APRENDENDO DOCKER

Wellington Figueira da Silva

Novatec

© Novatec Editora Ltda. 2016.

Todos os direitos reservados e protegidos pela Lei 9.610 de 19/02/1998. É proibida a reprodução desta obra, mesmo parcial, por qualquer processo, sem prévia autorização, por escrito, do autor e da Editora.

Editor: Rubens Prates
Assistente editorial: Priscila A. Yoshimatsu
Revisão gramatical: Solange Martins
Capa: Carolina Kuwabata
Editoração eletrônica: Carolina Kuwabata

ISBN: 978-85-7522-486-1

Histórico de impressões:

Março/2016 Primeira edição

Novatec Editora Ltda.
Rua Luís Antônio dos Santos 110
02460-000 – São Paulo, SP – Brasil
Tel.: +55 11 2959-6529
Email: novatec@novatec.com.br
Site: www.novatec.com.br
Twitter: twitter.com/novateceditora
Facebook: facebook.com/novatec
LinkedIn: linkedin.com/in/novatec

CAPÍTULO 1
Introdução

Neste primeiro capítulo abordaremos o que é o Docker e veremos um breve histórico sobre contêineres e máquinas virtuais. Veremos também a curta (por enquanto) e promissora história do Docker e por que ele está mudando o paradigma de ambientes de desenvolvimento, testes e até ambientes de produção nesses últimos anos.

1.1 Início do Docker

O primeiro anúncio público do Docker foi feito pelo rapaz da figura 1.1, chamado Solomon Hykes, CEO da dotCloud, em 15 de março de 2013 em uma lightning talk, uma espécie de palestra rápida, durante uma conferência de Python na Califórnia.

Figura 1.1 – Solomon Hykes (fonte: www.crn.com).

Dica: o vídeo dessa minipalestra ainda está disponível no YouTube no endereço *http://bit.ly/solomon-light-talk* e vale muito a pena conferir.

Inicialmente o Docker foi inventado para que a dotCloud pudesse suportar de forma mais simples a gerência de seu PaaS (Platform as a Service), onde desenvolvedores poderiam fazer deploy de suas aplicações de uma maneira similar ao Heroku, mas, em vez de máquinas virtuais por debaixo dos panos, tudo rodaria em contêineres Linux.

1.2 O que é o Docker

Para explicar o que é a ferramenta Docker vamos partir da frase disponível no próprio site *https://www.docker.com/what-docker*: "Docker allows you to package an application with all of its dependencies into a standardized unit for software development." Traduzindo: "O Docker lhe permite empacotar uma aplicação com todas as suas dependências em uma unidade padronizada para desenvolvimento de software."

Esta definição permite imaginar o empacotamento da aplicação. Fazendo um paralelo com o problema do transporte de carga em que existiam diversas maneiras de se transportar – navio, trem, caminhões –, a padronização usando contêineres resolveu essa problema.

Em vez de transportar arroz, soja, eletrônicos, carros, agora transportamos contêineres, dentro dos quais temos os eletrônicos, os carros, os grãos e todo o resto.

De maneira análoga, a ideia do Docker é construirmos nossas aplicações, sejam baseadas na web, sejam workers, sejam compiladas ou não, sejam escritas em PHP, Python, Java, Go, em contêineres que trafegam em qualquer rede e rodam em qualquer servidor Linux.

Nota: a ideia inspirou inclusive seu logotipo visto na figura 1.2 em que a baleia transporta os contêineres.

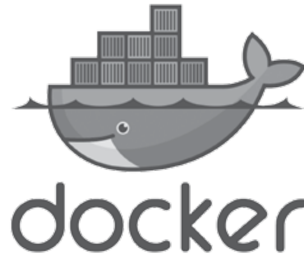

Figura 1.2 – Logotipo do Docker (fonte: www.docker.com).

1.3 Mudança de paradigma

Paradigma vem do latim e é definido como um conceito das ciências e da epistemologia (teoria do conhecimento) que define um exemplo típico ou modelo de algo.

Por um tempo o paradigma era comprar ou alugar máquinas físicas, servidores, para hospedar nossas aplicações. Nos últimos anos o paradigma se tornou computação em nuvem, onde alugamos não a máquina, mas o recurso que essas máquinas representam, CPU, memória ou armazenamento em disco, por exemplo. Isso graças à virtualização.

Arrisco dizer que vamos mudar novamente o paradigma, em vez de virtualizar o sistema operacional para trabalhar em nuvem, vamos isolar esses recursos em contêineres para aproveitar melhor nossas máquinas, sejam físicas ou virtuais.

Mais informações sobre paradigma em *https://pt.wikipedia.org/wiki/Paradigma*.

1.3.1 Virtualização

A maneira mais popular de subir um serviço baseado em web atualmente é por meio de máquinas virtuais, potencializado pela popularização crescente da computação em nuvem, onde não se contratam mais máquinas físicas, mas sim pacotes de CPU e memória alocados em máquinas virtuais.

Nessa técnica chamada virtualização temos um sistema operacional inteiro chamado de guest com seu kernel, bibliotecas e binários rodando sobre um outro sistema operacional chamado de host com a ajuda de um hypervisor

que faz o mapeamento do sistema operacional guest para os recursos reais de hardware. Alguns dos principais hypervisors hoje em dia são: VirtualBox, VMware, Xen, KVM e Hyper-V; mas existem muitos outros, acredite.

Na figura 1.3 temos uma imagem extraída do site da Docker representando de maneira geral aplicações rodando em uma arquitetura de virtualização.

