

Servidores

Conceitos básicos e complexos de manutenção e reparo de servidores.

- Introdução
 - O que é um servidor?
 - Onde consigo um servidor?
 - O que é a cloud?
- Setup inicial na cloud
 - DigitalOcean
- Primeiras intuições
 - O que servidores são capazes de fazer?
 - Criando uma página web no servidor
- Containerização
 - Introdução
 - Docker I: um prólogo prático
 - Docker II: Familiarização
 - Docker III: A Dockerfile
- Self-hosted
 - Como ter sua própria VPN com OpenVPN e Docker
- Teoria

Introdução

Introdução ao conceito de ter um servidor.

O que é um servidor?

Antes de entender como manter um servidor, é preciso se perguntar: O que é um servidor?

De acordo com a [wikipedia](#), um servidor "*é um software ou computador, com sistema de computação centralizada que fornece serviços a uma rede de computadores, chamada de cliente.*"

. Ou, de uma forma mais extensa:

A internet funciona na base de **protocolos**: padronizações de como enviar e receber dados entre computadores. Alguns protocolos famosos são:

- IP (Internet Protocol)
- TCP (Transmission Control Protocol)
- HTTP (HyperText Transfer Protocol)
- SMTP (Simple Mail Transfer Protocol)
- DNS (Domain Name System)

Entre muitos outros.

Todos os protocolos têm computadores como origem e destino. Geralmente, são computadores diferentes.

Logicamente, os dois computadores tem que estar acessíveis (conectados a rede) e disponíveis (ligados, prontos para receber mensagens) quando a comunicação for feita.

Ou seja: Se você quiser mandar uma mensagem de texto diretamente para um computador de um amigo, **o computador dele precisa estar ligado e conectado na internet**. Não parece a melhor ideia, parece?

(PS: protocolos que falam de cliente a cliente diretamente são chamados de protocolos peer-to-peer, ou p2p).

Logo, introduziu-se o conceito de servidores. Um servidor é um computador feito para estar conectado e disponível (quase) 100% do tempo. Assim, ao invés de computadores se

comunicarem diretamente (*cliente a cliente*), os computadores se comunicam com um servidor. Quem fica com a responsabilidade de manter a mensagem e ser consultado sobre mensagens novas é o servidor.

Nesse livro, você irá aprender a parte de *software* de um servidor. Desde quais sistemas operacionais escolher até a parte de automação de servidores em cloud.

Está pronto?

Onde consigo um servidor?

Para conseguir fazer um *setup* em um servidor, é preciso primeiro *ter* um servidor...

Felizmente, temos algumas opções na mesa:

- Servidores são computadores! Você pode fazer todos os tutoriais daqui em uma máquina virtual, ou mesmo na sua própria máquina (Não recomendado).
- Existem clouds que alugam servidores por um preço muito barato. Se você é estudante, todas as grandes clouds (como Amazon AWS, Google GCP, Microsoft Azure, DigitalOcean, entre outros) oferecem crédito grátis em suas nuvens. Se não é mais estudante, sem problema! Os preços são tão baratos quanto 5 dólares por mês.
- Caso aplicável, saiba também que o IMEsec tem um servidor físico no bloco A do IME.

Uma pergunta seguinte seria: *O que é a cloud?*

O que é a cloud?

