.1 netmask 255.255.255.0 broadcast 0.0.0.0
 inet6 fe80*b8a9:3dff:fe4a:c845 prefixlen 64
 scopeid 0x20<link>
 ether ba:a9:3d:4a:c8:45 txqueuelen 0 (Ethernet)
 RX packets 0 bytes 0 (0.0 B)
 TX errors 0 dropped 0 overruns 0 frame 0
 TX packets 8 bytes 648 (648.0 B)
 TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
```

Agora, só precisamos informar ao Docker que ele deve usar a nova interface virtual criada antes de ser inicializado novamente. Para isso, vamos adicionar as opções que vimos no arquivo de configurações do Docker. Geralmente, esse arquivo reside em `/etc/default/docker`:

```
$ echo 'DOCKER_OPTS="-b=bridge0"' >> /etc/default/docker
```

Com a nova configuração, agora vamos inicializar o Docker, criar alguns containers e verificar se estão utilizando os novos endereços IP que definimos:

```
$ sudo service docker start
$ sudo docker run -d -p 8080:8080 nginx
$ sudo docker run -d -p 3306:3306 -e MYSQL_ROOT_PASSWORD=xpto1234
mysql
$ sudo docker exec -it afef56adadf5 ifconfig eth0
eth0 Link encap:Ethernet HWaddr 02:42:c0:a8:0d:02
 inet addr:192.168.13.2 Bcast:0.0.0.0 Mask:255.255.255.0
...
```

E o novo IP atribuído ao container `nginx` faz uso da faixa que definimos manualmente apenas para verificar se está tudo funcionando. Teste a comunicação entre ele e o container `mysql` :

```
$ sudo docker exec -it e17d51e4a156 ping 192.168.13.2
PING 192.168.13.2 (192.168.13.2) 56(84) bytes of data.
64 bytes from 192.168.13.2: icmp_seq=1 ttl=64 time=0.195 ms
64 bytes from 192.168.13.2: icmp_seq=2 ttl=64 time=0.137 ms
```

O resultado da nossa configuração é que o Docker agora está preparado para operar com a nova *bridge* e delegar os IPs para os containers com a faixa que optamos.

## 5.3 COMUNICAÇÃO DE CONTAINERS NO MESMO HOST

Vimos na primeira seção deste capítulo um pouco sobre a comunicação de containers. Agora veremos, de forma prática, como uma aplicação faz uso dessa comunicação.

No exemplo, vamos criar dois containers: um para Nginx, que chamaremos de `app` , e outro para MySQL, que nomearemos como `db` apenas. O objetivo é estabelecer acesso entre os

containers de forma que `app` possa executar instruções em `db` .

Antes de tudo, precisamos criar os dois containers:

```
$ sudo docker run -d --name app nginx
$ sudo docker run -d -e MYSQL_ROOT_PASSWORD=xpto1234 --name db
mysql
```

Agora podemos acessar o shell do container `app` usando a opção `exec` e, em seguida, instalar o pacote `mysql-client-5.5` :

```
$ sudo docker exec -it app bash
root@1500621d9024:/# apt-get install -y mysql-client-5.5
```

Com o `mysql-client-5.5` instalado no container `app` , agora podemos testar o acesso ao MySQL no container `db` :

```
root@1500621d9024:/# mysql -h 192.168.13.10 -u root -p
Enter password:
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 1
Server version: 5.5.40-0ubuntu0.14.04.1-log (Ubuntu)

Copyright (c) 2000, 2014, Oracle and/or its affiliates.
All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help.
Type '\c' to clear the current input statement.

mysql>
```

Lembre-se apenas de verificar os IPs atribuídos aos containers.

## Linkando containers

Uma outra forma de estabelecer comunicação entre containers no mesmo host é utilizando a opção `link` . Essa opção

providencia o mapeamento de rede entre os containers que estão sendo *linkados*.

Refazendo o último exemplo, teríamos o mesmo resultado desta forma:

```
$ sudo docker run -d -e MYSQL_ROOT_PASSWORD=xpto1234 --name mysql
mysql
$ sudo docker run -d --name app --link mysql:db nginx
```

Note que na opção `link` é informado o nome do container que queremos linkar e um apelido para ele – neste caso, `mysql` é o nome do container, e `db` o seu apelido.

Verificaremos se o link está funcionando entrando no container `app` e testando a comunicação com o container `mysql`, através do apelido que criamos:

```
$ sudo docker exec -it app bash
root@c358d19468df:/# ping db
PING db (172.17.0.89) 56(84) bytes of data.
64 bytes from mysql (172.17.0.89): icmp_seq=1 ttl=64 time=0.211ms
64 bytes from mysql (172.17.0.89): icmp_seq=2 ttl=64 time=0.138ms
64 bytes from mysql (172.17.0.89): icmp_seq=3 ttl=64 time=0.137ms

--- db ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2000ms
rtt min/avg/max/mdev = 0.137/0.162/0.211/0.034 ms
root@c358d19468df:/#
```

Veremos mais detalhes sobre o uso da opção `link` em outro capítulo, mais à frente.

## 5.4 COMUNICAÇÃO DE CONTAINERS EM HOSTS DIFERENTES

Até agora, vimos como funciona a comunicação entre

containers no mesmo host. Agora, imagine que temos vários containers funcionando e estes estão espalhados em diferentes hosts, e precisamos estabelecer a comunicação de rede entre eles.

Supondo que temos dois hosts com IPs diferentes, em um dos hosts temos o container `app` e, no outro, o `db`. Precisamos configurar o Docker de forma que ele mantenha uma rota direta entre os containers. Para isso, criaremos uma rota entre as interfaces privadas dos hosts. A configuração será parecida com o seguinte esquema:

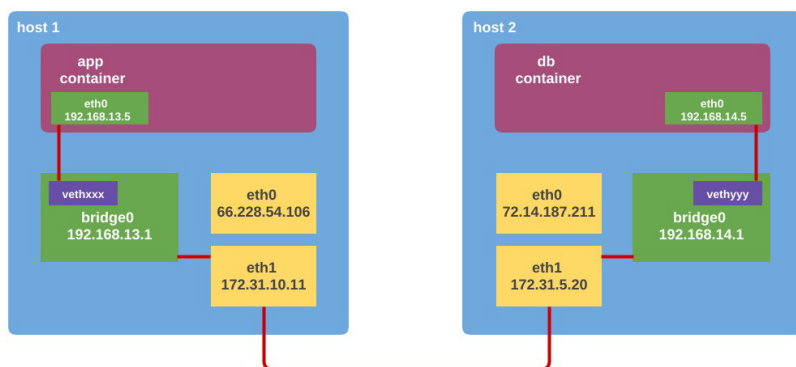


Figura 5.1: Comunicando containers de hosts diferentes

Em cada uma das instâncias, vamos configurar uma faixa IP diferente para ser usada por *bridge*. Então, na instância de `app`, pode ser feito da seguinte forma:

```
$ sudo brctl addbr bridge0
$ sudo ifconfig bridge0 192.168.13.1 netmask 255.255.255.0
$ echo 'DOCKER_OPTS="-b=bridge0"' >> /etc/default/docker
$ sudo service docker restart
```

Já na instância de `db`, vamos criar a *bridge* com outra faixa IP:



```
$ sudo brctl addbr bridge0
$ sudo ifconfig bridge0 192.168.14.1 netmask 255.255.255.0
$ echo 'DOCKER_OPTS="-b=bridge0"' >> /etc/default/docker
$ sudo service docker restart
```

Agora, precisaremos adicionar as rotas em cada uma das instâncias, de forma que exista conexão entre os IPs 192.168.13.1 e 192.168.14.1 . A rota que será adicionada deve trafegar os pacotes usando a rede privada. No host `app` , vamos criar a rota desta forma:

```
$ sudo route add -net 192.168.14.0 netmask 255.255.255.0 gw 172.31.5.20
```

Já no host de `db` , podemos adicionar a outra rota:

```
$ sudo route add -net 192.168.13.0 netmask 255.255.255.0 gw 172.31.10.11
```

Em um teste rápido, verificamos a comunicação funcionando entre os containers:

```
$ sudo docker exec -it app ping 192.168.14.5
PING 192.168.14.5 (192.168.14.5) 56(84) bytes of data.
64 bytes from 192.168.14.5: icmp_seq=1 ttl=64 time=0.165 ms
64 bytes from 192.168.14.5: icmp_seq=2 ttl=64 time=0.147 ms
```

## 5.5 COMUNICAÇÃO COM AUTOMAÇÃO

Na seção anterior, foi possível notar que estabelecer a comunicação entre containers existentes em hosts separados não é uma tarefa muito trivial. Todo esse trabalho pode ser minimizado utilizando o *Weave*, um serviço projetado para automatizar o processo de configuração de redes entre os hosts, e prover a comunicação entre containers.

A instalação é simples e pode ser feita desta forma:

```
$ sudo wget -O /usr/local/bin/weave \
 https://github.com/zettio/weave/releases/download/
 latest_release/weave
$ sudo chmod a+x /usr/local/bin/weave
```

Seguindo o exemplo da seção anterior, após instalar o Weave em ambos hosts, poderíamos inicializá-lo primeiro no host do container app ( host1 ):

```
host1$ weave launch
host1$ APP_CONTAINER=$(weave run 10.2.1.1/24 -t -i --name app
nginx)
```

O primeiro comando inicia o weave em um novo container e já baixa a docker image , caso seja necessário. O segundo roda o nosso container app , usando o comando weave run , que basicamente roda o docker run -d , mas já passando os parâmetros de IP automaticamente.

Em seguida, devemos fazer o mesmo no host do container db ( host2 ), informando a outra faixa IP:

```
host2$ weave launch 172.31.10.11
host2$ DB_CONTAINER=$(weave run 10.2.1.2/24 -t -i \
-e MYSQL_ROOT_PASSWORD=xpto1234 --name mysql mysql)
```

O 172.31.10.11 é o IP de host1 e, no lugar dele, também poderíamos ter colocado o hostname . Precisamos fazer isso para que o weave do host2 saiba que precisa se comunicar com o weave do host1 . Note que isso poderia ser feito ao contrário também, iniciando primeiro o weave no host2 e, depois, no host1 , passando o IP do host2 .

Para testar, vamos entrar no container app do host1 e enviar um ping para o IP do container db que está rodando no host2 :

```
host1$ docker attach $APP_CONTAINER
root@28841bd02eff:/# ping -c 1 -q 10.2.1.2
PING 10.2.1.2 (10.2.1.2): 48 data bytes
--- 10.2.1.2 ping statistics ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max/stddev = 1.048/1.048/1.048/0.000 ms
```

É importante citar que os dois hosts ( host1 e host2 ) precisam ter a porta 6783 acessível para TCP e UDP , para que o weave funcione.

# DOCKER HUB

Até agora, vimos como usar a linha de comando para executar o Docker no host local e remoto. Aprendemos a utilizar imagens prontas e como criar imagens próprias.

O *Docker Hub* (<http://hub.docker.com>) é um registro público que contém milhares de imagens para baixar e usar na construção de containers. Ele conta com autenticação, organização de grupos de trabalho e ferramentas para melhorar o fluxo de trabalho, além de repositórios privados para armazenar imagens que você não deseja compartilhar publicamente.

## 6.1 BUSCANDO E PUXANDO IMAGENS

Por padrão, o Docker vem configurado com acesso somente leitura ao Docker Hub, e podemos pesquisar na linha de comando por imagens nele ao usarmos a opção `search` :

```
$ sudo docker search rails
NAME DESCRIPTION STARS OFFICIALAUTOMATED
rails Rails is an open-source web... 75 [OK]
jonh/rails Ubuntu 12.04 LTS, Rails 4.0... 6
...
```

Uma vez que encontramos a imagem que estávamos a procura, vamos baixá-la com o comando `pull` :

