Jeferson Fernando Noronha Vitalino
Marcus André Nunes Castro

BEST SELLER

Descomplicando o Docker

2ª EDIÇÃO ATUALIZADA

Prefácio de
Jérôme Petazzoni

BRASPORT

Descomplicando o

Docker

Jeferson Fernando Noronha Vitalino
Marcus André Nunes Castro

Descomplicando o Docker

2ª edição

1ª edição: 2016
2ª edição: 2018

Para uma melhor visualização deste e-book sugerimos que mantenha seu software constantemente atualizado.

Editor: Sergio Martins de Oliveira

Diretora Editorial: Rosa Maria Oliveira de Queiroz

Gerente de Produção Editorial: Marina dos Anjos Martins de Oliveira

Editoração Eletrônica: SBNigri Artes e Textos Ltda.

Capa: Use Design

Produçao de e-pub: SBNigri Artes e Textos Ltda.

Técnica e muita atenção foram empregadas na produção deste livro. Porém, erros de digitação e/ou impressão podem ocorrer. Qualquer dúvida, inclusive de conceito, solicitamos enviar mensagem para brasport@brasport.com.br, para que nossa equipe, juntamente com o autor, possa esclarecer. A Brasport e o(s) autor(es) não assumem qualquer responsabilidade por eventuais danos ou perdas a pessoas ou bens, originados do uso deste livro.

ISBN Digital: 978-85-7452-902-8

BRASPORT Livros e Multimídia Ltda.
Rua Teodoro da Silva, 536 A – Vila Isabel
20560-001 Rio de Janeiro-RJ
Tels. Fax: (21) 2568.1415/2568.1507

e-mails:
marketing@brasport.com.br
vendas@brasport.com.br
editorial@brasport.com.br

site: www.brasport.com.br

Filial

Av. Paulista, 807 – conj. 915
01311-100 – São Paulo-SP

Agradecimentos

Jeferson Fernando

Ao meu companheiro de longas jornadas de trampo, Marcus André, pessoa fundamental para o desenvolvimento deste livro. Além de ser um superamigo, orgulha-me muito acompanhar seu crescimento profissional e pessoal! *Ieiiiii*!

Aos alunos que acumulei e com quem fiz amizades ao longo desses 15 anos ministrando treinamentos pelo Brasil, especialmente aos alunos do treinamento *Descomplicando o Docker* da LINUXtips! :D

Ao Carlos "do CNI" de'Villa, lá de Itanhaém, que foi o primeiro a acreditar no meu potencial em repassar o conhecimento, isso quando eu tinha apenas 16 anos.

A todos os amigos que fiz por onde passei, especialmente o Juliano "ncode" Martinez, João "orelhinhas" Gabriel, Jean Feltrin, Guilherme "Lero" Schroeder, Rodolfo "enetepe" Ponteado, Guilherme Almeida, Luiz Salvador Jr, Jhonatan Araujo, Francisco "chico" Freire, Adinan Paiva, Adilson Nunes, Pedro Vara, Eduardo Lipolis, Fábio "santa" Santiago, Rodrigo "birgui" Inacio, Giordano Fechio, Rafael Benvenuti, João Borges, Mateus Prado, Leo Martins, Guilherme Calcette e o Ricardo Iorio.

Com muito carinho, gostaria de agradecer a duas pessoas que foram muito importantes para mim nesses últimos anos, principalmente em relação a minha carreira: Eduardo Scarpellini e Leonardo Lorieri, muito obrigado! Como sempre digo, vocês mudaram a minha vida! <3

Meus chefes:

Rogerio Lelis, que sempre me ajudou e me apoiou em tudo que precisei! André Brandão, que, além de ser o melhor gerente que já tive, é o meu *coach* e ajudou-me a quebrar algumas barreiras bem importantes em minha vida. Rodrigo Campos, que tenho como espelho pela sensacional habilidade de palestrar com maestria.

Quero agradecer a minha sogra, Suely, meu sogro, Mario Sergio, e minha cunhada, Camila Silvano, que sempre me apoiam e torcem por mim. Amo vocês!

Quero agradecer muito aos meus irmãos Magno Junior e Camila Vitalino, sempre ao meu lado e me apoiando em tudo. Amo vocês!

Meu filhote Eduardo, que hoje já está um homem! Como o tempo passa! Eu te amo, Brooo!

O agradecimento mais especial é para as mulheres da minha vida:

- Minha esposa e fiel companheira Aletheia, que sempre está ao meu lado em todos os momentos, tristes ou felizes, fáceis ou difíceis, e que sempre acreditou em mim. Eu te amo!
- Minhas princesas e as coisas mais lindas, Maria Fernanda e Maria Eduarda, que são meu tudo, meu ar, minha vida! Eu amo vocês, princesas!
- E finalmente minha rainha, minha mãe, Cidinha Tavares, que passou por diversas dificuldades para conseguir educar o homem que hoje sou! Eu te amo!

Marcus André

Eu agradeço, primeiramente, ao Jeferson Vitalino (quem?) pela ideia, oportunidade e parceria neste livro. Gostaria, também, de continuar agradecendo a ele pelo tempo, pela dedicação e, acima de tudo, por acreditar em mim. Esses últimos anos têm sido, sem dúvida, os melhores da minha carreira profissional e você tem uma bela parcela de culpa nisso.

Queria também agradecer a todas as pessoas que contribuíram direta ou indiretamente para a minha formação como profissional de TI: colegas de trabalho, professores da faculdade, chefes, ex-clientes, pessoas que me agregaram muito valor e às quais eu serei eternamente grato.

Um agradecimento especial aos meus amigos/irmãos: Mário-Anderson, Márcio Ângelo, Bruno Henrique e Manuela Guedes. Meus amigos Jucianio Werllen, Rondineli Gomes, Renan Vicente e amigos de outras áreas: Jémina Diógenes, Juliana Mesquita, Ana Clara Rezende e Rodrigo Peixe.

À minha família: ao meu pai, José Monte, e à minha mãe, dona-Evani, muito obrigado por tudo e parabéns, seu projeto de homem feliz deu certo. :)

À minha mais nova família, Elisângela Oliveira. Agradeço imensamente pela compreensão nesses últimos tempos, pelo carinho, pela emoção, pelo cuidado. Muito obrigado por me mostrar a complexidade nas coisas simples e o universo que existe entre o **0** e o **1**. Eu te amo.

Agradecimento especial:

Gostaríamos ainda de agradecer ao Vicente Marçal, ao Carlos Augusto Malucelli, ao Eduardo Fonseca e ao Henrique Serrat Guimarães por dar aquela conferida final no livro, para que

pudéssemos ter a certeza de que você, leitor, teria uma experiência sensacional durante a leitura e execução dos exemplos. Bom aprendizado, divirtam-se!

Prefácio da primeira edição

If you are a developer, you probably have heard about Docker by now: Docker is the ideal platform to run applications in containers. Fine, but what are those containers? Containers are a lightweight virtualization technique providing us with lots of possibilities: thanks to them, we can ship applications faster; we can easily implement CI/CD (continuous integration and continuous delivery); we can set up development environments faster than ever; we can ensure parity between those development environments and our production servers; and much more. If you write code, if you deploy code, if you operate code, then I promise that containers are going to make your life easier.

But containers have been around for *decades*, and Docker was only released in 2013. So what's the big deal? Why is everybody so excited about Docker, if the technology behind it is more than ten years old?

Because the main innovation of Docker is not the technology. The main innovation of Docker is to make this technology available to every developer and sysadmin, very easily, without having to spend years of practice to become an expert, and without requiring to develop tons of custom tools.

I will share a secret with you: the really important thing is not Docker and containers. What is really important is to improve the quality of our software, reduce our development and deployment costs, release better, more reliable code, faster. It turns out that Docker is an insanely efficient way to achieve those goals. That's why it is so popular.

This book will teach you all you need to know to get started with Docker, and use it to build, ship, and run your applications. It will be your guide in the world of containers, and on the path to ship code better than you ever did before.

Jérôme Petazzoni

Docker Tinkerer Extraordinaire

Sobre o Livro

A proposta deste livro é ajudá-lo a construir ambientes utilizando *containers* com a ferramenta que está revolucionando as empresas de tecnologia ao redor do mundo: vamos aprender e brincar bastante com o sensacional Docker!

De maneira leve e totalmente prática, vamos aprender desde o que é o Docker até a criação de um *cluster* Docker com diversos *containers* em execução! Sempre de forma prática, vamos abordar temas importantes para que consiga administrar ambientes que utilizam ou pretendam utilizar o Docker.

Inclusive, vamos aprender a montar *dockerfiles* personalizados para construção de imagens de *containers*, além de conhecer como melhor administrá-las.

Também veremos na prática como utilizar o Docker Machine para criação de *hosts* Docker, seja local ou na nuvem. Vamos criar um *cluster* utilizando o Swarm e aprender como escalar o nosso ambiente através do Compose.

Nesta segunda edição adicionamos muitas novidades, como o Docker Secret e o Docker Stack, e atualizamos praticamente todo o restante do livro, visando torná-lo super atual e didático. Adicionamos mais exemplos práticos, para que sua experiência possa ser ainda mais agradável e proveitosa.

Sobre os Autores

Jeferson Fernando

Profissional com mais de 15 anos de experiência em administração de servidores Unix/Linux em ambientes críticos de grandes empresas como Motorola, TIVIT, Votorantim, Alog, Locaweb, Walmart.com e Santander. Possuindo mais de 12 anos de experiência como instrutor em grandes centros de treinamento, passou por Impacta Tecnologia, Utah Linux Center, Green, 4Linux e foi instrutor oficial da Red Hat. Já ministrou treinamentos como consultor em empresas como Petrobras, Vale, Ministério do Exército, Polícia Militar do Estado de SP e HP. Possui diversas certificações, como *Docker Certified Associate*, *Certified Kubernetes Administrator*, LPI, RHCE, RHCI, Solaris, entre outras. É um grande entusiasta do software livre e especialista na administração de ambientes críticos com cultura DevOps. Possui ainda um canal no YouTube chamado LinuxTips, onde aborda temas interessantes para todos os níveis de administradores de sistemas Linux.

Marcus André

Profissional com mais de 10 anos de experiência em administração de redes e sistemas, tendo passado por grandes empresas nos últimos anos, como Walmart.com, IBM, Santander,

atuando em diversas plataformas e ambiente homogêneo (*nix, Windows).

Focado em FOSS (*Free and Open Source Software*), é apaixonado e evangelista da linguagem Python e de toda a cultura DevOps.

Sumário

Introdução

Se você é um desenvolvedor que sempre sonhou em poder reproduzir o ambiente de produção em sua máquina ou a aplicação que você desenhou e desenvolveu poder rodar em produção facilmente, com o mesmo "binário" que foi utilizado nos momentos de testes...

...Imagine conseguir instalar e testar diversas aplicações rapidamente e ainda simular um ambiente completo, bem mais leve e inteligente do que as máquinas virtuais em questão de minutos.

Se você é um administrador de sistemas que sempre sonhou em não precisar mais preparar os servidores para suportar milhares de aplicações diferentes, feitas em linguagens diferentes, com bibliotecas diferentes, com versões diferentes (ufa!)…

...Imagine economizar milhões de horas em instalações de novos servidores, atualizações de aplicações, *deploys*, acionamentos na madrugada por conta de serviços que "falharam" e não poderiam ser executados em outros servidores por incompatibilidades diversas.

Se você precisa escalar seu ambiente em questão de minutos para suportar uma demanda que o pegou de surpresa após uma campanha do departamento de marketing de sua empresa...

Se você é um gestor e está preocupado com a fatura do *datacenter* no final do mês por conta da quantidade de recursos necessários para seu ambiente funcionar....

E mesmo quando você possui todos os recursos computacionais para suportar a demanda, se você percebe que na maioria das vezes os seus recursos estão ociosos e sua equipe está com dificuldades para equalizá-los...

...Então este livro e o Docker foram feitos para você!

Boa leitura e divirta-se!

1. O que é *container*?

1.1. Então vamos lá, o que é um *container*?

Container é, em português claro, o agrupamento de uma aplicação junto com suas dependências, que compartilham o *kernel* do sistema operacional do *host*, ou seja, da máquina (virtual ou física) onde está rodando. Deu para entender?

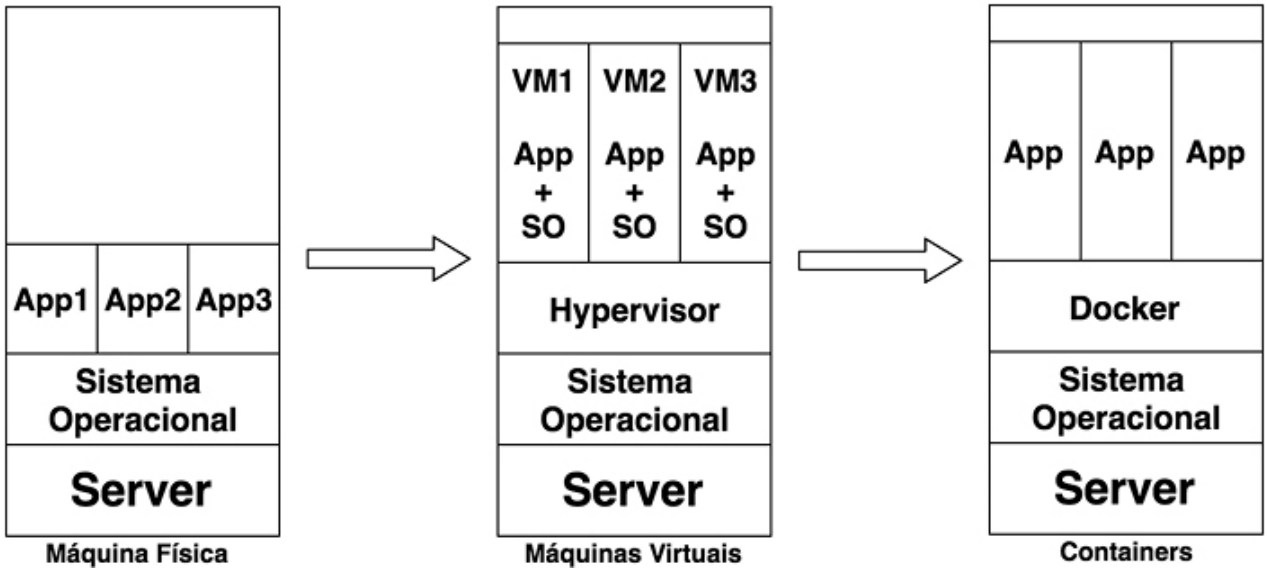
Containers são bem similares às máquinas virtuais, porém mais leves e mais integrados ao sistema operacional da máquina *host*, uma vez que, como já dissemos, compartilha o seu *kernel*, o que proporciona melhor desempenho por conta do gerenciamento único dos recursos.

Na maioria dos casos, a imagem de um *container* é bastante enxuta, havendo somente o necessário para o funcionamento da aplicação, que, quando em execução, possui um pequeno *overhead* se comparada à mesma aplicação rodando nativamente no sistema operacional, grande parte disso por conta do compartilhamento dos recursos com a máquina *host*.

Quando estamos utilizando máquinas virtuais, nós emulamos um novo sistema operacional e virtualizamos todo o seu hardware utilizando mais recursos da máquina *host*, o que não ocorre quando utilizamos *containers*, pois os recursos são compartilhados. O ganho

óbvio disso é a capacidade de rodar mais *containers* em um único *host*, se comparado com a quantidade que se conseguiria com máquinas virtuais.

A seguir, na figura, podemos notar as diferenças de quando temos aplicações sendo executadas nativamente, máquinas virtuais e por fim em *containers*. Repare que não é necessário emular um novo sistema operacional quando estamos utilizando *containers*, diferentemente das máquinas virtuais.



Outro ponto interessante na utilização de *containers* é a portabilidade. Não importa em qual ambiente você criou o seu *container*, ele irá rodar em qualquer outro que possua, no nosso caso, o Docker instalado, seja ele no Linux, MacOS ou Windows. Você não precisa se preocupar com suas dependências, está tudo dentro do *container*. :D

O desenvolvedor consegue, na sua própria máquina, criar uma aplicação em *container* e depois executá-la em um servidor de produção sem nenhum problema de dependência ou algo do tipo –

nem mesmo o bom e velho “engraçado, na minha máquina funciona” escapa, hein?

Lembre-se: na máquina virtual você emula um novo sistema operacional dentro do sistema operacional do *host*. Já no *container* você emula somente as aplicações e suas dependências e as torna portáteis.

1.2. E quando começou que eu não vi?

Apesar de o termo ter se tornado *hype* nos últimos anos, durante décadas já utilizávamos *containers* em sistemas Unix através do comando *chroot*. Sim, bem mais simplório, é verdade, pois era apenas uma forma de isolar o *filesystem*, mas já era o começo!

Em seguida vieram os *jails* do FreeBSD, que, além do isolamento do *filesystem*, permitiam também o isolamento de processos, seguidos de perto pela Sun, que desenvolveu o *Solaris Zones*, mais uma solução baseada em *containers*, porém somente para sistemas Solaris.

O grande passo rumo ao cenário que temos hoje foi a criação, pela Parallels do Virtuozzo, de um painel que permitia o fácil gerenciamento de *containers* e a disponibilização do *core* do Virtuozzo como *open source* com o nome de OpenVZ.

O OpenVZ foi uma ferramenta que ganhou bastante destaque no gerenciamento de *containers* e ajudou e muito na popularização do VPS (*Virtual Private Server*) e, consequentemente, na criação de centenas de empresas de *hosting* espalhadas pelo mundo. O principal ponto negativo do OpenVZ era a necessidade de aplicar um *patch* no *kernel* Linux.

Logo após surgir o OpenVZ, o Google iniciou o desenvolvimento do CGroups para o *kernel* do Linux e iniciou a utilização de *containers* em seus *datacenters*.

Em 2008, desenvolvedores de empresas como Virtuozzo, IBM e o próprio Google iniciaram o projeto LXC, que trazia consigo o CGroups, *namespaces* e *chroot* para prover uma completa e estável solução para a criação e o gerenciamento de *containers*.

Porém, foi no ano de 2013 que os *containers* conquistaram o *mainstream*, saíram do *underground* através da utilização massiva pelas empresas de internet e gigantes de tecnologia e invadiram os principais eventos de tecnologia ao redor do mundo, com palestras sobre o sucesso na utilização de *containers* e com o melhor aproveitamento dos recursos físicos como CPU e memória, maior agilidade no *deployment* de novas aplicações em fração de segundos e tudo isso com uma facilidade que impressiona. Amigo, estamos falando do simplesmente sensacional **Docker**.

2. O que é o Docker?

2.1. Onde entra o Docker nessa história?

Tudo começou em 2008, quando Solomon Hykes fundou a dotCloud, empresa especializada em PaaS com um grande diferencial: o seu *Platform-as-a-Service* não era atrelado a nenhuma linguagem de programação específica, como era o caso, por exemplo, da Heroku, que suportava somente aplicações desenvolvidas em Ruby.

A grande virada na história da dotCloud ocorreu em março de 2013, quando decidiram tornar *open source* o *core* de sua plataforma – assim nascia o Docker!

As primeiras versões do Docker nada mais eram do que um *wrapper* do LXC integrado ao *Union Filesystem*, mas o seu crescimento foi fantástico e muito rápido, tanto que em seis meses seu GitHub já possuía mais de seis mil *stars* e mais de 170 pessoas contribuindo para o projeto ao redor do mundo.

Com isso, a dotCloud passou a se chamar Docker e a versão 1.0 foi lançada apenas 15 meses após sua versão 0.1. A versão 1.0 do Docker trouxe muito mais estabilidade e foi considerada "production ready", além de trazer o Docker Hub, um repositório público para *containers*.

Por ser um projeto *open source*, qualquer pessoa pode visualizar o código e contribuir com melhorias para o Docker. Isso traz maior transparência e faz com que correções de *bugs* e melhorias aconteçam bem mais rápido do que seria em um software proprietário com uma equipe bem menor e poucos cenários de testes.

Quando o Docker 1.0 foi lançado e anunciado que estava pronto para produção, empresas como Spotify já o utilizavam em grande escala; logo AWS e Google começaram a oferecer suporte a Docker em suas nuvens. Outra gigante a se movimentar foi a Red Hat, que se tornou uma das principais parceiras do Docker, inclusive o incorporando-o ao *OpenShift*.

Atualmente, o Docker é oficialmente suportado apenas em máquinas Linux 64 *bits*. Isso significa que seus *containers* também terão que ser um Linux 64 *bits*, pois lembre que o *container* utiliza o mesmo *kernel* da máquina *host*. ;)

Hoje o Docker pode ser executado tranquilamente em outras plataformas como Windows e MacOS, porém ainda não com a mesma performance e estabilidade do Docker sendo executado no Linux. Ahhh, o Linux! <3 <3 <3

2.2. E esse negócio de camadas?

2.2.1. *Copy-On-Write* (COW) e Docker

Antes de entender as camadas propriamente ditas, precisamos entender como um dos principais requisitos para essa coisa acontecer, o *Copy-On-Write* (ou COW para os íntimos), funciona. Nas palavras do próprio Jérome Petazzoni:

> It's a little bit like having a book. You can make notes in that book if you want, but each time you approach the pen to the page, suddenly someone shows up and takes the page and makes a xerox copy and hand it back to you, that's exactly how copy on write works.

Em tradução livre, seria como se você tivesse um livro e que fosse permitido fazer anotações nele caso quisesse, porém, cada vez que você estivesse prestes a tocar a página com a caneta, de repente alguém aparecesse, tirasse uma xerox dessa página e entregasse a cópia para você. É exatamente assim que o *Copy-On-Write* funciona.

Basicamente, significa que um novo recurso, seja ele um bloco no disco ou uma área em memória, só é alocado quando for modificado.

Tá, mas o que isso tudo tem a ver com o Docker? Bom, como você sabe, o Docker usa um esquema de camadas, ou *layers*, e para montar essas camadas são usadas técnicas de *Copy-On-Write*. Um *container* é basicamente uma pilha de camadas compostas por N camadas *read-only* e uma, a superior, *read-write*.

2.3. *Storage drivers*

Apesar de um *container* possuir uma camada de escrita, na maior parte do tempo você não quer escrever dados diretamente nele, por vários motivos, dentre eles a sua natureza volátil. Em situações onde sua aplicação gera dados, você vai preferir usar volumes "atachados" ao *container* e escrever neles (veremos mais à frente como fazer isso). Porém, em algumas situações é, sim, necessária a escrita local no *container*, e é aí que o *storage driver* entra na história. *Storage driver* é o mecanismo utilizado pela *engine* do Docker para ditar a forma como esses dados serão manipulados no *filesystem* do *container*. A seguir, os principais *storage drivers* e suas peculiaridades.

2.3.1. AUFS (*Another Union File System*)

O primeiro *filesystem* disponível para o Docker foi o AUFS, um dos mais antigos *Copy-On-Write filesystems*, e inicialmente teve que passar por algumas modificações a fim de melhorar a estabilidade.

O AUFS funciona no nível de arquivos (não em bloco), e a ideia é ter múltiplos diretórios (camadas) que ele apresenta para o SO como um ponto único de montagem.

Quando você tenta ler um arquivo, a busca é iniciada pela camada superior, até achar o arquivo ou concluir que ele não existe. Para escrever em um arquivo, este precisa primeiro ser copiado para a camada superior (*writable*) – e, sim, você adivinhou: escrever em arquivos grandes pode causar certa degradação da performance, já que o arquivo precisaria ser copiado completamente para a primeira camada, mesmo quando uma parte bem pequena vai sofrer alteração.

Já que estamos falando de coisa chata, outra coisa que pode degradar a sua performance usando AUFS é o fato de que ele procura cada diretório de um *path* em cada camada do *filesystem* toda vez que você tentar executar um comando. Por exemplo, se você tem um *path* com cinco camadas, serão realizadas 25 buscas (*stat()*, uma *system call*). Isso pode ser bem complicado em aplicações que fazem *load* dinâmico, como os *apps* Python que importam os .py da vida.

Outra particularidade é quando algum arquivo é deletado. Quando isso acontece é criado um *whiteout* para esse arquivo. Em outras palavras, ele é renomeado para ".wh.arquivo" e fica indisponível para o *container*, já que, né, não dá para apagar de verdade, pois as outras camadas são *read-only*.

2.3.2. *Device Mapper*

Device Mapper é um *kernel-based framework* da Red Hat usado para algumas abstrações, como, por exemplo, o mapeamento de "blocos físicos" em "blocos lógicos", permitindo técnicas como LVM e RAID. No contexto do Docker, porém, ele se resume ao "thin

provisioning target" ou ao *storage driver* "devicemapper". Assim que essa coisa de Docker começou a andar, o pessoal da Red Hat (e toda a galera que usava alguma *distro* relacionada com Red Hat) se interessou bastante, só que havia um problema: eles não queriam usar AUFS. Para resolver isso, eles reuniram uma equipe de engenheiros muito habilidosos que adicionaram suporte ao *Device Mapper* no Docker.