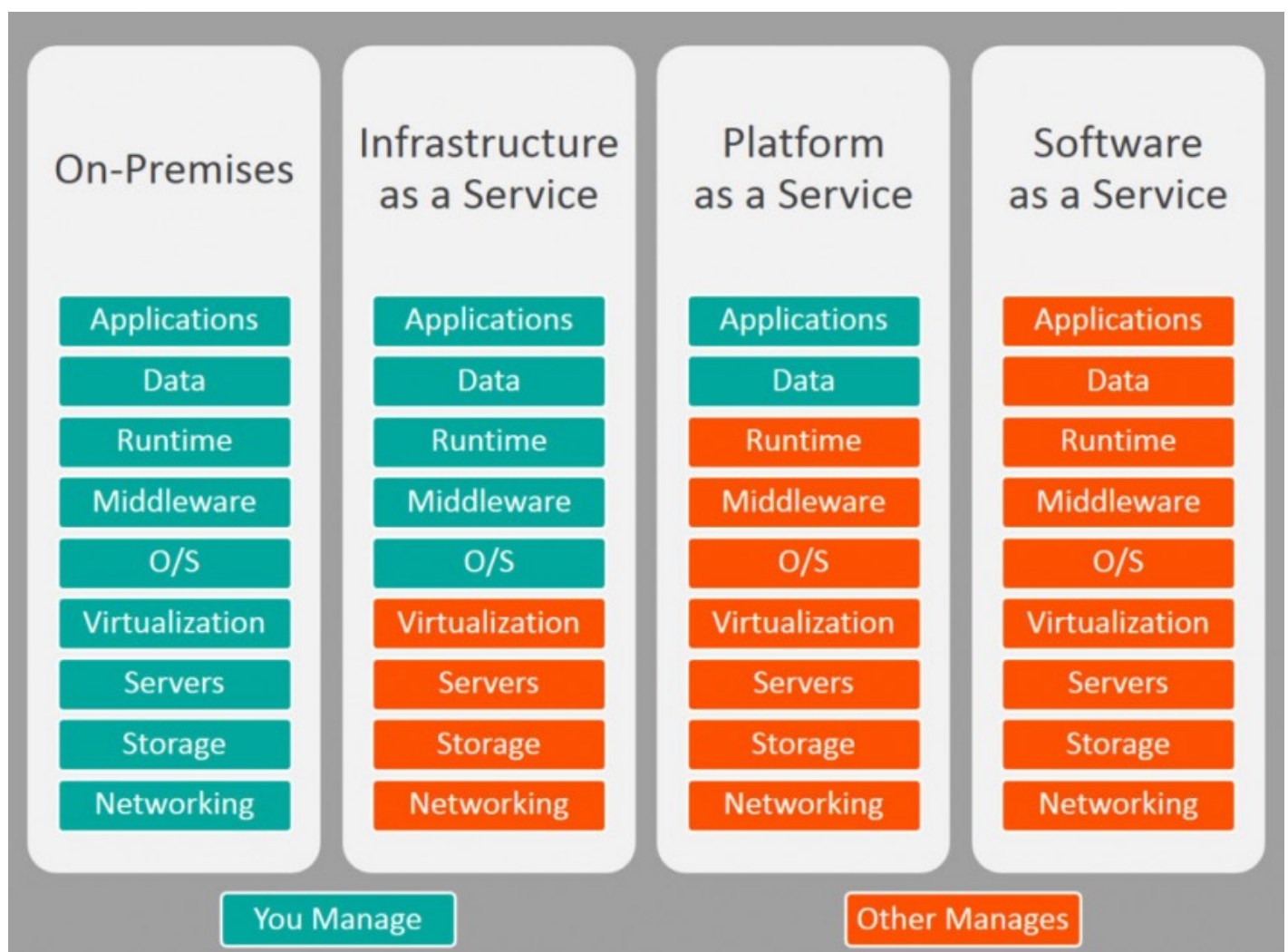


Clouds são apenas um arranjo de computadores prontos para serem alugados de várias formas. O jeito mais simples de se vender um serviço cloud é simplesmente oferecer o aluguel do servidor crú, apenas com um sistema operacional instalado, e deixar quem está alugando fazer todo o trabalho de manter seus serviços. Essa forma de alugar servidores é chamada de **laaS, ou Infrastructure as a Service**. É exatamente o que queremos!

Mas, para completude de conversa, clouds podem ser vendidas de outras maneiras: em vez de

oferecer o servidor crú, muitas clouds também oferecem um serviço de 'rodar a sua aplicação, independente de quantos servidores forem necessários'. Ou seja, você fornece o código, eles rodam o código. A parte de como rodar o código fica a cargo da própria cloud. Essa forma de alugar servidores é comumente chamada de **PaaS, ou Platform As A Service**. Outro nome comum para isso é **Serverless**, dado que para você não existem servidores, só existe código rodando.

Outra maneira que irei citar de como vender uma cloud é o jeito da *iCloud*: Ela não oferece servidores, muito menos roda o seu código nas máquinas da Apple. Em vez disso tudo, o que a *iCloud* faz é vender serviços que rodam nos servidores deles, como por exemplo guardar imagens, vídeos e dados miscelâneos dos seus dispositivos Apple. Essa forma de vender servidores é chamada de **SaaS, ou Software as a Service**, e é a que impõe a maior quantidade de abstrações em cima do servidor. Para todo efeito, não é necessário nem saber o que é um servidor ou código!



Setup inicial na cloud

Decidiu pegar um servidor em uma nuvem? OK!

DigitalOcean

A DigitalOcean é uma **cloud provider** muito simples de usar, e apenas com as *features* necessárias. É muito simples começar um servidor virtual por lá! Vamos lá, passo a passo.

Criando uma conta na DigitalOcean

Caso você seja estudante e não tenha uma conta na DO, recomendo fortemente que você crie utilizando o **GitHub student developer pack**, pois ele te dá **50 dólares** em créditos durante os primeiros 2 meses! Isso é suficiente para rodar 2 máquinas potentes por esses dois meses.

Configuração inicial

Antes de criar um servidor, é preciso fazer algumas configurações iniciais. Aqui estão as instruções para **bash** para sistemas operacionais Unix-like (OSX, GNU/Linux).

1. Os servidores que você irá criar na DigitalOcean são acessados através de **ssh keys**, não por usuário e senha. Para acessá-los, portanto, é necessário criar uma chave ssh. No terminal, utilize o comando

```
ssh-keygen
```

para gerar um par de chaves pública (`id_rsa.pub`) e privada (`id_rsa`).

As chaves são geradas em pares, de tal forma que apenas a chave privada consegue descriptografar o que a chave pública criptografa, e vice-versa. A chave pública será a que você coloca nos servidores da DigitalOcean, e a chave privada fica no seu próprio computador, para que você consiga se autenticar no servidor.

2. Após gerar as chaves, é preciso colocá-la na DigitalOcean.

As chaves encontram-se no diretório `~/.ssh` do seu computador.

Para exportá-las para a DigitalOcean, faça login e entre no dashboard, na página de `security`:

PROJECTS

MANAGE

DISCOVER

ACCOUNT

Profile

Billing

Security

Referrals

Search by Droplet name or IP (Ctrl+B)

Create

SSH keys

Add SSH Key

Name	Fingerprint	
Rede linux	4c:bcc:b:c0:28:8b:5f:ad:9b:c2:d9:d4:d3:c5:c7:47	More
Mobiusv2	94:34:a2:67:d6:95:2b:fc:01:33:3d:29:d2:80:be:14	More

Certificates

Add Certificate

Secure your data in transit by establishing TLS/SSL connections. Add your own certificate manually or generate a free one using [our integration with Let's Encrypt](#).

To use the Let's Encrypt feature, your domain must be managed on DigitalOcean. [Learn how to manage your DNS on DigitalOcean](#)

clique em `Add SSH Key`.

×

New SSH key

Paste a copy of your **public key** in the space below. It should end in `.pub`. [Learn more](#). This does not add an SSH key to your existing Droplets. To do so, [follow the instructions here](#).