```
$ sudo docker pull rails
rails:latest: The image you are pulling has been verified
848d84b4b2ab: Pull complete
71d9d77ae89e: Pull complete
9c7db00d9eab: Pull complete
e512573c88ba: Pull complete
aeb49701ff5e: Pull complete
511136ea3c5a: Already exists
Status: Downloaded newer image for rails:latest
```

Perceba que, anteriormente, quando usamos a imagem do Ubuntu no nosso primeiro exemplo do livro, o que o Docker fez por baixo dos panos para nós foi: verificar que não tínhamos a imagem localmente, procurar por uma imagem no Docker Hub que possuía o nome `ubuntu`, baixá-la (fazendo `pull`) automaticamente, e aí sim criar o nosso container.

## 6.2 ENVIANDO IMAGENS PARA UM REPOSITÓRIO

Para que possamos enviar nossas imagens para o Docker Hub, precisamos criar uma nova conta (caso você ainda não possua), ou fazer login em uma já existente:

```
$ sudo docker login
Username: infoslack
Password:
Email: infoslack@gmail.com
Login Succeeded
```

Com o login feito, vamos enviar nossas imagens geradas para Docker Hub e fazer uso delas nos projetos em produção.

Para enviar um repositório para o Docker Hub, usamos a opção `push`. O formato deve obedecer a seguinte regra: `username/image`. Esse formato é elaborado durante o processo

de build da imagem, utilizando a opção `-t` :

```
$ sudo docker build -t="infoslack/mysql:v1" .
```

Observe a identificação da imagem usando o nome do usuário. A opção `:v1` no final é uma tag – no exemplo, a nova imagem gerada `infoslack/mysql` está marcada como `v1` :

```
$ sudo docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	VIRTUAL SIZE
mysql	latest	937abb0828b6	15 minutes ago	342.9 MB
infoslack/mysql	v1	937abb0828b6	15 minutes ago	342.9 MB

Também podemos adicionar e alterar tags em imagens já criadas. Para isso, basta usar a opção `tag` e o ID da imagem:

```
$ sudo docker tag 937abb0828b6 infoslack/mysql:latest
$ sudo docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	VIRTUAL SIZE
mysql	latest	937abb0828b6	24 minutes ago	342.9 MB
infoslack/mysql	v1	937abb0828b6	24 minutes ago	342.9 MB
infoslack/mysql	latest	937abb0828b6	24 minutes ago	342.9 MB

Por fim, vamos enviar a imagem ao repositório do Docker Hub com a opção `push` :

```
$ sudo docker push infoslack/mysql
The push refers to a repository [infoslack/mysql] (len: 2)
Sending image list
Pushing repository infoslack/mysql (2 tags)
...
```

Agora, temos disponível a imagem `infoslack/mysql` publicamente, que pode ser usada em qualquer Dockerfile.

# TRABALHANDO COM DOCKER

No processo de desenvolvimento de software, a agilidade e a automação fazem parte do cotidiano, e com Docker não é diferente. Até aqui, vimos vários comandos para utilizar a ferramenta. Agora veremos na prática como automatizar boa parte das funcionalidades que aprendemos.

Em nossos exemplos, usaremos uma aplicação Ruby on Rails. No primeiro momento, o foco será configurar o ambiente de desenvolvimento.

Para automatizar a criação e a gestão de containers, vamos utilizar o **Docker Compose**, que é uma ferramenta elaborada para reduzir a complexidade de configuração e isolamento de ambientes de execução para aplicações com Docker.

## 7.1 PREPARANDO O AMBIENTE

Antes de começar, precisamos instalar o **Compose**. A instalação é muito simples e pode ser feita com apenas um comando:

```
$ sudo pip install -U docker-compose
```

Podemos prosseguir com a configuração do nosso environment de desenvolvimento.

O uso do compose consiste em 3 passos:

- Definição do Dockerfile da aplicação;
- Definição dos serviços em um arquivo de configuração;
- Execução do compose para iniciar os serviços.

Neste exemplo, precisamos definir no Dockerfile a imagem base que será usada. Neste caso, utilizaremos uma imagem oficial do Ruby. Além disso, é necessário resolver algumas dependências para o projeto Rails, como a instalação de alguns pacotes: `build-essential`, `postgresql-client` e `nodejs`.

Algumas tarefas não devem ser executadas sempre que a imagem for recriada, como por exemplo, o `bundle install`. Para evitar isso, faremos uso de uma estratégia de cache, copiando os arquivos `Gemfile` e `Gemfile.lock` para o diretório `/tmp` e, em seguida, executando o `bundle install`. Não se preocupe com isso, pois veremos formas melhores de resolver esse problema mais adiante.

Também é preciso dizer que queremos os arquivos do projeto dentro do container, quando ele for criado e, por fim, definimos o comando para dar `start` em nosso projeto. O nosso Dockerfile de exemplo ficará assim:

```
FROM ruby:2.2.1
```

```
RUN apt-get update -qq && apt-get install -y \
 build-essential \
 postgresql-client \
 nodejs
```



```
nodejs

WORKDIR /tmp
COPY Gemfile Gemfile
COPY Gemfile.lock Gemfile.lock
RUN bundle install

RUN mkdir /myapp
ADD . /myapp

WORKDIR /myapp
RUN RAILS_ENV=development bundle exec rake assets:precompile
--trace
CMD ["rails", "server", "-b", "0.0.0.0"]
```

Ainda no diretório da aplicação, precisamos criar um arquivo de configuração indicando os serviços que deverão ser inicializados pelo `compose`. Então, teremos um arquivo chamado `docker-compose.yml`:

```
db:
 image: postgres:9.3
 volumes:
 - ~/.docker-volumes/blog/db/:/var/lib/postgresql/data/
 expose:
 - '5432'

app:
 build: .
 volumes:
 - ./myapp
 ports:
 - '8080:3000'
 links:
 - db
```

Finalmente, podemos executar o comando `docker-compose up` e ver a aplicação funcionando:

```
$ docker-compose up
```

```
Creating railsdockerdemo-db_1...
```

```

Creating railsdockerdemo_app_1...
Attaching to railsdockerdemo_db_1, railsdockerdemo_app_1
db_1 | LOG: database last known up at 2015-02-28 20:14:40 UTC
db_1 | LOG: database automatic recovery in progress
db_1 | LOG: record with zero length at 0/17FEB40
db_1 | LOG: redo is not required
db_1 | LOG: database system is ready to accept connections
db_1 | LOG: autovacuum launcher started
app_1 | Puma starting in single mode...
app_1 | * Version 2.10.2 (ruby 2.2.0-p0), codename: Robots on
Comets
app_1 | * Min threads: 0, max threads: 16
app_1 | * Environment: development
app_1 | * Listening on tcp://0.0.0.0:3000
app_1 | Use Ctrl-C to stop

```

Para testar rapidamente, utilizaremos o `curl` ou o browser, e acessar o endereço `http://localhost:8080` :

```

$ curl -IL http://localhost:8080

HTTP/1.1 200 OK
Cache-Control: max-age=0, private, must-revalidate
Content-Type: text/html; charset=utf-8
ETag: W/"414873b379f496748fe9cd9d2da675db"
Set-Cookie: _
blog_session=M1Q1cDZLanQ1TVZTM3JnTjIvSG10QzRjdVRMbmlrQXZJ
--df46ac772d83165b3e21c47d19d120d3b4476a68; path=/; HttpOnly
Transfer-Encoding: chunked
X-Content-Type-Options: nosniff
X-Frame-Options: SAMEORIGIN
X-Request-Id: 3a428722-24dc-47ae-ba3a-a5887d3fa0f0
X-Runtime: 0.193664
X-XSS-Protection: 1; mode=block

```

## 7.2 ENTENDENDO COMO FUNCIONA

Vamos entender agora o que aconteceu e como os containers de `app` e `db` estabeleceram comunicação.

Olhando para o arquivo de configuração `docker-`

`compose.yml` , na primeira instrução temos um bloco chamado `db` que faz referência ao serviço que será iniciado – neste caso, o `PostgreSQL` .

Observe os detalhes da configuração. Em `image` , estou informando o nome da imagem que será usada e, caso ela não exista no host local, um *pull* será feito no Docker Registry, que vimos no capítulo anterior.

Em seguida, na declaração `volumes` , é informado o `path` local que será mapeado com um diretório do container e, por fim, temos a opção `expose` , onde a porta `5432` está sendo liberada de forma privada, somente para comunicação entre containers.

Já em `app` , é iniciado com a opção `build` , onde é informado que a imagem que será usada por este serviço vem de um Dockerfile local. A instrução para iniciar o web server padrão do Rails na porta `3000` ficou definido no próprio Dockerfile do projeto, na opção `CMD` .

Como tudo está sendo mantido dentro do diretório do projeto Rails, a configuração de `volumes` é declarada partindo do local corrente e mapeada para um diretório dentro do container, de forma que toda alteração nos arquivos do projeto seja refletida no container, em tempo de execução.

Note que em `app` não estamos usando a opção `expose` , em vez disso usamos `ports` , pois queremos mapear a porta `3000` do container para `8080` no host local; assim, acessaremos a aplicação.

A última opção `links` é bastante interessante, pois, como já vimos no capítulo *Configurações de rede*, ela efetiva a comunicação

entre os containers – neste caso, entre `db` e `app` –, sem a necessidade de trabalharmos com configurações de rede.

## 7.3 EXECUTANDO EM PRODUÇÃO

Imagine que finalizamos o desenvolvimento de uma nova aplicação e agora queremos enviá-la para o ambiente de produção. Seguindo a mesma linha de raciocínio da seção anterior, o Dockerfile será algo como:

```
FROM ruby:2.2.2-onbuild
```

Nesta etapa, estamos utilizando a imagem oficial do Ruby – neste caso, com regras que funcionam como gatilhos chamados de `ONBUILD`. Analisando as instruções que criaram a imagem que estamos chamando no Dockerfile, podemos ver o seguinte:

```
FROM ruby:2.2.2

RUN bundle config --global frozen 1

RUN mkdir -p /usr/src/app
WORKDIR /usr/src/app

ONBUILD COPY Gemfile /usr/src/app/
ONBUILD COPY Gemfile.lock /usr/src/app/
ONBUILD RUN bundle install

ONBUILD COPY . /usr/src/app
```

As instruções que apresentam `ONBUILD` não serão executadas quando a imagem for gerada, todas serão adiadas. Esses gatilhos serão disparados apenas quando essa imagem for herdada em outro Dockerfile. Esta abordagem é ideal para a criação de imagens `stand-alone` para um projeto.

Pensando ainda no ambiente de produção, não precisamos alterar a receita existente em `docker-compose.yml`, pois o `docker-compose` permite a definição de um arquivo separado com as configurações apropriadas para o ambiente de produção, neste caso, `production.yml`.

Antes de continuar, vamos sobrescrever o nosso `Dockerfile` para instalar duas coisas necessárias para funcionamento da nossa aplicação: o `NodeJS` como *engine* JavaScript para o Rails, e o `client` de conexão para o `postgresql`:

```
FROM ruby:2.2.2-onbuild
RUN apt-get update && apt-get install -y nodejs postgresql-client
```

Como não queremos que a nossa aplicação funcione sob a porta `3000`, vamos adicionar o servidor web `nginx` para trabalharmos com a porta padrão `80`. A configuração de produção será como a seguir:

```
db:
 image: postgres:9.3
 volumes:
 - ~/.docker-volumes/blog/db:/var/lib/postgresql/data/
 expose:
 - '5432'

app:
 build: .
 command: bundle exec puma -p 9001 -e production
 environment:
 - RAILS_ENV=production
 volumes:
 - ../usr/src/app
 expose:
 - '9001'
 links:
 - db

web:
```

```
image: infoslack/nginx-puma
volumes_from:
 - app
ports:
 - '80:80'
links:
 - app
```

Agora, temos mais um serviço sendo criado, chamado de `web`, que estabelece comunicação com o container de `app` e aplica o mapeamento de portas entre o host utilizando a porta padrão `80`.

Note a opção `command`; ela será responsável por executar o servidor de aplicações Rails Puma na porta `9000`. Na opção `environment`, estamos indicando que a aplicação deve funcionar em modo de produção.

Para que tudo funcione, é necessário configurar o `nginx` de modo que ele tenha conectividade com o servidor de aplicações Rails. Ou seja, devemos inserir o bloco `upstream` nas configurações do `nginx`, informando o host e a porta que será utilizada na aplicação Rails:

```
upstream rails_app {
 server app:9001 fail_timeout=0;
}
```

Felizmente, isso já foi aplicado na construção da imagem e está disponível no Docker Hub.

É importante notar a diretiva `volumes_from` no container `web`, pois, sem ela, o `nginx` não saberá o *path* para a aplicação Rails e, portanto, não funcionará.

Com o arquivo de configuração alternativo ( `production.yml` ), podemos definir uma variável ambiente

chamada `COMPOSE_FILE` . Isso faz com que o `compose` use o arquivo que definimos como padrão:

```
$ COMPOSE_FILE=production.yml
$ docker-compose up -d
```

Ou podemos utilizar o parâmetro `-f` para especificar o arquivo durante a execução:

```
$ docker-compose -f production.yml up -d
```

Agora, vamos pensar em estratégias de deploy para nossa aplicação. No momento, a forma mais simples seria clonar o projeto direto no servidor de produção e criar os containers com `docker-compose` :

```
$ ssh server
$ sudo git clone
https://github.com/infoslack/rails_docker_demo.git
$ cd rails_docker_demo
$ docker-compose -f production.yml up -d
```

O resultado deste projeto no browser será algo como:

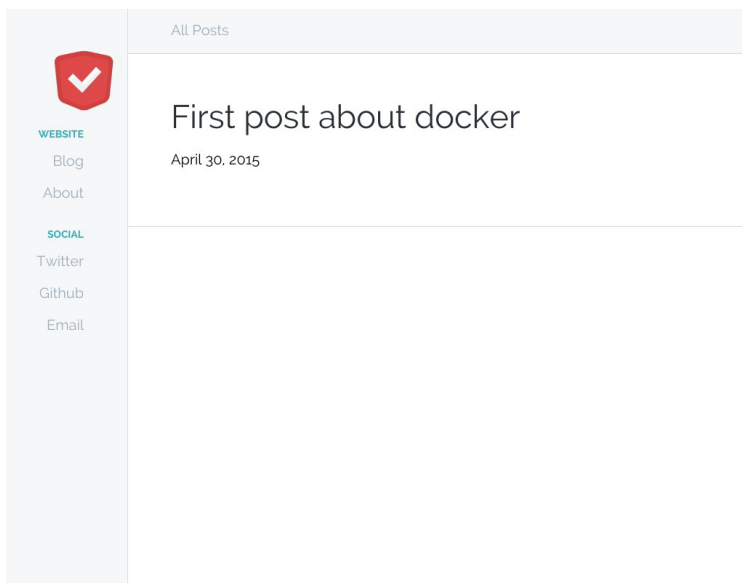


Figura 7.1: Demo Rails com Docker em produção

## 7.4 UM POUCO DE INTEGRAÇÃO CONTÍNUA

Agora que concluímos os ajustes para produção e providenciamos o deploy, veremos algumas formas de melhorar e automatizar o processo. Para isso, faremos uso da integração contínua e elaborar uma forma melhor de implantar nossos containers.

Neste exemplo, faremos uso do *CircleCI* (<https://circleci.com/>), um serviço de integração contínua e deploy, bastante rápido e simples de se usar e que fornece um ótimo suporte para usuários Docker.

Inicialmente, vamos refazer o deploy da seção anterior, desta vez de forma automática. A primeira coisa a ser feita é criar um



arquivo de configuração na raiz do projeto para o CircleCI, chamado `circle.yml` :

```
machine:
 services:
 - docker

dependencies:
 override:
 - docker build -t infoslack/docker-book .

deployment:
 production:
 branch: master
 commands:
 - sudo ./deploy.sh
```

Em `machine` , definimos que o serviço utilizado no CI é para Docker. Em seguida, na opção `dependencies` , informamos que desejamos construir uma imagem, partindo do `Dockerfile` existente no projeto. Por fim, na opção `deployment` , executamos um script para o deploy.

O script `deploy.sh` contém as seguintes informações:

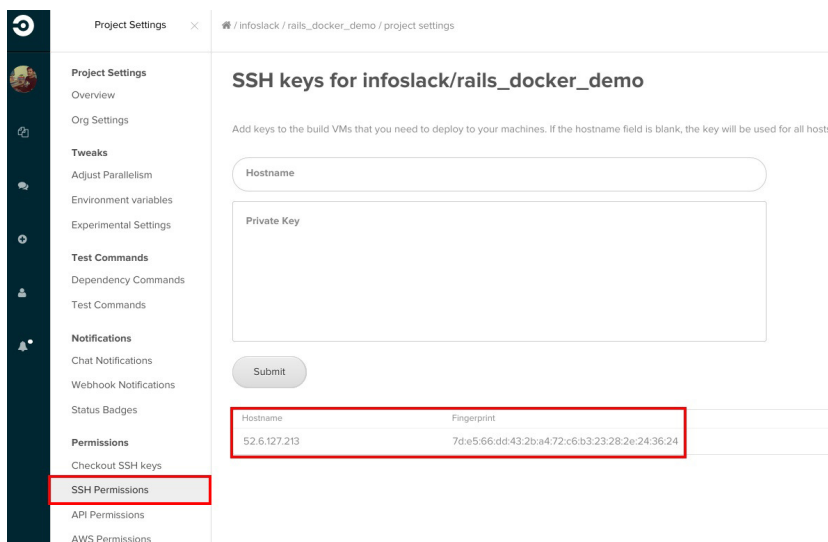
```
#!/bin/bash

ssh deploy@infoslack.com \
 "git clone https://github.com/infoslack/rails_docker_demo.git \
 && cd rails_docker_demo \
 && sudo docker-compose -f production.yml up -d"
```

Ao rodar o script, o CI entra no servidor de produção via `SSH` , executa uma sequência de comandos, realiza o clone do projeto, entra em seu diretório e cria os containers utilizando `docker-compose` .

Para que tudo funcione, é necessário configurar o CircleCI de forma que ele possa acessar o servidor via `SSH`. Essa configuração

pode ser feita com troca de chaves entre o servidor e o CI:



Project Settings

SSH keys for infoslack/rails\_docker\_demo

Add keys to the build VMs that you need to deploy to your machines. If the hostname field is blank, the key will be used for all hosts.

Hostname

Private Key

Submit

Hostname	Fingerprint
52.6.127.213	7d:e5:66:dd:43:2b:a4:72:c6:b3:23:28:2e:24:36:24

Figura 7.2: Configurando o CircleCI para acesso SSH

Clicando nas configurações do projeto ( **Project Settings** ) e em permissões de SSH ( **SSH Permissions** ), podemos adicionar a chave privada para o CI; a chave pública deve ser inserida no servidor. Desta forma, sempre que o projeto for atualizado no GitHub, o CI entra em ação executando as tarefas, e realiza o deploy para produção.

Agora, note que estamos pedindo ao CI para gerar uma imagem, partindo do nosso Dockerfile na task `dependencies` em `circle.yml` , mas não estamos usando para nada. Vamos melhorar isso com o `docker-compose` , aplicar um teste no CI para criar os containers com base na imagem gerada e verificar se tudo está funcionando, antes de realizar o deploy.

O arquivo `circle.yml` ficará assim:

```
machine:
 services:
 - docker

dependencies:
 override:
 - sudo ./install-compose.sh

test:
 override:
 - docker-compose run -d --no-deps app
 - curl --retry 10 --retry-delay 5 -v http://localhost:8080

deployment:
 production:
 branch: master
 commands:
 - sudo ./deploy.sh
```

Em `dependencies`, mudamos para que o CI instale o `docker-compose` e adicionamos a task `test` que cria somente o container de `app` via `compose`. Assim, não precisamos criar todos os containers, pois o CI providencia o `Postgresql` automaticamente.

Em seguida, temos uma instrução `curl` que verifica rapidamente se a aplicação está respondendo, e finalmente realiza o `deploy`.

Veremos mais adiante outra forma de trabalhar com o `CircleCI`.

## 7.5 LIMITANDO O USO DE RECURSOS

Até agora, todos os containers que criamos estão utilizando o total de recursos do host: *CPU*, *RAM*, *I/O* etc. Este é o padrão do

Docker; entretanto, podemos controlar o consumo ao definirmos limites aos containers.

Imagine que temos um container com o web server `nginx` em ação:

```
$ docker run -d -p 80:80 --name nginx infoslack/docker-nginx
```

Vamos acompanhar o consumo de recursos desse container usando a opção `stats`. O resultado seria algo como:

```
$ docker stats nginx
```

CONTAINER	CPU %	MEM USAGE/LIMIT	MEM %	NET I/O
nginx	0.00%	8.293 MiB/7.686 GiB	0.11%	648 B/648 B

Por padrão, o container criado é preparado para fazer uso de toda a memória RAM disponível no host; o mesmo ocorre com a CPU. Podemos recriar o container `nginx`, mas, desta vez, limitando o uso de CPU e RAM com as opções `-c` e `-m`, em seguida, verificar com `stats`:

```
$ docker run -d -p 80:80 -c 512 -m 1024m infoslack/docker-nginx
$ docker stats sick_perlman
```

CONTAINER	CPU %	MEM USAGE/LIMIT	MEM %	NET I/O
nginx	0.00%	8.34 MiB/1 GiB	0.81%	648 B/648 B

Perceba que, ao utilizarmos os limites para memória `-m 1024m`, o container agora está limitado a um total de 1Gb de RAM em vez do total do existente no host. A opção `-c 512` indica que estamos limitando o uso de CPU neste container a 50% da capacidade total contida no host. Por padrão, o Docker usa 1024 nesta diretiva, que representa 100%.

Fazendo uma pequena análise nos `cgroups`, é possível verificar que a opção `-c 512` realmente está configurando o

container para utilizar apenas 50% do total de CPU existente no host. Porém, primeiro vamos listar o total:

```
$ lscpu
Architecture: x86_64
CPU op-mode(s): 32-bit, 64-bit
Byte Order: Little Endian
CPU(s): 4
On-line CPU(s) list: 0-3
...
```

O comando `lscpu` retorna as informações da CPU. Neste exemplo, o meu host possui 4 núcleos, que vão de 0 a 3. Agora, verificando os `cgroups`, notamos a utilização de apenas 50% do total pelo container:

```
$ cat /sys/fs/cgroup/cpuset/docker/d398ff46511a/cpuset.cpus
0-1
```

A diretiva responsável por controlar o uso de CPU neste container é `cpuset.cpus`, e o seu limite está setado para usar os núcleos 0 e 1. Ou seja, apenas 50%.

Não se preocupe sobre limitar os recursos. Veremos na sequência que isso pode ser automatizado.

## 7.6 ORQUESTRANDO CONTAINERS

Chegou o momento de automatizar a construção e a manutenção de containers. Para isso, vamos conhecer uma excelente opção: o projeto `Rancher`.

O `Rancher` é uma plataforma de gestão de containers para a execução de `Docker` em grande escala. Ele é ideal para trabalhar com `Docker` em produção.

O projeto foi construído para reduzir a complexidade de manter projetos e infraestrutura em execução, em qualquer lugar, com o uso de APIs e ferramentas nativas do Docker. Veremos mais detalhes sobre a API do Docker no próximo capítulo.

A instalação é bem simples. Tudo o que precisamos fazer é criar um container que utilize a imagem base do projeto rancher :