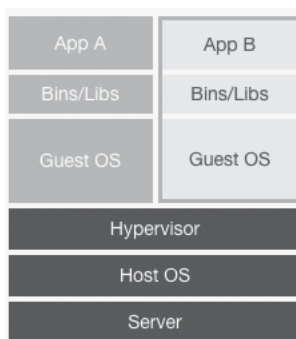

Figura 1.3 – Aplicações em máquinas virtuais (fonte: www.docker.com).

Podemos notar que o item que faz a interface entre o sistemas guest e o sistema host é o hypervisor. Ele que usa os recursos do sistema host e simula o hardware (CPU, memória e disco principalmente) para o sistema guest.

Notamos também o kernel de cada sistema guest sendo carregado sobre o host.

1.3.2 LXC – Linux Containers e o Docker

Os LXC (Linux Containers) existem desde o kernel 2.6.26, que foi distribuído a partir de julho de 2008, porém, só nos últimos anos vem popularizando graças principalmente ao Docker que, por meio da sua arquitetura baseada em API, a utilização de imagens em camadas (técnica de copy-on-write), o repositório público de imagens, e todo o ecossistema em volta evoluiu e simplificou o gerenciamento desses contêineres Linux.

Contêineres Linux são como os sistemas operacionais guests, mas eles compartilham recursos como o kernel do sistema operacional host, rodam dentro de Cgroups, portanto, têm PID único no host e são isolados por namespaces.

Cgroup – abreviação de control groups – é um recurso do kernel do Linux responsável por isolar a utilização de recursos como rede, memória, disco

e CPU. Namespace é uma funcionalidade que permite definir uma área, ou região, muito parecido com as Zones dos Solaris.

O grande trunfo do LXC e muito facilitado no Docker é que conseguimos criar namespaces para processos, para rede, para o hostname, para usuário, para montagem de sistema de arquivos e para comunicação entre processos. E, isolando esses recursos com cgroups, conseguimos criar um contexto que dentro dele temos praticamente tudo que um sistema operacional precisa para rodar.

Ainda veremos neste livro um exemplo em que, devido ao namespace de PID dentro de um contêiner, um processo como um simples ping tem dentro de um contêiner um PID com uma numeração e, fora dele, no host, outra numeração.

Na figura 1.4 vemos que com contêineres não precisamos de toda aquela camada que envolve o kernel do sistema operacional guest. Os binários são executados compartilhando recursos do host.

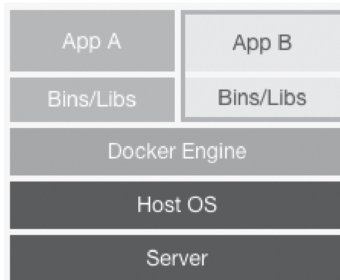

Figura 1.4 – Aplicações no Docker (fonte: www.docker.com).

Também percebemos que quem faz esse meio de campo entre o contêiner e o host é o Docker Engine. Aliás, sem o Docker Engine a gerência de contêineres Linux é muito trabalhosa e ineficiente.

1.3.3 Docker versus máquinas virtuais

Vamos levantar agora vantagens e desvantagens do modelo de aplicações construídas sobre contêineres e do modelo de aplicações construídas sobre máquinas virtuais.

Dentre inúmeras vantagens do Docker, destacamos:

- Padroniza ambientes de desenvolvimento, teste e produção.
- Melhora a utilização de recursos físicos de infraestrutura.
- Facilita a recuperação de dados.
- Melhora também a reutilização, tanto para compor novas camadas com técnica de copy-on-write quanto migrando um contêiner de um host para outros.
- Faz o isolamento utilizando namespaces que isola recursos, utiliza processos PID, faz comunicações interprocessos, rede e filesystem.
- Como iniciamos só o processo e não a pilha toda do sistema operacional, essa inicialização é praticamente instantânea.
- Utilizando o Docker Hub ou registry privado conseguimos distribuir imagens e classificar de maneira simples as oficiais.
- Limita memória e CPU de maneira mais simples, já que podemos designar o limite de memória e utilização de CPU do contêiner no comando de execução.
- Builds automatizados usando arquivos de instrução chamados Dockerfiles são muito mais simples que sistemas de provisionamento como Chef, Puppet, Salt e Ansible.
- O Docker Engine fornece uma API que podemos consumir local ou remotamente para enviar comandos.
- Sempre em inovação, a Docker libera quase um release novo por mês.

As desvantagens do Docker em relação às máquinas virtuais são:

- Por causa do sistema de imagens em camada, o overhead de IO no disco é muito maior.
- Se o host Docker cai, todos os contêineres nele caem também.
- Dificulta troubleshooting, já que adicionamos mais uma camada na investigação do problema.
- Por mais isolado que estejam os processos (por causa dos namespaces), como há compartilhamento de recursos, há a possibilidade de um ataque sofisticado, ou a exploração de uma configuração falha para assumir controle da máquina host.

- Há menor portabilidade,. já que não temos como rodar contêineres Linux em hosts Windows ou hosts BSDs. A recíproca também é verdadeira: Apenas contêineres Linux em hosts Linux[1].