SSH key content must be a valid SSH key *

SSH key content

Name *

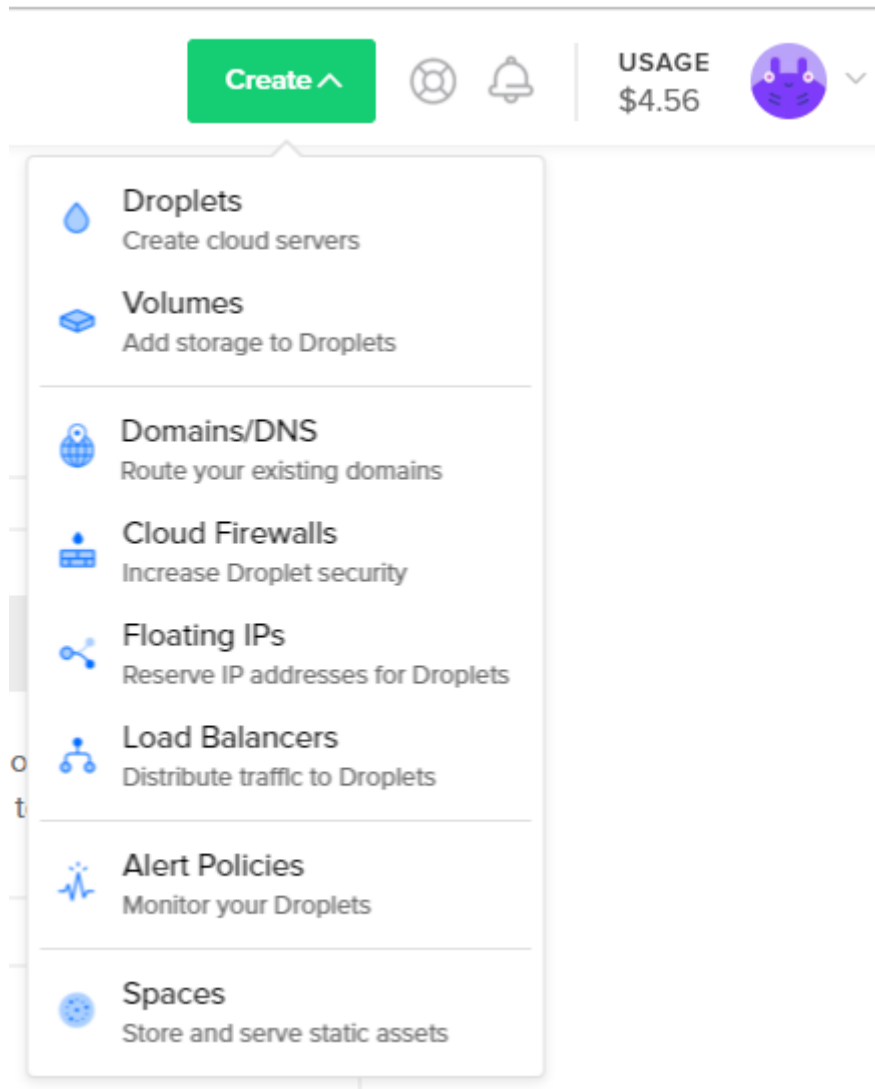
Add SSH Key

Cole os conteúdos de `id_rsa.pub` na caixa acima, e dê um nome amigável para lembrar a qual computador essa chave SSH pertence.

Agora, é possível criar servidores com essa chave SSH.

Criando um servidor virtual na nuvem

Para alugar um servidor na DigitalOcean, clique no botão **Create**, na barra do topo do site.

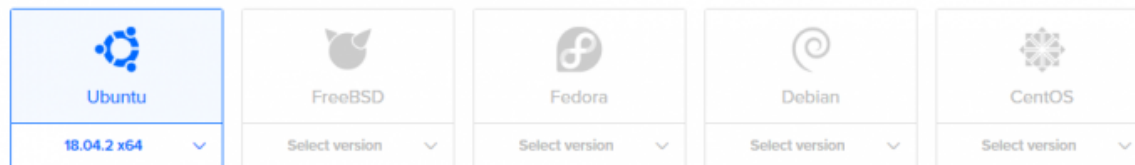


Como queremos criar um servidor, clique em [Droplets](#). Ele irá abrir essa página:

Create Droplets

Choose an image ?

[Distributions](#) [Container distributions](#) [Marketplace](#) [Custom images](#)



Choose a plan



Standard virtual machines with a mix of memory and compute resources. Best for small projects that can handle variable levels of CPU performance, like blogs, web apps and dev/test environments.

\$5/mo \$0.007/hour	\$10/mo \$0.015/hour	\$15/mo \$0.022/hour	\$15/mo \$0.022/hour	\$15/mo \$0.022/hour	\$20/mo \$0.030/hour
1 GB / 1 CPU 25 GB SSD disk 1000 GB transfer	2 GB / 1 CPU 50 GB SSD disk 2 TB transfer	3 GB / 1 CPU 60 GB SSD disk 3 TB transfer	2 GB / 2 CPUs 60 GB SSD disk 3 TB transfer	1 GB / 3 CPUs 60 GB SSD disk 3 TB transfer	4 GB / 2 CPUs 80 GB SSD disk 4 TB transfer

- Para o sistema operacional, escolha Ubuntu 18.04 (ou o mais novo que tiver).
- Para o plano, escolha o mais barato (\$5/mo). Ele vai ser suficiente para esse tutorial.
- Para a região do datacenter, qualquer uma delas funciona, mas escolha a mais perto de você.
- **Não esqueça de adicionar sua chave SSH no servidor!** Basta apenas clicar no nome da chave que você criou nos passos anteriores.

Add your SSH keys ?

New SSH Key

☐ Rede linux

☒ Mobiusv2

```
ChmJnVdE2nrxKcGpwKnlXZWS9pwpmb0nKlWw5WjY3CkKQ  
FKwlEvP3BxIMD/Ew4FE4uCBDh1nehSQxiZcCuOebNUL59oVI  
jkiJBiPSkApO6pX2RY3MT0He2YQ8mLtoQCBewGKM3VPZG  
ojigaK3MEysYb/InI9Ocrk5cLTMTTsa7I MOBIUSv2
```

- Crie apenas 1 droplet com essas configurações.

Após clicar em 'Criar', você vai ver uma barra de progressão. Quando ela acabar, você terá o **Endereço de IP do servidor**. Este já tem **sshd configurado**, ou seja, é possível conectar nele

com SSH, o comando de terminal (falaremos mais dele depois).

Para se conectar no seu novo servidor, utilize o comando

```
ssh root@ip.do.servidor.aqui
```

Por exemplo, se o IP do servidor é 100.101.102.103, então o comando ficaria:

```
ssh root@100.101.102.103
```

Por ultimo, note que **para esse comando executar com sucesso, é necessário que sua chave privada esteja na pasta `.ssh`**. Se você escolheu uma pasta diferente para esse arquivo, é necessário explicitá-la na chamada ssh:

```
ssh -i ~/caminho/ate/o/arquivo.txt root@100.101.102.103
```

Conseguiu conectar no servidor? Ótimo! Vamos para o que interessa agora.