Em se tratando de Docker, o *Device Mapper* e o AUFS são bem similares: a grande diferença entre eles é que, no *Device Mapper*, quando você precisa escrever no arquivo, a cópia é feita em nível de blocos, que era um problema lá no AUFS, e com isso você ganha uma granularidade bem maior. Em teoria, o problema que você tinha quando escrevia um arquivo grande desaparece. Por padrão, *Device Mapper* escreve em arquivos de *loopback*, o que deixa as coisas mais lentas, mas agora na versão 1.17+ você já pode configurá-lo em modo *direct-lvm*, que escreve em blocos e, em teoria, resolveria esse problema. É um pouco mais chatinho de configurar, mas é uma solução mais elegante para ambientes em produção.

Além de AUFS e *Device Mapper*, você também pode usar BRTFS e OverlayFS como *storage driver*. Por serem tecnologias relativamente jovens, aprecie com moderação.

2.3.3. OverlayFS e OverlayFS2

A bola da vez. Uma versão melhorada do AUFS, o OverlayFS e sua versão seguinte e oficialmente recomendada pelo Docker, o OverlayFS2, são ambos *other union filesystems*, mas dessa vez muito mais eficientes, rápidos e com uma implementação muito mais simples.

Por serem *union file systems*, também compartilham da ideia de juntar vários diretórios em um único ponto de montagem como nosso

amigo AUFS, porém, no caso do OverlayFS, apenas dois diretórios são suportados, o que não acontece no OverlayFS2, que tem suporte *multi-layer*. Ambos suportam *page caching sharing*, ou seja, múltiplos *containers* acessando o mesmo arquivo dividem a mesma entrada no arquivo de paginação, o que é um uso mais eficiente de memória.

Aquele problema antigo do AUFS de ter de copiar todo o arquivo para a camada de cima para escrever nele ainda persiste, porém no OverlayFS ele só é copiado uma vez e fica lá para que as outras escritas no mesmo arquivo possam acontecer mais rápido, então tem uma pequena vantagem.

Nota-se um consumo excessivo de *inodes* quando se usa OverlayFS. Esse é um problema resolvido no OverlayFS2, então sempre que possível busque usá-lo – até porque, no geral, tem uma performance superior. Lembrando que *kernel* 4.0+ é pré-requisito para usar OverlayFS2.

2.3.4. BTRFS

BTRFS é a geração seguinte de *union filesystem*. Ele é muito mais *space-efficient*, suporta muitas tecnologias avançadas de *storage* e já está incluso no *mainline* do *kernel*. O BTRFS, diferentemente do AUFS, realiza operações a nível de bloco e usa um esquema de *thin provision* parecido com o do *Device Mapper* e suporta *copy-on-write snapshots*. Você pode inclusive combinar vários *devices* físicos em um único BTRFS *filesystem*, algo como um LVM.

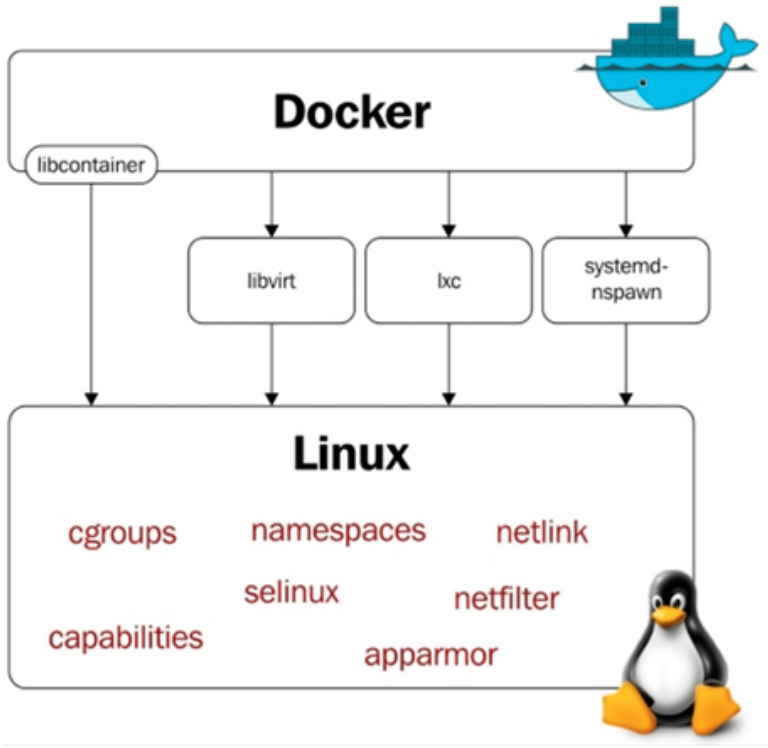
O BTRFS é suportado atualmente na versão CE apenas em distribuições *debian-like* e na versão EE apenas em SLES (*Suse Linux Enterprise Server*).

IMPORTANTE: alterar o *storage drive* fará com que qualquer *container* já criado se torne inacessível ao sistema local.

Cuidado!

2.4. Docker Internals

O Docker utiliza algumas *features* básicas do *kernel* Linux para seu funcionamento. A seguir temos um diagrama no qual é possível visualizar os módulos e *features* do *kernel* de que o Docker faz uso:



2.5. *Namespaces*

Namespaces foram adicionados no *kernel* Linux na versão 2.6.24 e são eles que permitem o isolamento de processos quando estamos utilizando o Docker. São os responsáveis por fazer com que cada *container* possua seu próprio *environment*, ou seja, cada *container* terá a sua árvore de processos, pontos de montagens, etc., fazendo com que um *container* não interfira na execução de outro. Vamos saber um pouco mais sobre alguns *namespaces* utilizados pelo Docker.

2.5.1. PID *namespace*

O PID *namespace* permite que cada *container* tenha seus próprios identificadores de processos. Isso faz com que o *container* possua um PID para um processo em execução – e quando você procurar por esse processo na máquina *host* o encontrará; porém, com outra identificação, ou seja, com outro PID.

A seguir temos o processo "testando.sh" sendo executado no *container*.

Perceba o PID desse processo na árvore de processos dele:

```
root@c774fa1d6083:/# bash testando.sh &
[1] 7
root@c774fa1d6083:/# ps -ef
UID PID PPID C STIME TTY TIME CMD
root 1 0 0 18:06 ?           00:00:00 /bin/bash
root 7 1 0 18:07 ?           00:00:00 bash testando.sh
root 8 7 0 18:07 ?           00:00:00 sleep 60
root 9 1 0 18:07 ?           00:00:00 ps -ef
root@c774fa1d6083:/#
```

Agora, perceba o PID do mesmo processo exibido através do *host*:

```
root@linuxtips:~# ps -ef | grep testando.sh
root 2958 2593 0 18:12 pts/2 00:00:00 bash testando.sh
root 2969 2533 0 18:12 pts/0 00:00:00 grep --color=auto testando.sh
root@linuxtips:~#
```

Diferentes, né? Porém, são o mesmo processo. :)

2.5.2. *Net Namespace*

O *Net Namespace* permite que cada *container* possua sua interface de rede e portas. Para que seja possível a comunicação entre os *containers*, é necessário criar dois *Net Namespaces* diferentes, um responsável pela interface do *container* (normalmente

utilizamos o mesmo nome das interfaces convencionais do Linux, por exemplo, a eth0) e outro responsável por uma interface do *host*, normalmente chamada de veth* (veth + um identificador aleatório). Essas duas interfaces estão *linkadas* através da *bridge* Docker0 no *host*, que permite a comunicação entre os *containers* através de roteamento de pacotes.

Conforme falamos, veja as interfaces. Interfaces do *host*:

```
root@linuxtips:~# ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN
group default
   link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
   inet 127.0.0.1/8 scope host lo
     valid_lft forever preferred_lft forever
   inet6 ::1/128 scope host
     valid_lft forever preferred_lft forever
2: eth1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast
state UP group default qlen 1000
   link/ether 00:1c:42:c7:bd:d8 brd ff:ff:ff:ff:ff:ff
   inet 10.211.55.35/24 brd 10.211.55.255 scope global eth1
     valid_lft forever preferred_lft forever
   inet6 fdb2:2c26:f4e4:0:21c:42ff:fec7:bdd8/64 scope global dynamic
     valid_lft 2591419sec preferred_lft 604219sec
   inet6 fe80::21c:42ff:fec7:bdd8/64 scope link
     valid_lft forever preferred_lft forever
3: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
state UP group default
   link/ether 02:42:c7:c1:37:14 brd ff:ff:ff:ff:ff:ff
   inet 172.17.0.1/16 scope global docker0
     valid_lft forever preferred_lft forever
   inet6 fe80::42:c7ff:fec1:3714/64 scope link
     valid_lft forever preferred_lft forever
5: vetha2e1681: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc
noqueue master docker0 state UP group default
   link/ether 52:99:bc:ab:62:5e brd ff:ff:ff:ff:ff:ff
   inet6 fe80::5099:bcff:feab:625e/64 scope link
     valid_lft forever preferred_lft forever
root@linuxtips:~#
```

Interfaces do *container*:

```
root@6ec75484a5df:/# ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN
group default
   link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
   inet 127.0.0.1/8 scope host lo
     valid_lft forever preferred_lft forever
   inet6 ::1/128 scope host
     valid_lft forever preferred_lft forever
6: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
state UP group default
   link/ether 02:42:ac:11:00:03 brd ff:ff:ff:ff:ff:ff
   inet 172.17.0.3/16 scope global eth0
     valid_lft forever preferred_lft forever
   inet6 fe80::42:acff:fe11:3/64 scope link
     valid_lft forever preferred_lft forever
root@6ec75484a5df:/#
```

Conseguiu visualizar as interfaces Docker0 e veth* do *host*? E a eth0 do *container*? Sim? *Otémooo*! :D

2.5.3. *Mnt Namespace*

É evolução do *chroot*. Com o *Mnt Namespace* cada *container* pode ser dono de seu ponto de montagem, bem como de seu sistema de arquivos raiz. Ele garante que um processo rodando em um sistema de arquivos não consiga acessar outro sistema de arquivos montado por outro *Mnt Namespace*.

2.5.4. IPC *namespace*

Ele provê um SystemV IPC isolado, além de uma fila de mensagens POSIX própria.

2.5.5. UTS *namespace*

Responsável por prover o isolamento de *hostname*, nome de domínio, versão do SO, etc.

2.5.6. *User namespace*

O mais recente *namespace* adicionado no *kernel* Linux, disponível desde a versão 3.8. É o responsável por manter o mapa de identificação de usuários em cada *container*.

2.6. *Cgroups*

É o *cgroups* o responsável por permitir a limitação da utilização de recursos do *host* pelos *containers*. Com o *cgroups* você consegue gerenciar a utilização de CPU, memória, dispositivos, I/O, etc.

2.7. *Netfilter*

A já conhecida ferramenta *iptables* faz parte de um módulo chamado *netfilter*. Para que os *containers* consigam se comunicar, o Docker constrói diversas regras de roteamento através do *iptables*; inclusive utiliza o NAT, que veremos mais adiante no livro.

```
root@linuxtips:~# iptables -t nat -L
Chain PREROUTING (policy ACCEPT)
target                          prot opt source         destination
DOCKER                          all -- anywhere         anywhere
ADDRTYPE match dst-type LOCAL

Chain INPUT (policy ACCEPT)
target                          prot opt source         destination

Chain OUTPUT (policy ACCEPT)
target                          prot opt source         destination
DOCKER                          all -- anywhere         !127.0.0.0/8
ADDRTYPE match dst-type LOCAL

Chain POSTROUTING (policy ACCEPT)
target                          prot opt source         destination
MASQUERADE                      all -- 172.17.0.0/16    anywhere
```

```
Chain DOCKER (2 references)
target                          prot opt source          destination
RETURN                          all -- anywhere          anywhere
root@linuxtips:~#
```

2.8. Para quem ele é bom?

O Docker é muito bom para os desenvolvedores, pois com ele você tem liberdade para escolher a sua linguagem de programação, seu banco de dados e sua distribuição predileta. Já para os *sysadmins* é melhor ainda, pois, além da liberdade de escolher a distribuição, não precisamos preparar o servidor com todas as dependências da aplicação. Também não precisamos nos preocupar se a máquina é física ou virtual, pois o Docker suporta ambas.

A empresa como um todo ganha, com a utilização do Docker, maior agilidade no processo de desenvolvimento de aplicações, encurtando o processo de transição entre os ambientes de QA STAGING e PROD, pois é utilizada a mesma imagem. Traz menos custos com hardware por conta do melhor gerenciamento e aproveitamento dos recursos, além do *overhead*, que é bem menor se comparado com outras soluções, como a virtualização.

Com Docker fica muito mais viável a criação de *microservices* (microsserviços, a ideia de uma grande aplicação ser quebrada em várias pequenas partes e estas executarem tarefas específicas), um assunto que tem ganhado cada vez mais espaço no mundo da tecnologia e que vamos abordar com mais detalhes no final deste livro.

Ainda temos diversos outros motivos para utilizar *containers* e que vamos descobrindo conforme evoluímos com a utilização do Docker. :D

3. Instalando o Docker

3.1. Quero instalar, vamos lá?

Bom, dado que você já sabe o que é um *container* e o que é o tal do Docker, chegou a hora de pôr a mão na massa. Vamos instalar o Docker pela primeira vez!

O *daemon* do Docker roda nativo em distribuições Linux, e por isso a instalação em sistemas operacionais que não sejam Linux consiste basicamente em subir uma VM e rodar o *daemon* de lá. O cliente, no entanto, pode ser instalado nos principais sistemas operacionais disponíveis atualmente.

Para realizar a instalação do Docker em máquinas Linux é bastante simples. Precisamos somente observar alguns pontos:

- ⇨ O Docker não suporta processadores 32 *bits*.
- ⇨ O Docker é suportado (*stable*) somente na versão do *kernel* 3.8 ou superior.
- ⇨ O *kernel* deve ter suporte aos sistemas de arquivos utilizados pelo Docker, como o AUFS, *Device Mapper*, OverlayFS, etc.
- ⇨ O *kernel* deverá ter suporte a *cgroups* e *namespaces*, o que normalmente já vem por *default* habilitado na maioria das *distros*.

Você também pode acessar a URL: <https://docs.docker.com/install/>. Lá é possível aprender a instalar o Docker em diversas distribuições Linux, nos principais *clouds* e também no MacOS e no Windows.

Neste livro vamos utilizar a distribuição Ubuntu Linux, porém não muda nada para outras distribuições. Chega de conversa, vamos lá!

Primeiro, vamos verificar a versão do *kernel* para saber se ele é compatível com o Docker:

```
# uname -r
```

3.2. Instalando no Debian/Centos/Ubuntu/Suse/Fedora

A instalação do Docker é bastante simples. Você pode optar por instalá-lo utilizando os pacotes disponíveis para sua *distro* – por exemplo, o *apt-get* ou *yum*.

Nós preferimos fazer a instalação através da execução do *curl* a seguir, que irá executar um *script* e detectará qual a distribuição que estamos utilizando, para então adicionar o repositório oficial do Docker em nosso gerenciador de pacotes, o *rpm* ou *apt*, por exemplo.

```
# curl -fsSL https://get.docker.com/ | sh
```

Assim ele sempre buscará a versão mais recente do Docker. :)

3.3. Instalando 'manualmente' no Debian

Caso você esteja utilizando o Debian e queira realizar a instalação através dos pacotes disponíveis no repositório, faça:

```
#apt-key adv --keyserver \ hkp://pgp.mit.edu:80 --recv-keys \
58118E89F3A912897C070ADBF76221572C52609D
```

Agora vamos criar/editar o arquivo "/etc/apt/sources.list.d/docker.list" e adicionar o endereço do repositório correspondente à versão do seu Debian. No nosso caso estamos utilizando a versão Debian 8, também conhecida como Jessie.

```
# vim /etc/apt/sources.list.d/docker.list # Debian Jessie
deb https://apt.dockerproject.org/repo debian-jessie main
```

Após adicionar a linha anterior, é necessário atualizar a lista de repositórios executando:

```
# apt-get update
```

Após finalizar a atualização da lista de repositórios disponíveis, já podemos fazer a instalação do Docker. O nome do pacote é "docker-ce". :)

```
# apt-get install docker-ce
```

Vamos verificar se o Docker está em execução. Digite na linha de comando o seguinte:

```
#/etc/init.d/docker status
```

Ou:

```
# service docker status
docker container stop/waiting
```

Com isso, podemos verificar se o processo está em execução. Como podemos notar, o *daemon* do Docker não está em execução, portanto vamos iniciá-lo.

```
# service docker start
docker container start/running, process 4303

# service docker status
docker container start/running, process 4303
```

Perfeito! Agora já temos o Docker instalado e pronto para começar a brincar com os *containers*. \o/

3.3.1. Dica importante

Por padrão, o *daemon* do Docker faz *bind* em um *socket* Unix, e não em uma porta TCP. *Sockets* Unix, por sua vez, são de propriedade e de uso exclusivo do usuário *root* (por isso o Docker sempre é iniciado como *root*), mas também podem ser acessados através do *sudo* por outros usuários.

Para evitar que você tenha que ficar usando *sudo* ao rodar comandos do Docker, crie um grupo chamado *docker* e adicione o seu usuário a ele. Pare o serviço e inicie-o novamente.

Infelizmente, nem tudo são flores. Esse procedimento faz com que o usuário tenha **os mesmos privilégios do usuário *root*** em operações relacionadas ao Docker. Mais informações no link: <https://docs.docker.com/engine/security/>.

Para criar um grupo no Linux e adicionar um usuário não tem segredo, basta rodar:

```
$ sudo usermod -aG docker user
```

Dica de um milhão de dólares: **user** = **seu usuário**. :D

4. Executando e administrando *containers* Docker

4.1. Então vamos brincar com esse tal de *container*!

Como todos sabemos, o Docker utiliza a linha de comando para que você possa interagir com ele – basicamente você utiliza o comando "docker".

Bom, agora que já iniciamos o Docker, vamos rodar nosso primeiro *container*.

Como é de costume quando alguém está aprendendo uma nova linguagem de programação, é bem comum fazer como o primeiro código um ***hello world*!**

Apesar de o Docker não ser uma linguagem de programação, vamos utilizar esse costume com o nosso primeiro exemplo de um *container* em execução.

O Docker possui uma imagem personalizada de *hello-world* e serve para que você possa testar a sua instalação e validar se tudo funciona conforme o esperado. :D

Para que possamos executar um *container*, utilizamos o parâmetro "run" do subcomando "container" do comando "docker".

Simples, não? :D

```
root@linuxtips:~# docker container run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world

03f4658f8b78: Pull complete a3ed95caeb02: Pull complete
Digest:           sha256:8be990ef2aeb16dbcb9271ddfe2610fa6658d13f6dfb8b
c72074cc1ca36966a7
Status: Downloaded newer image for hello-world:latest

Hello from Docker.
This message shows that your installation appears to be working
correctly.

To generate this message, Docker took the following steps:
The Docker client contacted the Docker daemon.
The Docker daemon pulled the "hello-world" image from the Docker Hub.
The Docker daemon created a new container from that image which runs
the executable that produces the output you are currently reading.
The Docker daemon streamed that output to the Docker client, which
sent it to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
$ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker
Hub account: https://hub.docker.com

For more examples and ideas, visit: https://docs.docker.com/userguide/

root@linuxtips:~#
```

No exemplo anterior, estamos executando um *container* utilizando a imagem personalizada do *hello-world*.

Apesar de ser uma tarefa simples, quando você executou o comando "docker container run hello-world" foram necessárias quatro etapas para sua conclusão, vamos ver quais:

1. O comando "docker" se comunica com o *daemon* do Docker informando a ação desejada.

2. O *daemon* do Docker verifica se a imagem "hello-world" existe em seu *host*; caso ainda não, o Docker faz o *download* da imagem diretamente do Docker Hub.
3. O *daemon* do Docker cria um novo *container* utilizando a imagem que você acabou de baixar.
4. O *daemon* do Docker envia a saída para o comando "docker", que imprime a mensagem em seu terminal.

Viu? É simples como voar! :)

Muito bem, agora que nós já temos uma imagem em nosso *host*, como eu faço para visualizá-la?

Muito simples, basta digitar o seguinte comando:

```
root@linuxtips:~# docker image ls
REPOSITORY          TAG       IMAGE ID            CREATED       SIZE
hello-world         latest    690ed74de00f        5 months      960 B
root@linuxtips:~#
```

Como você pode notar no código, a saída traz cinco colunas:

⇨ **REPOSITORY** – O nome da imagem.

⇨ **TAG** – A versão da imagem.

⇨ **IMAGE ID** – Identificação da imagem.

⇨ **CREATED** – Quando ela foi criada.

⇨ **SIZE** – Tamanho da imagem.

Quando executamos o comando "docker container run hello-world", ele criou o *container*, imprimiu a mensagem na tela e depois o *container* foi finalizado automaticamente, ou seja, ele executou sua tarefa, que era exibir a mensagem, e depois foi finalizado.

Para ter certeza de que ele realmente foi finalizado, digite:

```
root@linuxtips:~# docker container ls
CONTAINER ID IMAGE COMMAND CREATED STATUS PORT NAMES
```

```
root@linuxtips:~#
```

Com o "docker container ls", você consegue visualizar todos os *containers* em execução e ainda obter os detalhes sobre eles. A saída do "docker container ls" é dividida em sete colunas; vamos conhecer o que elas nos dizem:

- ⇨ **CONTAINER ID** – Identificação única do *container*.
- ⇨ **IMAGE** – A imagem que foi utilizada para a execução do *container*.
- ⇨ **COMMAND** – O comando em execução.
- ⇨ **CREATED** – Quando ele foi criado.
- ⇨ **STATUS** – O seu status atual.
- ⇨ **PORT** – A porta do *container* e do *host* que esse *container* utiliza.
- ⇨ **NAMES** – O nome do *container*.

Uma opção interessante do "docker container ls" é o parâmetro "-a".

```
root@linuxtips:~# docker container ls -a
CONTAINER ID      IMAGE           COMMAND     CREATED     STATUS
PORTS NAMES
6e45cf509282      hello-world     "/hello"    4seconds    Exited(0)
tracted_ardinghelli
root@linuxtips:~#
```

Com a opção "-a" você consegue visualizar não somente os *containers* em execução, como também *containers* que estão parados ou que foram finalizados.

4.2. Legal, quero mais!

Agora que vimos como criar um simples *container*, bem como visualizar as imagens e os *containers* que estão em nosso *host*, vamos criar um novo, porém conhecendo três parâmetros que irão

trazer maior flexibilidade no uso e na administração de nossos *containers*. Estou falando dos parâmetros "-t", "-i" e "-d".

- ⇨ **-t** – Disponibiliza um TTY (console) para o nosso *container*.
- ⇨ **-i** – Mantém o STDIN aberto mesmo que você não esteja conectado no *container*.
- ⇨ **-d** – Faz com que o *container* rode como um *daemon*, ou seja, sem a interatividade que os outros dois parâmetros nos fornecem.

Com isso temos dois modos de execução de nossos *containers*: modo interativo ou *daemonizando* o *container*.

4.2.1. Modo interativo

Na maior parte das vezes você vai subir um *container* a partir de uma imagem que já está pronta, toda ajustadinha. Porém, há alguns casos em que você precisa interagir com o seu *container* – isso pode acontecer, por exemplo, na hora de montar a sua imagem personalizada.

Nesse caso, usar o modo interativo é a melhor opção. Para isso, basta passar os parâmetros "-ti" ao comando "docker container run".

4.2.2. *Daemonizando* o *container*

Utilizando o parâmetro "-d" do comando "docker container run", é possível *daemonizar* o *container*, fazendo com que o *container* seja executado como um processo *daemon*.

Isso é ideal quando nós já possuímos um *container* que não iremos acessar (via *shell*) para realizar ajustes. Imagine uma imagem já com a sua aplicação e tudo que precisa configurado; você irá subir o *container* e somente irá consumir o serviço entregue por sua aplicação. Se for uma aplicação *web*, basta acessar no *browser*

passando o IP e a porta onde o serviço é disponibilizado no *container*. Sensacional, não?

Ou seja, se você quer subir um *container* para ser utilizado como uma máquina Linux convencional com *shell* e que necessita de alguma configuração ou ajuste, utilize o modo interativo, ou seja, os parâmetros "-ti".

Agora, se você já tem o *container* configurado, com sua aplicação e todas as dependências sanadas, não tem a necessidade de usar o modo interativo – nesse caso utilizamos o parâmetro "-d", ou seja, o *container daemonizado*. Vamos acessar somente os serviços que ele provê, simples assim. :D

4.3. Entendi, agora vamos praticar um pouco?

Perfeito. Vamos iniciar um novo *container* utilizando dois desses novos parâmetros que aprendemos.

Para o nosso exemplo, vamos subir um *container* do Centos 7:

```
root@linuxtips:~# docker container run -ti centos:7
Unable to find image 'centos:7' locally
7: Pulling from library/centos

a3ed95caeb02: Pull complete 196355c4b639: Pull complete
Digest:           sha256:3cdc0670fe9130ab3741b126cfac6d7720492dd2c1c8ae
033dcd77d32855bab2
Status: Downloaded newer image for centos:7
[root@3c975fb7fbb5 /]#
```

Como a imagem não existia em nosso *host*, ele começou a baixar do Docker Hub, porém, caso a imagem já estivesse em nosso *host*, ele a utilizaria, não sendo necessário o *download*.