1.4 Resumindo o Docker

Open source desde o release 0.9 (março de 2013), o Docker é praticamente um ecossistema e hoje é composto pelos seguintes componentes:

Contêineres

Os contêineres Docker inicialmente eram como os antigos LXC (Linux Containers) que existem em toda distribuição Linux desde julho de 2008. Com algumas diferenças, principalmente no sistema de armazenamento de dados e no sistema de imagem de camadas, são gerenciados por um Docker Engine. A partir da versão 1 a Docker substituiu o LXC pela libcontainer, e somente a partir de junho de 2015 passou a utilizar o runC (Runtime Container), uma evolução da libcontainer, que posteriormente foi doado ao OCI (Open Container Iniciative), projeto que abordaremos no capítulo 6.

Engine

Essa engine é um daemon que gerencia a construção e execução dos contêineres, faz o trabalho de criar o CHROOT e controlar os recursos de rede, de CPU, de memória e demais recursos dos host que os contêineres utilizem.

Cliente

A engine expõe uma API onde, com um cliente consumindo o socket ou a API, conseguimos passar os comandos para o daemon (Docker Engine) para criar, executar, parar, remover ou adicionar processos, além de listar imagens, contêineres, redes e volumes, entre outros comandos.

1 A Microsoft, com a ajuda da Docker, vai disponibilizar no Windows Server 2016 a possibilidade de rodar contêineres Windows rodando em hosts Windows. A FreeBSD tem o sistema de Jails para isolar processos similar ao contêiner disponível desde a versão 4, lançado em meados de 2000.

Registry

Quando salvamos o estado de um contêiner, geramos uma imagem que podemos reutilizar em outro contêiner por meio de uma técnica já conhecida chamada copy-on-write, sobre a qual falaremos mais no capítulo 6.

O Registry público da Docker, também chamado de Docker Hub, é um repositório similar ao GitHub, onde hospedamos as imagens dos contêineres utilizando comandos como `commit`, `push` e `pull`, por exemplo.

Compose

É uma ferramenta que nos ajuda a organizar a execução de diversos contêineres e a forma como eles vão utilizar recursos de rede, de persistência de dados e comunicar-se entre eles. Com um comando e um arquivo de configuração podemos gerenciar diversos contêineres simultaneamente. Abordaremos melhor o Docker Compose no capítulo 12.

Machine

Foi inspirado no Boot2docker, projeto que criava uma máquina virtual Linux enxuta e já com o Docker daemon rodando para utilizarmos em computadores Windows ou Mac OS X. O Docker Machine permite criar máquinas virtuais rodando o Docker não só em ambiente local, mas também em provedores de serviço de computação em nuvem. Basta escolher um driver entre Amazon AWS, DigitalOcean, Rackspace, VirtualBox ou outros para provisionar uma máquina pronta para rodar seus contêineres. Veremos mais informações no capítulo 13.

Swarm

De modo similar ao que o Compose faz levantando contêineres de maneira simples em um Docker host, o Swarm consegue juntar diversas Docker Machines em uma espécie de cluster, facilitando a gerência de contêineres em diversos Docker hosts. Mais informações serão mostradas no capítulo 14.

Kitematic

Ferramenta que possibilita criar seus contêineres usando uma interface GUI já integrada ao Docker Hub e que cria automaticamente uma máquina virtual local em seu VirtualBox. Em março de 2015, quando a Kitematic foi comprada pela Docker, a interface GUI havia sido desenvolvida exclusivamente para Mac OS X; atualmente, o Kitematic já é instalado via Docker Toolbox e o temos rodando também em Windows.

1.5 Arquitetura do ecossistema Docker

Podemos representar a arquitetura do Docker com a figura 1.5, extraída do próprio site da Docker:

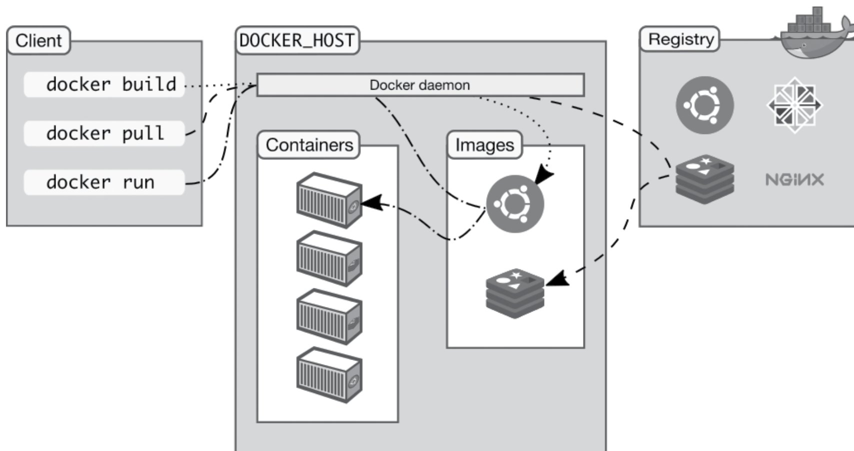

Figura 1.5 – Arquitetura do Docker (fonte: www.docker.com).

Percebemos o Docker host, que pode ser uma máquina física ou uma máquina virtual.

Percebemos também que os comandos rodados no cliente executam ações no daemon (Docker Engine) dentro do Docker host. Estas ações diversas gerenciam contêiner e imagens, podendo criar, apagar, executar, alterar, remover, consultar estado, exportar, gravar etc.

1.6 Consumindo a API

Como mencionamos, o Docker daemon expõe uma API que é consumida pelo Docker client e pode ser utilizada por sistemas que automatizam e simplificam a gerência de contêineres. O Kitematic, o Docker Compose e outras ferramentas, como Shipyard, Panamax e DockerUI, trabalham sobre essa API.