Primeiras intuições

Antes de pormos código de verdade rodando nos servidores, vamos brincar um pouco com suas capacidades!

O que servidores são capazes de fazer?

TL;DR: Tudo o que um computador consegue fazer. Algumas coisas são muito mais fáceis de fazer neles, inclusive.

Se você chegou até aqui no livro, deve estar com um servidor pronto na mão, mas ainda se perguntando sobre o que servidores são capazes de fazer.

Citarei aqui, portanto, algumas capacidades muito legais que esses computadores conseguem fazer.

1. Servidores que tem um **IP público** são (de uma maneira simples) acessíveis por todos os computadores conectados na internet. Isso possibilita a criação de:
 - Páginas web acessíveis por todos
 - Servidores de arquivos
 - jogos multi-jogador nos quais os clientes se conectam a um servidor específico
2. A habilidade de **distribuir computação** pode ser extremamente útil. Há problemas que um computador sozinho não consegue lidar em tempo hábil, mas milhares de computadores tornam o problema factível.
3. Pode ser que **centralizar informação** seja importante na sua aplicação. Uma única fonte de verdade pode ser importante -- imagina se uma LAN House precisasse gerenciar logins e senhas dos computadores dela individualmente?
4. **Offload de computação** é importante também. Em uma aplicação complexa, pode ser que seja necessário realizar muitos cálculos, e utilizar bastante processamento. Em alguns casos, não é uma boa prática deixar o cliente fazer essa computação, pois este pode estar em um celular ou dispositivo baseado em bateria. Por que não realizar a computação em um servidor e devolver o resultado para o cliente?

Espero que essa lista tenha ajudado a notar alguns usos de um servidor. Vamos realizar alguns exemplos nas próximas páginas.

Criando uma página web no servidor

Primeiro de tudo, vamos criar uma página web muito simples no servidor, para podermos acessar no próprio navegador!

Considerando que você já realizou os passos anteriores para setup de um servidor e agora tem uma máquina com IP público ao seu dispor, realize os seguintes comandos:

```
# 1. Vá para a sua pasta "home".
cd ~

# 2. Crie uma pasta. Nomeie-a como quiser.
mkdir meu-website

# 3. Entre nessa pasta.
cd meu-website

# 4. Crie um arquivo chamado "index.html", com um HTML bem simples dentro.
echo "<h1>Bem vindo ao meu website! </h1>" > index.html

# 5. Comece um servidor super-simples de python.

# Essa parte requer python3. Caso o servidor não tenha python3 instalado,
# instale como se fosse qualquer outro computador linux:

# em ubuntu/debian, use
# apt install python3

# em redhat/fedora, use
# yum install python3

python3 -m http.server 80
```

Os comandos acima deverão começar um processo escutando na porta 80 (o padrão do HTTP). Utilize `Ctrl+C` para parar o processo. Mas antes, abra o seu navegador favorito e coloque o IP público do seu servidor na barra de endereço.

O resultado deveria ser algo parecido com

Bem vindo ao meu website!

Yay! Tudo funcionando corretamente. Essa é uma forma rápida e suja de servir páginas web, mas não é recomendada como uma solução definitiva. Vamos mostrar como fazer uma solução mais robusta daqui a pouco.

Agora, vamos verificar os processos atados a portas que estão rodando no nosso servidor. Esperamos dois processos: o processo de SSH (que é como conseguimos acessar o nosso servidor remotamente), e o processo de HTTP (o processo em python que começamos agora). Explicarei sobre portas nos próximos capítulos.

Para isso, pare o processo em python com `Ctrl+C` e rode-o em plano de fundo:

```
# Note que estamos ignorando a saída do programa
python -m http.server 80 >/dev/null &
```

Agora, vamos utilizar o comando `netstat` para verificar os processos. Utilize o comando

```
netstat -tln
```

O output do programa deve parecer com isso:

```
root@ubuntu-tutorial:~# netstat -tln
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State       PID/Program name
tcp        0      0 0.0.0.0:80              0.0.0.0:*               LISTEN      1208/python3
tcp        0      0 127.0.0.53:53           0.0.0.0:*               LISTEN      593/systemd-resolve
tcp        0      0 0.0.0.0:22              0.0.0.0:*               LISTEN      947/sshd
tcp6       0      0 :::22                   :::*                    LISTEN      947/sshd
root@ubuntu-tutorial:~#
```

Quase exatamente o que previmos. No caso, são 3 processos escutando em 4 portas. A primeira linha,

tcp	0	0	0.0.0.0:80	0.0.0.0:*	LISTEN	1208/python3
-----	---	---	------------	-----------	--------	--------------

é o processo de python que começamos nos passos anteriores. O número do processo é o logo antes do nome do executável: `1208`. para matar esse processo, digite

```
kill 1208
```

A segunda linha é um processo interno da DigitalOcean e pode ser ignorado por enquanto.

As duas últimas linhas são do SSH.

O SSH está escutando tanto em IPv6 quanto IPv4 (mais sobre isso depois).

Limpando o servidor deste exercício

Para eliminar qualquer resíduo desse exercício, primeiro mate o processo em python que foi iniciado no começo dessa página. No meu caso, o número do processo é `1208`, mas isso sempre varia. Utilize o comando `netstat -tln` para descobrir o número.

Por fim, é preciso apagar a pasta que criamos no exercício.

```
cd ~  
rm -r meu-website
```

Containerização

Um dos grandes problemas de desenvolvimento de aplicações hoje em dia é o fato de que o seu computador local e o seu servidor são computadores diferentes, que funcionam de jeitos diferentes. A containerização de serviços tenta resolver esse problema.

Introdução

Digamos que você acabou de criar um novo app e quer deixar o mundo todo usar. Como é possível fazer isso?

Primeiramente, o app que você criou está rodando no seu próprio computador. Isso é legal, mas não é bom que as pessoas acessem o seu app *através* do seu computador. Imagina só? Você quer desligá-lo, mas não pode pois outras pessoas dependem dele.

Essa discussão já tivemos: precisamos de um servidor. Entretanto, isso gera outro problema, especialmente para *softwares* mais complexos: **O servidor é um computador diferente do seu.**

Ou seja, ele possivelmente está rodando outro *sistema operacional*, ou talvez em outra versão, talvez não tenha todos os programas instalados que o seu tem, ou em versões diferentes...