Perceba que mudou o seu *prompt* (variável $PS1), pois agora você já está dentro do *container*. Para provar que estamos dentro do nosso *container* Centos, execute o seguinte comando:

```
[root@3c975fb7fbb5 /]# cat /etc/redhat-release
CentOS Linux release 7.2.1511 (Core)
[root@3c975fb7fbb5 /]#
```

O arquivo "/etc/redhat-release" indica qual a versão do Centos que estamos utilizando, ou seja, estamos realmente em nosso *container* Centos 7. :D

4.4. Tá, agora quero sair...

Idealmente, no *container* vai haver apenas um processo rodando. No nosso caso, como estamos interagindo (opção "-ti"), é o processo do *bash*; logo, você não pode utilizar o comando "exit" para sair do console, pois dessa forma esse único processo para de rodar e seu *container* morre. Caso queira sair do *container* e mantê-lo em execução, é necessário sair com o seguinte atalho do teclado:

mantenha o botão Ctrl pressionado + p + q

Assim, você sairá do *container* e ele continuará em execução. Para confirmar se o *container* continua em execução, faça:

```
root@linuxtips:~# docker ps[1]
CONTAINER                          IDIMAGE  COMMAND      CREATED STATUS
PORTS NAMES
3c975fb7fbb5 centos:7 "/bin/bash"  2minutes Up 2 minutes
angry_wescoff
root@linuxtips:~#
```

4.5. Posso voltar ao *container*?

Deixamos o nosso *container* em execução e agora queremos acessá-lo novamente. Como podemos fazer?

Simples! Basta digitar o seguinte comando:

```
root@linuxtips:~# docker container attach <CONTAINER ID>
```

O parâmetro “attach” do comando “docker container” possibilita nos conectarmos a um *container* em execução. Para isso, basta passar como parâmetro o “CONTAINER ID”, que você consegue através da saída do “docker ps”, conforme mostramos no exemplo anterior.

4.6. Continuando com a brincadeira...

Existe a possibilidade de criar um *container,* porém não o executar imediatamente. Quando fazemos o uso do parâmetro “create” do comando “docker container”, ele apenas cria o *container*, não o inicializando, conforme notamos no exemplo a seguir:

```
root@linuxtips:~# docker container create -ti ubuntu
Unable to find image 'ubuntu:latest' locally
latest: Pulling from library/ubuntu

5a132a7e7af1: Pull complete
fd2731e4c50c: Pull complete
28a2f68d1120: Pull complete
a3ed95caeb02: Pull complete
Digest:
sha256:4e85ebe01d056b43955250bbac22bdb8734271122e3c78d21e55ee235fc6802
d
Status: Downloaded newer image for ubuntu:latest
3e63e65db85a6e36950959dc6bdc00279e2208a335580c478e01723819de9467
root@linuxtips:~#
```

Perceba que quando você digita “docker container ls” ele não traz o *container* recém-criado, afinal a saída do “docker container ls” somente traz os *containers* em execução. Para visualizar o *container* recém-criado foi necessário utilizar o parâmetro “-a”.

```
root@linuxtips:~# docker container ls -a
CONTAINER ID     IMAGE    COMMAND         CREATED              STATUS
PORTS NAMES
3e63e65db85a     ubuntu   "/bin/bash"     18 seconds ago       Created
```

```
elo_visves
root@linuxtips:~#
```

Para que o nosso *container* recém-criado seja executado, basta utilizar o "docker container start [CONTAINER ID]", conforme segue:

```
root@linuxtips:~# docker container start [CONTAINER ID]
root@linuxtips:~# docker container attach [CONTAINER ID]
root@b422f04df14c:/#
```

Verificando se estamos realmente utilizando o *container* do Ubuntu:

```
root@b422f04df14c:/# cat /etc/issue
Ubuntu 18.04 LTS \n \l
root@b422f04df14c:/#
```

Lembrando que para sair do *container* e mantê-lo em execução é necessário utilizar o atalho: **Ctrl + p + q**.

4.7. Subindo e matando *containers*...

Caso eu queira parar um *container* em execução, basta utilizar o parâmetro "stop" seguido do "CONTAINER ID":

```
# docker container stop [CONTAINER ID]
```

Verificando se o *container* continua em execução:

```
# docker container ls
```

Lembrando que para visualizar os *containers* que não estão em execução é necessário utilizar o parâmetro "-a".

Para colocar novamente em execução um *container* que está parado, é necessário utilizar o parâmetro "start" do comando "docker container" seguido do "CONTAINER ID":

```
# docker container start [CONTAINER ID]
```

Da mesma forma como podemos utilizar o *stop/start* para desligar/iniciar um *container*, podemos também fazer o uso do "restart", como notamos a seguir:

```
# docker container restart [CONTAINER ID]
```

Para pausar um *container*, execute:

```
# docker container pause [CONTAINER ID]
```

E verifique o status do *container*:

```
root@linuxtips:~# docker container ls
CONTAINER   IMAGE COMMAND    CREATED      STATUS
ID
PORTS   NAMES
b34f4987bdc ubunt "/bin/bash 12    seconds Up         11seconds (Paused)
e           u     "          ago          drunk_turi
root@linuxtips:~#
```

Para "despausar" o *container*:

```
# docker container unpause [CONTAINER ID]
```

4.8. Visualizando o consumo de recursos pelo *container*...

Caso você queira visualizar informações referentes ao consumo de recursos pelo *container*, também é bastante simples: basta utilizar o parâmetro "stats" para verificar o consumo de CPU, memória e rede pelo *container* em tempo real.

```
# docker container stats [CONTAINER ID]
CONTAINER   CPU% MEM USAGE / LIMIT  MEM  NET I/O
                                    %
BLOCK I/O   PIDS
b34f4987bdc 0.00 503.8  kB  /  2.094 0.02 648 B / 648 B   0 B / 0 B
```

```
e           %     GB                  %     2
```

Para sair, pressione **Ctrl + C.**

Para visualizar todos os *containers* de uma só vez, basta não especificar o [CONTAINER ID],conforme segue:

```
# docker container stats
```

Agora, se você quer visualizar quais processos estão em execução em determinado *container*, utilize o parâmetro "top". Com ele você consegue informações sobre os processos em execução, como, por exemplo, UID e o PID do processo.

```
# docker container top [CONTAINER ID]
UID    PID      PPID   C STIME    TTY      TIME          COMMAND
root   10656    4303   0 20:24    pts/3    00:00:00      /bin/bash
```

Para verificar os *logs* de um determinado *container*, utilize o parâmetro "logs", simples assim. :D

```
# docker container logs [CONTAINER ID]
```

Lembre-se: ele exibe o STDOUT, a saída padrão. Ou seja, normalmente você irá visualizar o histórico de mensagens que aparecerem em primeiro plano durante a execução do *container*.

Para exibir os *logs* de forma dinâmica, ou seja, conforme aparecem novas mensagens ele atualiza a saída no terminal, utilizamos a opção "-f".

```
# docker container logs -f [CONTAINER ID]
```

Com isso seu terminal ficará travado, apenas escutando o *log*, e qualquer nova entrada ele exibirá na tela. Saída parecida com o "tail -f" no Linux. Lembre-se, utilize o Ctrl+C para cancelar a exibição dos *logs*.

4.9. Cansei de brincar de *container*, quero removê-lo!

Bem, remover um *container* é mais simples ainda do que sua criação. Quando removemos um *container*, a imagem que foi utilizada para a sua criação permanece no *host*; somente o *container* é apagado.

```
root@linuxtips:~# docker container rm b34f4987bdce
Failed to remove container (b34f4987bdce): Error response from daemon: Conflict, You cannot remove a running container. Stop the container before attempting removal or use -f
root@linuxtips:~#
```

Perceba que, quando você tentou remover o *container*, ele retornou um erro dizendo que falhou em remover, pois o *container* estava em execução. Ele inclusive recomenda que você pare o *container* antes de removê-lo ou então utilize a opção "-f", forçando assim sua remoção.

```
root@linuxtips:~# docker container rm -f b34f4987bdce
b34f4987bdce
root@linuxtips:~#
```

Para confirmar a remoção do *container*, utilize o comando "docker container ls -a".

---

[1] Aqui usamos a maneira antiga para listar *containers*, ainda funciona! :D

5. Configurando CPU e memória para os meus *containers*

Vamos imaginar que você precise subir quatro *containers* para um projeto novo. Esses *containers* possuem as seguintes características:

⇨ Dois *web servers*.

⇨ Dois DB MySQL.

Evidentemente, por se tratar de serviços diferentes, na maioria dos casos possuem características de consumo de recursos, como CPU e memória, diferentes um do outro.

⇨ *Web server* – 0,5 CPU | 128 MB de memória

⇨ DB MySQL – 1 CPU | 256 MB de memória

E agora, como fazemos? :(

Por padrão, quando você executa um *container* sem especificar a quantidade de recursos que ele irá utilizar, ele sobe sem um controle, podendo inclusive impactar o *host* onde está sendo executado.

Portanto, é muito importante limitar a utilização de recursos de seus *containers*, visando um melhor aproveitamento de seus recursos computacionais, como veremos agora. :)

5.1. Especificando a quantidade de memória

Primeiro, vamos executar um *container* para realizarmos o nosso exemplo.

```
root@linuxtips:~# docker container run -ti --name teste debian
```

Agora vamos visualizar a quantidade de memória que está configurada para esse *container*. Uma forma fácil é utilizar a saída do comando "docker container inspect":

```
root@linuxtips:~# docker container inspect teste | grep -i mem
        "CpusetMems": "",
        "KernelMemory": 0,
        "Memory": 0,
        "MemoryReservation": 0,
        "MemorySwap": 0,
        "MemorySwappiness": -1,
```

Como percebemos, os valores correspondentes à memória estão zerados, ou seja, sem nenhum limite estabelecido.

Vamos agora subir um novo *container*, porém passando os valores para que utilize 512 MB de memória:

```
root@linuxtips:~# docker container run -ti -m 512M --name novo_container debian
```

Utilizamos o parâmetro "-m" para especificar a quantidade de memória que desejamos disponibilizar ao *container*. Vamos utilizar o "docker container inspect" novamente para verificar se a nossa configuração funcionou:

```
root@linuxtips:~# docker container inspect novo_container | grep -i mem
        "CpusetMems": "",
        "KernelMemory": 0,
        "Memory": 536870912,
        "MemoryReservation": 0,
        "MemorySwap": -1,
```

```
        "MemorySwappiness": -1,
```

Muito bom, parece que deu certo!

Podemos ver no campo "Memory" o valor utilizado na criação do *container*. Vale ressaltar que o valor exibido é em *bytes*.

Quando você utiliza o parâmetro "-m" ou "--memory", você está passando o máximo de memória que o *container* utilizará do *host*.

5.2. Especificando a quantidade de CPU

Para que você consiga limitar a quantidade de CPUs que os *containers* irão consumir, basta utilizar o parâmetro "--cpu". Com ele é possível dizer a quantidade de CPUs que você deseja disponibilizar para determinado *container*. Por exemplo, se utilizarmos "--cpus=0.5" estamos dizendo ao Docker para limitar o consumo pelo *container* em meio CPU. Fácil, não?

```
#docker container run --cpus=0.5 --name teste1 nginx
```

Com o comando anterior, estamos subindo um *container* utilizando a imagem do Nginx – até aqui nenhuma novidade, certo? Porém, agora passamos o parâmetro "--cpus=0.5" falando para o Docker que esse *container* poderá utilizar no máximo 0,5 CPU, ou seja, metade de 1 *core*. :D Simples como voar.

Para verificar, vamos utilizar o comando "docker container inspect":

```
root@linuxtips:~# docker container inspect teste1 | grep -i cpu
        "CpuShares": 0,
        "NanoCpus": 500000000,
        "CpuPeriod": 0,
        "CpuQuota": 0,
        "CpusetCpus": "",
        "CpusetMems": "",
```

O campo "NanoCpus" traz a informação que configuramos. :)

Simples, fácil e rápido!

5.3. Eu consigo alterar CPU e memória dos meus *containers* em execução?

Sim! \o/

Com o lançamento da versão 1.10 do Docker, temos o comando "docker update", que permite alterar as configurações referentes a CPU, memória e I/O com o *container* em execução de forma muito simples! Isso é fantástico!

Como exemplo, iremos subir um *container* e também alterar as informações referentes a memória e CPU:

```
root@linuxtips:~# docker container run -ti --cpus=4 -m 512m --name teste1 nginx

root@linuxtips:~# docker container inspect teste1 | grep -i cpu
        "CpuShares": 0,
        "NanoCpus": 4000000000,
        "CpuPeriod": 0,
        "CpuQuota": 0,
        "CpusetCpus": "",
        "CpusetMems": "",
root@linuxtips:~# docker container inspect teste1 | grep -i mem
        "CpusetMems": "",
        "KernelMemory": 0,
        "Memory": 536870912,
        "MemoryReservation": 0,
        "MemorySwap": -1,
        "MemorySwappiness": -1,
```

Agora, vamos alterar os valores de limite de CPU e memória:

```
root@linuxtips:~# docker container update -m 256m --cpus=1 teste1
teste1
root@linuxtips:~# docker container inspect teste1 | grep -i cpu
        "CpuShares": 0,
        "NanoCpus": 1000000000,
        "CpuPeriod": 0,
```

```
        "CpuQuota": 0,
        "CpusetCpus": "",
        "CpusetMems": "",
root@linuxtips:~# docker container inspect teste1 | grep -i mem
        "CpusetMems": "",
        "KernelMemory": 0,
        "Memory": 268435456,
        "MemoryReservation": 0,
        "MemorySwap": -1,
        "MemorySwappiness": -1,
```

Funcionou? SIMMM!

Assim, com os *containers* em execução, mudamos as informações referentes à memória e ao CPU!

Existem outros parâmetros do "docker container update". Para verificar a lista completa, digite "docker update --help".

6. Meu primeiro e tosco *dockerfile*...

Tudo que nós fizemos até agora foi escrever na linha de comando, o que é OK para aprender. Porém, principalmente nos dias de hoje, não dá para viver mais sem automatizar as coisas – se você, assim como nós, adora automatizar tudo que for possível, vai gostar bastante desse assunto.

O *dockerfile* nada mais é do que um arquivo onde você determina todos os detalhes do seu *container*, como, por exemplo, a imagem que você vai utilizar, aplicativos que necessitam ser instalados, comandos a serem executados, os volumes que serão montados, etc., etc., etc.!

É um *makefile* para criação de *containers*, e nele você passa todas as instruções para a criação do seu *container*.

Vamos ver como isso funciona na prática?

Primeira coisa: vamos criar um diretório onde deixaremos o nosso arquivo *dockerfile*, somente para ficar organizado. :D

Depois basta criar o *dockerfile* conforme exemplo a seguir:

```
# mkdir /root/primeiro_dockerfile
# cd /root/primeiro_dockerfile
# vim Dockerfile
```

Vamos adicionar as instruções que queremos para essa imagem de *container* que iremos criar:

```
FROM debian
RUN /bin/echo "HELLO DOCKER"
```

Apenas isso por enquanto. Salve e saia do *vim*.

Agora vamos rodar o comando "docker build" para fazer a criação dessa imagem de *container* utilizando o *dockerfile* criado.

```
root@linuxtips:~/primeiro_dockerfile# docker build -t tosko:1.0 .
Sending build context to Docker daemon 2.048 kB
Step 1/2 : FROM debian
latest: Pulling from library/debian

fdd5d7827f33: Pull complete a3ed95caeb02: Pull complete
Digest:           sha256:e7d38b3517548a1c71e41bffe9c8ae6d6d29546ce46bf6
2159837aad072c90aa
Status: Downloaded newer image for debian:latest
---> f50f9524513f
Step 2/2 : RUN /bin/echo "HELLO DOCKER"
---> Running in df60a0644bed HELLO DOCKER
---> fd3af97a8940
Removing intermediate container df60a0644bed
Successfully built fd3af97a8940
Successfully tagged tosko:1.0
root@linuxtips:~/primeiro_dockerfile#
```

Veja que usamos o diretório corrente, representado pelo caractere ".", para indicar o *path* do meu arquivo *dockerfile*, mas você não precisa necessariamente estar no mesmo diretório, basta passar o *path* do diretório onde o arquivo se encontra.

Lembre apenas que é o *path* do diretório e não do arquivo.

7. Entendendo volumes

7.1. Introdução a volumes no Docker

Bom, volumes nada mais são que diretórios externos ao *container*, que são montados diretamente nele, e dessa forma *bypassam* seu *filesystem*, ou seja, não seguem aquele padrão de camadas que falamos. Decepcionei você? Que bom, sinal de que é bem simples e você não vai ter problemas para entender. :)

A principal função do volume é persistir os dados. Diferentemente do *filesystem* do *container,* que é volátil e toda informação escrita nele é perdida quando o *container* morre, quando você escreve em um volume aquele dado continua lá, independentemente do estado do *container*.

Existem algumas particularidades entre os volumes e *containers* que valem a pena ser mencionadas:

- ⇨ O volume é inicializado quando o *container* é criado.
- ⇨ Caso ocorra de já haver dados no diretório em que você está montando como volume, ou seja, se o diretório já existe e está "populado" na imagem base, aqueles dados serão copiados para o volume.
- ⇨ Um volume pode ser reusado e compartilhado entre *containers*.
- ⇨ Alterações em um volume são feitas diretamente no volume.

- ⇨ Alterações em um volume não irão com a imagem quando você fizer uma cópia ou *snapshot* de um *container*.
- ⇨ Volumes continuam a existir mesmo se você deletar o *container*.

Dito isso, chega de papo. Vamos aprender a adicionar um volume em um *container*.

Primeiro, vamos ver como funciona da maneira antiga, que ainda é suportada, porém não é elegante. :)

Essa maneira é muito utilizada quando se quer montar um diretório específico do *host* dentro do *container*. Isso é ruim quando estamos trabalhando em *cluster*, uma vez que teríamos que garantir esse diretório criado em todos os *hosts* do *cluster*. Não seria legal.

Porém, podemos aprender como funciona e utilizar em algum momento, caso se faça necessário. Para evitar erros, primeiro crie o diretório "/volume" na sua máquina.

```
root@linuxtips:~# mkdir /volume
root@linuxtips:~#     docker     container     run     -ti     --mount
type=bind,src=/volume,dst=/volume ubuntu


root@7db02e999bf2:/# df -h
Filesystem                      Size Used Avail Use% Mounted on
none                            13G  6.8G 5.3G  57%  /
tmpfs                           999M 0    999M  0%   /dev
tmpfs                           999M 0    999M  0%   /sys/fs/cgroup
/dev/mapper/ubuntu--vg-root     13G  6.8G 5.3G  57%  /volume
shm                             64M  0    64M   0%   /dev/shm
root@7db02e999bf2:/# ls
bin boot dev etc home lib lib64 media mnt opt proc root run sbin srv
sys tmp usr var volume
root@7db02e999bf2:/#
```

No exemplo anterior, conhecemos um novo parâmetro do comando "docker container run", o "--mount".

O parâmetro “--mount” é o responsável por indicar o volume, que em nosso exemplo é o “/volume”, e onde ele será montado no *container*. Perceba que, quando passamos o parâmetro “--mount type=bind,src=/volume,dst=/volume”, o Docker montou esse diretório no *container*, porém sem nenhum conteúdo.

Podemos também montar um volume no *container linkando-o* com um diretório do *host* já com algum conteúdo. Para exemplificar, vamos compartilhar o diretório “/root/primeiro_container”, que utilizamos para guardar o nosso primeiro *dockerfile*, e montá-lo no *container* em um volume chamado “/volume” da seguinte forma:

```
# docker container run -ti --mount
type=bind,src=/root/primeiro_container,dst=/volume ubuntu


root@3d372a410ea2:/# df -h
Filesystem                      Size Used Avail Use% Mounted on
none                            13G  6.8G 5.3G  57%  /
tmpfs                           999M 0    999M  0%   /dev
tmpfs                           999M 0    999M  0%   /sys/fs/cgroup
/dev/mapper/ubuntu--vg-root     13G  6.8G 5.3G  57%  /volume
shm                             64M  0    64M   0%   /dev/shm
root@3d372a410ea2:/#
```

Com isso, estamos montando o diretório “/root/primeiro_dockerfile” do *host* dentro do *container* com o nome de “/volume”.

No *container*:

```
root@3d372a410ea2:/# ls /volume/
Dockerfile
root@3d372a410ea2:/#
```

No *host*:

```
root@linuxtips:~# ls /root/primeiro_dockerfile/
Dockerfile
```

```
root@linuxtips:~#
```

Caso eu queira deixar o volume no *container* apenas como *read-only,* é possível. Basta passar o parâmetro "ro" após o destino onde será montado o volume:

```
# docker container run -ti --mount
type=bind,src=/root/primeiro_container,dst=/volume,ro ubuntu


root@8d7863b1d9af:/# df -h
Filesystem                      Size Used Avail Use% Mounted on
none                            13G  6.8G 5.3G  57%  /
tmpfs                           999M 0    999M  0%   /dev
tmpfs                           999M 0    999M  0%   /sys/fs/cgroup
/dev/mapper/ubuntu--vg-root     13G  6.8G 5.3G  57%  /volume
shm                             64M  0    64M   0%   /dev/shm
root@8d7863b1d9af:/# cd /volume/
root@8d7863b1d9af:/volume# ls
Dockerfile
root@8d7863b1d9af:/volume# mkdir teste
mkdir: cannot create directory 'teste': Read-only file system
root@8d7863b1d9af:/volume#
```

Assim como é possível montar um diretório como volume, também é possível montar um arquivo:

```
# docker container run -ti --mount
type=bind,src=/root/primeiro_container/Dockerfile,dst=/Dockerfile/
ubuntu


root@df0e3e58280a:/# df -h
Filesystem                      Size Used Avail Use% Mounted on
none                            13G  6.8G 5.3G  57%  /
tmpfs                           999M 0    999M  0%   /dev
tmpfs                           999M 0    999M  0%   /sys/fs/cgroup
/dev/mapper/ubuntu--vg-root     13G  6.8G 5.3G  57%  /Dockerfile
shm                             64M  0    64M   0%   /dev/shm
root@df0e3e58280a:/# cat Dockerfile
```

```
FROM debian
RUN /bin/echo "HELLO DOCKER"
root@df0e3e58280a:/#
```

Isso faz com que o arquivo "/root/primeiro_dockerfile/Dockerfile" seja montado em "/Dockerfile" no *container*.

7.2. Criando volumes

Agora vamos criar os volumes da maneira mais elegante e atual. Hoje temos a possibilidade de realizar o gerenciamento de volumes de maneira muito simples e inteligente.

Sempre que criamos um volume, ele cria um diretório com o mesmo nome dentro de "/var/lib/docker/volumes/".

No exemplo a seguir, o volume "giropops" seria então criado em "/var/lib/docker/volumes/giropops"; com isso, todos os arquivos disponíveis nesse diretório também estariam disponíveis no local indicado no *container*. Vamos aos exemplos! :D

É possível fazer a criação de volumes e toda a sua administração através do comando:

```
# docker volume create giropops
```

É possível removê-lo através do comando:

```
# docker volume rm giropops
```

Para verificar detalhes sobre esse volume:

```
# docker volume inspect giropops
```

Para remover os volumes que não estão sendo utilizados (use com extrema moderação! :D):

```
# docker volume prune
```

Para que você possa montar o volume criado em algum *container*/*service*, basta executar o seguinte comando:

```
# docker container run -d --mount type=volume,source=giropops,destination=/var/opa nginx
```

Onde:

- ⇨ **--mount** – Comando utilizado para montar volumes.
- ⇨ **type=volume** – Indica que o tipo é "volume". Ainda existe o tipo "bind", onde, em vez de indicar um volume, você indicaria um diretório como *source*.
- ⇨ **source=giropops** – Qual o volume que pretendo montar.
- ⇨ **destination=/var/opa** – Onde no *container* montarei esse volume.

Simples como voar, não?

7.3. Localizando volumes

Caso você queira obter a localização do seu volume, é simples. Mas para isso você precisa conhecer o comando "docker volume inspect".

Com o "docker volume inspect" você consegue obter detalhes do seu *container*, como, por exemplo, detalhes do volume.

A saída do comando "docker volume inspect" retorna mais informação do que somente o *path* do diretório no *host*. Vamos usar a opção "--format" ou "-f" para filtrar a saída do "inspect".

```
docker volume inspect --format '{{ .Mountpoint }}' giropops
/var/lib/docker/volumes/giropopos/_data
```

7.4. Criando e montando um *data-only container*

Uma opção bastante interessante em relação aos volumes diz respeito ao *data-only container*, cuja principal função é prover

volumes para outros *containers*. Lembra do NFS *server* e do Samba? Ambos centralizavam diretórios com a finalidade de compartilhar entre outros servidores. Pois bem, o *data-only container* tem a mesma finalidade: prover volumes a outros *containers*.

Um dos grandes baratos do Docker é a portabilidade. Um *container* criado no seu *laptop* deve ser portátil a ponto de rodar em qualquer outro ambiente que utilize o Docker, em todos os cantos do universo!

Sendo assim, o que acontece se eu criar um ambiente em Docker que diz para os *containers* montarem um diretório do *host* local? Depende. Depende de como está esse tal *host* local. Perguntas como "o diretório existe?" "as permissões estão ajustadas?", entre outras mais, definirão o sucesso na execução dos *containers*, o que foge completamente do escopo do Docker.

Vamos ver como funciona isso na prática! :)

Para o nosso exemplo, primeiro vamos criar um *container* chamado "dbdados", com um volume chamado "/data", que guardará os dados de um banco PostgreSQL.