Com exceção de alguns comandos como `pull`, `push` e `exec` que necessitam de transporte de dados, a API tende a ser REST e pode ser consumida com requisições web simples.

1.6.1 Mostrando a versão

Neste exemplo vamos rodar o comando para consultar a versão e, em seguida, consumir a API usando o comando `wget` presente nas distribuições Linux e no Mac OS X.

```
$ docker version
Client:
 Version:      1.10.2
 API version:  1.22
 Go version:   go1.5.3
 Git commit:   c3959b1
 Built:        Mon Feb 22 22:37:33 2016
 OS/Arch:      darwin/amd64
Server:
 Version:      1.10.2
 API version:  1.22
 Go version:   go1.5.3
 Git commit:   c3959b1
 Built:        Mon Feb 22 22:37:33 2016
 OS/Arch:      linux/amd64
$ wget \
    --no-check-certificate \
    --certificate=$DOCKER_CERT_PATH/cert.pem \
    --private-key=$DOCKER_CERT_PATH/key.pem  \
 https://$(docker-machine ip default):2376/version -O - -q | python -m
json.tool
```

```
{
    "ApiVersion": "1.22",
    "Arch": "amd64",
    "BuildTime": "2016-02-22T22:37:33.778002647+00:00",
    "GitCommit": "c3959b1",
    "GoVersion": "go1.5.3",
    "KernelVersion": "4.1.18-boot2docker",
    "Os": "linux",
    "Version": "1.10.2"
}
```

Podemos utilizar também o comando `curl`, porém ele não funciona nas versões do Mac OS X 10.9 e 10.10, pois a Apple utiliza uma versão de `curl` diferente por causa do OpenSSL que não foi atualizado. Para ver a versão utilizando `curl` nos Mac OS X anteriores ao 10.9 e nas distribuições de Linux, podemos utilizar o seguinte exemplo:

```
curl --insecure \
     --cert $DOCKER_CERT_PATH/cert.pem \
     --key $DOCKER_CERT_PATH/key.pem \
     https:/$(docker-machine ip default):2376/images/json
```

Devemos ficar atentos ao colocar o endereço IP de onde está o Docker host em nossas requisições. Nos exemplos específicos estamos usando comandos do shell para ele buscar o endereço IP por meio do comando `docker-machine ip`, informando também o nome do Docker host: `default`.

1.6.2 Criando um contêiner

No seguinte exemplo listamos os contêineres usando o Docker client, iniciamos um contêiner do Nginx via API usando o comando `wget` e listamos, novamente com o Docker client, os contêineres disponíveis.

```
$ docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED
STATUS              PORTS               NAMES
$ wget \
>     --no-check-certificate \
>     --certificate=$DOCKER_CERT_PATH/cert.pem \
```

```
>     --private-key=$DOCKER_CERT_PATH/key.pem \
>    --method=POST \
>     --header='Content-Type: application/json' \
>     --post-data '{"Image":"nginx:latest"}' \
> https://$(docker-machine ip default):2376/containers/create -O - -q
{"Id":"47ba2886f6f48b2d10b0ddb85670d762792fd7c0e1e92f4eb0328b140759b966",
"Warnings":null}
$ docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED
STATUS              PORTS               NAMES
47ba2886f6f4       nginx:latest        "nginx -g 'daemon off"   54 seconds
ago      Created                                 admiring_visvesvaraya
```

1.6.3 Removendo um contêiner

Neste exemplo listamos os contêineres usando o Docker client, removemos o contêiner do Nginx criado anteriormente via API usando o comando `wget` e listamos, novamente com o Docker client, os contêineres disponíveis.

```
$ docker ps -a
CONTAINER ID        IMAGE               COMMAND                  CREATED
STATUS              PORTS               NAMES
47ba2886f6f4        nginx:latest        "nginx -g 'daemon off"   7 minutes
ago       Created                                  admiring_visvesvaraya
$ wget \
>     --no-check-certificate \
>     --certificate=$DOCKER_CERT_PATH/cert.pem \
>     --private-key=$DOCKER_CERT_PATH/key.pem \
>     --method=DELETE \
> https://$(docker-machine ip default):2376/containers/47ba2886f6f4 -O - -q
$ docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED
STATUS              PORTS               NAMES
```

1.6.4 Listando contêineres e imagens

De maneira similar, podemos também listar contêineres e imagens disponíveis em nosso host:

```
$ wget \
>     --no-check-certificate \
>     --certificate=$DOCKER_CERT_PATH/cert.pem \
>     --private-key=$DOCKER_CERT_PATH/key.pem \
>     --method=GET \
> https://$(docker-machine ip defaul):2376/containers/json -O - -q
[]
$ wget \
>     --no-check-certificate \
>     --certificate=$DOCKER_CERT_PATH/cert.pem \
>     --private-key=$DOCKER_CERT_PATH/key.pem \
>     --method=GET \
> https://$(docker-machine ip defaul):2376/images/json -O - -q | python -m
json.tool
[
    {
        "Created": 1456916363,
        "Id":
"sha256:fd19524415dc0e3ea56cd8fe9400400d47aacef1d05c2f503638525794d30fec",
        "Labels": {},
        "ParentId": "",
        "RepoDigests": null,
        "RepoTags": [
            "nginx:latest"
        ],
        "Size": 134622725,
        "VirtualSize": 134622725
    }
]
```

1.6.5 Documentação

A documentação da API é pública e está disponível no site da Docker em *https://docs.docker.com/engine/reference/api/docker_remote_api/*. Neste endereço é possível ver as diferenças entre cada versão de API e também a documentação detalhada de todas as versões disponíveis.