Isso pode dar uma grande dor de cabeça para instalar todos os requisitos que o seu app novo precisa. Isso também cria uma dependência no **servidor no qual você está rodando o app**, no sentido de 'se não for esse servidor, não funciona'.

Para isso, foi criado o conceito de **Containerização**.

O que é containerização?

Essencialmente, é o conceito de **isolar** uma aplicação das outras, de forma que tudo o que ela precise esteja dentro do *container*, **e nada mais**.

Além de um grande ganho na segurança do sistema (i.e. se alguém comprometer a segurança da aplicação, ela ganha acesso total ao container, mas não à máquina inteira), essa prática facilita muito a distribuição da aplicação em um ou mais servidores. Isso porque ao invés de distribuir a aplicação aos servidores, é possível distribuir **um container com a aplicação**, contendo todos os requisitos do software junto ao próprio software.

A diferença entre um container e uma VM

A primeira pergunta que eu me fiz quando li essa ideia foi "Peraí, qual a diferença entre um container e uma máquina virtual?" já que os conceitos são muito parecidos.

Para alguém que está usando o produto final cegamente, é completamente possível imaginar containeres de linux como micro-VMs, e trabalhar com isso.

Mas somos hackers. Queremos saber como as coisas funcionam. E antes de tudo, precisamos falar um pouco sobre como um sistema operacional funciona.

Todo sistema operacional tem um núcleo. O Windows tem o kernel NT, o OS X e o iOS usam o kernel Darwin, o GNU/Linux e o Android usam o Linux. (Kernel é a palavra inglesa de núcleo, usarei kernel e núcleo intermitentemente)

O kernel é responsável por fazer as partes mais essenciais de um sistema operacional, como por exemplo:

- O sistema de arquivos (ler e escrever em arquivos, e como transcrever isso para protocolos que um HD ou SSD entende);
- Lidar com todas as redes do computador (conversar com a placa de Wi-Fi, descobrir para qual IP é preciso mandar o pacote);
- Administração de processos (multi-threading, escalonamento de processos)

A parte mais superficial do sistema operacional lida com a interface do usuário para esse núcleo. Em outras palavras, transcrever cliques em um arquivo ou toques em uma tela para operações nucleares.

Dito isso, **O que é uma VM?**

Uma VM tem como sua parte principal uma camada que traduz instruções entre um núcleo de

sistema operacional e outro. Essa camada é chamada de **Hypervisor**, várias outras responsabilidades como salvar o estado da máquina virtual em um arquivo, mas a responsabilidade principal é **transcrever comandos dados por um programa dentro da máquina virtual para comandos no sistema operacional externo**, de maneira segura.

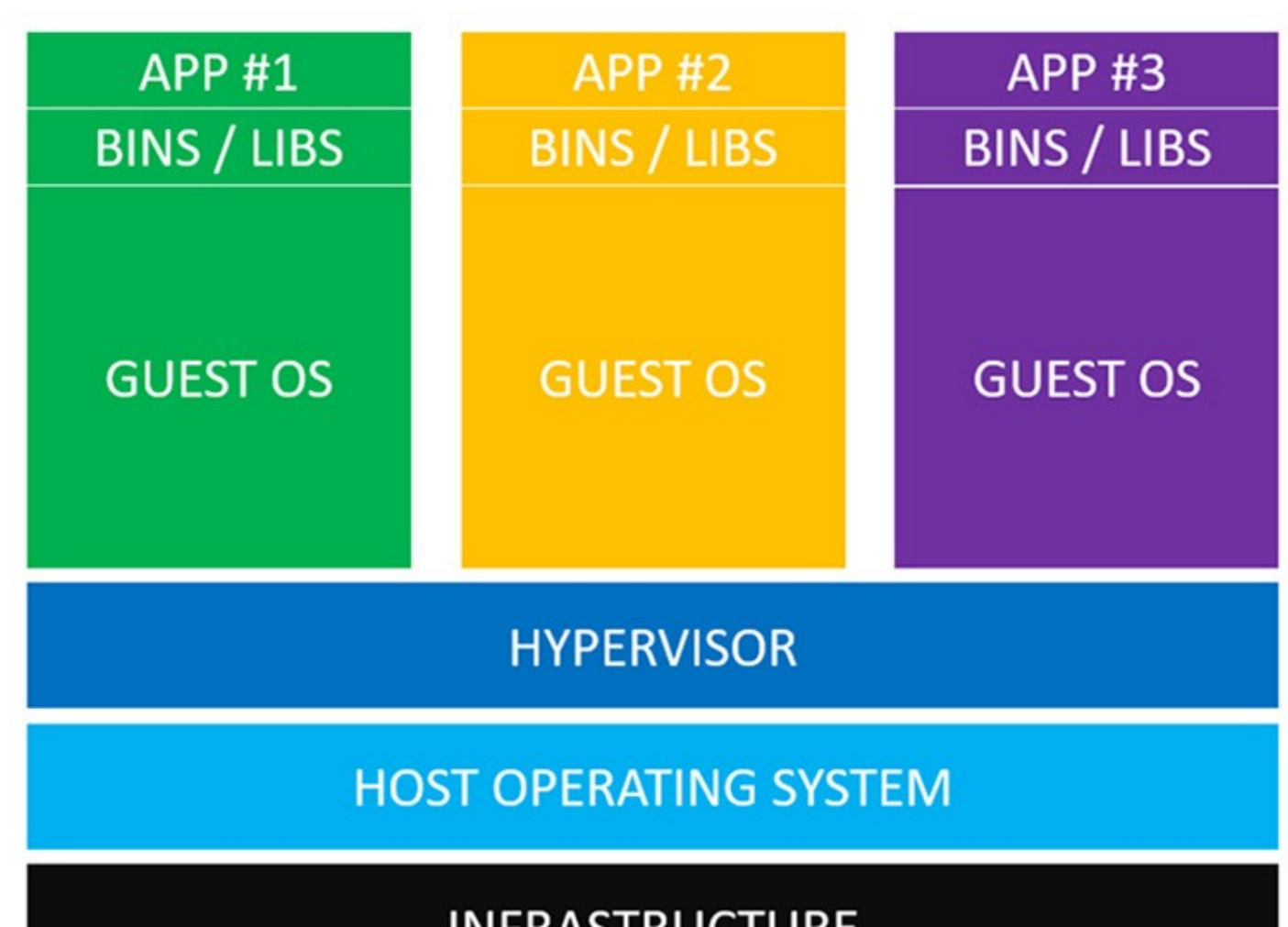
Isso implica que para n máquinas virtuais rodando num computador, há $n + 1$ núcleos de sistema operacional rodando, junto a no mínimo 1 Hypervisor para transcrever instruções das máquinas virtuais para a máquina real.