```
$ docker run -d -p 8080:8080 rancher/server
Pulling repository rancher/server
3e91cb8f0168: Download complete
511136ea3c5a: Download complete
27d47432a69b: Download complete
Status: Downloaded newer image for rancher/server:latest
6574bb291453
```

A porta 8080 é definida para termos acesso à interface web que usaremos para orquestrar os containers. Após concluir a etapa de instalação, vamos acessar a interface do Rancher – neste exemplo, ficou `http://localhost:8080` :

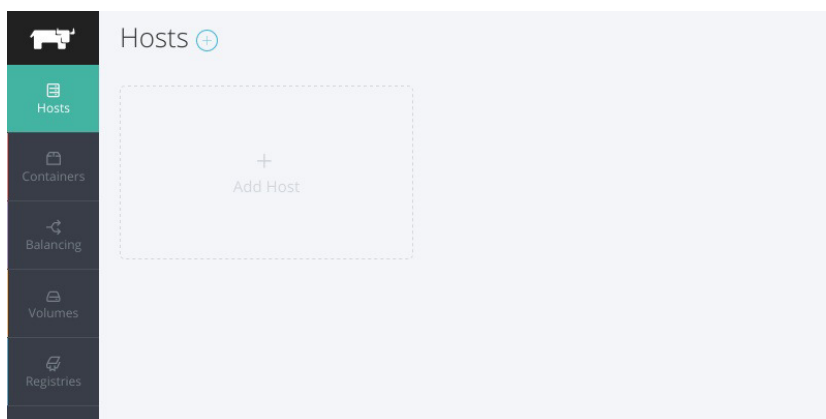


Figura 7.3: Rancher UI

Para começar, é preciso adicionar um host. Para obtermos o melhor potencial desta ferramenta, vamos escolher a opção Amazon EC2 e informar os detalhes, para que o Rancher possa fazer isso de forma automática:

The screenshot shows the 'Add Host' interface in Rancher, specifically for the Amazon EC2 provider. The form is titled 'Add Host' and has a sidebar on the left with navigation options. The main form area contains the following fields and values:

- PROVIDER:** Amazon EC2 (selected)
- NAME:** demo01
- DESCRIPTION:** Demo Rancher
- ACCOUNT ACCESS:**
  - ACCESS KEY:** AKIAIOSFODNN7EXAMPLE
  - SECRET KEY:** [Redacted]
  - SESSION TOKEN:** [Empty]
- INSTANCE:**
  - AMI:** ami-0b94a560
  - INSTANCE TYPE:** t2.micro
  - ROOT SIZE:** 16
  - IAM PROFILE:** [Empty]
- REGION:** us-east-1
- ZONE:** a
- NETWORK:**
  - VPC/SUBNET ID:** vpc-12379a77
  - SECURITY GROUP:** docker-machine

Below the security group field, there is a note: 'Your security group will need to allow traffic:' followed by two bullet points: 'From the internet to TCP ports 8042 and 8046 (for UI hosts, standalone)' and 'From and To all other hosts on UDP ports 500 and 4500 (for IPsec networking)'. A final note states: 'Machine will automatically create this group if needed, but will not add these rules.'

At the bottom right, there are two buttons: 'Create' and 'Cancel'.

Figura 7.4: Preenchendo os dados

As informações mais importantes são o ACCESS KEY e o SECRET KEY, chaves de API da Amazon, pois é por meio delas que o Rancher realizará a autenticação para criar o nosso host. Esses dados podem ser obtidos no console da Amazon:

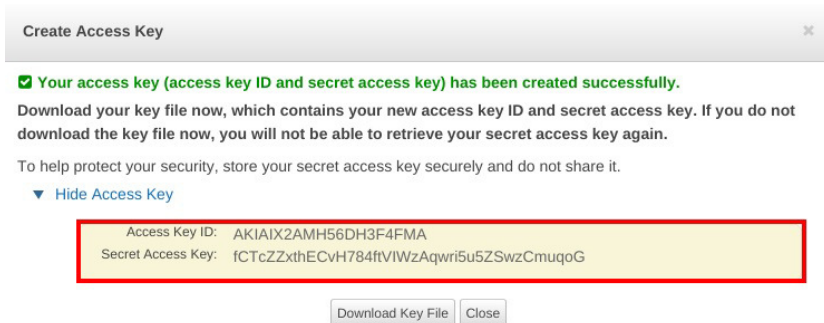


Figura 7.5: AWS API Keys

Depois de preencher os campos, podemos solicitar ao Rancher que crie o host. Esta etapa pode demorar alguns minutos.

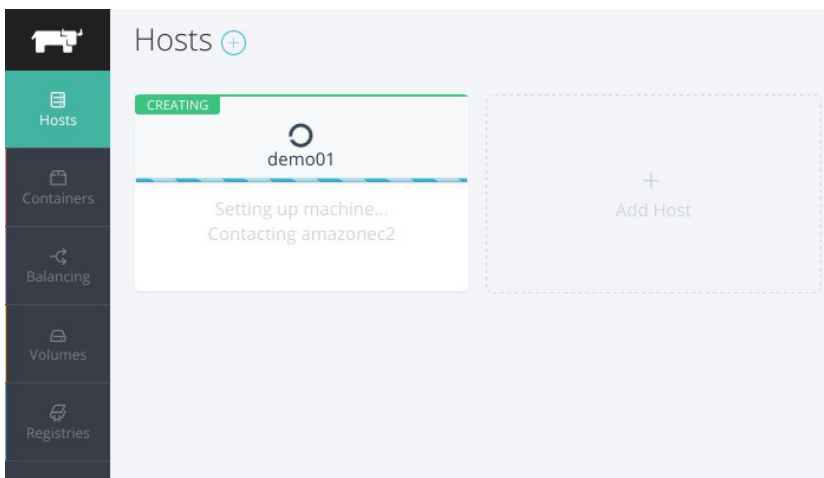


Figura 7.6: Rancher criando um novo host

Finalmente, o host foi criado com sucesso. Note as informações sobre a instância que foi criada na Amazon:



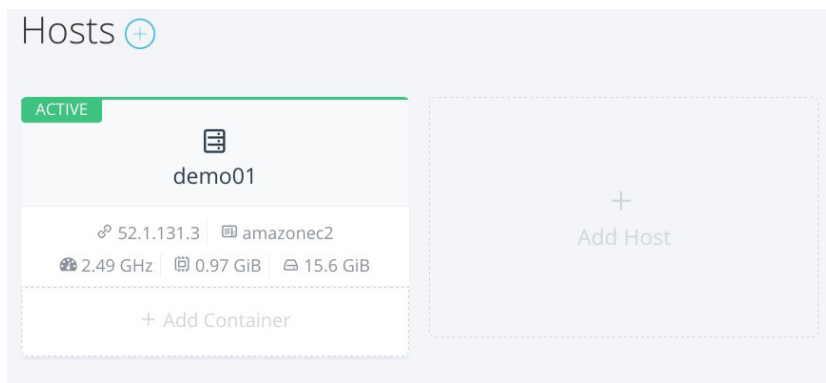


Figura 7.7: Host criado com sucesso!

Com o host configurado, vamos criar alguns containers e ver as facilidades fornecidas pelo Rancher. Neste exemplo, usaremos imagens oficiais do Docker Hub, uma do MySQL e outra do WordPress.

Primeiro criaremos um container para o MySQL , onde é necessário informar apenas 2 opções, a imagem oficial e a variável de ambiente que define a senha de root para acesso:

Add Container

NAME

DESCRIPTION

SELECT IMAGE

PORT MAP  ☐ Publish all ports to random host port

ADVANCED OPTIONS ^

Command Volumes Networking Security/Host

COMMAND

ENTRY POINT

CONSOLE ☐ No ☐ TTY (-t)

AUTO RESTART ☒ Never ☐ Always

ENVIRONMENT VARS

Name  Value

Create Cancel

Figura 7.8: Criando container MySQL

Em seguida, criamos um container para o WordPress, fazendo uso também de sua imagem oficial. Para este container, precisamos mapear a porta 80 para o host e definir a variável ambiente para ter acesso ao banco. Nas opções de rede, criamos um link entre os containers do WordPress e MySQL. Confira as figuras a seguir:

### Add Container

NAME

DESCRIPTION

SELECT IMAGE ▼ docker: wordpress

PORT MAP Public (on Host) 80 → Private (in Container) 80 / Protocol TC Publish all ports to random host port

ADVANCED OPTIONS ^

Command Volumes Networking Security/Host

COMMAND

ENTRY POINT

USER

CONSOLE No TTY (-t)

AUTO RESTART Never Always

ENVIRONMENT VARS +

Name	Value
WORDPRESS_DB_PASSWORD	test123

Figura 7.9: Criando container WordPress

ADVANCED OPTIONS ^

Command Volumes Networking Security/Host

NETWORK

LINKS Destination Container db01 → As Name mysql

HOST NAME

DOMAIN NAME

RESOLVING SERVERS +

SEARCH DOMAINS +

Figura 7.10: Linkando os containers WordPress e MySQL

Com os containers criados, vamos verificar se tudo está funcionando. Ao acessarmos o IP do host, devemos ter como resposta a página de instalação do WordPress:

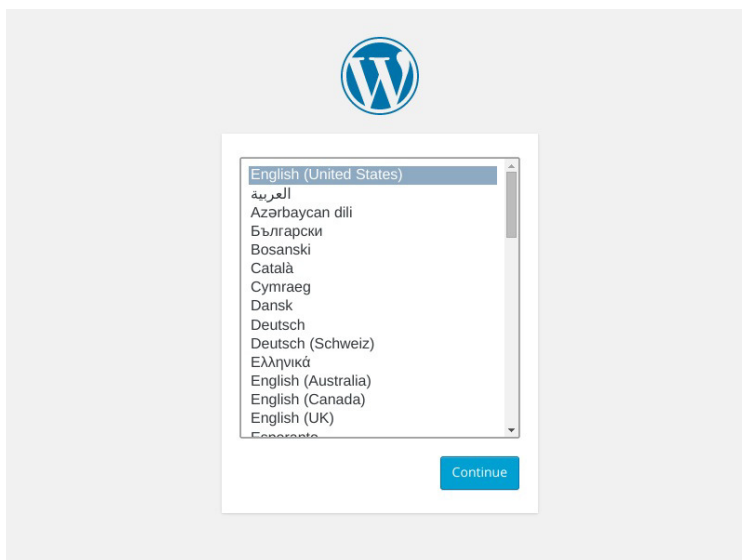


Figura 7.11: WordPress instalação

Explorando um pouco mais o Rancher, é possível ter acesso a um `web shell` nos containers. Isso pode ajudar em operações de debug, por exemplo. Clicando na lista de opções de um container, podemos ter acesso ao shell:

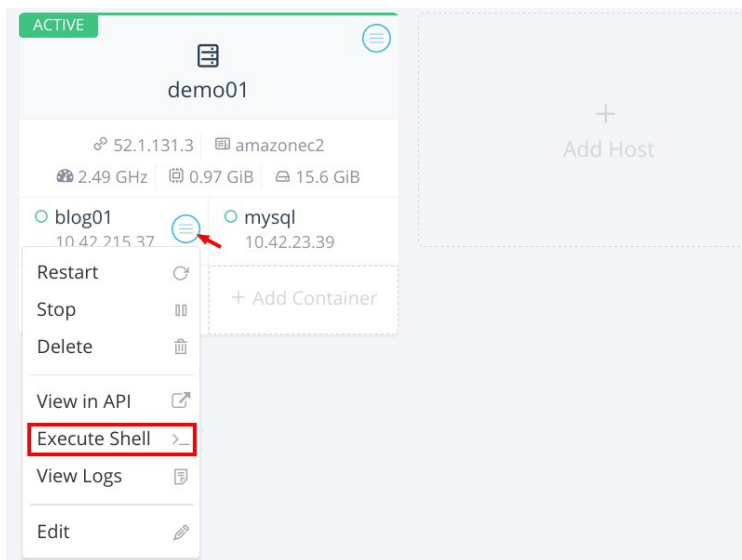


Figura 7.12: Lista de opções de um container

Clicando na opção, temos o seguinte resultado:

>\_ Shell: blog01

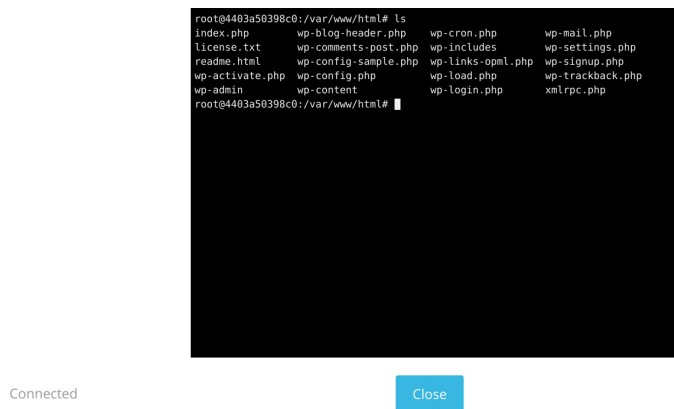


Figura 7.13: Web shell de um container

É possível também limitar o uso de recursos como CPU e memória por cada container de forma prática:

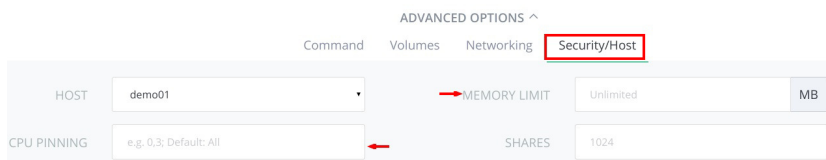


Figura 7.14: Limitando CPU/RAM em um container

Além disso, vamos monitorar o uso desses recursos em cada host adicionado e container criado:

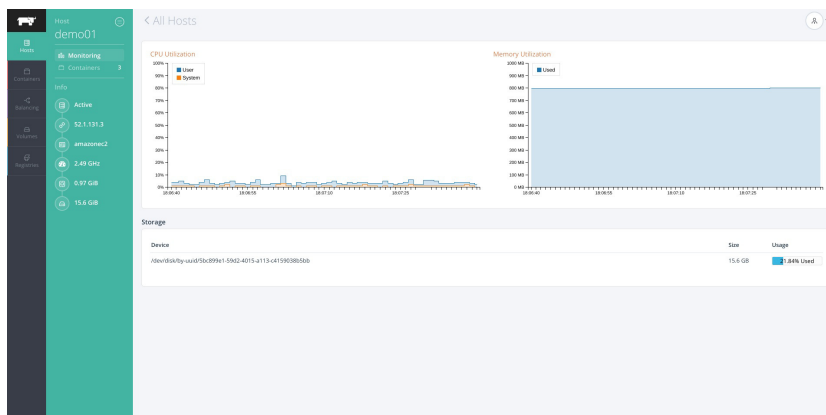


Figura 7.15: Monitorando o consumo de recursos

Como vimos ao criar um container, o Rancher busca as imagens direto no Docker Hub. Em um projeto real, teríamos uma conta privada no Docker Hub integrada a um CI para gerar a imagem do projeto, que, por sua vez, poderia ser chamada no Rancher. Neste caso, o nosso `circle.yml` seria alterado para:

```
machine:
 services:
```

```
- docker

dependencies:
 override:
 - docker build -t infoslack/docker-book .

deployment:
 hub:
 branch: master
 commands:
 - docker login -e $DOCKER_EMAIL -u $DOCKER_USER -p
 $DOCKER_PASS
 - docker push infoslack/docker-book
```

Em vez de enviar direto para produção como fizemos anteriormente, a *task* deployment está fazendo o deploy no Docker Hub. A novidade aqui são as opções de login e push , que recebem variáveis ambiente que foram configuradas no CI.

Com esta abordagem, teríamos a imagem infoslack/docker-book disponível no Docker Hub, pronta para ser usada.

O Rancher é uma das várias opções que pode ser usada para orquestrar containers. Vale a pena explorar todo o potencial da ferramenta.

No próximo capítulo, veremos mais detalhes sobre como a API do Docker funciona e é utilizada por ferramentas, como a que acabamos de ver.

# EXPLORANDO UM POUCO MAIS

Este capítulo destina-se a apresentar recursos mais avançados para você continuar explorando o Docker. Veremos um pouco sobre API, TDD (*Test-Driven Development*) e formas de uso não convencionais, como isolamento de programas desktop em containers.

## 8.1 DOCKER REMOTE API

No final do capítulo anterior, vimos como orquestrar containers utilizando a plataforma *Rancher*, que faz uso da API padrão do Docker em seu background. Agora, vamos explorar essa API e entender um pouco sobre como ela funciona.

A *API REST* que o Docker possui nos permite executar comandos remotamente, ou seja, podemos gerar imagens, criar containers e estabelecer total controle de forma remota. Para isso, é preciso alterar a configuração padrão do Docker em `/etc/default/docker` e adicionar os parâmetros de uso da API, a variável `DOCKER_OPTS` :

```
$ echo "DOCKER_OPTS='-H tcp://0.0.0.0:2375 -H
unix:///var/run/docker.sock'" \
```



```
> /etc/default/docker
$ service docker restart
```

Por padrão, o Docker funciona via *unix socket*. O que fizemos foi adicionar na configuração a opção para que ele responda via HTTP, em sua porta padrão TCP/IP 2375 .

Utilizando o `curl` , podemos testar e verificar como a API funciona, e listar as imagens da seguinte forma:

```
$ curl -X GET http://127.0.0.1:2375/images/json
```

```
[
 {
 "Created":1420935607,
 "Id":"4106803b0e8f",
 "ParentId":"1870b4c5265a",
 "RepoTags":[
 "railsdockerdemo_app:latest"
],
 "Size":2064519,
 "VirtualSize":838022842
 },
 {
 "Created":1420685151,
 "Id":"facc3d0d228ec",
 "ParentId":"9a9e0eaaf857",
 "RepoTags":[
 "inkscape:latest"
],
 "Size":0,
 "VirtualSize":672397560
 },
 {
 "Created":1420102338,
 "Id":"b36113199f789",
 "ParentId":"258a0a9eb8af",
 "RepoTags":[
 "postgres:9.3"
],
 "Size":0,
```

```
 "VirtualSize":213151121
 }
]
```

Ou simplesmente verificar a versão do Docker:

```
$ curl -X GET http://127.0.0.1:2375/version
```

```
{
 "ApiVersion":"1.17",
 "Arch":"amd64",
 "GitCommit":"5bc2ff8",
 "GoVersion":"go1.4",
 "KernelVersion":"3.14.33",
 "Os":"linux",
 "Version":"1.5.0"
}
```

## Manipulando containers

A API do Docker suporta todas as opções que usamos no terminal de comandos para criar e controlar containers. Utilizando o método POST HTTP, vamos inicializar um container desta forma:

```
$ curl -X POST -H "Content-Type: application/json" \
> http://127.0.0.1:2375/containers/create -d '{
 "Hostname": "",
 "User": "",
 "Memory": 0,
 "MemorySwap": 0,
 "AttachStdin": false,
 "AttachStdout": true,
 "AttachStderr": true,
 "PortSpecs": null,
 "Privileged": false,
 "Tty": false,
 "OpenStdin": false,
 "StdinOnce": false,
 "Env": null,
 "Dns": null,
 "Image": "postgres:9.3",
```

```

 "Volumes": {},
 "VolumesFrom": {},
 "WorkingDir": ""
 }'
{"Id": "3e9879113012"}

```

Como resposta, a API retornou o `Id 3e9879113012` , indicando que o container foi criado com sucesso. Com o `Id` do container criado, podemos inicializá-lo de forma bem simples, ao enviarmos a opção `start` como parâmetro na URL:

```

$ curl -X POST
http://127.0.0.1:2375/containers/3e9879113012/start

```

Agora, vamos conferir se o container foi inicializado, listando todos que estão em execução:

```

$ curl -X GET http://127.0.0.1:2375/containers/json

[
 {
 "Command": "/docker-entrypoint.sh postgres",
 "Created": 1424801791, "Id": "3e9879113012",
 "Image": "postgres:9.3",
 "Names": ["/sleepy_galileo"],
 "Ports": [{"PrivatePort": 5432, "Type": "tcp"}],
 "Status": "Up 8 seconds"
 }
]

```

Da mesma forma como usamos o método `start` , podemos enviar o `stop` para finalizar as atividades do container, e a opção `DELETE` para remover o container:

```

$ curl -X POST http://127.0.0.1:2375/containers/3e9879113012/stop
$ curl -X DELETE http://127.0.0.1:2375/containers/3e9879113012

```

## DockerUI

Caso você queira se aprofundar um pouco mais nos estudos

sobre a API do Docker, é interessante explorar o projeto *DockerUI*, uma interface web de manipulação da API. O projeto é aberto e está disponível em <https://github.com/crosbymichael/dockerui>.

A instalação é simples e lembra um pouco o Rancher. Veja como funciona:

```
$ docker run -d -p 9000:9000 --privileged \
-v /var/run/docker.sock:/var/run/docker.sock dockerui/dockerui
```

Um container foi criado, e a porta 9000 foi mapeada para o host local. Agora, basta acessar a interface web em <http://localhost:9000> e explorar.

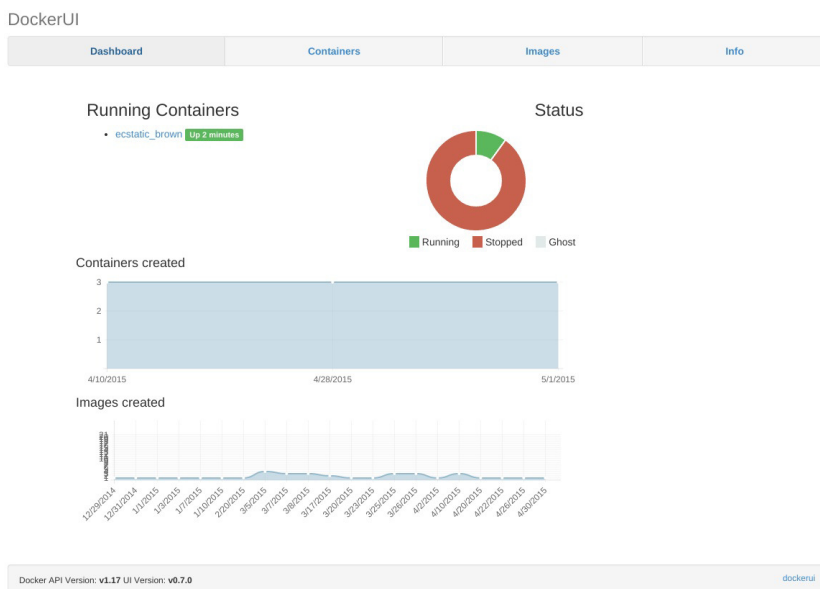


Figura 8.1: DockerUI web interface para API

## 8.2 TDD PARA DOCKER

TDD (*Test-Driven Development*) é uma boa prática em desenvolvimento de software, pois garante a qualidade do software que está sendo desenvolvido. Isso porque todas as funcionalidades têm testes para garantir que o que foi desenvolvido está funcionando conforme esperado, e também aumenta a segurança para modificações posteriores no software.

Quando seu software tem testes, você pode adicionar novas funcionalidades e, por meio dos testes, garantir que ele continue com o comportamento esperado em todas as situações previstas e cobertas pelos testes. Calma, este ainda é um livro sobre Docker!

Agora, imagine poder aplicar a mesma prática de TDD nos seus Dockerfiles para testar o comportamento dos containers. Isto é possível utilizando uma ferramenta chamada *Serverspec*, que permite a escrita de testes *RSpec* (Ruby) para verificar a configuração de servidores. A verificação normalmente é feita via SSH – no caso do Docker, tudo é realizado via API.

Se você não está familiarizado com a ferramenta RSpec ou com a prática de TDD, recomendo a leitura do livro *Test-Driven Development: teste e design no mundo real com Ruby*, disponível na Casa do Código (<http://www.casadocodigo.com.br/products/livro-tdd-ruby>).

Antes de começarmos, é necessário ter o Ruby instalado e as gem `s rspec` e `serverspec` :