Para que possamos criar um *container* especificando um nome para ele, utilizamos o parâmetro "--name", conforme veremos no exemplo a seguir:

```
# docker container create -v /data --name dbdados centos
```

Com isso, apenas criamos o *container* e especificamos um volume para ele, mas ainda não o iniciamos.

Sabemos que no *container* o volume se encontra montado em "/data". Porém, qual a localização desse volume no *host*?

Lembra do "docker inspect"? Vamos utilizá-lo novamente:

```
root@linuxtips:~# docker inspect -f {{.Mounts}} dbdados
[{46255137fe3f6d5f593e9ba9aaaf570b2f8b5c870f587c2fb34f29b79f97c30c
```

```
/var/lib/docker/volumes/46255137fe3f6d5f593e9ba9aaaf570b2f8b5
c870f587c2fb34f29b79f97c30c/_data /data local true }]
```

Perceba que agora utilizamos o nome do *container* em vez do "CONTAINER ID". Totalmente possível e muito mais intuitivo.

Quando executamos o "docker inspect", ele nos retornou o caminho do nosso volume. Vamos ver se existe algum conteúdo dentro dele:

```
root@linuxtips:~# ls \ /var/lib/docker/volumes/46255137fe3f6d
5f593e9ba9aaaf570b2f8b5c870f \ 587c2fb34f29b79f97c30c/_data
```

Como vimos, o diretório ainda não possui conteúdo.

Agora vamos criar os *containers* que rodarão o PostgreSQL utilizando o volume "/data" do *container* "dbdados" para guardar os dados.

Para que possamos fazer o exemplo, precisamos conhecer mais dois parâmetros superimportantes:

- ⇨ **--volumes-from** – É utilizado quando queremos montar um volume disponibilizado por outro *container*.
- ⇨ **-e** – É utilizado para informar variáveis de ambiente para o *container*. No exemplo, estamos passando as variáveis de ambiente do PostgreSQL.

Pronto, agora estamos preparados! Vamos criar os *containers* com o PostgreSQL:

```
# docker run -d -p 5432:5432 --name pgsql1 --volumes-from dbdados \
-e POSTGRESQL_USER=docker -e POSTGRESQL_PASS=docker \
-e POSTGRESQL_DB=docker kamui/postgresql

# docker run -d -p 5433:5432 --name pgsql2 --volumes-from dbdados \
-e POSTGRESQL_USER=docker -e POSTGRESQL_PASS=docker \
-e POSTGRESQL_DB=docker kamui/postgresql
```

Para verificar os dois *containers* com o PostgreSQL em execução, utilize o "docker container ls".

Pronto, agora temos os dois *containers* com PostgreSQL em execução! Será que já temos algum dado no volume "/data" do *container* "dbdados"?

Vamos verificar novamente no *host* se o volume agora possui algum dado:

```
root@linuxtips:~# ls /var/lib/docker/volumes/46255137fe3f6d5f593e9ba9aaaf570b2f8b5c870f587c2fb34f29b79f97c30c/_data
base  pg_clog  pg_ident.conf  pg_notify  pg_snapshots  pg_stat_tmp  pg_tblspc  PG_VERSION  postgresql.conf  postmaster.pid  server.key  global  pg_hba.conf  pg_multixact  pg_serial  pg_stat  pg_subtrans  pg_twophase  pg_xlog  postmaster.opts  server.crt
root@linuxtips:~#
```

Sensacional! Como percebemos, os dois *containers* do PostgreSQL estão escrevendo seus dados no volume "/data" do *container* "dbdados". Chega a ser lacrimejante! :D

7.5. Sempre é bom um *backup*...

Outra coisa bem bacana é a possibilidade de fazer *backups* dos seus *containers* de dados de forma muito simples e rápida.

Digamos que você queira fazer o *backup* do diretório "/data" do *container* "dbdados" que nós criamos há pouco; como faríamos?

```
root@linuxtips:~# cd backup/
root@linuxtips:~/backup# docker run -ti --volumes-from dbdados -v $(pwd):/backup debian tar -cvf /backup/backup.tar /data
```

Quando executamos o comando anterior, é criado um novo *container* montando o(s) volume(s) do *container* "dbdados" (que no caso é o "/data", lembra?). Além disso, será montado o diretório corrente do *host* no volume "/backup" do *container*, e em seguida será

executado o comando do *tar* para empacotar o diretório "/data" dentro do diretório "/backup". Bem simples, né?

```
root@linuxtips:~/backup# ls
backup.tar
root@linuxtips:~/backup#
```

Lembrando que os volumes são sempre criados dentro de "/var/lib/docker/volumes". Caso queira fazer o *backup* de todos os volumes, basta tratar esse diretório em suas rotinas de *backup*. ;)

8. Criando e gerenciando imagens

8.1. Agora eu quero criar minha imagem, posso?

Claro que pode!

E mais, vamos aprender de duas formas simples e intuitivas.

Uma das coisas mais interessantes do Docker é a possibilidade de usar imagens criadas por outras pessoas ao redor do mundo através de algum *registry* como o Docker Hub. Isso agiliza muito a sua vida, ainda mais quando você precisa apenas testar uma determinada tecnologia. O POC (*Proof of Concept* – em português, prova de conceito) se torna muito mais ágil, fazendo com que você consiga testar diversas ferramentas no mesmo tempo em que levaria para testar somente uma sem o Docker.

Entretanto, em determinados momentos precisamos criar a nossa própria imagem do zero, ou então modificar uma imagem criada por terceiros e salvar essas alterações em uma nova imagem.

Agora vamos ver os dois casos: como montar uma distribuição praticamente do zero utilizando somente instruções através do *dockerfile* e outra realizando modificações em uma imagem já existente e salvando em uma imagem nova.

8.2. Vamos começar do começo então, *dockerfile*!

Vamos montar a nossa primeira imagem utilizando como roteiro de criação um *dockerfile*. Você verá o quanto é simples a criação de um *dockerfile* bem completo e prático. :)

Para começar, vamos criar um diretório chamado "/root/Dockerfiles".

```
# mkdir /root/Dockerfiles
```

Agora começaremos a criação do nosso *dockerfile*, nosso mapa de criação da imagem. Para que possamos organizá-lo melhor, vamos criar um diretório chamado "apache", onde guardaremos esse nosso primeiro exemplo:

```
# cd /root/Dockerfiles/
# mkdir apache
```

Por enquanto, vamos apenas criar um arquivo chamado "Dockerfile" e adicionar o conteúdo conforme exemplo a seguir:

```
# cd apache
# vim Dockerfile

FROM debian

RUN apt-get update && apt-get install -y apache2 && apt-get clean
ENV APACHE_LOCK_DIR="/var/lock"
ENV APACHE_PID_FILE="/var/run/apache2.pid"
ENV APACHE_RUN_USER="www-data"
ENV APACHE_RUN_GROUP="www-data"
ENV APACHE_LOG_DIR="/var/log/apache2"

LABEL description="Webserver"

VOLUME /var/www/html/
EXPOSE 80
```

Muito bom! Agora que você já adicionou as informações conforme o exemplo, vamos entender cada seção utilizada nesse nosso primeiro *dockerfile*:

- ⇨ **FROM** – Indica a imagem a servir como base.
- ⇨ **RUN** – Lista de comandos que deseja executar na criação da imagem.
- ⇨ **ENV** – Define variáveis de ambiente.
- ⇨ **LABEL** – Adiciona *metadata* à imagem, como descrição, versão, etc.
- ⇨ **VOLUME** – Define um volume a ser montado no *container*.

Após a criação do arquivo, vamos *buildar* (construir a nossa imagem) da seguinte forma:

```
# docker build .
```

Lembre-se: você deverá estar no diretório onde está o seu *dockerfile*.

Todos os passos que definimos em nosso *dockerfile* serão realizados, como a instalação dos pacotes solicitados e todas as demais tarefas.

```
Successfully built 53de2cee9e71
```

Muito bem! Como podemos notar na última linha da saída do "docker build", a imagem foi criada com sucesso! :D

Vamos executar o "docker image ls" para ver se está tudo certo com a nossa primeira imagem!

```
root@linuxtips:~/Dockerfile/apache# docker image ls
REPOSITORY    TAG      IMAGE ID         CREATED            SIZE
<none>        <none>   53de2cee9e71     2 minutes ago      193.4 MB
```

A nossa imagem foi criada! Porém, temos um problema. :/

A imagem foi criada e está totalmente funcional, mas, quando a *buildamos*, não passamos o parâmetro "-t", que é o responsável por adicionar uma *tag* ("nome:versão") à imagem.

Vamos executar novamente o *build*, porém passando o parâmetro '-t', conforme o exemplo a seguir:

```
# docker build -t linuxtips/apache:1.0 .
```

Agora vamos ver se realmente a imagem foi criada, adicionando um nome e uma versão a ela:

```
root@linuxtips:~/Dockerfile/apache# docker image ls
REPOSITORY            TAG IMAGE ID         CREATED           SIZE
linuxtips/apache      1.0 53de2cee9e71     5 minutes ago     193.4 MB
```

Maravilha! Funcionou conforme esperávamos!

Vamos executar um *container* utilizando nossa imagem como base:

```
# docker container run -ti linuxtips/apache:1.0
```

Agora já estamos no *container*. Vamos verificar se o Apache2 está em execução. Se ainda não estiver, vamos iniciá-lo e verificar se a porta 80 está "LISTEN".

```
root@70dd36fe2d3b:/# ps -ef
UID    PID   PPID    C STIME TTY        TIME CMD
root   1     0       1 21:33 ?          00:00:00 /bin/bash
root   6     1       0 21:33 ?          00:00:00 ps -ef

root@70dd36fe2d3b:/# /etc/init.d/apache2 start
[....] Starting web server: apache2AH00558: apache2: Could not
reliably determine the server's fully qualified domain name, using
172.17.0.4. Set the 'ServerName' directive globally to suppress this
message
. ok

root@70dd36fe2d3b:/# ps -ef
UID       PID PPID C STIME TTY  TIME CMD
root      1   0    0 21:33 ?    00:00:00 /bin/bash
root      30  1    0 21:33 ?    00:00:00 /usr/sbin/apache2 -k start
```

```
www-data  33  30   0 21:33 ?    00:00:00 /usr/sbin/apache2 -k start
www-data  34  30   0 21:33 ?    00:00:00 /usr/sbin/apache2 -k start
root      109 1    0 21:33 ?    00:00:00 ps -ef
```

```
root@70dd36fe2d3b:/# ss -atn
State   Recv-Q  Send-Q  Local Address:Port      Peer Address:Port
LISTEN  0       128     :::80                   :::*
```

```
root@70dd36fe2d3b:/# ip addr show eth0
50: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
state UP group default
   link/ether 02:42:ac:11:00:04 brd ff:ff:ff:ff:ff:ff
   inet 172.17.0.4/16 scope global eth0
     valid_lft forever preferred_lft forever
   inet6 fe80::42:acff:fe11:4/64 scope link
     valid_lft forever preferred_lft forever
root@70dd36fe2d3b:/#
```

No código anterior é possível observar o IP do *container* na saída do "ip addr". Vamos testar a comunicação com o *container* a partir do *host*.

No *host*, digite:

```
# curl <IP DO CONTAINER>
```

O "curl" retornou a página de boas-vindas do Apache2, ou seja, tudo está funcionando muito bem e o Apache2, respondendo conforme esperado!

Maaaaaasssss, não é interessante que eu tenha que entrar no *container* para subir o meu processo do Apache. Todo *container* deve executar seu processo em primeiro plano e esse processo deve subir de forma automática e não com um *serumaninho* acessando o *container* e subindo o serviço. Vimos antes somente como primeiro exemplo, agora vamos aperfeiçoá-lo e deixá-lo como deve estar! :D

A primeira coisa é deixar o nosso *dockerfile* como segue:

```
# vim Dockerfile
```

```
FROM debian

RUN apt-get update && apt-get install -y apache2 && apt-get clean
ENV APACHE_LOCK_DIR="/var/lock"
ENV APACHE_PID_FILE="/var/run/apache2/apache2.pid"
ENV APACHE_RUN_USER="www-data"
ENV APACHE_RUN_DIR="/var/run/apache2"
ENV APACHE_RUN_GROUP="www-data"
ENV APACHE_LOG_DIR="/var/log/apache2"

LABEL description="Webserver"

VOLUME /var/www/html/
EXPOSE 80

ENTRYPOINT ["/usr/sbin/apachectl"]
CMD ["-D", "FOREGROUND"]
```

Perceba que agora nós adicionamos mais duas opções: o ENTRYPOINT e o CMD!

Ficou curioso sobre o que eles fazem? Então 'bora aprender muito mais opções possíveis de serem adicionadas em um *dockerfile*!

8.3. Vamos aprender um pouco mais sobre *dockerfile*?

Vamos agora aprender um pouco mais sobre as opções que podemos utilizar quando estamos criando um *dockerfile*:

- ⇨ **ADD** – Copia novos arquivos, diretórios, arquivos TAR ou arquivos remotos e os adiciona ao *filesystem* do *container*.
- ⇨ **CMD** – Executa um comando. Diferentemente do RUN, que executa o comando no momento em que está "buildando" a imagem, o CMD irá fazê-lo somente quando o *container* for iniciado.
- ⇨ **LABEL** – Adiciona metadados à imagem, como versão, descrição e fabricante.

- ⇨ **COPY** – Copia novos arquivos e diretórios e os adiciona ao *filesystem* do *container*.
- ⇨ **ENTRYPOINT** – Permite que você configure um *container* para rodar um executável. Quando esse executável for finalizado, o *container* também será.
- ⇨ **ENV** – Informa variáveis de ambiente ao *container*.
- ⇨ **EXPOSE** – Informa qual porta o *container* estará ouvindo.
- ⇨ **FROM** – Indica qual imagem será utilizada como base. Ela precisa ser a primeira linha do *dockerfile*.
- ⇨ **MAINTAINER** – Autor da imagem.
- ⇨ **RUN** – Executa qualquer comando em uma nova camada no topo da imagem e "commita" as alterações. Essas alterações você poderá utilizar nas próximas instruções de seu *dockerfile*.
- ⇨ **USER** – Determina qual usuário será utilizado na imagem. Por *default* é o *root*.
- ⇨ **VOLUME** – Permite a criação de um ponto de montagem no *container*.
- ⇨ **WORKDIR** – Responsável por mudar do diretório "/" (raiz) para o especificado nele.

Um detalhe superimportante de mencionar é que quando estamos trabalhando com o ENTRYPOINT e o CMD dentro do mesmo *dockerfile*, o CMD somente aceita parâmetros do ENTRYPOINT, conforme nosso exemplo do *dockerfile* anterior:

```
ENTRYPOINT ["/usr/sbin/apachectl"]
CMD ["-D", "FOREGROUND"]
```

Onde:

- ⇨ **"/usr/sbin/apachectl"** – Esse é o comando.
- ⇨ **"-D", "FOREGROUND"** – Esse é o argumento, o parâmetro.

No *shell*, por exemplo, a execução ficaria assim:

```
# /usr/sbin/apachectl -D FOREGROUND
```

Ou seja, assim você está iniciando o Apache passando a instrução para que ele seja iniciado em primeiro plano, como deve ser. :D

Para maiores detalhes sobre como criar imagens, veja essa apresentação criada pelo Jeferson: <https://www.slideshare.net/jfnredes/images-deep-dive>.

8.4. *Multi-stage*

Um importante e recente recurso adicionado ao *dockerfile* visa facilitar a vida de quem pretende criar imagens de *containers* de forma efetiva. Esse cara é o *multi-stage*!

O *multi-stage* nada mais é do que a possibilidade de você criar uma espécie de *pipeline* em nosso *dockerfile*, podendo inclusive ter duas entradas FROM.

Esse recurso é muito utilizado quando queremos, por exemplo, compilar a nossa aplicação em um *container* e executá-la, porém não queremos ter aquela quantidade enorme de pacotes instalados em nossos *containers* necessários sempre quando se quer compilar códigos de alguma linguagem, como C, Java ou Golang.

Vamos a um exemplo para que possamos entender melhor como isso funciona!

Para isso, preparamos uma *app* escrita em Golang superavançada para o nosso teste:

```
# vim goapp.go

package main
import "fmt"

func main() {
   fmt.Println("GIROPOPS STRIGUS GIRUS - LINUXTIPS")
}
```

Achou que seria algo avançado? Impossível, fomos nós que fizemos. :D

Bem, agora vamos criar um *dockerfile* para criar a nossa imagem e assim executar a nossa *app*.

```
# vim Dockerfile
FROM golang

WORKDIR /app
ADD . /app
RUN go build -o goapp
ENTRYPOINT ./goapp
```

Pronto! Agora vamos realizar o *build*.

```
# docker build -t goapp:1.0 .
```

Listando a nossa imagem:

```
# docker image ls | grep goapp
goapp    1.0   50451808b384         11 seconds ago           781MB
```

Agora vamos executá-la e ver a nossa fantástica *app* em execução:

```
# docker container run -ti goapp:1.0
GIROPOPS STRIGUS GIRUS - LINUXTIPS
```

Pronto! Nossa *app* e nossa imagem estão funcionando! Sucesso!

Porém, podemos melhorar muita coisa se começarmos a utilizar o nosso poderoso recurso, o *multi-stage*!

Vamos refazer o nosso *dockerfile* utilizando o *multi-stage*, entender como ele funciona e a diferença entre as duas imagens.

Vamos deixar nosso *dockerfile* dessa maneira:

```
# vim Dockerfile
FROM golang AS buildando

ADD . /src
```

```
WORKDIR /src
RUN go build -o goapp

FROM alpine:3.1

WORKDIR /app
COPY --from=buildando /src/goapp /app
ENTRYPOINT ./goapp
```

Perceba que agora nós temos duas entradas FROM, o que não era possível antes do *multi-stage*. Mas por que isso?

O que está acontecendo é que agora temos o *dockerfile* dividido em duas seções. Cada entrada FROM define o início de um bloco, uma etapa.

Então, em nosso primeiro bloco temos:

- ⇨ **FROM golang AS buildando** – Estamos utilizando a imagem do Golang para criação da imagem de *container*, e aqui estamos apelidando esse bloco como "buildando".
- ⇨ **ADD . /src** – Adicionando o código de nossa *app* dentro do *container* no diretório "/src".
- ⇨ **WORKDIR /src** – Definindo que o diretório de trabalho é o "/src", ou seja, quando o *container* iniciar, estaremos nesse diretório.
- ⇨ **RUN go build -o goapp** – Vamos executar o *build* de nossa *app* Golang.

Já no segundo bloco temos o seguinte:

- ⇨ **FROM alpine:3.1** – Iniciando o segundo bloco e utilizando a imagem do Alpine para criação da imagem de *container*.
- ⇨ **WORKDIR /app** – Definindo que o diretório de trabalho é o "/app", ou seja, quando o *container* iniciar, estaremos nesse diretório.

- ⇨ **COPY --from=buildando /src/goapp /app** – Aqui está a mágica: vamos copiar do bloco chamado "buildando" um arquivo dentro de "/src/goapp" para o diretório "/app" do *container* que estamos tratando nesse bloco, ou seja, copiamos o binário que foi compilado no bloco anterior e o trouxemos para esse.
- ⇨ **ENTRYPOINT ./goapp** – Aqui vamos executar a nossa sensacional *app*. :)

Agora que já entendemos todas as linhas do nosso novo *dockerfile*, 'bora realizar o *build* dele.

```
# docker build -t goapp_multistage:1.0 .
```

Vamos executar a nossa imagem para ver se está tudo funcionando:

```
# docker container run -ti goapp_multistage:1.0
GIROPOPS STRIGUS GIRUS - LINUXTIPS
```

Será que existe diferença de tamanho entre elas? Vamos conferir:

```
# docker image ls | grep goapp
goapp_multistage      1.0 dfe57485b7f0     22 seconds ago      7.07MB
goapp                 1.0 50451808b384     15 minutes ago      781MB
```

A diferença de tamanho é brutal, pois em nossa primeira imagem precisamos ter um monte de pacotes para que o *build* da *app* Golang ocorra. Já em nossa segunda imagem também utilizamos a imagem do Golang e todos os seus pacotes para *buildar* a nossa *app*, porém descartamos a primeira imagem e somente copiamos o binário para o segundo bloco, onde estamos utilizando a imagem do Alpine, que é superenxuta.

Ou seja, utilizamos o primeiro bloco para compilar a nossa *app* e o segundo bloco somente para executá-la. Simples assim, simples

como voar! :D

8.5. Vamos customizar uma imagem base agora?

Vamos agora criar uma nova imagem, porém sem utilizar o *dockerfile*. Vamos executar um *container* com uma imagem base, realizar as modificações que desejarmos e depois salvar esse *container* como uma nova imagem!

Simples, rápido e fácil!

Bem, primeiro precisamos criar um *container*. Vamos dessa vez utilizar um *container* Debian, somente para variar. :D

```
root@linuxtips:~# docker container run -ti debian:8 /bin/bash
root@0b7e6f606aae:/#
```

Agora vamos fazer as alterações que desejamos. Vamos fazer o mesmo que fizemos quando montamos nossa primeira imagem com o *dockerfile*, ou seja, fazer a instalação do Apache2. :D

```
root@0b7e6f606aae:/# apt-get update && apt-get install -y apache2 &&
apt-get clean
```

Agora que já instalamos o Apache2, vamos sair do *container* para que possamos *commitar* a nossa imagem com base nesse *container* em execução.

Lembre-se de que para sair do *container* e deixá-lo ainda em execução é necessário pressionar **Ctrl + p + q**. ;)

```
# docker container ls
# docker commit -m "meu container" CONTAINERID
# docker image ls
REPOSITORY     TAG      IMAGE ID         CREATED            SIZE
<none>         <none>   fd131aedd43a     4 seconds ago      193.4 MB
```

Repare que nossa imagem ficou com o "<none>" em seu nome e "TAG". Para que possamos ajustar e dar um nome e uma versão à nossa imagem, vamos usar o comando "docker tag", conforme mostramos a seguir:

```
# docker tag IMAGEID linuxtips/apache_2:1.0
REPOSITORY              TAG IMAGE ID        CREATED          SIZE
linuxtips/apache_2      1.0 fd131aedd43a    2 minutes ago    193.4 MB
```

Agora sim!!! Temos a nossa imagem criada e nome e versão especificados.

Vamos iniciar um *container* utilizando a imagem que acabamos de criar:

```
# docker container run -ti linuxtips/apache_2:1.0 /bin/bash
```

Vamos subir o Apache2 e testar a comunicação do *container*:

```
root@57094ec894ce:/# ps -ef

UID    PID   PPID    C STIME TTY        TIME CMD
root   1     0       0 21:48 ?          00:00:00 /bin/bash
root   6     1       0 21:48 ?          00:00:00 ps -ef
root@57094ec894ce:/# /etc/init.d/apache2 start
[....] Starting web server: apache2AH00558: apache2: Could not
reliably determine the server's fully qualified domain name, using
172.17.0.6. Set the 'ServerName' directive globally to suppress this
message
. ok


root@70dd36fe2d3b:/# ps -ef
UID        PID PPID C STIME TTY  TIME CMD
root       1   0    0 21:43 ?    00:00:00 /bin/bash
root       30  1    0 21:44 ?    00:00:00 /usr/sbin/apache2 -k start
www-data   33  30   0 21:44 ?    00:00:00 /usr/sbin/apache2 -k start
www-data   34  30   0 21:44 ?    00:00:00 /usr/sbin/apache2 -k start
root       111 1    0 21:44 ?    00:00:00 ps -ef
```

```
root@70dd36fe2d3b:/# ss -atn
State   Recv-Q  Send-Q  Local Address:Port      Peer Address:Port
LISTEN  0       128     :::80                   :::*

root@57094ec894ce:/# ip addr show eth0
54: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
state UP group default
   link/ether 02:42:ac:11:00:06 brd ff:ff:ff:ff:ff:ff
   inet 172.17.0.6/16 scope global eth0
     valid_lft forever preferred_lft forever
   inet6 fe80::42:acff:fe11:6/64 scope link
     valid_lft forever preferred_lft forever
```

Boaaa! Agora já temos o Apache2 em execução. Vamos sair do *container* e testar a comunicação com o Apache2 a partir do *host*:

```
# curl <Container IP>
```

Ele retornará a página de boas-vindas do Apache2! Tudo funcionando conforme esperado!

9. Compartilhando as imagens

Bem, já aprendemos como criar uma imagem de *container*, seja via *dockerfile* ou através da modificação de um *container*, e conhecemos alguns comandos interessantes, como o "docker build" e o "docker commit".

Agora vamos aprender a compartilhar essas imagens, seja em um *registry* local ou então no *registry* do próprio Docker Hub.

9.1. O que é o Docker Hub?

Docker Hub é um repositório público e privado de imagens que disponibiliza diversos recursos, como, por exemplo, sistema de autenticação, *build* automático de imagens, gerenciamento de usuários e departamentos de sua organização, entre outras funcionalidades.

Pessoas e empresas se juntam, criam seus *containers* seguindo as melhores práticas, testam tudo direitinho e depois disponibilizam lá para você usar sem ter nenhum trabalho. Isso é uma mão na roda gigantesca, uma vez que você não vai ter que perder tempo instalando coisas. Você também pode usá-lo para aprender como configurar determinado serviço. Para isso, basta ir no Docker Hub e procurar; provavelmente alguém já criou um *container* que você poderá usar pelo menos de base!