1.7 Ferramentas de orquestração

Algumas ferramentas lançadas pela Docker para facilitar a disponibilização e o controle de aplicações são o Compose, Machine e o Swarm. Abordaremos cada um deles em capítulos separados.

1.8 Produtos

Como vimos durante a introdução, o Docker foi criado para dar suporte internamente ao dotCloud, que vendia serviços em nuvem. Com o sucesso do Docker provavelmente o foco mudou, pois a dotCloud encerrou suas operações no dia 29 de fevereiro de 2016 e vem anunciando uma série de novas plataformas e novos serviços, alguns deles pagos. Estes serviços se baseiam na combinação do uso das ferramentas já lançadas.

1.8.1 Docker Hub

Com o principal objetivo de ser um registry público (um repositório de imagens) o Docker Hub vem evoluindo. Inicialmente escrito em Python e posteriormente reescrito em Go, o serviço passou a contar com imagens de aplicações, como Nginx, MySQL e Redis, e de distribuições Linux, como Debian, Fedora e Ubuntu, assinadas oficialmente. Também passou a contar com a construção automática de imagens – basta vincularmos um repositório com os arquivos necessários para a construção das imagens no GitHub ou no Bitbucket. Para quem não conhece, o GitHub e o Bitbucket são serviços de controle de versão de códigos-fonte muito utilizados para hospedar projetos de código aberto.

O Docker Hub disponibiliza uma série de planos em que o básico, que é gratuito, permite hospedar uma imagem privada e quantas imagens públicas desejar. Nos demais planos a quantidade de imagens privadas varia.

1.8.2 Docker Datacenter

Este serviço é pago e tem o objetivo de auxiliar empresas que queiram migrar suas aplicações para o padrão Docker. Podemos ver na figura 1.6 que o serviço engloba o Docker Trusted Registry, uma espécie de Docker

Hub privado, e o Docker Universal Control Plane, um painel que gerencia visualmente a criação e o monitoramento de contêineres. Ambas as ferramentas têm integração entre si e o Docker Content Trust como base que implementa uma camada de segurança, onde podemos definir políticas de criação e alteração de imagens e contêineres por times e criação de usuários e times; também suporta o vínculo com serviços externos como LDAP e ActiveDirectory.

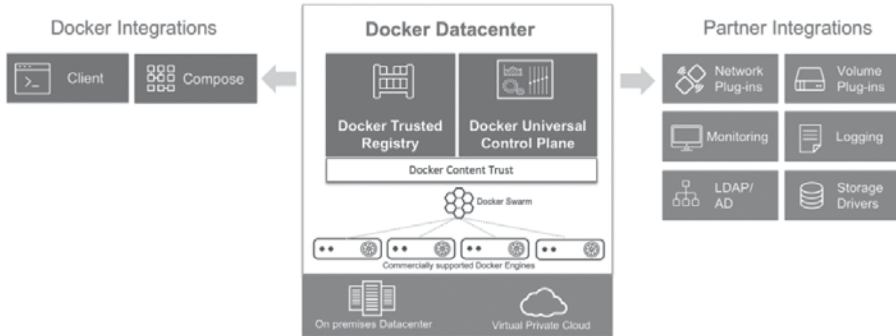

Figura 1.6 – Arquitetura no Docker Datacenter (fonte: www.docker.com).

O Docker Datacenter pode ser montado em máquinas físicas, no modelo chamado "on premises", ou em serviços de nuvem privada (VPC – Virtual Private Cloud).

1.8.3 Docker Cloud

O Docker Cloud também é pago e é similar ao Docker Datacenter. É um serviço de gerência de imagens, gerência de contêineres e de monitoramento, só que todo baseado em serviços de nuvem. Foi inspirado no Tutum, que gerencia, por meio de um painel muito bem montado, serviços rodando em contêineres, desde que estes estejam em algum provedor de nuvem suportado por eles. O Tutum também foi adquirido pelo Docker em outubro de 2015.

CAPÍTULO 2

Como instalar

A Docker suporta instalações no Mac OS X, Microsoft Windows e diversas distribuições de Linux, desde que o servidor suporte arquitetura de 64 bits. Para arquiteturas de 32 bits é necessário baixar as dependências, o código-fonte do GitHub e compilar manualmente. Reforçamos que é um processo não suportado pela Docker e pode trazer problemas.

Como vimos durante o capítulo de introdução, a arquitetura instalada é composta pelo Docker Engine, daemon que roda o serviço do docker, pelo Docker client, para podermos rodar comandos sobre o Docker Engine, e no caso de Microsoft Windows e Mac OS X um Docker host, uma máquina virtual Linux leve e pronta para rodar o Docker Engine.

Como a Docker está sempre liberando atualizações e com uma frequência relativamente alta, em média um release a cada dois meses, pode ser que as instalações que abordaremos tenham sido alteradas depois da publicação do livro.

Por exemplo, quando comecei a escrever este livro, a instalação-padrão indicada para Mac OS X era via Boot2docker, atualmente é via Docker Toolbox. Portanto, sempre consulte o guia oficial de instalação em *https://docs.docker.com/installation/*.

Vamos à instalação nos principais sistemas operacionais em uso atualmente.

2.1 Instalação no Arch Linux

2.1.1 Pré-requisitos para instalação no Arch Linux

Para o Arch Linux, os pacotes bridge-utils, device-mapper, iproute2, lxc e sqlite devem estar instalados previamente.