```
$ gem install rspec
$ gem install serverspec
```

Com as dependências resolvidas, vamos ao exemplo prático de TDD para Docker. Primeiro, vamos escrever um teste e vê-lo

falhar. Crie um arquivo chamado `Dockerfile_spec.rb` com o código a seguir:

```
require "serverspec"
require "docker"

describe "Dockerfile" do
 before(:all) do
 image = Docker::Image.build_from_dir('.')

 set :os, family: :debian
 set :backend, :docker
 set :docker_image, image.id
 end

 it "installs the last version of ubuntu" do
 expect(os_version).to include("Ubuntu 14")
 end

 def os_version
 command("lsb_release -a").stdout
 end
end
```

O nosso primeiro teste está verificando antes de tudo se a imagem existe. Durante sua execução, ele cria um container que captura a saída do programa `lsb_release -a`, que no nosso exemplo serve para verificar a versão instalada do Ubuntu. Caso a versão seja igual a descrita no teste – ou seja, Ubuntu 14 –, o teste passará.

Agora, vamos executar o teste e vê-lo falhar:

```
$ rspec spec/Dockerfile_spec.rb
F
```

Failures:

```
1) Dockerfile installs the last version of ubuntu
Failure/Error: image = Docker::Image.build_from_dir('.')
Docker::Error::ServerError:
Cannot locate specified Dockerfile: Dockerfile
```

```
...
Finished in 0.02554 seconds (files took 0.74568 seconds to load)
 1 example, 1 failure
```

A mensagem acusa que não encontrou um Dockerfile para gerar a imagem, criar um container e testar a versão. Para fazer o teste passar, precisamos criar o Dockerfile com a versão da imagem base do Ubuntu:

```
FROM ubuntu:14.04
```

Agora rodamos o teste novamente para vê-lo funcionar:

```
$ rspec spec/Dockerfile_spec.rb
.
```

```
Finished in 0.87696 seconds (files took 0.63847 seconds to load)
1 example, 0 failures
```

Esse foi um teste muito simples apenas para explicar o fluxo. Vamos escrever um novo teste, mas desta vez para verificar se um pacote foi instalado e estará presente quando um container for criado:

```
...
 it "installs the last version of ubuntu" do
 expect(os_version).to include("Ubuntu 14")
 end

 it "installs required packages" do
 expect(package("nginx")).to be_installed
 end
...
```

O novo teste verifica se o pacote do `nginx` está presente no container. Ao executá-lo, veremos que ele falha novamente:

```
$ rspec spec/Dockerfile_spec.rb
.F
```

Failures:

```
1) Dockerfile installs required packages
Failure/Error: expect(package("nginx")).to be_installed
expected Package "nginx" to be installed

./spec/Dockerfile_spec.rb:18:in `block (2 levels) in <top
(required)>'

Finished in 1.63 seconds (files took 0.63878 seconds to load)
 2 examples, 1 failure
```

O teste falha pois não temos o Nginx instalado. Para resolver o problema, basta alterar o Dockerfile para que instale o nosso pacote:

```
FROM ubuntu:14.04

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

Com isso, veremos o teste passar novamente:

```
$ rspec spec/Dockerfile_spec.rb
..

Finished in 1 minute 29.61 seconds (files took 0.64142 seconds
to load)
 2 examples, 0 failures
```

O Serverspec facilita bastante a escrita dos testes, você pode testar muito mais coisas além dos exemplos mostrados aqui, como: verificar o mapeamento de portas, a conectividade entre containers e realizar testes específicos para uma aplicação. Dedique um tempo para explorar com mais detalhes essa ferramenta.

## 8.3 DOCKER NO DESKTOP

Além do uso convencional, o Docker pode ser usado para isolar aplicações desktop em containers. Não estou falando só de



aplicações modo texto, estou falando de aplicações GUI.

Imagine limitar a quantidade de CPU e RAM para o Google Chrome, e continuar usando ele normalmente.

Para que isso funcione, tudo o que precisamos fazer é mapear o volume do container, para que use o *socket* gráfico `x11` do Linux. Veja o exemplo:

```
$ docker run -it \
 --net host \
 --cpuset 0 \
 -m 512mb \
 -v /tmp/.X11-unix:/tmp/.X11-unix \
 -e DISPLAY=unix$DISPLAY \
 --name chrome \
 infoslack/chrome
```

A resposta ao criar o container para o Google Chrome será a sua janela de abertura. Note a configuração mapeando o volume para utilizar o *socket* `/tmp/.X11-unix`.

Além disso, a variável ambiente `DISPLAY` é definida para o container com o mesmo valor que é usado no host local. Confira o resultado:

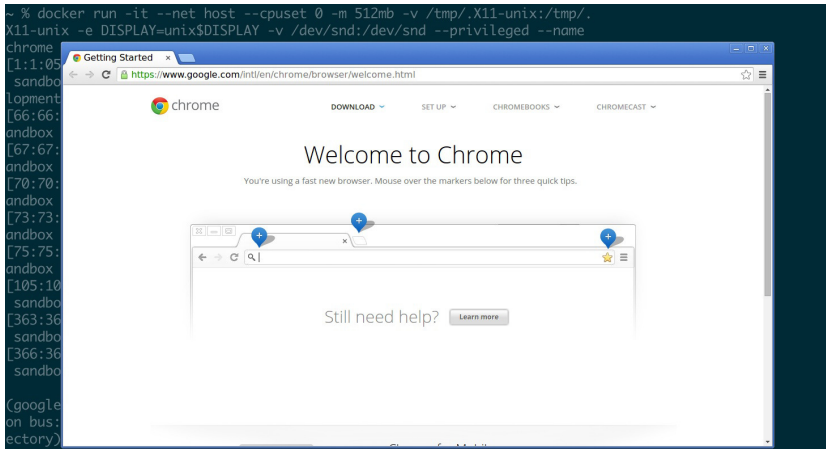


Figura 8.2: Google Chrome com Docker

Além disso, observe as configurações para limitar o uso do processador e o consumo de memória `--cpuset 0` e `-m 512mb`. Com essas opções, estou limitando o uso de apenas 1 núcleo de CPU e disponibilizando um total de 512mb para o container. Podemos acompanhar o consumo com `docker stats`:

```
$ docker stats chrome
```

CONTAINER	CPU %	MEM USAGE/LIMIT	MEM %
chrome	1.25%	376.5 MiB/512 MiB	73.54%

Existe ainda outros complementos para esse container, como por exemplo, o uso da placa de áudio do host. Da mesma forma como mapeamos o *socket* para o *servidor X* criando um volume, podemos fazer com o dispositivo de áudio. Basta adicionar mais um volume na nossa receita:

```
$ docker run -it \
 --net host \
 --cpuset 0 \
 -m 512mb \
 -v /tmp/.X11-unix:/tmp/.X11-unix \
```

```
-e DISPLAY=unix$DISPLAY \
-v /dev/snd:/dev/snd --privileged \
--name chrome \
infoslack/chrome
```

Com o volume `-v /dev/snd:/dev/snd --privileged`, estamos criando um compartilhamento de vários *devices* do host com o container, isso inclui a placa de áudio, a *webcam* e o microfone.

Imagine as possibilidades de uso para outros aplicativos, como Skype, Spotify e até mesmo o Steam. Além do controle fornecido para limitar o consumo de recursos, você tem a flexibilidade de adicionar, remover e manter aplicações em várias versões diferentes, sem a preocupação de poluir o sistema operacional base, pois está tudo em containers.

# UM POUCO SOBRE COREOS LINUX

*CoreOS* é uma distribuição Linux totalmente repensada para fornecer os recursos necessários a uma infraestrutura moderna, de fácil gerenciamento e altamente escalável. Calma, este ainda é um livro sobre Docker.

Neste capítulo, veremos um pouco sobre esse sistema projetado para trabalhar com containers.

## 9.1 PRINCIPAIS DIFERENÇAS

Ao contrário de distribuição Linux, como *Ubuntu* ou *Debian*, o *CoreOS* não possui um gerenciador de pacotes. Portanto, todo e qualquer software que for instalado será um container Docker por padrão.

Outra grande diferença entre o *CoreOS* e a maioria das distribuições é que o sistema é mantido como um todo, ou seja, não teremos atualizações independentes de partes do sistema. Por possuir uma dupla partição raiz, significa que uma atualização pode ser instalada em uma partição diferente da que está em uso.

Por exemplo, o sistema é inicializado na partição A ; o CoreOS verifica se existe atualizações disponíveis e, então, faz o download da atualização e a instala na partição B :



Figura 9.1: CoreOS update inteligente

Desta forma, o sistema garante que se houver algum problema na atualização, tudo poderá ser desfeito. Além disso, as taxas de *limites de rede e I/O* não provocam sobrecarga em aplicações que estejam funcionando, pois esse procedimento é isolado com *cgroups* .

No final do update, a máquina deve ser reiniciada para que o sistema utilize a partição B . Caso ocorra algum problema, o CoreOS vai realizar uma tarefa de *rollback*, voltando a fazer uso da partição A novamente.

## 9.2 ETCD

Um dos principais requisitos para o funcionamento de um *cluster* é a capacidade de estabelecer comunicação entre os nós. O CoreOS faz isso muito bem através do *ETCD*, um esquema

global de *chave-valor*, responsável pela gestão de descoberta de serviços.

A configuração do ETCD no CoreOS é feita por meio de um arquivo chamado `cloud-config`. Nele, é possível personalizar as configurações de rede, scripts para *Systemd* e demais opções de nível de sistema operacional.

O arquivo `cloud-config` é salvo utilizando o formato `YAML` e sempre é processado durante a inicialização da máquina. Além disso, é responsável por armazenar as configurações que permitem ao ETCD ter o controle de entrada de novas máquinas ao `cluster`. Um exemplo do arquivo de configuração `cloud-config` seria assim:

```
coreos:
 etcd:
 # generate a new token for each unique
 # cluster from https://discovery.etcd.io/new
 discovery: https://discovery.etcd.io/<token>

 addr: $private_ipv4:4001
 peer-addr: $private_ipv4:7001

 fleet:
 public-ip: $private_ipv4 # used for fleetctl ssh command

 units:
 - name: etcd.service
 command: start
 - name: fleet.service
 command: start
```

Cada host possui o seu ETCD configurado para manter comunicação com os demais hosts que integram o cluster:

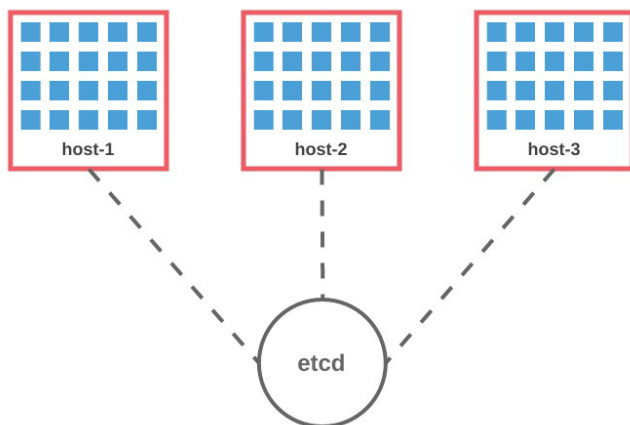


Figura 9.2: Cluster CoreOS, ETCD em ação

Todas as operações realizadas por ETCD, para compartilhar os dados de configuração entre os nós do cluster, são baseadas em uma API/REST. Outro fato interessante é a forma como os containers Docker são tratados, pois um container pode ser replicado em diferentes partes do cluster.