Apesar dessa facilidade de encontrar coisas prontas, provavelmente você não vai querer baixar da internet, mesmo que do *registry* do próprio Docker (sério), e subir no seu ambiente de produção algo que você não tem certeza de como funciona, o que é, etc.

Para resolver esse problema o Docker disponibiliza algumas funcionalidades, como o comando "docker image inspect", que já vimos antes, quando falávamos de volumes, lembra? Naquele momento usamos a *flag* "-f" e especificamos um campo de pesquisa, pois o intuito era mostrar somente a parte que estava sendo discutida naquele capítulo. Mas o "docker image inspect" vai muito além disso; sem passar o "-f" ele vai retornar todas as informações contidas naquela imagem, desde a imagem base que foi utilizada até pontos de montagem, configurações, enfim, muita coisa. Teste aí:

```
root@linuxtips:~# docker image inspect debian
[
   {
     "Id":
"sha256:f50f9524513f5356d952965dc97c7e831b02bb6ea0619da9bfc1997e4b9781
b7",
     "RepoTags": [
        "debian:8",
        "debian:latest"
     ],
     "RepoDigests": [],
     "Parent": "",
     "Comment": "",
     "Created": "2016-03-01T18:51:14.143360029Z",
     "Container":              "557177343b434b6797c19805d49c37728a4445d
2610a6647c27055fbe4ec3451",
     "ContainerConfig": {
        "Hostname": "e5c68db50333",
        "Domainname": "",
        "User": "",
        "AttachStdin": false,
        "AttachStdout": false,
```

```
        “AttachStderr”: false,
        “Tty”: false,
        “OpenStdin”: false,
        “StdinOnce”: false,
        “Env”: null,
        “Cmd”: [
          “/bin/sh”,
          “-c”,
          “#(nop) CMD [\”/bin/bash\”]”
        ],
        “Image”:               “d8bd0657b25f17eef81a3d52b53da5bda4de0cf
5cca3dcafec277634ae4b38fb”,
        “Volumes”: null,
        “WorkingDir”: “”,
        “Entrypoint”: null,
        “OnBuild”: null,
        “Labels”: {}
    },
    “DockerVersion”: “1.9.1”,
    “Author”: “”,
    “Config”: {
        “Hostname”: “e5c68db50333”,
        “Domainname”: “”,
        “User”: “”,
        “AttachStdin”: false,
        “AttachStdout”: false,
        “AttachStderr”: false,
        “Tty”: false,
        “OpenStdin”: false,
        “StdinOnce”: false,
        “Env”: null,
        “Cmd”: [
          “/bin/bash”
        ],
        “Image”:               “d8bd0657b25f17eef81a3d52b53da5bda4de0cf
5cca3dcafec277634ae4b38fb”,
        “Volumes”: null,
        “WorkingDir”: “”,
        “Entrypoint”: null,
        “OnBuild”: null,
        “Labels”: {}
    },
```

```
    "Architecture": "amd64",
    "Os": "linux",
    "Size": 125110803,
    "VirtualSize": 125110803,
    "GraphDriver": {
        "Name": "aufs",
        "Data": null
    }
  }
]
root@linuxtips:~#
```

Às vezes será disponibilizado junto com a imagem o seu respectivo *dockerfile* e aí fica bem mais fácil: basta ler esse arquivo para saber exatamente como ela foi criada. :)

Um comando bastante interessante, que nos faz entender como uma imagem é dividida em camadas (e, principalmente, o que foi feito em cada camada), é o "docker history".

```
root@linuxtips:~#
docker      history
linuxtips/apache:1.
0
IMAGE               CREATED   CREATED BY  SIZE COMMENT
4862def18dfd        36        /bin/sh   -c EXPOSE 80/tcp          0 B
                    minutes   #(nop)
                    ago
06210ac863da        36        /bin/sh   -c VOLUME                 0 B
                    minutes   #(nop)       [/var/www/html/]
                    ago
fed9b6bc7ad9        36        /bin/sh   -c LABEL                  0 B
                    minutes   #(nop)       description=Webserver
                    ago
68f6e8de3df3        36        /bin/sh   -c ENV                    0 B
                    minutes   #(nop)       APACHE_LOG_DIR=/var/lo
                    ago                    g
1a129e753d1e        36        /bin/sh   -c ENV                    0 B
                    minutes   #(nop)       APACHE_RUN_GROUP=www-
                    ago                    da
```

```
f0f9d7be7c70        36 minutes ago      /bin/sh -c #(nop)   ENV APACHE_RUN_USER=www-dat      0 B
3dafea4a403a        36 minutes ago      /bin/sh -c #(nop)   ENV APACHE_PID_FILE=/var/ru       0 B
f31eb176ecc8        36 minutes ago      /bin/sh -c #(nop)   ENV APACHE_LOCK_DIR=/var/lo       0 B
0bbefd91da05        36 minutesago       /bin/sh -c apt-get  update && apt-get install      68.29 MB
f50f9524513f        12 days ago         /bin/sh -c #(nop)   CMD ["/bin/bash"]               0 B
<missing>           12 days ago         /bin/sh -c #(nop)   ADD file:b5393172fb513d125.1 MB
root@linuxtips:~#
```

Perceba que as primeiras linhas da saída do comando anterior são referentes às informações que pedimos para adicionar à imagem no *dockerfile*. As demais camadas são originais da imagem que pegamos do Docker Hub através da instrução "FROM".

Existe também um site chamado "ImageLayers": ele faz exatamente a mesma coisa que o "docker history", mas você não precisa baixar a imagem – e, bom, é *web*. O ImageLayers pode ser acessado em: <https://imagelayers.io/>.

O Docker Hub, como já falamos, tem muitos componentes, dentre eles o responsável pelo repositório de imagens: o *registry*.

É possível utilizar um *registry* local em vez de um na nuvem, como o Docker Hub ou outros *registries* que são facilmente encontrados na internet. Falaremos disso com detalhes mais à frente. :P

Para que você possa utilizar o Docker Hub para gerenciar as suas imagens, é necessário criar uma conta.

9.2. Vamos criar uma conta?

Para que consiga realizar a criação da sua conta, é necessário acessar a URL <https://hub.docker.com>. Antes era possível através do comando "docker login". Hoje, o comando "docker login" somente é utilizado para autenticar no Docker Hub após a criação da conta.

A partir da confirmação do seu e-mail, já é possível se autenticar e começar a fazer uso do Docker Hub.

```
root@linuxtips:~# docker login
Login with your Docker ID to push and pull images from Docker Hub. If
you don't have a Docker ID, head over to https://hub.docker.com to
create one.
Username: linuxtips01
Password:
Login Succeeded
root@linuxtips:~#
```

Você consegue criar repositórios públicos à vontade, porém na conta *free* você somente tem direito a um repositório privado. Caso precise de mais do que um repositório privado, é necessário o *upgrade* da sua conta e o pagamento de uma mensalidade. :)

Caso você queira especificar outro *registry* em vez do Docker Hub, basta passar o endereço como parâmetro, como segue:

```
# docker login registry.seilaqual.com
```

9.3. Agora vamos compartilhar essas imagens na *interwebs*!

Uma vez que já criamos a nossa conta no Docker Hub, podemos começar a utilizá-la!

Como exemplo, vamos utilizar a imagem que montamos com o *dockerfile* no capítulo anterior chamada "linuxtips/apache". Quando

realizarmos o *upload* dessa imagem para o Docker Hub, o repositório terá o mesmo nome da imagem, ou seja, "linuxtips/apache".

Uma coisa muito importante! A sua imagem deverá ter o seguinte padrão, para que você consiga fazer o *upload* para o Docker Hub:

```
seuusuario/nomedaimagem:versão
```

Assim, sabemos que "linuxtips/apache:1.0" significa:

⇨ **linuxtips** – Usuário do Docker Hub.

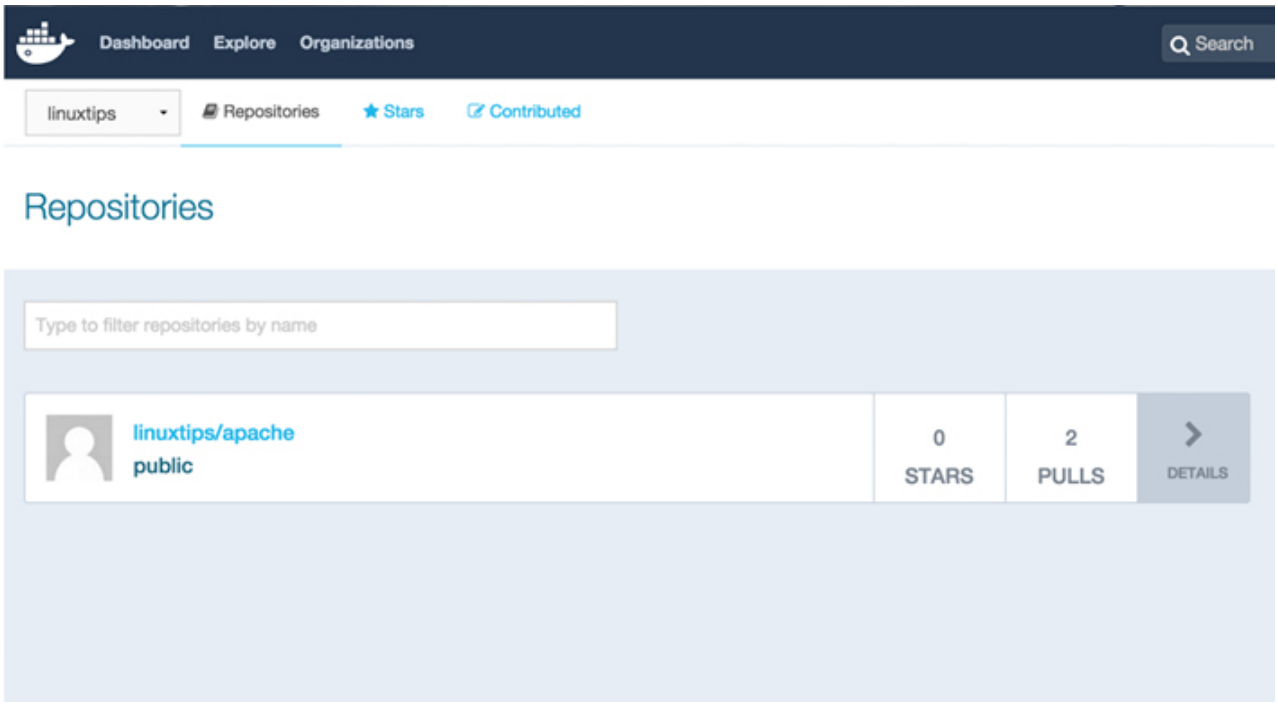
⇨ **apache** – Nome da imagem.

⇨ **1.0** – Versão.

Agora vamos utilizar o comando "docker push", responsável por fazer o *upload* da imagem da sua máquina local para o *registry* do Docker Hub, como mostrado no exemplo a seguir:

```
root@linuxtips:~# docker push linuxtips/apache:1.0
The push refers to a repository [docker.io/linuxtips/apache]
b3a691489ee1: Pushed
5f70bf18a086: Layer already exists
917c0fc99b35: Pushed
1.0: digest:
sha256:c8626093b19a686fd260dbe0c12db79a97ddfb6a6d8e4c4f44634f66991d93d
0 size: 6861
root@linuxtips:~#
```

Acessando a URL <https://hub.docker.com/> você conseguirá visualizar o repositório que acabou de criar, conforme a imagem a seguir:



Por padrão, ele cria o repositório como público. ;)

Caso você queira visualizar o seu novo repositório pela linha de comando, basta utilizar o comando "docker search" seguido de seu usuário do Docker Hub:

```
root@linuxtips:~# docker search <seu_usuario>
NAME                    DESCRIPTION     STARS   OFFICIAL    AUTOMATED
linuxtips/apache                        0
```

Já que possuímos a imagem no *registry* do Docker Hub, vamos testá-la fazendo o *pull* da imagem e em seguida vamos executando-a para saber se realmente tudo isso funciona de forma simples e fácil assim. :)

Primeiro vamos parar os *containers* que utilizam a imagem "linuxtips/apache":

```
# docker container ls | grep seu_usuario/sua_imagem
# docker container stop CONTAINERID
```

Não é possível remover uma imagem se algum *container* estiver em execução utilizando-a como imagem base. Por isso é necessário parar os *containers* conforme fizemos antes.

Para que você possa remover uma imagem é necessário utilizar o comando "docker image rm", responsável por remover imagens do disco local.

É importante mencionar que, caso possua *containers* parados que utilizam essa imagem como base, é necessário forçar a remoção da imagem utilizando o parâmetro "-f":

```
# docker image rm -f linuxtips/apache:1.0
Untagged: linuxtips/apache:1.0
```

Pronto! Removemos a imagem!

Agora vamos realizar o *pull* da imagem diretamente do *registry* do Docker Hub para que tenhamos a imagem novamente em nosso disco local e assim possamos subir um *container* utilizando-a.

```
root@linuxtips:~# docker pull linuxtips/apache:1.0
1.0: Pulling from linuxtips/apache
fdd5d7827f33: Already exists
a3ed95caeb02: Already exists
11b590220174: Already exists
Digest:
sha256:c8626093b19a686fd260dbe0c12db79a97ddfb6a6d8e4c4f44634f66991d93d
0
Status: Downloaded newer image for linuxtips/apache:1.0
root@linuxtips:~#
```

Podemos novamente visualizá-la através do comando "docker image ls".

```
root@linuxtips:~# docker image ls
REPOSITORY          TAG IMAGE ID       CREATED              SIZE
linuxtips/apache    1.0 4862def18dfd   About an hour ago    193.4 MB
root@linuxtips:~#
```

Para criar o *container* utilizando nossa imagem:

```
root@linuxtips:~# docker container run -d linuxtips/apache:1.0
```

Simples como voar, não?

9.4. Não confio na internet; posso criar o meu *registry* local?

Como muitas empresas não gostam de manter seus dados na nuvem em serviços de terceiros, existe a possibilidade de você configurar um *registry* local. Assim, você não precisa utilizar o *registry* do Docker Hub, por exemplo. Isso permite a você compartilhar suas imagens com outras pessoas de sua empresa, funcionando como repositório de imagens Docker. Sensacional!

A URL do projeto fica em <https://github.com/docker/distribution>. O Docker Distribution é um *registry* que serve para guardar e compartilhar suas imagens. Ele substitui o Docker Registry, que se tornou obsoleto.

Para que possamos ter o Docker Distribution de forma simples e totalmente funcional, guardando e distribuindo nossas imagens Docker localmente, basta rodá-lo como um *container*! :D

```
root@linuxtips:~# docker container run -d -p 5000:5000 --
restart=always --name registry registry:2
Unable to find image 'registry:2' locally
2: Pulling from library/registry
fdd5d7827f33: Already exists
a3ed95caeb02: Pull complete
a79b4a92697e: Pull complete
6cbb75c7cc30: Pull complete
4831699594bc: Pull complete
Digest:
sha256:20f5d95004b71fe14dbe7468eff33f18ee7fa52502423c5d107d4fb0abb05c1
d
Status: Downloaded newer image for registry:2
```

```
4f8efc8a71531656dc74e99dea74da203645c0f342b0706bc74200ae0a50cb20
root@linuxtips:~#
```

Com o comando anterior, criamos um *container* chamado "registry" que utiliza a imagem "registry:2" como base e usamos a opção "--restart=always". Caso ocorra qualquer problema com o *container* ou com o Docker, ele será iniciado automaticamente. Passamos também que a porta de comunicação com o *container* será a 5000, que também utilizará a porta 5000 do *host* com o mesmo propósito. Vamos ver sobre o parâmetro "-p" em breve, no capítulo relacionado a redes. ;)

Você pode verificar o *container* do *registry* em execução, bem como sua imagem:

```
# docker container ls
# docker image ls
```

Muito bom, nosso *registry* já está em execução! Agora vamos testá-lo tentando realizar um *push* da nossa imagem para ele.

Primeiramente, teremos que adicionar uma nova *tag* em nossa imagem mencionando o endereço do novo *registry* em vez do nome do usuário que utilizávamos quando queríamos fazer o *push* para o Docker Hub. Para isso, vamos utilizar novamente o comando "docker tag":

```
# docker tag IMAGEMID localhost:5000/apache
# docker image ls
```

Agora basta fazer o *push* para o nosso *registry* local da seguinte forma:

```
root@linuxtips:~# docker push localhost:5000/apache

The push refers to a repository [localhost:5000/apache]
b3a691489ee1: Pushed
5f70bf18a086: Pushed
917c0fc99b35: Pushed
```

```
latest: digest:
sha256:0e69b8d5cea67fcfedb5d7128a9fd77270461aa5852e6fe9b465565ec8e4e12
f size: 925
root@linuxtips:~#
```

Pronto! Agora nós possuímos um *registry* local!

Fizemos um *registry* totalmente funcional, porém simples. Caso queira utilizar recursos como controle de usuários, certificados, outras opções de *storage*, etc., visite a página do projeto no GitHub: <https://github.com/docker/distribution>.

10. Gerenciando a rede dos *containers*

Quando o Docker é executado, ele cria uma *bridge* virtual chamada "docker0", para que possa gerenciar a comunicação interna entre o *container* e o *host* e também entre os *containers*.

Vamos conhecer alguns parâmetros do comando "Docker container run" que irão nos ajudar com a rede em que os *containers* irão se comunicar.

- ⇨ **--dns** – Indica o servidor DNS.
- ⇨ **--hostname** – Indica um *hostname*.
- ⇨ **--link** – Cria um *link* entre os *containers*, sem a necessidade de se saber o IP um do outro.
- ⇨ **--net** – Permite configurar o modo de rede que você usará com o *container*. Temos quatro opções, mas a mais conhecida e utilizada é a "--net=host", que permite que o *container* utilize a rede do *host* para se comunicar e não crie um *stack* de rede para o *container*.
- ⇨ **--expose** – Expõe a porta do *container* apenas.
- ⇨ **--publish** – Expõe a porta do *container* e do *host*.
- ⇨ **--default-gateway** – Determina a rota padrão.

⇨ **--mac-address** – Determina um MAC *address*.

Quando o *container* é iniciado, a rede passa por algumas etapas até a sua inicialização completa:

1. Cria-se um par de interfaces virtuais.
2. Cria-se uma interface com nome único, como "veth1234", e em se- guida *linka-se* com a *bridge* do Docker, a "docker0".
3. Com isso, é disponibilizada a interface "eth0" dentro do *container*, em um *network namespace* único.
4. Configura-se o MAC *address* da interface virtual do *container*.
5. Aloca-se um IP na "eth0" do *container*. Esse IP tem que pertencer ao *range* da *bridge* "docker0".

Com isso, o *container* já possui uma interface de rede e já está apto a se comunicar com outros *containers* ou com o *host*. :D

10.1. Consigo fazer com que a porta do *container* responda na porta do *host*?

Sim, isso é possível e bastante utilizado.

Vamos conhecer um pouco mais sobre isso em um exemplo utilizando aquela nossa imagem "linuxtips/apache".

Primeira coisa que temos que saber é a porta pela qual o Apache2 se comunica. Isso é fácil, né? Se estiver com as configurações padrões de porta de um *web server*, o Apache2 do *container* estará respondendo na porta 80/TCP, correto?

Agora vamos fazer com que a porta 8080 do nosso *host* responda pela porta 80 do nosso *container*, ou seja, sempre que alguém bater na porta 8080 do nosso *host*, a requisição será encaminhada para a porta 80 do *container*. Simples, né?

Para conseguir fazer esse encaminhamento, precisamos utilizar o parâmetro "-p" do comando "docker container run", conforme

faremos no exemplo a seguir:

```
root@linuxtips:~#  #  docker  container  run  -ti  -p  8080:80
linuxtips/apache:1.0 /bin/bash
root@4a0645de6d94:/# ps -ef
UID        PID PPID C STIME TTY          TIME CMD
root         1   0    1 18:18 ?        00:00:00 /bin/bash
root         6   1    0 18:18 ?        00:00:00 ps -ef
root@4a0645de6d94:/# /etc/init.d/apache2 start
[....] Starting web server: apache2AH00558: apache2: Could not
reliably determine the server's fully qualified domain name, using
172.17.0.3. Set the 'S erverName' directive globally to suppress this
message
. ok
root@4a0645de6d94:/# ps -ef
UID        PID PPID C STIME TTY          TIME CMD
root         1   0    0 18:18 ?        00:00:00 /bin/bash
root        30   1    0 18:19 ?        00:00:00 /usr/sbin/apache2 -k start
www-data    33  30    0 18:19 ?        00:00:00 /usr/sbin/apache2 -k start
www-data    34  30    0 18:19 ?        00:00:00 /usr/sbin/apache2 -k start
root       109   1    0 18:19 ?        00:00:00 ps -ef

root@4a0645de6d94:/# ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN
group default
   link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
   inet 127.0.0.1/8 scope host lo
     valid_lft forever preferred_lft forever
   inet6 ::1/128 scope host
     valid_lft forever preferred_lft forever
74:  eth0:  <BROADCAST,MULTICAST,UP,LOWER_UP>  mtu  1500  qdisc  noqueue
state UP group default
   link/ether 02:42:ac:11:00:03 brd ff:ff:ff:ff:ff:ff
   inet 172.17.0.3/16 scope global eth0
     valid_lft forever preferred_lft forever
   inet6 fe80::42:acff:fe11:3/64 scope link
     valid_lft forever preferred_lft forever
root@4a0645de6d94:/#
```

Repare que passamos o parâmetro "-p" da seguinte forma:

⇨ **-p 8080:80** – Onde "8080" é a porta do host e "80", a do *container*.

Com isso, estamos dizendo que toda requisição que chegar na porta 8080 do meu *host* deverá ser encaminhada para a porta 80 do *container*.

Já no *container*, subimos o Apache2 e verificamos o IP do *container*, correto?

Agora vamos sair do *container* com o atalho "Ctrl + p + q". :)

A partir do *host*, vamos realizar um "curl" com destino ao IP do *container* na porta 80, depois com destino à porta 8080 do *host* e em seguida analisar as saídas:

```
root@linuxtips:~# curl <IPCONTAINER>:80
```

Se tudo ocorreu bem até aqui, você verá o código da página de boas-vindas do Apache2.

O mesmo ocorre quando executamos o "curl" novamente, porém batendo no IP do *host*. Veja:

```
root@linuxtips:~# curl <IPHOST>:8080
```

Muito fácil, chega a ser lacrimejante! \o/

10.2. E como ele faz isso? Mágica?

Não, não é mágica! Na verdade, o comando apenas utiliza um módulo bastante antigo do *kernel* do Linux chamado *netfilter*, que disponibiliza a ferramenta *iptables*, que todos nós já cansamos de usar.

Vamos dar uma olhada nas regras de *iptables* referentes a esse nosso *container*. Primeiro a tabela *filter*:

```
root@linuxtips:~# iptables -L -n
```

```
Chain INPUT (policy ACCEPT)
target                                  prot opt source    destination

Chain FORWARD (policy ACCEPT)
target                                  prot opt source    destination
DOCKER-ISOLATION all -- 0.0.0.0/0       0.0.0.0/0
DOCKER                                  all -- 0.0.0.0/0   0.0.0.0/0
ACCEPT                                  all -- 0.0.0.0/0   0.0.0.0/0
ctstate RELATED,ESTABLISHED
ACCEPT                                  all -- 0.0.0.0/0   0.0.0.0/0
ACCEPT                                  all -- 0.0.0.0/0   0.0.0.0/0

Chain OUTPUT (policy ACCEPT)
target                                  prot opt source    destination

Chain DOCKER (1 references)
target                                  prot opt source    destination
ACCEPT                                  tcp -- 0.0.0.0/0   172.17.0.2
tcp dpt:5000
ACCEPT                                  tcp -- 0.0.0.0/0   172.17.0.3
tcp dpt:80

Chain DOCKER-ISOLATION (1 references)
target                                  prot opt source    destination
RETURN                                  all -- 0.0.0.0/0   0.0.0.0/0
root@linuxtips:~#
```

Agora a tabela NAT:

```
root@linuxtips:~# iptables -L -n -t nat
Chain PREROUTING (policy ACCEPT)
target             prot opt source               destination
DOCKER             all -- 0.0.0.0/0              0.0.0.0/0
ADDRTYPE match dst-type LOCAL

Chain INPUT (policy ACCEPT)
target             prot opt source               destination

Chain OUTPUT (policy ACCEPT)
```

```
target             prot opt source               destination
DOCKER             all -- 0.0.0.0/0              !127.0.0.0/8
ADDRTYPE match dst-type LOCAL

Chain POSTROUTING (policy ACCEPT)
target             prot opt source               destination
MASQUERADE         all -- 172.17.0.0/16          0.0.0.0/0
MASQUERADE         tcp -- 172.17.0.2             172.17.0.2
tcp dpt:5000
MASQUERADE         tcp -- 172.17.0.3             172.17.0.3
tcp dpt:80

Chain DOCKER (2 references)
target             prot opt source               destination
RETURN             all -- 0.0.0.0/0              0.0.0.0/0
DNAT               tcp -- 0.0.0.0/0              0.0.0.0/0
tcp dpt:5000 to:172.17.0.2:5000
DNAT               tcp -- 0.0.0.0/0              0.0.0.0/0
tcp dpt:8080 to:172.17.0.3:80
root@linuxtips:~#
```

Como podemos notar, temos regras de NAT configuradas que permitem o DNAT da porta 8080 do *host* para a 80 do *container*. Veja a seguir:

```
MASQUERADE         tcp -- 172.17.0.3             172.17.0.3
tcp dpt:80
DNAT               tcp -- 0.0.0.0/0              0.0.0.0/0
tcp dpt:8080 to:172.17.0.3:80
```

Tudo isso feito “automagicamente” pelo Docker, sem a necessidade de precisar configurar diversas regras de *iptables*. <3

11. Controlando o *daemon* do Docker

Antes de tudo, vamos tentar entender o que é um *daemon*. Sabemos que, em sistemas operacionais *multitask*, isto é, em um sistema operacional capaz de executar mais de uma tarefa por vez (*not really*), um *daemon* é um software que roda de forma independente em *background*. Ele executa certas ações predefinidas em resposta a certos eventos. Pois bem, o *daemon* do Docker é exatamente isso: uma espécie de processo-pai que controla tudo, *containers*, imagens, etc., etc., etc.