2.1.2 Como instalar no Arch Linux

Caso estejamos utilizando pacman como gerenciador de pacotes, basta rodar o seguinte comando:

```
$ sudo pacman -S docker
```

Se estivermos utilizando o AUR como gerenciador de pacotes, então rodaremos:

```
$ sudo yaourt -S docker-git
```

2.1.3 Rodando Docker no Arch Linux

Para rodar o daemon (Docker Engine), basta executar o serviço:

```
$ sudo systemctl start docker
```

2.2 Instalação no CentOS

2.2.1 Pré-requisitos para instalação no CentOS

Apesar de Linux kernel suportar contêineres desde a versão 2.6.26, o Docker precisa que no CentOS esteja rodando kernel 3.10 ou superior. Podemos testar com o comando `uname -r` para saber a versão do kernel em nosso sistema.

2.2.2 Como instalar no CentOS

Primeiro assumimos o papel de root:

```
$ sudo -i
```

Em seguida, rodamos o script de autoinstalação que instala o repositório no sistema e baixa o pacote correto de acordo com a distribuição de Linux utilizada.

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

2.2.3 Rodando Docker no CentOS

Para rodar o daemon (Docker Engine), basta executar o serviço:

```
$ sudo service docker start

$ sudo service docker stop
```

2.3 Instalação no Debian

2.3.1 Pré-requisitos para instalação no Debian

Já vimos que, apesar de o Linux kernel suportar contêineres desde a versão 2.6.26, no Debian precisamos de um kernel 3.10 ou superior. Podemos testar com o comando `uname -r` para saber a versão do nosso kernel. As versões atuais Debian 7 (Wheeze) e Debian 8 (Jessie) já atendem esse requisito.

2.3.2 Como instalar no Debian Jessie

No Jessie, para instalar basta executar os comandos:

```
$ sudo apt-get update

$ sudo apt-get install docker.io
```

2.3.3 Como instalar no Debian Wheezy

No Wheezy, para instalar basta executar os comandos:

```
$ echo 'deb http://http.debian.net/debian wheezy-backports main' | sudo
tee -a /etc/apt/sources.list.d/docker

$ sudo apt-get update

$ sudo apt-get install -t wheezy-backports linux-image-amd64
```

Enquanto o Jessie já tem nos sources lists (repositório de pacotes) os pacotes para instalação do Docker, no Wheezy temos que instalar o source list chamado backports.

2.3.4 Rodando Docker no Debian

Para rodar o daemon (Docker Engine), basta executar o serviço:

```
$ sudo service docker start

$ sudo service docker stop
```

2.4 Instalação no Fedora

2.4.1 Pré-requisitos para instalação no Fedora

Novamente reforçando que apesar de o Linux kernel suportar contêineres desde a versão 2.6.26, no Fedora deve estar rodando o kernel 3.10 ou superior. Podemos testar com o comando `uname -r` para saber a versão do nosso kernel. As versões 20, 21 e 22 do Fedora já atendem este requisito.

2.4.2 Como instalar no Fedora

Para instalar, primeiramente assumimos o papel de root:

```
$ sudo -i
```

Em seguida rodamos o script de autoinstalação, que instala o repositório no sistema e baixa o pacote correto de acordo com a distribuição de Linux utilizada.

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

2.4.3 Rodando Docker no Fedora

Para rodar o daemon (Docker Engine), basta executar o serviço:

```
$ sudo service docker start

$ sudo service docker stop
```

2.5 Instalação no Ubuntu

2.5.1 Pré-requisitos para instalação no Ubuntu

No Ubuntu Precise 12.04 (LTS) devemos ter o kernel 3.13 ou superior para instalar. Para as demais versões até o Ubuntu Vivid 15.04 (atualmente o último), também devemos instalar o pacote curl para rodar o script de autoinstalação.

2.5.2 Como instalar no Ubuntu

Para instalar, primeiramente assumimos o papel de root:

```
$ sudo -i
```

Rodamos o script de autoinstalação, que instala o repositório no sistema e baixa o pacote correto de acordo com a distribuição de Linux utilizada.

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

2.5.3 Rodando Docker no Ubuntu

Para rodar o daemon (Docker Engine), basta executar o serviço:

```
$ sudo service docker start
```

```
$ sudo service docker stop
```

2.6 Dica para usar Docker no Linux

Em todos as distribuições Linux recomendamos (apenas em ambiente de desenvolvimento) adicionar seu usuário ao grupo do Docker para que não seja obrigatório executar o comando `sudo` em todos os comandos docker.

```
$ sudo gpasswd -a ${USER} docker
```

2.7 Instalação no Mac OS X

Quando comecei escrever este livro, a instalação-padrão no Mac OS X e no Microsoft Windows era via Boot2docker. Era instalado um script que

permitia iniciar, parar, restaurar, remover uma máquina virtual rodando a distribuição Linux Boot2docker *http://boot2docker.io/*. Essa é uma distribuição leve e baseada no Tiny Core Linux.

Atualmente a instalação-padrão é via Docker Toolbox que já traz um pacote de ferramentas, dentre elas o Docker Machine, onde é possível criar, iniciar, parar, restaurar mais de uma máquina virtual rodando boot2docker. Na verdade o Docker Machine faz muito mais que isso, o que veremos posteriormente no capítulo 13.