## 9.3 FLEET

Para o controle de clusters, o sistema conta com uma ferramenta chamada *Fleet*, que funciona como um gestor de processos para facilitar a configuração de aplicações no cluster, partindo de um único nó.

Em um cluster, cada nó possui o seu próprio sistema de inicialização e gestão de serviços locais, conhecido por *systemd*. O Fleet nos fornece uma interface controladora para cada *systemd* presente nos nós do cluster. Isso torna possível a inicialização (ou a

parada) de serviços e coleta de informações de processos em execução, em todo o cluster.

Uma outra responsabilidade importante delegada ao Fleet é o controle do mecanismo de distribuição de processos, pois ele escolhe os nós de menor carga para inicializar os serviços. Isso garante a distribuição das tarefas no cluster de forma equilibrada:

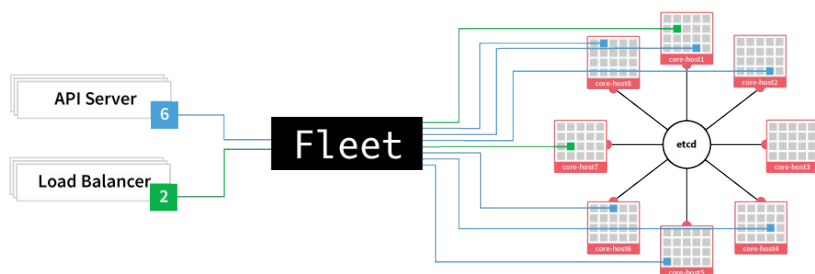


Figura 9.3: CoreOS, controle do cluster com Fleet

## 9.4 CRIANDO UM CLUSTER

Neste exemplo, veremos como criar um cluster de duas máquinas CoreOS para trabalhar com alguns containers Docker. Para isso, usaremos como *cloud provider* a Amazon.

A primeira coisa a ser feita é criar as duas instâncias CoreOS, neste caso, com a AMI (*Amazon Machine Image*) estável mais recente:



## Step 1: Choose an Amazon Machine Image (AMI)

An AMI is a template that contains the software configuration (operating system, application server, and applications)

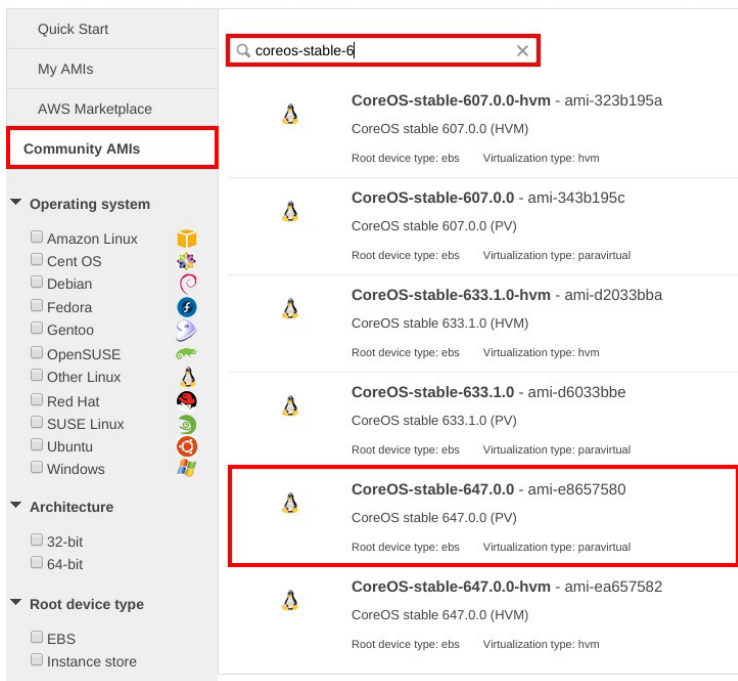


Figura 9.4: AMI CoreOS

Prosseguindo com a criação, é necessário indicar a quantidade de instâncias que vamos construir:

**Step 3: Configure Instance Details**  
 Configure the instance to suit your requirements. You can launch multiple instances from the same AMI, request Spot Instances to take advantage of the lower

Number of instances

Purchasing option ☐ Request Spot Instances

Network  [Create new VPC](#)

Subnet  [Create new subnet](#)

Auto-assign Public IP

IAM role

Shutdown behavior

Enable termination protection ☐ Protect against accidental termination

Monitoring ☐ Enable CloudWatch detailed monitoring  
[Additional charges apply.](#)

Tenancy   
[Additional charges will apply for dedicated tenancy.](#)

Figura 9.5: CoreOS quantidade de instâncias

Para a criação do `cluster`, é preciso armazenar os endereços dos nós do CoreOS e os metadados. Como vimos anteriormente, essa responsabilidade pertence ao *ETCD*. Seu uso é simples, e tudo o que precisamos fazer é gerar um token acessando a sua URL:

```
$ curl https://discovery.etcd.io/new
https://discovery.etcd.io/f1d7ee53c41260107016f1ef50b2ac0a
```

Ainda precisamos personalizar a inicialização das novas instâncias (para configurar detalhes de rede), a comunicação com o serviço de descobertas *ETCD* e o gerenciador de `cluster fleet`.

No painel de configuração, podemos inserir o arquivo `cloud-config` que, como vimos no início deste capítulo, é responsável por carregar essas configurações na inicialização do CoreOS:

```
#cloud-config
```

```
coreos:
 etcd:
```

```
generate a new token from https://discovery.etcd.io/new
discovery:
https://discovery.etcd.io/f1d7ee53c41260107016f1ef50b2ac0a

multi-region and multi-cloud deployments need to use
$public_ipv4
addr: $private_ipv4:4001
peer-addr: $private_ipv4:7001
units:
- name: etcd.service
 command: start
- name: fleet.service
 command: start
```

As informações do cloud-config devem ser inseridas na opção User data .



Figura 9.6: CoreOS adicionando cloud-config

Finalizando a configuração e com as instâncias criadas, podemos testar o cluster acessando qualquer um dos nós. Com o utilitário `fleetctl` , vamos listar todas as máquinas que pertencem ao cluster:

```
$ ssh core@54.174.248.174
```

```
CoreOS (stable)
core@ip-172-31-34-98 ~ $ fleetctl list-machines
MACHINE IP METADATA
610aa9e3... 172.31.34.98 -
0b735501... 172.31.34.97 -
```

## 9.5 CRIANDO UM SERVIÇO

Como sabemos, o `fleet` é responsável por gerenciar o agendamento de serviços para todo o cluster, funcionando como uma interface centralizada de controle que manipula o `systemd` de cada nó do cluster.

Para começar, podemos criar o primeiro arquivo de serviço `nginx@.service`. O `@` na descrição do arquivo indica que ele é apenas um modelo:

```
[Unit]
Description=Nginx web server service
After=etcd.service
After=docker.service
Requires=nginx-discovery@%i.service

[Service]
TimeoutStartSec=0
KillMode=none
ExecStartPre=/usr/bin/docker kill nginx%i
ExecStartPre=/usr/bin/docker rm nginx%i
ExecStartPre=/usr/bin/docker pull nginx
ExecStart=/usr/bin/docker run -d --name nginx%i -p 80:80 nginx

[X-Fleet]
X-Conflicts=nginx@*.service
```

Analisando por partes, temos um cabeçalho de seção representado por `[Unit]` e, logo em seguida, alguns metadados sobre a unidade criada.

Em `Description`, estamos informando a descrição do serviço e verificando as dependências, que, neste caso, está conferindo se os serviços *ETCD* e *Docker* estão disponíveis antes de prosseguir.

Na sequência, um outro arquivo de serviços é adicionado, o `nginx-discovery@%i.service`. Ele será responsável por

atualizar o *ETCD* com as informações sobre o container Docker. O sufixo `%i` no final de alguns nomes serve como variável para receber parâmetros enviados pelo `fleet`.

É preciso informar que serviços devem ser executados, e isso está sendo definido na seção `[Service]`. Como queremos controlar containers Docker, primeiro é preciso desativar o serviço de tempo limite, pois levará um tempo maior que o padrão durante a primeira inicialização do container em cada nó do cluster.

Um outro detalhe importante é o controle das ações de `start` e `stop` do nosso serviço. O *systemd* precisa ser informado que queremos ter o controle e, então, é preciso definir o modo `KillMode` para `none`.

Antes de inicializar o serviço, é necessário ter certeza de que o ambiente está limpo, porque o serviço será inicializado pelo nome e, como sabemos, o Docker só permite um nome único por container.

Note que as instruções `ExecStartPre` possuem `--` em sua sintaxe. Isso indica que essas rotinas podem falhar e, mesmo assim, a execução do arquivo deve prosseguir. Ou seja, caso exista um container com o nome `nginx`, essas tarefas vão obter êxito. Nas duas últimas instruções, serão executados o `pull` da imagem utilizada e o `run` para a criação do container.

Por fim, queremos que o serviço seja executado somente em nós que não possuam um container `Nginx`. É isso que a chamada na seção `[X-Fleet]` faz: armazena as instruções de comportamento do `fleet`. Neste caso, estamos inserindo uma

restrição e garantindo a execução de apenas um container Nginx , por cada nó em todo o cluster. Essa configuração fica interessante em clusters maiores.

## 9.6 REGISTRANDO UM SERVIÇO

É preciso registrar o estado atual dos serviços inicializados no cluster. Para isso, criaremos um outro arquivo de serviço chamado `nginx-discovery@.service` . O novo arquivo é bastante parecido com o anterior, porém este novo serviço é de acompanhamento para atualizar o *ETCD*, informando sobre a disponibilidade do servidor:

```
[Unit]
Description=Announce Nginx@%i service
BindsTo=nginx@%i.service

[Service]
EnvironmentFile=/etc/environment

ExecStart=/bin/sh -c "while true; do etcdctl set \
/announce/services/nginx%i ${COREOS_PUBLIC_IPV4}:%i --ttl 60; \
sleep 45; done"

ExecStop=/usr/bin/etcdctl rm /announce/services/nginx%i

[X-Fleet]
X-ConditionMachineOf=nginx@%i.service
```

Aqui temos a diretiva `BindsTo` , que é uma dependência para verificar o estado do serviço e coletar informações. Caso o serviço listado seja interrompido, o nosso serviço de acompanhamento parará também, porém estamos modificando isso. Caso o container falhe de forma inesperada, as informações no *ETCD* serão atualizadas.

Para manter as informações sempre atualizadas, temos um loop infinito e, dentro do loop, estamos executando o `etcdctl`, que é responsável por alterar os valores em `etcd` que estão sendo armazenados em `/announce/services/nginx%i`.

Finalmente, na última instrução, estamos garantindo que esse serviço seja inicializado na mesma máquina onde o container estiver em execução.