Até o Docker 1.7 as configurações referentes especificamente ao *daemon* se confundiam bastante com configurações globais – isso porque quando você digitava lá o "docker -help" um monte de coisas retornava, e você não sabia o que era o quê. A partir da versão 1.8 tivemos o "docker daemon", e agora, mais recentemente, acreditamos que na versão 18.03 do Docker, ele foi substituído pelo "dockerd", que resolve de vez esse problema e trata especificamente de configurações referentes, obviamente, ao *daemon* do Docker.

11.1. O Docker sempre utiliza 172.16.X.X ou posso configurar outro intervalo de IP?

Sim, você pode configurar outro *range* para serem utilizados pela *bridge* "docker0" e também pelas interfaces dos *containers*.

Para que você consiga configurar um *range* diferente para utilização do Docker é necessário iniciá-lo com o parâmetro "--bip".

```
# dockerd --bip 192.168.0.1/24
```

Assim, você estará informando ao Docker que deseja utilizar o IP "192.168.0.1" para sua *bridge* "docker0" e, consequentemente, para a *subnet* dos *containers*.

Você também poderá utilizar o parâmetro "--fixed-cidr" para restringir o *range* que o Docker irá utilizar para a *bridge* "docker0" e para a *subnet* dos *containers*.

```
# dockerd --fixed-cidr 192.168.0.0/24
```

11.2. Opções de *sockets*

Sockets são *end-points* com as quais duas ou mais aplicações ou processos se comunicam em um ambiente, geralmente um "IP:porta" ou um arquivo, como no caso do *Unix Domain Sockets*.

Atualmente o Docker consegue trabalhar com três tipos de *sockets*, Unix, TCP e FD, e por *default* ele usa *unix sockets*. Você deve ter notado que, ao *startar* seu Docker, foi criado um arquivo em "/var/run/docker.sock". Para fazer alterações nele você vai precisar ou de permissão de *root* ou de que o usuário que esteja executando as ações faça parte do grupo "docker", como dissemos no começo deste livro, lembra?

Por mais prático que isso seja, existem algumas limitações, como, por exemplo, o *daemon* só poder ser acessado localmente. Para resolver isso usamos geralmente o TCP. Nesse modelo nós definimos um IP, que pode ser tanto "qualquer um" (0.0.0.0 e uma porta) como um IP específico e uma porta.

Nos sistemas baseados em *systemd* você ainda pode se beneficiar do *systemd socket activation*, uma tecnologia que visa economia de recursos. Consiste basicamente em ativar um *socket* somente enquanto uma conexão nova chega e desativar quando não está sendo mais usado.

Além disso tudo, dependendo do seu ambiente, você também pode fazer o Docker escutar em diferentes tipos de *sockets*, o que é feito através do parâmetro "-H" do comando "dockerd".

Exemplos:

11.2.1. *Unix Domain Socket*

```
root@linuxtips:~# dockerd -H unix:///var/run/docker.sock
INFO[0000] [graphdriver] using prior storage driver "aufs"
INFO[0000] Graph migration to content-addressability took 0.00 seconds
INFO[0000] Firewalld running: false
INFO[0000] Default bridge (docker0) is assigned with an IP address
172.17.0.0/16. Daemon option --bip can be used to set a preferred IP
address
WARN[0000] Your kernel does not support swap memory limit.
INFO[0000] Loading containers: start.
.........................
INFO[0000] Loading containers: done.
INFO[0000] Daemon has completed initialization
INFO[0000] Docker daemon commit=c3959b1 execdriver=native-0.2
graphdriver=aufs version=1.10.2
INFO[0000] API listen on /var/run/docker.sock
```

11.2.2. TCP

```
root@linuxtips:~# dockerd -H tcp://0.0.0.0:2375
WARN[0000] /!\ DON'T BIND ON ANY IP ADDRESS WITHOUT setting -tlsverify
IF YOU DON'T KNOW WHAT YOU'RE DOING /!\
INFO[0000] [graphdriver] using prior storage driver "aufs"
INFO[0000] Graph migration to content-addressability took 0.01 seconds
INFO[0000] Firewalld running: false
INFO[0000] Default bridge (docker0) is assigned with an IP address
172.17.0.0/16. Daemon option --bip can be used to set a preferred IP
address
```

```
WARN[0000] Your kernel does not support swap memory limit.
INFO[0000] Loading containers: start.
.....................
INFO[0000] Loading containers: done.
INFO[0000] Daemon has completed initialization
INFO[0000] Docker daemon commit=c3959b1 execdriver=native-0.2
graphdriver=aufs version=1.10.2
INFO[0000] API listen on [::]:2375
```

11.3. Opções de *storage*

Sendo o cara que controla tudo, naturalmente é possível passar opções que mudam a forma como o Docker se comporta ao trabalhar com *storages*. Como falamos anteriormente, o Docker suporta alguns *storage drivers*, todos baseados no esquema de *layers*.

Essas opções são passadas para o *daemon* pelo parâmetro "--storage-opt", com o qual itens relacionados ao *Device Mapper* recebem o prefixo "dm" e "zfs" para (adivinha?) o ZFS. A seguir vamos demonstrar algumas opções mais comuns:

- ⇨ **dm.thinpooldev** – Com esta opção você consegue especificar o *device* que será usado pelo *Device Mapper* para desenvolver o *thin-pool* que ele usa para criar os *snapshots* usados por *containers* e imagens.

Exemplo:

```
root@linuxtips:~# dockerd --storage-opt
dm.thinpooldev=/dev/mapper/thin-pool
INFO[0000] [graphdriver] using prior storage driver "aufs" INFO[0000]
Graph migration to content-addressability took
0.00 seconds
INFO[0000] Firewalld running: false
INFO[0000] Default bridge (docker0) is assigned with an IP address
172.17.0.0/16. Daemon option --bip can be used to set a preferred IP
address
WARN[0000] Your kernel does not support swap memory limit. INFO[0000]
Loading containers: start.
.............................
```

```
INFO[0000] Loading containers: done. INFO[0000] Daemon has completed
initialization INFO[0000] Docker daemon
commit=c3959b1 execdriver=native-0.2 graphdriver=aufs version=1.10.2

INFO[0000] API listen on /var/run/docker.sock
```

⇨ **dm.basesize** – Este parâmetro define o tamanho máximo do *container*. O chato disso é que você precisa deletar tudo dentro de "/var/lib/docker" (o que implica em matar todos os *containers* e imagens) e *restartar* o serviço do Docker.

```
root@linuxtips:~# dockerd --storage-opt dm.basesize=10G
INFO[0000] [graphdriver] using prior storage driver "aufs"
INFO[0000] Graph migration to content-addressability took 0.00 seconds
INFO[0000] Firewalld running: false
INFO[0000] Default bridge (docker0) is assigned with an IP address
172.17.0.0/16. Daemon option --bip can be used to set a preferred IP
address
WARN[0000] Your kernel does not support swap memory limit.
INFO[0000] Loading containers: start.
.........................
INFO[0000] Loading containers: done.
INFO[0000] Daemon has completed initialization
INFO[0000] Docker daemon commit=c3959b1 execdriver=native-0.2
graphdriver=aufs version=1.10.2
INFO[0000] API listen on /var/run/docker.sock
```

⇨ **dm.fs** – Especifica o *filesystem* do *container*. As opções suportadas são: **EXT4** e **XFS**.

11.4. Opções de rede

Também é possível controlar como o *daemon* se comportará em relação à rede:

⇨ **--default-gateway** – Autoexplicativo, né? Todos os *containers* receberão esse IP como *gateway*.

⇨ **--dns** – Também sem segredo: é o DNS que será usado para consultas.

- ⇨ **--dns-search** – Especifica o domínio a ser procurado, assim você consegue pesquisar máquinas sem usar o fqdn.
- ⇨ **--ip-forward** – Esta opção habilita o roteamento entre *containers*. Por padrão, ela já vem *setada* como *true*.

11.5. Opções diversas

- ⇨ **--default-ulimit** – Passando isso para o *daemon*, todos os *containers* serão iniciados com esse valor para o "ulimit". Esta opção é sobrescrita pelo parâmetro "--ulimit" do comando "docker container run", que geralmente vai dar uma visão mais específica.
- ⇨ **--icc** – "icc" vem de *inter container comunication*. Por padrão, ele vem marcado como *true*; caso você não queira esse tipo de comunicação, você pode marcar no *daemon* como *false*.
- ⇨ **--log-level** – É possível alterar também a forma como o Docker trabalha com *log*; em algumas situações (geralmente *troubleshoot*) você pode precisar de um *log* mais "verboso", por exemplo.

12. Docker Machine

12.1. Ouvi dizer que minha vida ficaria melhor com o Docker Machine!

Certamente!

Com o Docker Machine você consegue, com apenas um comando, iniciar o seu projeto com Docker!

Antes do Docker Machine, caso quiséssemos montar um Docker Host, era necessário fazer a instalação do sistema operacional, instalar e configurar o Docker e outras ferramentas que se fazem necessárias.

Perderíamos um tempo valoroso com esses passos, sendo que já poderíamos trabalhar efetivamente com o Docker e seus *containers*.

Porém, tudo mudou com o Docker Machine! Com ele você consegue criar o seu Docker Host com apenas um comando. O Docker Machine consegue trabalhar com os principais *hypervisors* de VMs, como o VMware, Hyper-V e Oracle VirtualBox, e também com os principais provedores de infraestrutura, como AWS, Google Compute Engine, DigitalOcean, Rackspace, Azure, etc.

Para que você possa ter acesso a todos os *drivers* que o Docker Machine suporta, acesse: <https://docs.docker.com/machine/drivers/>.

Quando você utiliza o Docker Machine para instalar um Docker Host na AWS, por exemplo, ele disponibilizará uma máquina com Linux e com o Docker e suas dependências já instaladas.

12.1.1. Vamos instalar?

A instalação do Docker Machine é bastante simples, por isso, vamos parar de conversa e brincar! Vale lembrar que é possível instalar o Docker Machine no Linux, MacOS ou Windows.

Para fazer a instalação do Docker Machine no Linux, faça:

```
# curl -L https://github.com/docker/machine/releases/download/v0.12.0/docker-machine-`uname -s`-`uname -m` >/tmp/docker-machine
# chmod +x /tmp/docker-machine # sudo cp /tmp/docker-machine /usr/local/bin/docker-machine
```

Para seguir com a instalação no MacOS:

```
$ curl -L https://github.com/docker/machine/releases/download/v0.15.0/docker-machine-`uname -s`-`uname -m` >/usr/local/bin/docker-machine
$ chmod +x /usr/local/bin/docker-machine
```

Para seguir com a instalação no Windows caso esteja usando o Git *bash*:

```
$ if [[ ! -d "$HOME/bin" ]]; then mkdir -p "$HOME/bin"; fi
$ curl -L https://github.com/docker/machine/releases/download/v0.15.0/docker-machine-Windows-x86_64.exe > "$HOME/bin/docker-machine.exe"
$ chmod +x "$HOME/bin/docker-machine.exe"
```

Para verificar se ele foi instalado e qual a sua versão, faça:

```
root@linuxtips:~# docker-machine version
docker-machine version 0.15.0, build b48dc28
root@linuxtips:~#
```

Pronto. Como tudo que é feito pelo Docker, é simples de instalar e fácil de operar. :)

12.1.2. Vamos iniciar nosso primeiro projeto?

Agora que já temos o Docker Machine instalado em nossa máquina, já conseguiremos fazer a instalação do Docker Host de forma bastante simples – lembrando que, mesmo que tivéssemos feito a instalação do Docker Machine no Windows, conseguiríamos tranquilamente comandar a instalação do Docker Hosts na AWS em máquinas Linux. Tenha em mente que a máquina na qual você instalou o Docker Machine é o maestro que determina a criação de novos Docker Hosts, seja em VMs ou em alguma nuvem como a AWS.

Em nosso primeiro projeto, vamos fazer com que o Docker Machine instale o Docker Host utilizando o VirtualBox.

Como utilizaremos o VirtualBox, é evidente que precisamos ter instalado o VirtualBox em nossa máquina para que tudo funcione. ;)

Portanto:

```
root@linuxtips:~# apt-get install virtualbox
```

Para fazer a instalação de um novo Docker Host, utilizamos o comando "docker-machine create". Para escolher onde criaremos o Docker Host, utilizamos o parâmetro "--driver", conforme segue:

```
root@linuxtips:~# docker-machine create --driver virtualbox linuxtips
Running pre-create checks...
(linuxtips) Default Boot2Docker ISO is out-of-date, downloading the
latest release...
(linuxtips) Latest release for github.com/boot2docker/boot2docker is
v17.05.0-ce
(linuxtips) Downloading
/Users/linuxtips/.docker/machine/cache/boot2docker.iso from
https://github.com/boot2docker/boot2docker/releases/download/v17.05.0-
```

```
ce/boot2docker.iso...
(linuxtips)
0%....10%....20%....30%....40%....50%....60%....70%....80%....90%....1
00%
Creating machine...
(linuxtips)                                                    Copying
/Users/linuxtips/.docker/machine/cache/boot2docker.iso              to
/Users/linuxtips/.docker/machine/machines/linuxtips/boot2docker.iso...
(linuxtips) Creating VirtualBox VM...
(linuxtips) Creating SSH key...
(linuxtips) Starting the VM...
(linuxtips) Check network to re-create if needed...
(linuxtips) Waiting for an IP...
Waiting for machine to be running, this may take a few minutes...
Detecting operating system of created instance...
Waiting for SSH to be available...
Detecting the provisioner...
Provisioning with boot2docker...
Copying certs to the local machine directory...
Copying certs to the remote machine...
Setting Docker configuration on the remote daemon...
Checking connection to Docker...
Docker is up and running!
To see how to connect your Docker Client to the Docker Engine running
on this virtual machine, run: docker-machine env linuxtips
root@linuxtips:~#
```

Onde:

⇨ **docker-machine create** – Cria um novo Docker Host.

⇨ **--driver virtualbox** – Irá criá-lo utilizando o VirtualBox.

⇨ **linuxtips** – Nome da VM que será criada.

Para visualizar o *host* que acabou de criar, basta digitar o seguinte comando:

```
root@linuxtips:~# docker-machine ls
NAME    ACTIVE DRIVER    STATE URL SWARM DOCKER ERRORS
linuxtips - virtualbox Running tcp://192.168.99.100:2376 v18.06.0-
```

Como podemos notar, o nosso *host* está sendo executado perfeitamente! Repare que temos uma coluna chamada URL,

correto? Nela temos a URL para que possamos nos comunicar com o nosso novo *host*.

Outra forma de visualizar informações sobre o *host*, mais especificamente sobre as variáveis de ambiente dele, é digitar:

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

Serão mostradas todas as variáveis de ambiente do *host*, como URL, certificado e nome.

Para que você possa acessar o ambiente desse *host* que acabamos de criar, faça:

```
root@linuxtips:~# eval "$(docker-machine env linuxtips)"
```

O comando "eval" serve para definir variáveis de ambiente através da saída de um comando, ou seja, as variáveis que visualizamos na saída do "docker-machine env linuxtips".

Agora que já estamos no ambiente do *host* que criamos, vamos visualizar os *containers* em execução:

```
root@linuxtips:~# docker container ls
```

Claro que ainda não temos nenhum *container* em execução; vamos iniciar o nosso primeiro agora:

```
root@linuxtips:~# docker container run busybox echo "LINUXTIPS,
VAIIII"
Unable to find image 'busybox:latest' locally
latest: Pulling from library/busybox
385e281300cc: Pull complete
```

```
a3ed95caeb02: Pull complete
Digest:
sha256:4a887a2326ec9e0fa90cce7b4764b0e627b5d6afcb81a3f73c85dc29cea0004
8
Status: Downloaded newer image for busybox:latest
LINUXTIPS, VAIIII
root@linuxtips:~#
```

Como podemos observar, o *container* foi executado e imprimiu a mensagem "**LINUXTIPS, VAIIII**", conforme solicitamos.

Lembre-se de que o *container* foi executado em nosso Docker Host, que criamos através do Docker Machine.

Para verificar o IP do *host* que criamos, faça:

```
root@linuxtips:~# docker-machine ip linuxtips
192.168.99.100
root@linuxtips:~#
```

Para que possamos acessar o nosso *host*, utilizamos o parâmetro "ssh" passando o nome do *host* que queremos acessar:

```
root@linuxtips:~# docker-machine ssh linuxtips
```

Para saber mais detalhes sobre o *host*, podemos utilizar o parâmetro "inspect":

```
root@linuxtips:~# docker-machine inspect linuxtips
{
   "ConfigVersion": 3,
   "Driver": {
     "IPAddress": "192.168.99.100",
     "MachineName": "linuxtips",
     "SSHUser": "docker",
     "SSHPort": 57249,
     "SSHKeyPath":
"/Users/jeferson/.docker/machine/machines/linuxtips/id_rsa",
     "StorePath": "/Users/jeferson/.docker/machine",
     "SwarmMaster": false,
     "SwarmHost": "tcp://0.0.0.0:3376",
     "SwarmDiscovery": "",
```

```
  "VBoxManager": {},
  "HostInterfaces": {},
  "CPU": 1,
  "Memory": 1024,
  "DiskSize": 20000,
  "NatNicType": "82540EM",
  "Boot2DockerURL": "",
  "Boot2DockerImportVM": "",
  "HostDNSResolver": false,
  "HostOnlyCIDR": "192.168.99.1/24",
  "HostOnlyNicType": "82540EM",
  "HostOnlyPromiscMode": "deny",
  "UIType": "headless",
  "HostOnlyNoDHCP": false,
  "NoShare": false,
  "DNSProxy": true,
  "NoVTXCheck": false,
  "ShareFolder": ""
},
"DriverName": "virtualbox",
"HostOptions": {
  "Driver": "",
  "Memory": 0,
  "Disk": 0,
  "EngineOptions": {
     "ArbitraryFlags": [],
     "Dns": null,
     "GraphDir": "",
     "Env": [],
     "Ipv6": false,
     "InsecureRegistry": [],
     "Labels": [],
     "LogLevel": "",
     "StorageDriver": "",
     "SelinuxEnabled": false,
     "TlsVerify": true,
     "RegistryMirror": [],
     "InstallURL": "https://get.docker.com"
  },
  "SwarmOptions": {
     "IsSwarm": false,
     "Address": "",
```

```
        "Discovery": "",
        "Agent": false,
        "Master": false,
        "Host": "tcp://0.0.0.0:3376",
        "Image": "swarm:latest",
        "Strategy": "spread",
        "Heartbeat": 0,
        "Overcommit": 0,
        "ArbitraryFlags": [],
        "ArbitraryJoinFlags": [],
        "Env": null,
        "IsExperimental": false
    },
    "AuthOptions": {
        "CertDir": "/Users/jeferson/.docker/machine/certs",
        "CaCertPath": "/Users/jeferson/.docker/machine/certs/ca.pem",
        "CaPrivateKeyPath":  "/Users/jeferson/.docker/machine/certs/ca-
key.pem",
        "CaCertRemotePath": "",
        "ServerCertPath":
"/Users/jeferson/.docker/machine/machines/linuxtips/server.pem",
        "ServerKeyPath":
"/Users/jeferson/.docker/machine/machines/linuxtips/server-key.pem",
        "ClientKeyPath":
"/Users/jeferson/.docker/machine/certs/key.pem",
        "ServerCertRemotePath": "",
        "ServerKeyRemotePath": "",
        "ClientCertPath":
"/Users/jeferson/.docker/machine/certs/cert.pem",
        "ServerCertSANs": [],
        "StorePath":
"/Users/jeferson/.docker/machine/machines/linuxtips"
    }
  },
  "Name": "linuxtips"
}
```


Para parar o *host* que criamos:

```
root@linuxtips:~# docker-machine stop linuxtips
```

Para que você consiga visualizar o status do seu *host* Docker, digite:

```
root@linuxtips:~# docker-machine ls
```

Para iniciá-lo novamente:

```
root@linuxtips:~# docker-machine start linuxtips
```

Para removê-lo definitivamente:

```
root@linuxtips:~# docker-machine rm linuxtips
Successfully removed linuxtips
```

13. Docker Swarm

Bom, agora temos uma ferramenta muito interessante e que nos permite construir *clusters* de *containers* de forma nativa e com extrema facilidade, como já é de costume com os produtos criados pelo time do Docker. ;)

Com o Docker Swarm você consegue construir *clusters* de *containers* com características importantes como balanceador de cargas e *failover*.

Para criar um *cluster* com o Docker Swarm, basta indicar quais os *hosts* que ele irá supervisionar e o restante é com ele.

Por exemplo, quando você for criar um novo *container*, ele irá criá-lo no *host* que possuir a menor carga, ou seja, cuidará do balanceamento de carga e garantirá sempre que o *container* será criado no melhor *host* disponível no momento.

A estrutura de *cluster* do Docker Swarm é bastante simples e se resume a um *manager* e diversos *workers*. O *manager* é o responsável por orquestrar os *containers* e distribuí-los entre os *hosts workers*. Os *workers* são os que carregam o piano, que hospedam os *containers*.

13.1. Criando o nosso *cluster*!

Uma coisa importante que começou após a versão 1.12 foi a inclusão do Docker Swarm dentro do Docker, ou seja, hoje quando você realiza a instalação do Docker, automaticamente você está instalando o Docker Swarm, que nada mais é do que uma forma de orquestrar seus *containers* através da criação de um *cluster* com alta disponibilidade, balanceamento de carga e comunicação criptografada, tudo isso nativo, sem qualquer esforço ou dificuldade.

Para o nosso cenário, vamos utilizar três máquinas Ubuntu. A ideia é fazer com que tenhamos dois *managers* e 1 *worker*.

Precisamos ter sempre mais do que 1 *node* representando o *manager*, pois, se ficarmos sem *manager*, nosso *cluster* estará totalmente indisponível.

Com isso temos o seguinte cenário:

⇨ **LINUXtips-01** – *Manager* ativo.

⇨ **LINUXtips-02** – *Manager*.

⇨ **LINUXtips-03** – *Worker*.

Não precisa falar que precisamos ter o Docker instalado em todas essas máquinas, certo, amiguinho? :D

Para iniciar, vamos executar o seguinte comando na "LINUXtips-01":

```
root@LINUXtips-01:~# docker swarm init
Swarm initialized: current node (2qacv429fvnret8v09fqmjm16) is now a
manager.

To add a worker to this swarm, run the following command:

docker swarm join --token \ SWMTKN-1-
100qtga34hfnf14xdbbhtv8ut6ugcvuhsx427jtzwaw1td2otj-
18wccykydxte59gch2pix \
172.31.58.90:2377

To add a manager to this swarm, run 'docker swarm join-token manager'
and follow the instructions.
```

```
root@LINUXtips-01:~#
```

Com o comando anterior, iniciamos o nosso *cluster*!

Repare no seguinte trecho da saída do último comando:

```
# docker swarm join --token \ SWMTKN-1-
100qtga34hfnf14xdbbhtv8ut6ugcvuhsx427jtzwaw1td2otj-
18wccykydxte59gch2pix \
172.31.58.90:2377
```

Essa linha nada mais é do que toda informação que você precisa para adicionar *workers* ao seu *cluster*! Como assim?

Simples: o que você precisa agora é executar exatamente esse comando na próxima máquina que você deseja incluir no *cluster* como *worker*! Simples como voar, não?

De acordo com o nosso plano, a única máquina que seria *worker* é a máquina "LINUXtips-03", correto? Então vamos acessá-la e executar exatamente a linha de comando recomendada na saída do "docker swarm init".

```
root@LINUXtips-03:~# docker swarm join --token \ SWMTKN-1-
100qtga34hfnf14xdbbhtv8ut6ugcvuhsx427jtzwaw1td2otj-
18wccykydxte59gch2pix \
172.31.58.90:2377
This node joined a swarm as a worker.
root@LINUXtips-03:~#
```

Maravilha! Mais um *node* adicionado ao *cluster*!

Para que você possa ver quais os *nodes* que existem no *cluster*, basta digitar o seguinte comando no ***manager* ativo**:

```
root@LINUXtips-01:~# docker node ls
ID  HOSTNAME          STATU AVAILABILITY MANAGER STATUS ENGINE VERSION
                      S
2qa LINUXtips-01 Ready Activ Leader      18.03.1-ce
c                     e
nmx LINUXtips-03 Ready Activ
```

```
l                    e
18.03.1-ce
root@LINUXtips-01:~#
```

Como podemos notar na saída do comando, temos dois *nodes* em nosso *cluster*, um como *manager* e outro como *worker*. A coluna "MANAGER STATUS" traz a informação de quem é o "Leader", ou seja, quem é o nosso *manager*.