2.7.1 Pré-requisitos para instalação no Mac OS X

Como o Docker só roda sobre Linux, é necessário que seja instalado um hypervisor. O mais recomendado e padrão para ambientes locais é o VirtualBox, mas também é possível trabalhar com o VMware.

2.7.2 Como instalar no Mac OS X

Devemos baixar o último pacote em *https://www.docker.com/toolbox* e rodar o instalador como na figura 2.1.

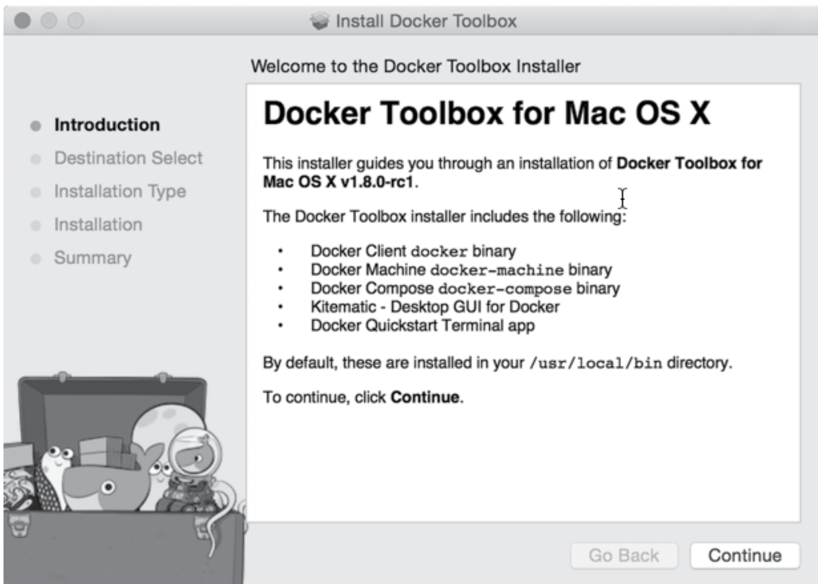

Figura 2.1 – Instalação do Toolbox no Mac OS X.

Depois basta seguir as instruções na tela de instalação, como na figura 2.2.

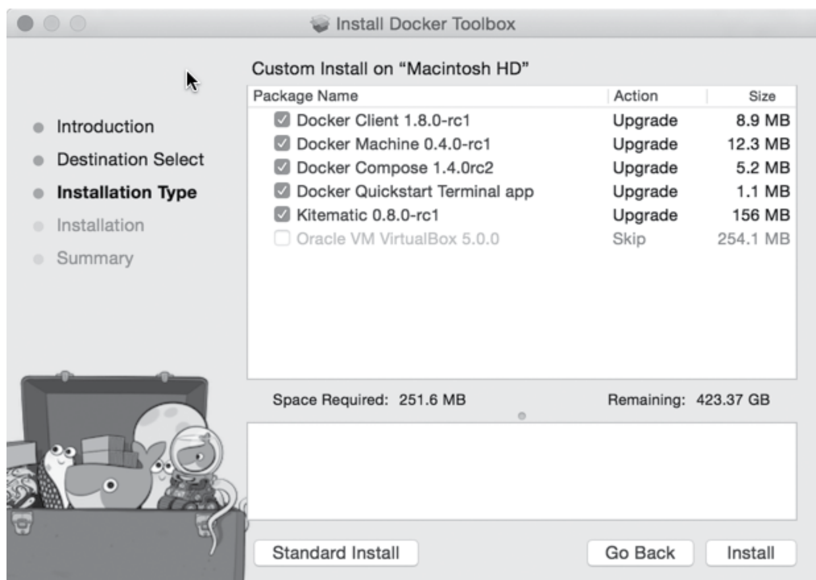

Figura 2.2 – Instalação do Toolbox no Mac OS X.

Ao final da instalação, veremos a tela de conclusão, como na figura 2.3.

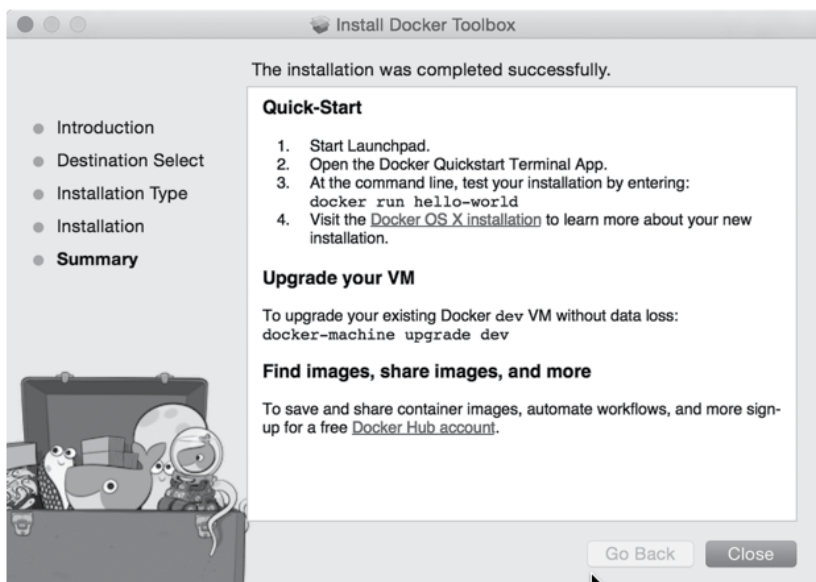

Figura 2.3 – Instalação do Toolbox no Mac OS X.