Containeres fazem isso um pouco diferente. Ao invés de executar vários núcleos de sistema operacional e transcrever instruções entre eles, um container executa instruções utilizando **o mesmo kernel** que o da máquina física.

Ou seja, para n containeres rodando na mesma infraestrutura, há apenas 1 núcleo de sistema operacional compartilhado entre os containeres. E mesmo assim, o administrador de containeres (os quais vamos falar nesse capítulo, que incluem **dockerd**, **LXC**, **podman**, **rkt**, entre outros) dá um jeito de criar uma experiência isolada entre os containeres.

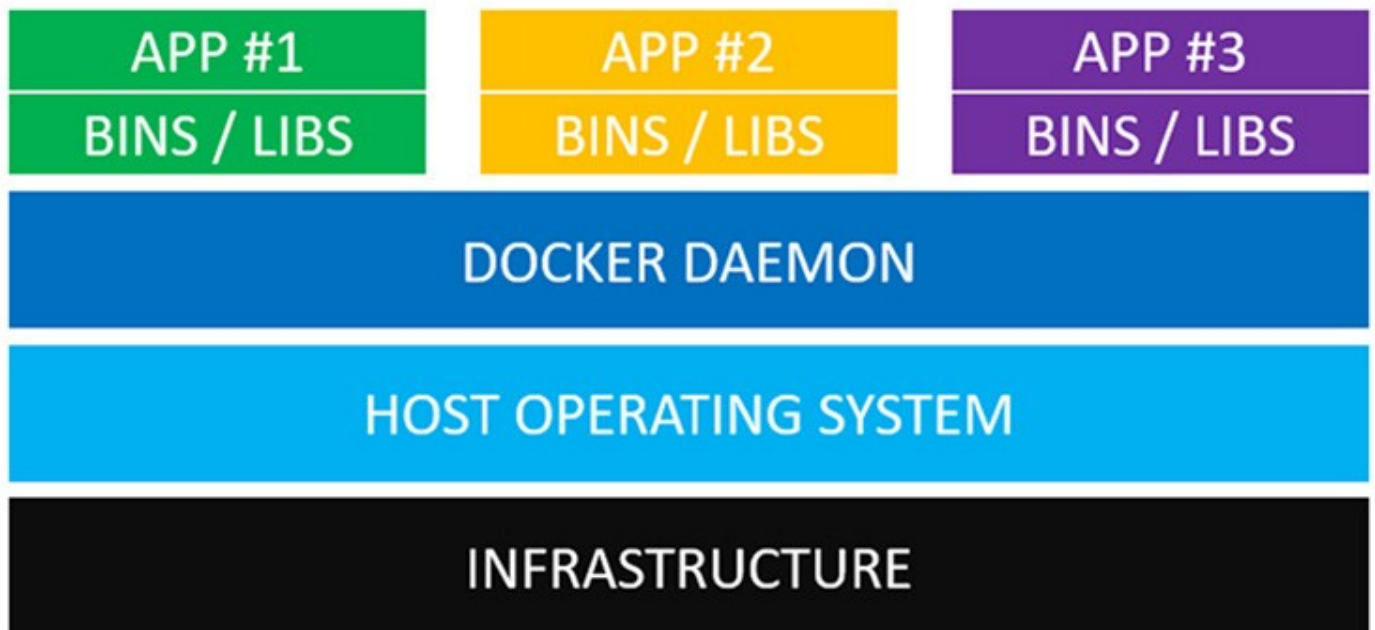
Para visualizar isso melhor, peguei os gráficos desse artigo.

Arquitetura de VMs



Note que há vários "Guest OS" sendo transcritos pelo Hypervisor para o sistema operacional da máquina física.

Arquitetura de Containeres (Docker, no caso)



Em contraste com a arquitetura de VMs, não há mais do que um sistema operacional rodando na infraestrutura. O daemon do Docker (`dockerd`) tem a responsabilidade de isolar as chamadas de sistema dos containeres, para que um não interfira na vida do outro.

Em termos práticos, Máquinas virtuais possibilitam rodar **núcleos** diferentes, o que possibilita coisas como Windows dentro de Linux (e vice-versa). Isso vem a um custo grande: máquinas virtuais são mais pesadas do que containeres, tanto em consumo de memória (mantendo dois sistemas operacionais em memória) quanto armazenamento (Uma VM básica pode ter uns 700MB de tamanho).

Um container possibilita criar ambientes isolados que usam o mesmo núcleo de SO do que a máquina anfitriã. Para rodar containeres de Docker em um Windows, por exemplo, o Docker executa secretamente uma VM de linux na qual o `dockerd` executa dentro. Com isso, os containeres são muito mais leves em termos de RAM, armazenamento e tempo de *startup* do que uma VM.

No nosso exemplo acima de construir um app, o ideal é construir um container linux que rodará numa máquina física GNU/linux. Nesse capítulo, vamos ver como.

Docker I: um prólogo prático

Docker I: um prólogo prático

Convencido de que containerização é útil? Então vamos containerizar uma aplicação bobinha para entendermos como isso é feito na prática.

Nesse tutorial, irei criar um app superficial em node.js, e encapsulá-lo em um container de Docker, e logo em seguida executá-lo.