Em nosso plano nós teríamos dois *managers*, correto?

Agora a pergunta é: como eu sei qual é o *token* que preciso utilizar para adicionar mais um *node* em meu *cluster*, porém dessa vez como outro *manager*?

Lembra que, quando executamos o comando para adicionar o *worker* ao *cluster*, nós tínhamos no comando um *token*? Pois bem, esse *token* é quem define se o *node* será um *worker* ou um *manager*, e naquela saída ele nos trouxe somente o *token* para adicionarmos *workers*.

Para que possamos visualizar o comando e o *token* referente aos *managers*, precisamos executar o seguinte comando no *manager* ativo:

```
root@LINUXtips-01:~# docker swarm join-token manager
To add a manager to this swarm, run the following command:
   docker swarm join \
   --token SWMTKN-1-100qtga34hfnf14xdbbhtv8ut6ugcvuhsx427jtzwaw1td2otj-
3i4jsv4i70odu1mes0ebe1l1e \
   172.31.58.90:2377
root@LINUXtips-01:~#
```

Para visualizar o comando e o *token* referente aos *workers*:

```
root@LINUXtips-01:~# docker swarm join-token worker
To add a worker to this swarm, run the following command:
   docker swarm join \
   --token SWMTKN-1-100qtga34hfnf14xdbbhtv8ut6ugcvuhsx427jtzwaw1td2otj-
```

```
18wccykydxte59gch2pixq9av \
   172.31.58.90:2377
root@LINUXtips-01:~#
```

Fácil, não?

Agora o que precisamos é executar na "LINUXtips-02" o comando para inclusão de mais um *node* como *manager*. Portanto, execute:

```
root@LINUXtips-02:~# docker swarm join \
--token    SWMTKN-1-100qtga34hfnf14xdbbhtv8ut6ugcvuhsx427jtzwaw1td2otj-
3i4jsv4i70odu1mes0ebe1l1e \
172.31.58.90:2377
This node joined a swarm as a manager.
root@LINUXtips-02:~#
```

Pronto! Agora temos o nosso *cluster* completo com dois *managers* e um *worker*!

Vamos visualizar os *nodes* que fazem parte de nosso *cluster*. Lembre-se: qualquer comando para administração do *cluster* ou criação de serviços deverá obrigatoriamente ser executado no *manager* ativo, sempre!

```
root@LINUXtips-01:~# docker node ls
ID  HOSTNAME          STATU AVAILABILITY MANAGER STATUS ENGINE VERSION
                      S
2qa LINUXtips-01 Ready Activ Leader      18.03.1-ce
c                     e
j6l LINUXtips-02 Ready Activ Reachable   18.03.1-ce
m                     e
nmx LINUXtips-03 Ready Activ             18.03.1-ce
l                     e
root@LINUXtips-01:~#
```

Perceba que o "MANAGER STATUS" da "LINUXtips-02" é "Reachable". Isso indica que ela é um *manager*, porém não é o *manager* ativo, que sempre carrega o "MANAGER STATUS" como "Leader".

Se nós quisermos saber detalhes sobre determinado *node*, podemos usar o subcomando “inspect”:

```
root@LINUXtips-01:~# docker node inspect LINUXtips-02
[
   {
     “ID”: “x3fuo6tdaqjyjl549r3lu0vbj”,
     “Version”: {
        “Index”: 27
     },
     “CreatedAt”: “2017-06-09T18:09:48.925847118Z”,
     “UpdatedAt”: “2017-06-09T18:09:49.053416781Z”,
     “Spec”: {
        “Labels”: {},
        “Role”: “worker”,
        “Availability”: “active”
     },
     “Description”: {
        “Hostname”: “LINUXtips-02”,
        “Platform”: {
           “Architecture”: “x86_64”,
           “OS”: “linux”
        },
        “Resources”: {
           “NanoCPUs”: 1000000000,
           “MemoryBytes”: 1038807040
        },
        “Engine”: {
        “EngineVersion”: “17.05.0-ce”,
        “Plugins”: [
             {
                “Type”: “Network”,
                “Name”: “bridge”
             },
             {
                “Type”: “Network”,
                “Name”: “host”
             },
             {
                “Type”: “Network”,
                “Name”: “macvlan”
             },
```

```
            {
               "Type": "Network",
               "Name": "null"
            },
            {
               "Type": "Network",
               "Name": "overlay"
            },
            {
               "Type": "Volume",
               "Name": "local"
            }
         ]
      }
   },
   "Status": {
      "State": "ready",
      "Addr": "172.31.53.23"
   }
 }
]
root@LINUXtips-01:~#
```

E se nós quisermos promover um *node worker* para *manager*, como devemos fazer? Simples como voar, confira a seguir:

```
root@LINUXtips-01:~# docker node promote LINUXtips-03
Node LINUXtips-03 promoted to a manager in the swarm.

root@LINUXtips-01:~# docker node ls
ID  HOSTNAME          STATU AVAILABILITY MANAGER STATUS ENGINE VERSION
                      S
2qa LINUXtips-01 Ready Activ Leader       18.03.1-ce
c                     e
j6l LINUXtips-02 Ready Activ Reachable    18.03.1-ce
m                     e
nmx LINUXtips-03 Ready Activ Reachable    18.03.1-ce
l                     e
root@LINUXtips-01:~#
```

Se quiser tornar um *node manager* em *worker*, faça:

```
root@LINUXtips-01:~# docker node demote LINUXtips-03
Node LINUXtips-03 demoted to a manager in the swarm.
```

Vamos conferir:

```
root@LINUXtips-01:~# docker node ls
ID  HOSTNAME          STATU AVAILABILITY MANAGER STATUS ENGINE VERSION
                      S
2qa LINUXtips-01 Ready Activ Leader      18.03.1-ce
c                     e
j6l LINUXtips-02 Ready Activ Reachable   18.03.1-ce
m                     e
nmx LINUXtips-03 Ready Activ             18.03.1-ce
l                     e
root@LINUXtips-01:~#
```

Agora, caso você queira remover um *node* do *cluster*, basta digitar o seguinte comando no *node* desejado:

```
root@LINUXtips-03:~# docker swarm leave
Node left the swarm.
root@LINUXtips-03:~#
```

E precisamos ainda executar o comando de remoção desse *node* também em nosso *manager* ativo da seguinte forma:

```
root@LINUXtips-01:~# docker node rm LINUXtips-03
LINUXtips-03
root@LINUXtips-01:~#
```

Com isso, podemos executar o "docker node ls" e constatar que o *node* foi realmente removido do *cluster*. Caso queira adicioná-lo novamente, basta repetir o processo que foi utilizado para adicioná-lo, está lembrado? :D

Para remover um *node manager* de nosso *cluster*, precisamos adicionar a *flag* "--force" ao comando "docker swarm leave", como

mostrado a seguir:

```
root@LINUXtips-02:~# docker swarm leave --force
Node left the swarm.
root@LINUXtips-02:~#
```

Agora, basta removê-lo também em nosso *node manager*:

```
root@LINUXtips-01:~# docker node rm LINUXtips-02
LINUXtips-02
root@LINUXtips-01:~#
```

13.2. O sensacional *services*!

Uma das melhores coisas que o Docker Swarm nos oferece é justamente a possibilidade de fazer o uso dos *services*.

O *services* nada mais é do que um VIP ou DNS que realizará o balanceamento de requisições entre os *containers*. Podemos estabelecer um número x de *containers* respondendo por um *service* e esses *containers* estarão espalhados pelo nosso *cluster*, entre nossos *nodes*, garantindo alta disponibilidade e balanceamento de carga, tudo isso nativamente!

O *services* é uma forma, já utilizada no Kubernetes, de você conseguir gerenciar melhor seus *containers*, focando no serviço que esses *containers* estão oferecendo e garantindo alta disponibilidade e balanceamento de carga. É uma maneira muito simples e efetiva para escalar seu ambiente, aumentando ou diminuindo a quantidade de *containers* que responderá para um determinado *service*.

Meio confuso? Sim, nós sabemos, mas vai ficar fácil. :)

Imagine que precisamos disponibilizar o serviço do Nginx para ser o novo *web server*. Antes de criar esse *service*, precisamos de algumas informações:

⇨ Nome do *service* que desejo criar – **webserver**.

⇨ Quantidade de *containers* que desejo debaixo do *service* – **5**.

⇨ Portas que iremos "bindar", entre o *service* e o *node* – **8080:80**.

⇨ Imagem dos *containers* que irei utilizar – **Nginx**.

Agora que já temos essas informações, 'bora criar o nosso primeiro *service*. :)

```
root@LINUXtips-01:~# docker service create --name webserver --replicas
5 -p 8080:80 nginx
0azz4psgfpkf0i5i3mbfdiptk
root@LINUXtips-01:~#
```

Agora já temos o nosso *service* criado. Para testá-lo, basta executar:

```
root@LINUXtips-01:~# curl QUALQUER_IP_NODES_CLUSTER:8080
```

O resultado do comando anterior lhe trará a página de boas-vindas do Nginx.

Como estamos utilizando o *services*, cada conexão cairá em um *container* diferente, fazendo assim o balanceamento de cargas "automagicamente"!

Para visualizar o *service* criado, execute:

```
root@LINUXtips-01:~# docker service ls

ID      NAME       MODE         REPLICAS  IMAGE         PORTS
0azz4p  webserver  replicated   5/5       nginx:lates   *:8080->80/tcp
```

Conforme podemos notar, temos o *service* criado e com ele cinco réplicas em execução, ou seja, cinco *containers* em execução.

Se quisermos saber onde estão rodando nossos *containers*, em quais *nodes* eles estão sendo executados, basta digitar o seguinte comando:

```
root@LINUXtips-01:~# docker service ps webserver
```

```
ID   NAME       IMAGE       NODE      DESIRED        STAT CURREN STAT
                                                     E    T      E
ERRO PORTS
R
zbt1 webserver. nginx:lates LINUXtips Running
j    1          t           -01       Running      8
                                      minutes ago
iqm9 webserver. nginx:lates LINUXtips Running
p    2          t           -02       Running      8
                                      minutes ago
jlih webserver. nginx:lates LINUXtips Running
t    3          t           -01       Running      8
                                      minutes ago
qcft webserver. nginx:lates LINUXtips Running
h    4          t           -03       Running      8
                                      minutes ago
e17u webserver. nginx:lates LINUXtips Running
m    5          t           -02       Running      8
                                      minutes ago
root@LINUXtips-01:~#
```

Assim conseguimos saber onde está rodando cada *container* e ainda o seu *status*.

Se precisarmos saber maiores detalhes sobre o meu *service*, basta utilizar o subcomando “inspect”.

```
root@LINUXtips-01:~# docker service inspect webserver
[
   {
     “ID”: “0azz4psgfpkf0i5i3mbfdiptk”,
     “Version”: {
        “Index”: 29
     },
     “CreatedAt”: “2017-06-09T19:35:58.180235688Z”,
     “UpdatedAt”: “2017-06-09T19:35:58.18899891Z”,
     “Spec”: {
        “Name”: “webserver”,
        “Labels”: {},
        “TaskTemplate”: {
           “ContainerSpec”: {
```

```
            "Image":
"nginx:latest@sha256:41ad9967ea448d7c2b203c699b429abe1ed5af331cd925339
00c6d77490e0268",
            "StopGracePeriod": 10000000000,
            "DNSConfig": {}
         },
         "Resources": {
            "Limits": {},
            "Reservations": {}
         },
         "RestartPolicy": {
            "Condition": "any",
            "Delay": 5000000000,
            "MaxAttempts": 0
         },
         "Placement": {},
         "ForceUpdate": 0
      },
      "Mode": {
         "Replicated": {
            "Replicas": 5
         }
      },
      "UpdateConfig": {
         "Parallelism": 1,
         "FailureAction": "pause",
         "Monitor": 5000000000,
         "MaxFailureRatio": 0,
         "Order": "stop-first"
      },
      "RollbackConfig": {
         "Parallelism": 1,
         "FailureAction": "pause",
         "Monitor": 5000000000,
         "MaxFailureRatio": 0,
         "Order": "stop-first"
      },
      "EndpointSpec": {
         "Mode": "vip",
         "Ports": [
            {
               "Protocol": "tcp",
```

```
                "TargetPort": 80,
                "PublishedPort": 8080,
                "PublishMode": "ingress"
            }
         ]
      }
   },
   "Endpoint": {
      "Spec": {
         "Mode": "vip",
         "Ports": [
            {
               "Protocol": "tcp",
               "TargetPort": 80,
               "PublishedPort": 8080,
               "PublishMode": "ingress"
            }
         ]
      },
      "Ports": [
         {
            "Protocol": "tcp",
            "TargetPort": 80,
            "PublishedPort": 8080,
            "PublishMode": "ingress"
         }
      ],
      "VirtualIPs": [
         {
            "NetworkID": "89t2aobeik8j7jcre8lxhj04l",
            "Addr": "10.255.0.5/16"
         }
      ]
   }
}
]
root@LINUXtips-01:~#
```

Na saída do "inspect" conseguiremos pegar informações importantes sobre nosso *service*, como portas expostas, volumes, *containers*, limitações, entre outras coisas.

Uma informação muito importante é o endereço do VIP do *service*:

```
"VirtualIPs": [
          {
            "NetworkID": "89t2aobeik8j7jcre8lxhj04l",
            "Addr": "10.255.0.5/16"
          }
```

Esse é o endereço IP do "balanceador" desse *service*, ou seja, sempre que acessarem via esse IP, ele distribuirá a conexão entre os *containers*. Simples, não?

Agora, se quisermos aumentar o número de *containers* debaixo desse *service*, é muito simples. Basta executar o comando a seguir:

```
root@LINUXtips-01:~# docker service scale webserver=10
webserver scaled to 10
root@LINUXtips-01:~#
```

Pronto, simples assim!

Agora já temos dez *containers* respondendo requisições debaixo do nosso *service webserver*! Simples como voar!

Para visualizar, basta executar:

```
root@LINUXtips-01:~# docker service ls
ID   NAME       MODE         REPLICAS  IMAGE          PORTS
0azz webserver  replicated   10/10     nginx:latest   *:8080->80/tcp
root@LINUXtips-01:~#
```

Para saber em quais *nodes* eles estão em execução, lembre-se do "docker service ls webserver".

Para acessar os *logs* desse *service*, basta digitar:

```
root@LINUXtips-01:~# docker service logs -f webserver
webserver.5.e17umj6u6bix@LINUXtips-02     |     10.255.0.2     -     -
[09/Jun/2017:19:36:12   +0000]   "GET   /   HTTP/1.1"   200   612   "-"
"curl/7.47.0" "-"
```

Assim, você terá acesso aos *logs* de todos os *containers* desse *service*. Muito prático!

"Cansei de brincar! Quero remover esse meu *service*!" É tão simples quanto criá-lo. Digite:

```
root@LINUXtips-01:~# docker service rm webserver
webserver
root@LINUXtips-01:~#
```

Pronto! Seu *service* foi excluído e você pode conferir na saída do comando a seguir:

```
root@LINUXtips-01:~# docker service ls
ID NAME MODE REPLICAS IMAGE PORTS
root@LINUXtips-01:~#
```

Criar um *service* com um volume conectado é bastante simples. Faça:

```
root@LINUXtips-01:~# docker service create --name webserver --replicas
5 -p 8080:80 --mount type=volume,src=teste,dst=/app nginx
yfheu3k7b8u4d92jemglnteqa
root@LINUXtips-01:~#
```

Quando criamos um *service* com um volume conectado a ele, isso indica que esse volume estará disponível em todos os nossos *containers* desse *service*, ou seja, o volume com o nome de "teste" estará montado em todos os *containers* no diretório "/app".

14. Docker Secrets

Ninguém tem dúvida de que a arquitetura de microsserviços já se provou eficiente. Porém, implementar segurança, principalmente em um contexto de infraestrutura imutável, tem sido um belo desafio.

Com questões que envolvem desde como separar a senha do *template* em uma imagem até como trocar a senha de acesso a um serviço sem interrompê-lo, *workarounds* não faltam. Mas como sempre dá para melhorar, na versão 1.13 nossos queridos amigos do Docker lançaram o que foi chamado de Docker Secrets.

O Docker Secrets é a solução do Docker para trabalhar com toda a parte de *secrets* em um ambiente *multi-node* e *multi-container*. Em outras palavras, um *"swarm mode" cluster*. A *secret* pode conter qualquer coisa, porém deve ter um tamanho máximo de 500 KB. Por enquanto essa belezinha não está disponível fora do contexto do Docker Swarm – na verdade, não é claro se vai algum dia ser diferente. Por enquanto somos encorajados a usar um *service* para fazer *deploy* de *containers* individuais.

14.1. O comando *docker secret*

O comando "docker secret" vem com alguns subcomandos. São eles:

```
docker secret --help

Usage:   docker secret COMMAND

Manage Docker secrets

Options:
   --help Print usage


Commands:
  create    Create a secret from a file or STDIN as content
  inspect   Display detailed information on one or more secrets
  ls        List secrets
  rm        Remove one or more secrets

Run 'docker secret COMMAND --help' for more information on a command.
```

- ⇨ **create** – Cria uma *secret* a partir do conteúdo de um arquivo ou STDIN.
- ⇨ **inspect** – Mostra informações detalhadas de uma ou mais *secrets*.
- ⇨ **ls** – Lista as *secrets* existentes.
- ⇨ **rm** – Remove uma ou mais *secrets*.

Create

Como dito, o "create" aceita conteúdo tanto do STDIN...

```
root@linuxtips:~#  echo  'minha  secret'  |  docker  secret  create
minha_secret -
jxr0pilzhtqsiqi1f1fjmmg4t
root@linuxtips:~#
```

...quanto de um arquivo:

```
root@linuxtips:~# docker secret create minha_secret minha_secret.txt
ci7mse43i5ak378sg3qc4xt04
root@linuxtips:~#
```

Inspect

Fique tranquilo, o “inspect” não mostra o conteúdo da sua *secret*! :P

Em vez disso, ele mostra uma série de informações sobre a *secret*, incluindo sua criação e modificação (apesar de não ter, na verdade, como modificar uma *secret*; uma vez criada, ela não pode ser atualizada via CLI, porém já há um *endpoint* na API do Docker Swarm para *update* de *secret* – “/secrets/{id}/update”, vamos aguardar!)

```
root@linuxtips:~# docker secret inspect minha_secret
[
     {
     “ID”: “ci7mse43i5ak378sg3qc4xt04”,
     “Version”: {
        “Index”: 808
     },
     “CreatedAt”: “2017-07-02T17:17:18.143116694Z”,
     “UpdatedAt”: “2017-07-02T17:17:18.143116694Z”,
     “Spec”: {
        “Name”: “minha_secret”,
        “Labels”: {}
     }
     }
]
root@linuxtips:~#
```

O “inspect” aceita mais de uma *secret* por vez e mostrará o resultado na mesma sequência.

ls && rm

Servem respectivamente para listar suas *secrets* e removê-las.

```
root@linuxtips:~# docker secret ls
ID        NAME CREATED UPDATED
ci7mse43i5ak378sg3qc4xt04  minha_secret  About  a  minute  ago  About  a
minute ago
root@linuxtips:~#

root@linuxtips:~# docker secret rm minha_secret
```

```
minha_secret
root@linuxtips:~#
```

14.2. Tudo bem, mas como uso isso?

As *secrets* são consumidas por serviços, como já citamos, e isso acontece através de associação explícita, usando a *flag* "--secret" na hora de criar um serviço. Vamos para um exemplo.

Primeiro vamos criar uma *secret* com a senha do banco de dados da nossa aplicação *fake*.

```
root@linuxtips:~# echo 'senha_do_banco' | docker secret create db_pass
-
kxzgmhlu3ytv64hbqzg30nc8u
root@linuxtips:~#
```

Agora, vamos associá-la à nossa *app*, criando um serviço.

```
root@linuxtips:~# docker service create --name app --detach=false --
secret db_pass minha_app:1.0

npfmry3vcol61cmcql3jiljk2
overall progress: 1 out of 1 tasks
1/1: running [==================================================>]
verify: Waiting 1 seconds to verify that tasks are stable...
root@linuxtips:~# docker service ls
ID             NAME          MODE
REPLICAS       IMAGE         PORTS
npfmry3vcol6   app           replicated             1/1   minha_app:1.0

root@linuxtips:~# docker container ls
CONTAINER ID   IMAGE         COMMAND
CREATED        STATUS        PORTS                  NAMES
65d1533f5b50   minha_app:1.0 "/bin/sh -c ./scri..."
40 seconds ago Up 39 seconds
app.1.molbmj0649c7xkzfermkuwrx2
root@linuxtips:~#
```

Também é possível dar acesso a *keys* para serviços já criados, através da *flag* "--secret-add" do comando "docker service update", assim como revogá-las, usando a *flag* "--secret-rm" no mesmo comando. Ver o tópico "Atualizando a *secret* de um serviço".

14.3. Acessando a *secret*

Com o serviço criado, a *secret* ficará disponível para todos os *containers* daquele *service* e estará em arquivos dentro do diretório "/run/secrets", montado em *tmpfs*. Se a sua *secret* chamar "db_pass", como no exemplo, o conteúdo dela estará em "/run/secrets/db_pass".

É possível incluir alguns parâmetros na hora de adicionar uma *secret* a um serviço, como, por exemplo, o *target*, que altera o nome do arquivo no destino e até itens de segurança, como *uid*, *gid* e *mode*:

```
docker service create --detach=false --name app --secret
source=db_pass,target=password,uid=2000,gid=3000,mode=0400
minha_app:1.0
```

Dentro do *container* ficaria assim:

```
root@4dd6b9cbff1a:/app# ls -lhart /run/secrets/
total 12K
-r-------- 1 2000 3000   15 Jul   2 17:44 password
drwxr-xr-x 7 root root 4.0K Jul   2 17:44 ..
drwxr-xr-x 2 root root 4.0K Jul   2 17:44 .
root@4dd6b9cbff1a:/app#
```

E aí basta que a sua aplicação leia o conteúdo do arquivo.

```
root@8b16b5335334:/app# python
Python 2.7.12 (default, Nov 19 2016, 06:48:10)
[GCC 5.4.0 20160609] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
>>> secret = open('/run/secrets/password').read()
>>>
```

```
>>> print "minha secret e: %s" % (secret)
minha secret e: nova_senha

>>>
```

14.4. Atualizando a *secret* de um serviço

Secrets foram criadas para serem imutáveis, ou seja, caso você queira trocar a *secret* de um serviço, você precisará criar outra *secret*. A troca da *secret* é bem fácil. Por exemplo, para trocar a *secret* "db_pass" do exemplo anterior, teríamos que fazer o seguinte:

Criar uma nova *secret*:

```
root@linuxtips:~# echo 'nova_senha' | docker secret create db_pass_1 -
221uzilbsl1ybna7g7014hoqr
root@linuxtips:~#
```

Adicionar à *secret* a *app* criada anteriormente:

```
root@linuxtips:~# docker service update --secret-rm db_pass --
detach=false --secret-add source=db_pass_1,target=password app
app
overall progress: 1 out of 1 tasks
1/1: running [==================================================>]
verify: Waiting 1 seconds to verify that tasks are stable...
root@linuxtips:~#
```

Após a atualização, basta remover a *secret* antiga:

```
root@linuxtips:~# docker secret rm db_pass
db_pass
root@linuxtips:~#
```

15. Docker Compose

Bem, agora chegamos em uma das partes mais importantes do livro, o sensacional e completo Docker Compose!

O Docker Compose nada mais é do que uma forma de você conseguir escrever em um único arquivo todos os detalhes do ambiente de sua aplicação. Antes nós usávamos o *dockerfile* apenas para criar imagens, seja da nossa aplicação, do nosso BD ou do nosso *webserver*, mas sempre de forma unitária, pois temos um *dockerfile* para cada "tipo" de *container*: um para a nossa *app*, outro para o nosso BD e assim por diante.

Com o Docker Compose nós falamos sobre o ambiente inteiro. Por exemplo, no Docker Compose nós definimos quais os *services* que desejamos criar e quais as características de cada *service* (quantidade de *containers* debaixo daquele *service*, volumes, *network*, *secrets*, etc.).

O padrão que os *compose files* seguem é o YML, supersimples e de fácil entendimento, porém sempre é bom ficar ligado na sintaxe que o padrão YML lhe impõe. ;)

Bem, vamos parar de falar e começar a brincadeira!

Antes a gente precisava instalar o Docker Compose para utilizá-lo. Porém, hoje nós temos o subcomando "docker stack", já disponível

junto à instalação do Docker. Ele é responsável por realizar o *deploy* de nossos *services* através do Docker Compose de maneira simples, rápida e muito efetiva.

'Bora começar! A primeira coisa que devemos realizar é a própria criação do *compose file*. Vamos começar por um mais simples e vamos aumentando a complexidade conforme evoluímos.