2.7.3 Rodando Docker no Mac OS X

No capítulo 13, focado no Docker Machine, vamos abordar como criar, iniciar, remover uma máquina virtual com suporte ao Docker e como definir as variáveis de ambiente para que nosso Docker client consiga executar comandos no daemon (Docker Engine).

2.8 Instalação no Microsoft Windows

Assim como no Mac OS X, uma máquina virtual rodando Linux funcionará como o nosso Docker host.

2.8.1 Pré-requisitos para instalação no Windows

Também é necessário que seja instalado um hypervisor VirtualBox (preferencialmente) ou VMware.

2.8.2 Como instalar no Windows

Devemos baixar o último pacote em *https://www.docker.com/toolbox* e rodar o instalador, como na figura 2.4.

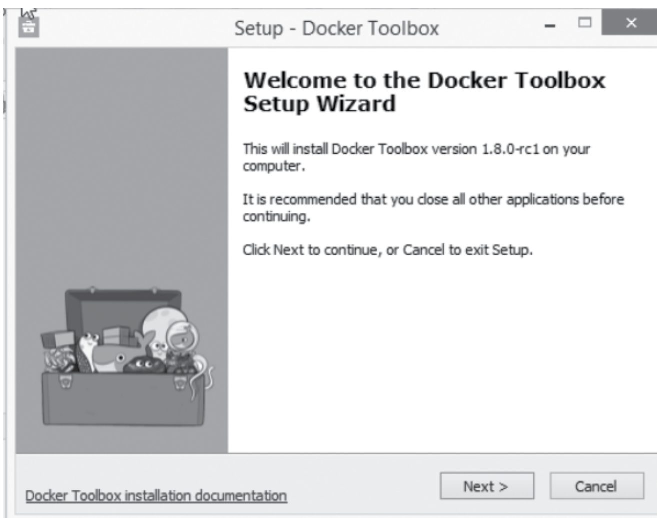

Figura 2.4 – Instalação do Toolbox no Windows.

Depois, seguir as instruções na tela, como na figura 2.5.

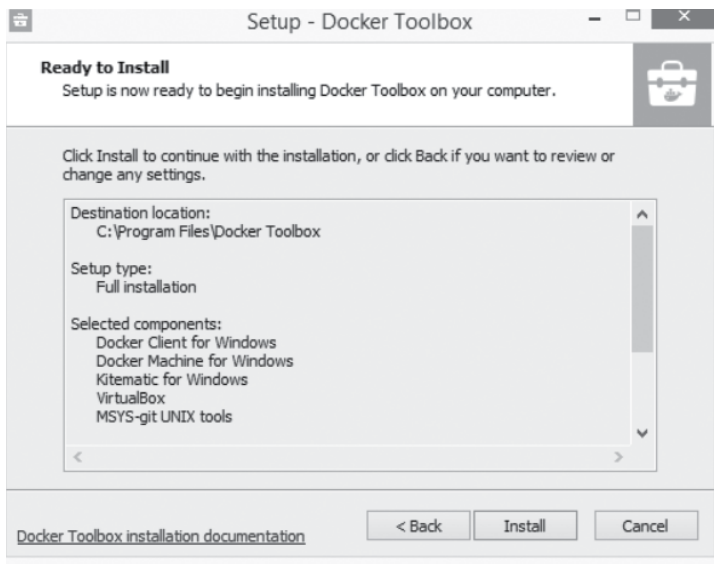

Figura 2.5 – Instalação do Toolbox no Windows.

Ao final da instalação, veremos a tela de conclusão, como na figura 2.6.

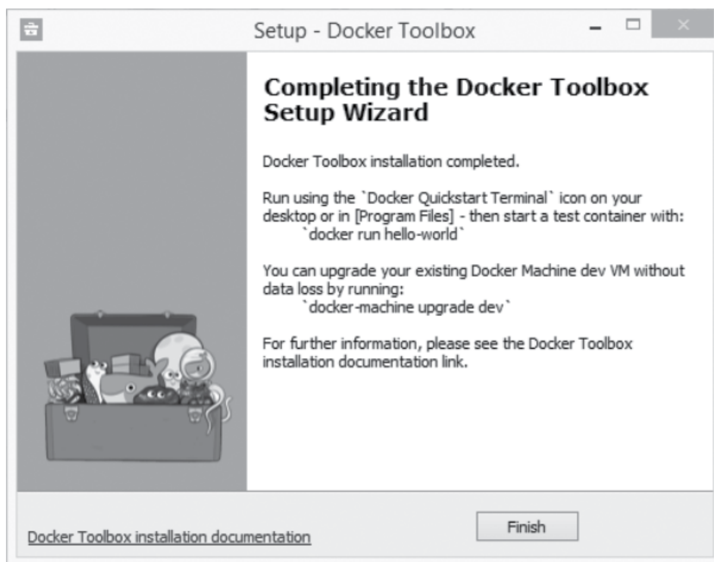

Figura 2.6 – Instalação do Toolbox no Windows.

Os comandos de Docker precisam que o *ssh.exe* estejam na variável de ambiente PATH. No prompt de comando basta executarmos o comando a seguir para adicionar.

```
set PATH=%PATH%;"c:\Program Files (x86)\Git\bin"
```

2.8.3 Rodando Docker no Windows

Como mencionado anteriormente no capítulo 13, falaremos sobre o Docker Machine e abordaremos como criar, iniciar, remover uma máquina virtual com suporte ao Docker e como definir as variáveis de ambiente para que nosso Docker client consiga executar comandos no daemon (Docker Engine).