Parte I: Construindo nosso app bobo

primeiro, vamos instalar o que é necessário: `npm` e `docker`. Essas instruções são para instalar npm e docker em Debian/Ubuntu, confira como é feito no seu próprio sistema.

```
sudo apt update
sudo apt install npm docker
```

agora, crie uma nova pasta. vamos trabalhar nela o tempo todo.

```
mkdir docker_intro
cd docker_intro
```

agora comece um repositório de NPM (Node Packet Manager) dentro da pasta. O NPM será útil pra instalarmos pacotes de Javascript para o nosso programa.

```
npm init -y
```

Isso cria um arquivo chamado `package.json` com as seguintes informações dentro:

```
{
  "name": "docker_1",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
```

```
"scripts": {
  "test": "echo \"Error: no test specified\" && exit 1"
},
"keywords": [],
"author": "",
"license": "ISC"
}
```

Vamos alterar uma linha: trocar a linha `"test": "echo \"Error: no test specified\" && exit 1"` para `"start": "node index.js"`. Assim, quando rodarmos `npm start`, o NPM executará `node index.js` para nós.

O arquivo fica assim então:

```
{
  "name": "docker_1",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "start": "node index.js"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

Para esse exercício, vamos instalar o pacote `ramda`, apenas para provar que o container conseguirá instalar esse pacote também.

```
npm install ramda
```

Agora vamos criar o arquivo `index.js` e colocar o seguinte "Hello World" dentro dele:

```
const { applyTo } = require('ramda')

const withHelloWorld = applyTo('Hello, World!')

withHelloWorld(console.log)
```

Adendo: Se você quer entender o que está acontecendo nessas linhas, você pode pensar a função `applyTo` como

```
const applyTo = (values) => (func) => func(values)
```

Ou seja, uma função que recebe valores e devolve uma função que recebe uma função e devolve ela aplicada nos valores. *Confuso, eu sei.* Se quiser saber mais, procure saber um pouco sobre **programação funcional**. Essa apresentação parece ser uma boa introdução ao assunto.

Voltando aos trilhos. Ao rodar `npm start` no nosso console, se tudo ocorreu certo, deveríamos ver isso:

```
$ npm start

> docker_1@1.0.0 start /home/raz/infosec/wiki/docker_1
> node index.js

Hello, World!
```

Ótimo! Fora algumas instruções de debug, vemos o nosso `Hello, World!`.

Parte II: Containerizando o app

Temos nosso hello world funcionando. Agora vem a parte de colocá-lo dentro de um container. E você vai ver que é *super* fácil.

Crie um arquivo chamado `Dockerfile` na pasta do projeto. Esse arquivo é uma "receita" de como construir um container com nosso app. Coloque isso dentro do arquivo:

```
# Herda da imagem do debian
FROM debian
```

```
# Instala o npm
RUN apt update
RUN apt install -y npm
# Atualiza o npm
RUN npm install -g npm

# Cria o diretório /app
# e instala as dependências do /app lá
# note que WORKDIR tanto cria o diretório
# quanto troca para ele (cd /app)
WORKDIR /app
COPY package.json .
COPY package-lock.json .
RUN npm install

# copia o nosso app para dentro da imagem
COPY index.js .
# Esse é o comando que será rodado quando você executar o `docker run`
CMD ["npm", "start"]
```

Não precisa pensar muito sobre o que tem dentro desse arquivo ainda não. Já vamos falar sobre ele.

Agora, rode o seguinte comando (**OBS: Você talvez precise rodar os próximos comandos com `sudo`**):

```
docker build . -t meu-app
```

Leia esse comando como "Construa o container da pasta `.` e dê um nome de `meu-app` a ele". Esse comando vai demorar para terminar, e uma conexão boa com a internet ajuda bastante.

Com esse ultimo comando acabando com sucesso, é só rodar o container!

```
docker run meu-app
```

No meu caso, aconteceu o seguinte:

```
$ docker run meu-app
```

```
> docker_1@1.0.0 start /app  
> node index.js
```

```
Hello, World!
```

Perfeito! Igualzinho ao nosso da parte I, só que esse comando rodou dentro de um container isolado!

Na próxima página, vamos discutir o que na verdade aconteceu por baixo dos panos.

Docker II: Familiarização

Docker II: Familiarização

Na parte I construímos um container meio que *tirando da cartola*. Antes de entender o que houve, preciso explicar mais algumas coisas.

Containers e imagens

O Docker introduz esses dois conceitos para nós: containeres e imagens. Nós viemos falando sobre o que é um container, mas não falamos sobre o que é uma imagem.

Note que, mesmo você rodando o comando `docker run` da página passada múltiplas vezes, o resultado é o mesmo.

```
$ docker run meu-app

> docker_1@1.0.0 start /app
> node index.js

Hello, World!
$ docker run meu-app

> docker_1@1.0.0 start /app
> node index.js

Hello, World!
```

Hm, há 2 coisas que podem estar acontecendo aqui:

1. O Docker criou um container com o `docker build` e você está invocando esse mesmo container toda vez
2. O Docker criou um modelo de como criar containers e está criando um novo toda vez que você roda o comando

A magia do Docker é que o que está acontecendo é a opção 2. Para provar mais ainda esse ponto, vamos tentar mudar nosso `index.js` para ler e escrever em um arquivo.

```
const fs = require('fs')

// Coloca "olá!" no final do arquivo teste.txt.
// Se o container é o mesmo que roda entre vários `docker run`s,
// O esperado é ver vários "olá!" no arquivo.
fs.appendFileSync('teste.txt', 'olá!')

// Agora lemos o arquivo para ver o que tem.
const fileData = fs.readFileSync('teste.txt').toString('utf8')

console.log({ fileData })
```

Ao tentar rodar isso fora do docker, temos o que esperamos: Um arquivo que fica cada vez mais longo toda vez que rodarmos o programa.

```
$ npm start

> docker_1@1.0.0 start /home/raz/infosec/wiki/docker_1
> node index.js

{ fileData: 'olá!' }
$ npm start

> docker_1@1.0.0 start /home/raz/infosec/wiki/docker_1
> node index.js

{ fileData: 'olá! olá!' }
$ npm start

> docker_1@1.0.0 start /home/raz/infosec/wiki/docker_1
> node index.js

{ fileData: 'olá! olá! olá!' }
$ rm teste.txt
```

Agora, vamos rodar o `docker build`. Dessa vez vai demorar uma fração do tempo do que a outra,

já que grande parte da "receita" já está cacheada.