Lembre-se: para que possamos seguir com os próximos exemplos, o seu *cluster swarm* deverá estar funcionando perfeitamente. Portanto, se ainda não estiver com o *swarm* ativo, execute:

```
# docker swarm init
```

Vamos criar um diretório chamado "Composes", somente para que possamos organizar melhor nossos arquivos.

```
# mkdir /root/Composes
# mkdir /root/Composes/1
# cd /root/Composes/1
# vim docker-compose.yml
```

```
version: "3"
services:
  web:
   image: nginx
   deploy:
     replicas: 5
     resources:
      limits:
        cpus: "0.1"
        memory: 50M
     restart_policy:
      condition: on-failure
   ports:
   - "8080:80"
   networks:
   - webserver
networks:
  webserver:
```

Pronto! Agora já temos o nosso primeiro *docker-compose*. O que precisamos agora é realizar o *deploy*, porém antes vamos conhecer algumas opções que utilizamos anteriormente:

- ⇨ **version: "3" –** Versão do *compose* que estamos utilizando.
- ⇨ **services:** – Início da definição de meu serviço.
- ⇨ **web: –** Nome do serviço.
- ⇨ **image: nginx** – Imagem que vamos utilizar.
- ⇨ **deploy: –** Início da estratégia de *deploy*.
- ⇨ **replicas: 5** – Quantidade de réplicas.
- ⇨ **resources:** – Início da estratégia de utilização de recursos.
- ⇨ **limits:** – Limites.
- ⇨ **cpus: "0.1"** – Limite de CPU.
- ⇨ **memory: 50M** – Limite de memória.
- ⇨ **restart_policy:** – Políticas de *restart*.
- ⇨ **condition: on-failure** – Somente irá "restartar" o *container* em caso de falha.
- ⇨ **ports:** – Quais portas desejamos expor.
- ⇨ **- "8080:80"** – Portas expostas e "bindadas".
- ⇨ **networks:** – Definição das redes que utilizaremos nesse serviço.
- ⇨ **- webserver** – Nome da rede desse serviço.
- ⇨ **networks:** – Declarando as redes que usaremos nesse *docker-compose*.
- ⇨ **webserver:** – Nome da rede a ser criada, caso não exista.

Simples como voar, não? :D

15.1. O comando *docker stack*

Agora precisamos realizar o *deploy* desse *service* através do *compose file* que criamos. Para isso, vamos utilizar o sensacional "docker stack":

```
root@LINUXtips-01:~/Composes/1# docker stack deploy -c docker-compose.yml primeiro
Creating network primeiro_webserver
Creating service primeiro_web
root@LINUXtips-01:~/Composes/1#
```

Simples assim, e nosso *service* já está disponível para uso. Agora vamos verificar se realmente o *service* subiu e se está respondendo conforme esperado:

```
root@LINUXtips-01:~/Composes/1# curl 0:8080
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
   body {
     width: 35em;
     margin: 0 auto;
     font-family: Tahoma, Verdana, Arial, sans-serif;
   }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

```
root@LINUXtips-01:~/Composes/1#
```

Sensacional, o nosso *service* está em pé, pois recebemos a página de boas-vindas do Nginx!

Vamos verificar se está tudo certo com o *service*:

```
root@LINUXtips-01:~/Composes/1# docker service ls
ID     NAME                              MODE  REPLICAS          IMAGE
PORTS
mw95t  primeiro_web replicated           5/5   nginx:latest
*:8080->80/tcp

root@LINUXtips-01:~/Composes/1#
docker service ps primeiro_web
ID    NAME    IMAGE    NODE    DESIRED STATE    CURRENT STATE    ERROR
PORTS
lrcqo8ifult primeiro_web. nginx:latest
q           1
LINUXtips-  Running       Running   2   minutes
02                        ago
ty16mkcqdwy primeiro_web. nginx:latest         Runnin Running        2
l           2             LINUXtips-03         g      minutes ago
dv670shw22o primeiro_web. nginx:latest         Runnin Running        2
2           3             LINUXtips-01         g      minutes ago
sp0k1tnjftn primeiro_web. nginx:latest         Runnin Running        2
r           4             LINUXtips-01         g      minutes ago
4fpl35llq1i primeiro_web. nginx:latest         Runnin Running        2
h           5             LINUXtips-03         g      minutes ago
root@LINUXtips-01:~/Composes/1#
```

Para listar todos os *stacks* criados, basta executar:

```
root@LINUXtips-01:~/Composes/1# docker stack ls
NAME                                SERVICES
primeiro                            1
root@LINUXtips-01:~/Composes/1#
```

Perceba: a saída diz que possuímos somente um *stack* criado e esse *stack* possui um *service*, que é exatamente o nosso do Nginx.

Para visualizar os *services* que existem em determinado *stack*, execute:

```
root@LINUXtips-01:~/Composes/1# docker stack services primeiro
ID                  NAME                MODE
REPLICAS            IMAGE               PORTS
mx0p4vbrzfuj        primeiro_web        replicated
5/5                 nginx:latest        *:8080->80/tcp
root@LINUXtips-01:~/Composes/1#
```

Podemos verificar os detalhes do nosso *stack* criado através do comando a seguir:

```
root@LINUXtips-01:~/Composes/1# docker stack ps primeiro
ID                  NAME                IMAGE
NODE                DESIRED STATE       CURRENT STATE
ERROR               PORTS
x3u03509w9u3        primeiro_web.1      nginx:latest
LINUXtips-03        Running             Running 5 seconds ago
3hpu5lo6yvld        primeiro_web.2      nginx:latest
LINUXtips-02        Running             Running 5 seconds ago
m82wbwuwoza0        primeiro_web.3      nginx:latest
LINUXtips-03        Running             Running 5 seconds ago
y7vizedqvust        primeiro_web.4      nginx:latest
LINUXtips-02        Running             Running 5 seconds ago
wk0acjny16jm        primeiro_web.5      nginx:latest
LINUXtips-01        Running             Running 5 seconds ago
root@LINUXtips-01:~/Composes/1#
```

Maravilha! Nosso *service* está *UP* e tudo está em paz!

Em poucos minutos subimos o nosso *service* do Nginx em nosso *cluster* utilizando o *docker-compose* e o "docker stack", simples como voar!

Agora vamos imaginar que nós queiramos remover esse meu *service*. Como fazemos? Simples:

```
root@LINUXtips-01:~/Composes/1# docker stack rm primeiro
Removing service primeiro_web
Removing network primeiro_webserver
root@LINUXtips-01:~/Composes/1#
```

Para verificar se realmente removemos o *service*:

```
root@LINUXtips-01:~/Composes/1# docker service ls
ID   NAME      MODE      REPLICAS            IMAGE        PORTS
root@LINUXtips-01:~/Composes/1#
```

Pronto! Nosso *service* está removido!

Vamos aumentar um pouco a complexidade na criação de nosso *docker-compose* nesse novo exemplo.

Vamos criar mais um diretório, onde criaremos o nosso novo *compose file*:

```
root@LINUXtips-01:~/Composes# mkdir 2
root@LINUXtips-01:~/Composes# cd 2
root@LINUXtips-01:~/Composes# vim docker-compose.yml
```

```
version: '3'
services:
  db:
   image: mysql:5.7
   volumes:
     - db_data:/var/lib/mysql
   environment:
     MYSQL_ROOT_PASSWORD: somewordpress
     MYSQL_DATABASE: wordpress
     MYSQL_USER: wordpress
     MYSQL_PASSWORD: wordpress

  wordpress:
   depends_on:
     - db
```

```
    image: wordpress:latest
    ports:
      - "8000:80"
    environment:
      WORDPRESS_DB_HOST: db:3306
      WORDPRESS_DB_USER: wordpress
      WORDPRESS_DB_PASSWORD: wordpress
volumes:
    db_data:
```

Perfeito!

Nesse exemplo estamos conhecendo mais algumas opções que podemos utilizar no *docker-compose*. São eles:

- ⇨ **volumes:** – Definição dos volumes utilizados pelo *service*.
- ⇨ **- db_data:/var/lib/mysql** – Volume e destino.
- ⇨ **environment:** – Definição de variáveis de ambiente utilizados pelo *service.*
- ⇨ **MYSQL_ROOT_PASSWORD: somewordpress** – Variável e valor.
- ⇨ **MYSQL_DATABASE: wordpress** – Variável e valor.
- ⇨ **MYSQL_USER: wordpress** – Variável e valor.
- ⇨ **MYSQL_PASSWORD: wordpress** – Variável e valor.
- ⇨ **depends_on:** – Indica que esse *service* depende de outro para subir.
- ⇨ **- db** – Nome do *service* necessário para sua execução.

Muito simples, não?!?

Agora vamos realizar o *deploy* desse exemplo. Como se pode perceber, o nosso *stack* é composto por dois *services*, o Wordpress e o MySQL.

```
root@LINUXtips-01:~/Composes/2# docker stack deploy -c docker-compose.yml segundo
Creating network segundo_default
Creating service segundo_db
```

```
Creating service segundo_wordpress
root@LINUXtips-01:~/Composes/2#
```

Conforme esperado, ele realizou a criação dos dois *services* e da rede do *stack*.

Para acessar o seu Wordpress, basta acessar em um navegador:

http://SEU_IP:8000

Seu Wordpress está pronto para uso!

Para verificar se correu tudo bem com os *services*, lembre-se dos comandos:

```
root@LINUXtips-01:~/Composes/1# docker stack ls
root@LINUXtips-01:~/Composes/1# docker stack services segundo
root@LINUXtips-01:~/Composes/1# docker service ls
root@LINUXtips-01:~/Composes/1# docker service ps segundo_db
root@LINUXtips-01:~/Composes/1# docker service ps segundo_wordpress
```

Para visualizar os *logs* de determinado *service*:

```
root@LINUXtips-01:~/Composes/2# docker service logs segundo_wordpress
segundo_wordpress.1.r6reuq8fsil0@LINUXtips-01 | WordPress not found in /var/www/html - copying now...
segundo_wordpress.1.r6reuq8fsil0@LINUXtips-01 | Complete! WordPress has been successfully copied to /var/www/html
segundo_wordpress.1.r6reuq8fsil0@LINUXtips-01 | AH00558: apache2: Could not reliably determine the server's fully qualified domain name, using 10.0.4.5. Set the 'ServerName' directive globally to suppress this message
segundo_wordpress.1.r6reuq8fsil0@LINUXtips-01 | AH00558: apache2: Could not reliably determine the server's fully qualified domain name, using 10.0.4.5. Set the 'ServerName' directive globally to suppress this message
segundo_wordpress.1.r6reuq8fsil0@LINUXtips-01 | [Sun Jun 11 10:32:47.392836 2017] [mpm_prefork:notice] [pid 1] AH00163: Apache/2.4.10 (Debian) PHP/5.6.30 configured -- resuming normal operations
segundo_wordpress.1.r6reuq8fsil0@LINUXtips-01 | [Sun Jun 11 10:32:47.392937 2017] [core:notice] [pid 1] AH00094: Command line: 'apache2 -D FOREGROUND'
root@LINUXtips-01:~/Composes/2#
```

E se for necessária uma modificação em nosso *stack* e depois um *re-deploy*, como fazemos? É possível?

Claro! Afinal, Docker é muita vida!

```
root@LINUXtips-01:~/Composes# mkdir 3
root@LINUXtips-01:~/Composes# cd 3
root@LINUXtips-01:~/Composes/3# vim docker-compose.yml
```

```
version: "3"
services:
  web:
   image: nginx
   deploy:
     replicas: 5
     resources:
      limits:
        cpus: "0.1"
        memory: 50M
     restart_policy:
      condition: on-failure
   ports:
     - "8080:80"
   networks:
     - webserver

  visualizer:
   image: dockersamples/visualizer:stable
   ports:
     - "8888:8080"
   volumes:
     - "/var/run/docker.sock:/var/run/docker.sock"
   deploy:
     placement:
      constraints: [node.role == manager]
   networks:
     - webserver

networks:
  webserver:
```

Perceba que apenas adicionamos um novo *service* ao nosso *stack*, o *visualizer*. A ideia é realizar o *update* somente no *stack* para adicionar o *visualizer*, sem deixar indisponível o *service web*.

Antes de realizarmos o *update* desse *stack*, vamos conhecer as novas opções que estão no *compose file* desse exemplo:

deploy:

- ⇨ **placement:** – Usado para definir a localização do nosso *service*.
- ⇨ **constraints: [node.role == manager]** – Regra que obriga a criação desse *service* somente nos *nodes manager*.

Agora vamos atualizar o nosso *stack*:

```
root@LINUXtips-01:~/Composes/3# docker stack deploy -c docker-compose.yml primeiro
Creating service primeiro_visualizer
Updating service primeiro_web (id: mx0p4vbrzfujk087c3xe2sjvo)
root@LINUXtips-01:~/Composes/3#
```

Perceba que, para realizar o *update* do *stack*, utilizamos o mesmo comando que usamos para realizar o primeiro *deploy* do *stack*, o "docker stack deploy".

Que tal aumentar ainda mais a complexidade e o número de *services* de um *stack*? 'Bora?

Para esse exemplo, vamos utilizar um projeto do próprio Docker- (<https://github.com/dockersamples/example-voting-app>), onde teremos diversos *services*. Vamos criar mais um diretório para receber o nosso projeto:

```
root@LINUXtips-01:~/Composes# mkdir 4
root@LINUXtips-01:~/Composes# cd 4
root@LINUXtips-01:~/Composes/4# vim compose-file.yml
version: "3"
services:
  redis:
```

```
 image: redis:alpine
 ports:
   - "6379"
 networks:
   - frontend
 deploy:
   replicas: 2
   update_config:
    parallelism: 2
    delay: 10s
   restart_policy:
    condition: on-failure
db:
 image: postgres:9.4
 volumes:
   - db-data:/var/lib/postgresql/data
 networks:
   - backend
 deploy:
   placement:
    constraints: [node.role == manager]
vote:
 image: dockersamples/examplevotingapp_vote:before
 ports:
   - 5000:80
 networks:
   - frontend
 depends_on:
   - redis
 deploy:
   replicas: 2
   update_config:
    parallelism: 2
   restart_policy:
    condition: on-failure
result:
 image: dockersamples/examplevotingapp_result:before
 ports:
   - 5001:80
 networks:
   - backend
 depends_on:
```

```
    - db
   deploy:
    replicas: 1
    update_config:
     parallelism: 2
     delay: 10s
    restart_policy:
     condition: on-failure
  worker:
   image: dockersamples/examplevotingapp_worker
   networks:
    - frontend
    - backend
   deploy:
    mode: replicated
    replicas: 1
    labels: [APP=VOTING]
    restart_policy:
     condition: on-failure
     delay: 10s
     max_attempts: 3
     window: 120s
    placement:
     constraints: [node.role == manager]
  visualizer:
   image: dockersamples/visualizer:stable
   ports:
    - "8080:8080"
   stop_grace_period: 1m30s
   volumes:
    - "/var/run/docker.sock:/var/run/docker.sock"
   deploy:
    placement:
     constraints: [node.role == manager]
networks:
  frontend:
  backend:
volumes:
  db-data:
```

Ficou mais complexo ou não? Achamos que não, pois no Docker tudo é bastante simples!

Temos algumas novas opções nesse exemplo, vamos conhecê-las:

deploy:

⇨ **mode: replicated** – Qual é o tipo de *deployment*? Temos dois, o *global* e o *replicated*. No *replicated* você escolhe a quantidade de réplicas do seu *service*, já no *global* você não escolhe a quantidade de réplicas, ele irá subir uma réplica por *node* de seu *cluster* (uma réplica em cada *node* de seu *cluster*).

update_config:

⇨ **parallelism: 2** – Como irão ocorrer os *updates* (no caso, de 2 em 2).

⇨ **delay: 10s** – Com intervalo de 10 segundos.

restart_policy:

⇨ **condition: on-failure** – Em caso de falha, *restart*.

⇨ **delay: 10s** – Com intervalo de 10 segundos.

⇨ **max_attempts: 3** – Com no máximo três tentativas.

⇨ **window: 120s** – Tempo para definir se o *restart* do *container* ocorreu com sucesso.

Agora vamos realizar o *deploy* do nosso *stack*:

```
root@LINUXtips-01:~/Composes/4# docker stack deploy -c
docker-compose.yml quarto
Creating network quarto_default
Creating network quarto_frontend
Creating network quarto_backend
Creating service quarto_worker
Creating service quarto_visualizer
Creating service quarto_redis
Creating service quarto_db
Creating service quarto_vote
Creating service quarto_result
root@LINUXtips-01:~/Composes/4#
```

Verificando os *services*:

```
root@LINUXtips-01:~/Composes/4# docker service ls
ID            NAME                                                MODE
REPLICAS      IMAGE
PORTS
3hi3sx2on3t5  quarto_worker                                       replicated
1/1           dockersamples/examplevotingapp_worker:latest
hbsp4fdcvgnz  quarto_visualizer                                   replicated
1/1           dockersamples/visualizer:stable
*:8080->8080/tcp
k6xuqbq7g55a  quarto_db                                           replicated
1/1           postgres:9.4
p2reijydxnsw  quarto_result                                       replicated
1/1           dockersamples/examplevotingapp_result:before        *:5001->80/tcp
rtwnnkwftg9  quarto_redis
u
replicated    2/2                                                 redis:alpine
*:0->6379/tcp
w2ritqiklpok  quarto_vote                                         replicated
2/2           dockersamples/examplevotingapp_vote:before          *:5000->80/tcp
root@LINUXtips-01:~/Composes/4#
```

Lembre-se de sempre utilizar os comandos que já conhecemos para visualizar *stack*, *services*, volumes, *container*, etc.

Para acessar os *services* em execução, abra um navegador e vá aos seguintes endereços:

- ⇨ **Visualizar a página de votação:** http://IP_CLUSTER:5000/
- ⇨ **Visualizar a página de resultados:** http://IP_CLUSTER:5001/
- ⇨ **Visualizar a página de com os *containers* e seus *nodes*:** http://IP_CLUSTER:8080/

Vamos para mais um exemplo. Agora vamos realizar o *deploy* de um *stack* completo de monitoração para o nosso *cluster* e todas as demais máquinas de nossa infraestrutura. Nesse exemplo vamos utilizar um arquivo YML que realizará o *deploy* de diversos

containers para que possamos ter as seguintes ferramentas integradas:

- ⇨ **Prometheus** – Para armazenar todas as métricas de nosso ambiente.
- ⇨ **cAdvisor** – Para coletar informações dos *containers*.
- ⇨ **Node Exporter** – Para coletar informações dos *nodes* do *cluster* e demais máquinas do ambiente.
- ⇨ **Netdata** – Para coletar mais de cinco mil métricas de nossas máquinas, além de prover um *dashboard* sensacional.
- ⇨ **Rocket.Chat** – Para que possamos nos comunicar com outros times e pessoas e também para integrá-lo ao sistema de monitoração, notificando quando os alertas acontecem. O Rocket.Chat é uma excelente alternativa ao Slack.
- ⇨ **AlertManager** – Integrado ao Prometheus e ao Rocket.Chat, é o responsável por gerenciar nossos alertas.
- ⇨ **Grafana** – Integrado à nossa solução de monitoração, ele é o responsável pelos *dashboards* que são produzidos através das métricas que estão armazenadas no Prometheus.

Com esse *stack* é possível monitorar *containers*, VMs e máquinas físicas. Porém, o nosso foco agora é somente no que se refere ao livro e a este capítulo, ou seja, as informações contidas no *compose file* que definirão nosso *stack*.

Para maiores detalhes em relação ao *Giropops-Monitoring*, acesse o repositório no endereço: <https://github.com/badtuxx/giropops--monitoring>.

Antes de conhecer nosso *compose file*, precisamos realizar o clone do projeto:

```
# git clone https://github.com/badtuxx/giropops-monitoring.git
```

Acesse o diretório “giropops-monitoring”:

```
# cd giropops-monitoring
```

O nosso foco aqui será em três caras: o arquivo "grafana.config", o diretório "conf" e o nosso querido e idolatrado "docker-compose.yml".

O arquivo "grafana.config" contém variáveis que queremos passar ao nosso Grafana. Nesse momento a única informação importante é o *password* do *admin*, usuário que utilizaremos para logar na interface web do Grafana.

O diretório "conf" possui os arquivos necessários para que a integração entre as aplicações de nosso *stack* funcione corretamente.

Já o nosso *compose file* traz todas as informações necessárias para que nós possamos realizar o *deploy* de nosso *stack*.

Como o nosso foco é o *compose file*, 'bora lá conhecê-lo!

```
# cat docker-compose.yml
version: '3.3'

services:
  prometheus:
    image: linuxtips/prometheus_alpine
    volumes:
      - ./conf/prometheus/:/etc/prometheus/
      - prometheus_data:/var/lib/prometheus
    networks:
      - backend
    ports:
      - 9090:9090

  node-exporter:
    image: linuxtips/node-exporter_alpine
    hostname: '{{.Node.ID}}'
    volumes:
      - /proc:/usr/proc
      - /sys:/usr/sys
      - /:/rootfs
    deploy:
      mode: global
    networks:
```

```
    - backend
  ports:
    - 9100:9100

alertmanager:
  image: linuxtips/alertmanager_alpine
  volumes:
    - ./conf/alertmanager/:/etc/alertmanager/
  networks:
    - backend
  ports:
    - 9093:9093

cadvisor:
  image: google/cadvisor
  hostname: '{{.Node.ID}}'
  volumes:
    - /:/rootfs:ro
    - /var/run:/var/run:rw
    - /sys:/sys:ro
    - /var/lib/docker/:/var/lib/docker:ro
    - /var/run/docker.sock:/var/run/docker.sock:ro
  networks:
    - backend
  deploy:
    mode: global
  ports:
    - 8080:8080

grafana:
  image: nopp/grafana_alpine
  depends_on:
    - prometheus
  volumes:
    - ./conf/grafana/grafana.db:/grafana/data/grafana.db
  env_file:
    - grafana.config
  networks:
    - backend
    - frontend
  ports:
    - 3000:3000
```

```
# If you already has a RocketChat instance running, just comment the
code of rocketchat, mongo and mongo-init-replica services bellow
  rocketchat:
   image: rocketchat/rocket.chat:latest
   volumes:
     - rocket_uploads:/app/uploads
   environment:
     - PORT=3080
     - ROOT_URL=http://YOUR_IP:3080
     - MONGO_URL=mongodb://giropops_mongo:27017/rocketchat
     - MONGO_OPLOG_URL=mongodb://giropops_mongo:27017/local
   depends_on:
     - giropops_mongo
   ports:
     - 3080:3080

  mongo:
   image: mongo:3.2
   volumes:
     - mongodb_data:/data/db
     #- ./data/dump:/dump
   command: mongod --smallfiles --oplogSize 128 --replSet rs0

  mongo-init-replica:
   image: mongo:3.2
   command: 'mongo giropops_mongo/rocketchat --eval "rs.initiate({ _id:
''rs0'', members: [ { _id: 0, host: ''localhost:27017'' } ]})"'
   depends_on:
     - giropops_mongo

networks:
  frontend:
  backend:

volumes:
   prometheus_data:
   grafana_data:
   rocket_uploads:
   mongodb_data:
```

Perceba que já conhecemos todas as opções que estão nesse exemplo, nada de novo. :D

O que precisamos agora é realizar o *deploy* de nosso *stack*:

```
# docker stack deploy -c docker-compose.yml giropops
Creating network giropops_backend
Creating network giropops_frontend
Creating network giropops_default
Creating service giropops_grafana
Creating service giropops_rocketchat
Creating service giropops_mongo
Creating service giropops_mongo-init-replica
Creating service giropops_prometheus
Creating service giropops_node-exporter
Creating service giropops_alertmanager
Creating service giropops_cadvisor
```

Caso queira verificar se os *services* estão em execução:

```
# docker service ls
```

Para listar os *stacks*:

```
# docker stack ls
```

Para acessar os serviços do quais acabamos de realizar o *deploy*, basta acessar os seguintes endereços:

- ⇨ **Prometheus**: http://SEU_IP:9090
- ⇨ **AlertManager**: http://SEU_IP:9093
- ⇨ **Grafana**: http://SEU_IP:3000
- ⇨ **Node_Exporter**: http://SEU_IP:9100
- ⇨ **Rocket.Chat**: http://SEU_IP:3080
- ⇨ **cAdivisor**: http://SEU_IP:8080

Para remover o *stack*:

```
# docker stack rm giropops
```

Lembrando: para conhecer mais sobre o *giropops-monitoring* acesse o repositório no GitHub e assista à série de vídeos em que o Jeferson fala detalhadamente como montou essa solução:

- ⇨ **Repo**: <https://github.com/badtuxx/giropops-monitoring>
- ⇨ **Vídeos**: <https://www.youtube.com/playlist?list=PLf-O3X2-mxDls9uH8gyCQTnyXNMe10iml>

E assim termina a nossa jornada no mundo do Docker. Esperamos que você tenha aprendido e, mais do que isso, tenha gostado de dividir esse tempo conosco para falar sobre o que nós mais amamos, tecnologia!

15.2. E já acabou?

Esperamos que você tenha curtido viajar conosco durante o seu aprendizado sobre *containers* e principalmente sobre o ecossistema do Docker, que é sensacional!

Não pare de aprender mais sobre Docker! Continue acompanhando o Canal LinuxTips no <https://www.youtube.com/linuxtips> e fique ligado no site do Docker, pois sempre tem novidades e ótima documentação!

Junte-se a nós no <https://www.facebook.com/linuxtipsbr> para que possa acompanhar e tirar dúvidas que possam ter surgido durante seus estudos!

#VAIIII