## 9.7 INICIALIZANDO O CLUSTER

De posse dos dois modelos de serviços, vamos enviá-los para o cluster utilizando o comando `fleetctl`:

```
core@node-1 ~ $ fleetctl submit nginx@.service
nginx-discovery@.service
```

Após o envio, podemos verificar se os arquivos de serviço estão disponíveis para o cluster:

```
core@node-1 ~ $ fleetctl list-unit-files
```

UNIT	HASH	DSTATE	STATE	TARGET
nginx-discovery@.service	9531802	inactive	inactive	-
nginx@.service	1e67818	inactive	inactive	-

Agora que os modelos estão registrados no sistema de inicialização para todo o cluster, precisamos carregá-los especificando o novo nome para cada serviço. No caso, a referência da porta `80` será usada indicando que o container está funcionando nesta porta:

```
core@node-1 ~ $ fleetctl load nginx@80.service
core@node-1 ~ $ fleetctl load nginx-discovery@80.service
```

Com isso, podemos conferir em quais nós do cluster o serviço

foi carregado:

```
core@node-1 ~ $ fleetctl list-unit-files
```

UNIT	HASH	DSTATE	STATE	TARGET
nginx-discovery@.service	9531802	inactive	inactive	-
nginx-discovery@80.service	9531802	loaded	loaded	172.31.46.2
nginx@.service	1e67818	inactive	inactive	-
nginx@80.service	1e67818	launched	launched	172.31.46.2

Como os serviços foram todos carregados nas máquinas do cluster, vamos finalmente iniciar, para que tudo funcione:

```
core@node-1 ~ $ fleetctl start nginx@80.service
```

Para saber rapidamente se o container do Nginx foi inicializado e está funcionando corretamente nos nós do cluster, podemos realizar requisições no endereço IP público de cada host:

```
$ http -h 54.174.248.174
```

```
HTTP/1.1 200 OK
Accept-Ranges: bytes
Connection: keep-alive
Content-Length: 612
Content-Type: text/html
Date: Fri, 23 Jan 2015 01:32:54 GMT
ETag: "5418459b-264"
Last-Modified: Tue, 16 Sep 2014 14:13:47 GMT
Server: nginx/1.6.2
```

```
$ http -h 54.174.226.238
```

```
HTTP/1.1 200 OK
Accept-Ranges: bytes
Connection: keep-alive
Content-Length: 612
Content-Type: text/html
Date: Fri, 23 Jan 2015 01:33:08 GMT
ETag: "5418459b-264"
Last-Modified: Tue, 16 Sep 2014 14:13:47 GMT
```



Server: nginx/1.6.2

Gerenciar containers Docker no CoreOS e distribuí-los em um cluster é bastante interessante, mesmo que possa levar um pouco de tempo até você ficar bastante familiarizado com todo o sistema e suas regras.

O projeto CoreOS tem um futuro promissor na gestão e distribuição de aplicações em containers. Dedique um tempo para compreendê-lo melhor.

# O QUE ESTUDAR ALÉM?

Estamos na reta final! Nas próximas seções, veremos algumas recomendações de tecnologias complementares, para que você siga descobrindo mais sobre o ecossistema Docker.

## 10.1 A ALTERNATIVA AO DOCKER COMPOSE, O AZK

Assim como o *Compose*, *azk* é uma ferramenta de orquestração de ambientes de desenvolvimento que, por meio de um arquivo manifesto `Azkfile.js`, ajuda o desenvolvedor a instalar, configurar e executar ferramentas comumente utilizadas para desenvolver aplicações web de forma rápida.

## 10.2 DOCKER MACHINE

*Docker Machine* é um facilitador para criar hosts configurados com Docker partindo da nossa máquina local. Com ele, podemos estabelecer uma ligação com os provedores de Cloud via API, e solicitar a criação de novos servidores com Docker instalado.

Por ser um projeto complementar, sua instalação é independente. No caso do Linux, pode ser instalado da seguinte

forma:

```
$ wget -O -
https://github.com/docker/machine/releases/download/v0.2.0/
> docker-machine_linux-amd64 > /usr/local/bin/docker-machine
$ chmod +x /usr/local/bin/docker-machine
$ docker-machine -v
docker-machine version 0.2.0 (8b9eaf2)
```

No momento em que escrevo este livro, o *Docker Machine* ainda é uma versão beta e significa que poderá sofrer mudanças. Para exemplificar o seu uso, vamos criar uma máquina local com VirtualBox. Desta forma, tudo o que precisamos fazer é informar ao `docker-machine`, com a opção `--driver`, que o provedor onde queremos criar o novo host é, na verdade, uma máquina virtual local. Veja em código:

```
$ docker-machine create --driver virtualbox test
INFO[0000] Creating CA: ~/.docker/machine/certs/ca.pem
INFO[0000] Creating client certificate:
~/.docker/machine/certs/cert.pem
INFO[0003] Creating SSH key...
INFO[0003] Image cache creating it at ~/.docker/machine/cache...
INFO[0003] No default boot2docker iso found locally,
downloading...
INFO[0004] Downloading latest boot2docker release to ...
INFO[0021] Creating VirtualBox VM...
INFO[0050] Starting VirtualBox VM...
INFO[0051] Waiting for VM to start...
INFO[0102] "test" has been created and is now the active machine.
INFO[0102] To point your Docker client at it, run this in your
shell:
eval "$(docker-machine env test)"
```

Concluída a etapa de criação, podemos observar que foi feito o download de uma imagem pequena e customizada chamada `boot2docker`. Essa imagem foi utilizada pelo Virtualbox para gerar o host. Verificaremos o host criado com a opção `ls`:

```
$ docker-machine ls
```

NAME	ACTIVE	DRIVER	STATE	URL
test	*	virtualbox	Running	tcp://192.168.99.100:2376

SWARM

Outro detalhe importante é que devemos dizer ao Docker Machine qual máquina vamos trabalhar. Para isso, vamos apontar o nosso client Docker local, para fixar a comunicação com o host criado:

```
$ eval "$(docker-machine env test)"
```

Agora, toda operação com containers realizada será aplicada no novo host existente no Virtualbox:

```
$ docker run busybox echo Hello Docker Machine
Unable to find image 'busybox:latest' locally
latest: Pulling from busybox
cf2616975b4a: Pull complete
6ce2e90b0bc7: Pull complete
8c2e06607696: Already exists
busybox:latest
Digest: sha256:38a203e1986c
Status: Downloaded newer image for busybox:latest
```

Hello Docker Machine

A documentação do projeto conta com exemplos de uso em VPS e no Cloud da Amazon. Acesse <https://docs.docker.com/machine/>, verifique e faça alguns testes.

## 10.3 DOCKER SWARM

*Swarm* é um sistema de cluster nativo para Docker que utiliza a API padrão. Ou seja, qualquer ferramenta que fizer uso da API Docker poderá usar Swarm para escalar containers, de forma transparente, entre vários hosts. Assim como o *Docker Machine*,

Swarm encontra-se em versão beta no momento em que este livro é escrito.

Tudo o que precisamos para usar Swarm é fazer o `pull` de sua imagem e, em seguida, criar um container de gestão do cluster:

```
$ docker pull swarm
$ docker run --rm swarm create
609c0cbd93b131d7ffa6a0c942acfbca
```

A opção `create` nos retorna um *token* que serve de autenticação para cada nó de um cluster. Um agente será instalado em cada ponto referenciando no host gestor, de forma que mantenha atualizado as informações sobre os containers em execução. O token pode ser chamado de `cluster_id`.

Podemos fazer login em cada nó e inicializar o Docker com suporte para uso de sua API via *HTTP*, lembre-se apenas das opções:

```
$ docker -H tcp://0.0.0.0:2375 -d
```

Agora, só precisamos registrar o agente Swarm que realiza o serviço de descoberta de nós para o cluster. Para isso, o IP do nó deve ser acessível ao nosso gestor. O seguinte comando vai criar o agente:

```
$ docker run -d swarm join --addr=66.228.54.103:2375 \
token://609c0cbd93b131d7ffa6a0c942acfbca
101d0a67fc59a10552fa84f64ed6739d3cef4b356f40c95f24b9c4daec827741
```

Note que foi informado o IP do nó e o token que geramos com `create` para estabelecer a autenticação entre o gestor do cluster e a nova máquina.

Assim, podemos inicializar o gestor Swarm com o seguinte comando:

```
$ docker run -d -p 2376:2375 swarm manage \
token://609c0cbd93b131d7ffa6a0c942acfbca
```

Observe o mapeamento de portas. Nele estou informando uma porta para Swarm que seja direcionada para a porta da API do Docker. Além disso, o token também é informado.

Agora podemos testar e ver se o cluster está funcionando. No exemplo a seguir, podemos coletar informações dos *daemons Docker* presentes no cluster, no caso apenas do nó:

```
$ docker -H tcp://0.0.0.0:2376 info

Containers: 11
Strategy: spread
Filters: affinity, health, constraint, port, dependency
Nodes: 1
 labs: 66.228.54.103:2375
 L Containers: 11
 L Reserved CPUs: 0 / 2
 L Reserved Memory: 0 B / 2.045 GiB
```

Nesta consulta, foi necessário informar o endereço do host responsável pela gestão, para que ele colete as informações usando a opção `info`. Como resposta, temos a informação de que, no cluster, existe apenas 1 nó presente e 11 containers criados. Outros comandos podem ser utilizados dessa mesma forma para listar e criar containers, verificar logs e trabalhar com imagens.

Também é possível solicitar ao Swarm que liste apenas os nós do cluster, com a opção `swarm list`:

```
$ docker run --rm swarm list
token://6856663cdefdec325839a4b7e1de38e8
172.31.40.100:2375
```

172.31.40.101:2375

172.31.40.102:2375

*Docker Swarm* também pode ser usado em conjunto com *Docker Machine* para integrar a um cluster cada novo host criado. Para saber mais, consulte a documentação <https://docs.docker.com/swarm/>.

## 10.4 ALPINE LINUX

Ao construir imagens Docker, não nos preocupamos com o seu tamanho. Porém, depois de um tempo, isso pode ser um problema, pois quanto maior a imagem maior será o tempo para pull e push no Docker Hub.

Pensando nisso, o time da *Glider Labs* vem se empenhando no projeto *Alpine Linux*, um sistema operacional projetado para ser pequeno, simples e seguro. Focaremos apenas em seu tamanho para comparar com as imagens base mais comuns.

Para esse exemplo, podemos fazer uso da imagem oficial do nginx :

```
$ docker pull nginx
```

Criaremos agora um Dockerfile usando o Alpine como imagem base, e instalaremos o Nginx:

```
FROM alpine:3.1
```

```
RUN apk add --update nginx && mkdir /tmp/nginx
```

```
EXPOSE 80 443
```

```
CMD ["nginx", "-g", "daemon off;"]
```

Note que a diferença está apenas no gerenciador de pacotes do Alpine, que no caso se chama `apk`. Assim, criaremos nossa imagem para comparar o tamanho com a oficial do Nginx, que é baseada em Debian:

```
$ docker build -t alpine-nginx .
```

Agora, comparando os tamanhos das imagens, note a diferença.

```
$ docker images
```

REPOSITORY	SIZE
alpine-nginx	6.492 MB
nginx	132.8 MB

Dedique um tempo para estudar mais sobre a gestão de pacotes do Alpine Linux. Este projeto é promissor e deve contribuir bastante com a comunidade Docker.

## 10.5 DÚVIDAS?

Você pode encontrar todos os códigos-fontes utilizados aqui em: <https://github.com/infoslack/docker-exemplos>.

Se você tiver alguma dúvida sobre este livro, junte-se à lista em <http://forum.casadocodigo.com.br>.

Lá, você pode enviar a sua pergunta! Eu e os participantes do grupo tentaremos ajudar.

**Muito obrigado!**