```
docker build . -t meu-app: teste-arquivo
docker run meu-app: teste-arquivo
```

Nota: vamos falar sobre esse `:` no nome da imagem daqui a pouco. Essencialmente, você pode dar nomes a diferentes versões do seu mesmo programa.

Rodando o `docker run` várias vezes, vemos que a opção 2 é verdade:

```
$ docker run meu-app: teste-arquivo

> docker_1@1.0.0 start /app
> node index.js

Hello, World!
{ fileData: 'olá!' }
$ docker run meu-app: teste-arquivo

> docker_1@1.0.0 start /app
> node index.js

Hello, World!
{ fileData: 'olá!' }
```

Não é mágico?

Isso nos introduz o conceito de **imagem**. Na verdade, `meu-app` não é um container. `meu-app` é uma imagem: um modelo de container. Ou seja, uma descrição de um estado que o container deve ter quando ele começar.

Toda vez que rodamos o `docker run`, o comando imprimiu as mesmas coisas. Isso é porque o `docker run` **cria um container a partir de uma imagem**, enquanto o `docker build` **cria a imagem**.

Assim, fica claro o motivo de imprimir sempre um "olá". **O container criado pelo docker run se baseia em uma imagem que não tem um arquivo teste.txt.**

Mas por que não a opção 1?

O problema com a opção 1 é que ela não provê isolamento completo. Assim como o nosso app pode ser isolado, um sistema operacional inteiro pode ser isolado. Tente você mesmo: `docker run -it ubuntu` começará um terminal novo em um sistema operacional Ubuntu (utilize Ctrl+D ou `exit` para sair).

Imagine agora que você rodou um programa ontem nesse container, e agora você quer rodar um outro programa, mas não consegue pois o programa anterior apagou um arquivo importante.

Concorda que a execução dos dois programas não foi isolada? Afinal, um programa afetou o outro.

A vantagem principal da opção 2 e desvantagem da opção 1 é que **imagens de Docker podem ser compartilhadas na internet**. Por exemplo, a imagem do Ubuntu que você executou foi uma imagem baixada da internet. Se na verdade a imagem fosse um container, ela poderia conter arquivos e dados de execuções de programas anteriores.

Propriedades

Um container de Docker é **efêmero**. Ou seja, ele é feito para executar até a conclusão do programa que está rodando dentro dele. Após isso, fim.

Uma imagem é **herdável**, sendo possível construir imagens em cima de imagens. Foi o que fizemos no nosso app bobo! Basta ler a primeira linha do `Dockerfile`:

```
FROM debian
```

Ou seja, a partir da imagem `debian` construímos a imagem `meu-app`. Todas as imagens herdam da imagem inicial `scratch`. A imagem `scratch` não é instanciável a um container, ou seja, não é possível rodar `docker run scratch` (Isso porque na verdade essa imagem não existe e `FROM scratch` é uma operação que não faz nada).

Na próxima página, vamos ler o Dockerfile de novo e entender pouco a pouco o que está acontecendo, além de explicar algumas questões que devem ter ficado, especialmente essa:

Se um container é efêmero, como eu poderia rodar algo permanente nele? Por exemplo, um banco de dados.

Docker I: um prólogo prático

Docker III: A Dockerfile

Docker III: A Dockerfile

Agora sim, vamos ler a nossa Dockerfile pouco a pouco e entender o que houve. A mentalidade que é preciso ter lendo esse arquivo é a de que **o Dockerfile é a interface entre seu computador real e a imagem sendo construída**, portanto é com ela que você irá mover arquivos de sua máquina para dentro da imagem.

Relembrando, a Dockerfile completa pode ser encontrada na página Docker I. A maior parte dela está comentada para fácil entendimento.

```
# Herda da imagem do debian
FROM debian
```

Falamos sobre herança de imagens na página anterior, é exatamente o que fazemos aqui. Todos os comandos a seguir levam em conta que estamos no ambiente provido pela imagem `debian`.

Note que essa imagem pode sempre mudar: Caso alguém atualize a imagem `debian`, os nossos builds seguintes irão utilizar a imagem nova, mantendo a nossa imagem sempre atualizada também. Quando essa não é a intenção, e controle de versão é mais importante, é possível especificar uma **versão** da imagem, por exemplo `debian:jessie` ou `debian:buster`. A parte depois do `:` é a versão específica da imagem.

```
# Instala o npm
RUN apt update
RUN apt install -y npm
# Atualiza o npm
RUN npm install -g npm
```

O comando `RUN` roda o comando na imagem. Por exemplo, `RUN ls` rodaria o comando `ls`, simples o suficiente.

Mas como assim, imagens podem rodar comandos?

O que o Docker faz, na verdade, é criar um container intermediário com a imagem anterior,

executar o comando, e em sucesso, criar uma imagem intermediária nova para os comandos seguintes.

```
# Cria o diretório /app
# e instala as dependências do /app lá
# note que WORKDIR tanto cria o diretório
# quanto troca para ele (cd /app)
WORKDIR /app
```

o comando `WORKDIR foo` é muito parecido com `RUN mkdir -p foo && cd foo`, exceto que `RUN cd foo` não é levado para os comandos seguintes, pois é parecido com criar um terminal novo, executar `cd` nele e logo após isso fechá-lo. Isso não muda o diretório no terminal anterior!

Logo, é necessário usar esse comando para dizer que todos os próximos comandos são executados dentro do diretório `/app`.

```
COPY package.json .
COPY package-lock.json .
```

Primeiramente, o comando `COPY` tem a seguinte sintaxe: `COPY <arquivo(s) da máquina física> <destino na imagem>`

Logo, estamos copiando o `package.json` e `package-lock.json` do nosso projeto para dentro da pasta `/app`.

Segundo, por que não estamos movendo a pasta inteira de uma vez? Afinal, tudo vai estar lá alguma hora ou outra.

A razão para essa escolha tem a ver com o cacheamento do Docker (o que vou falar mais a fundo em uma próxima página). Essencialmente, o Docker é esperto, e o `docker build` só irá realizar o trabalho necessário. Se esse trabalho já foi feito antes, o Docker só pega da memória o que ele fez anteriormente.

Um exemplo disso é a linha que contém `FROM debian`. O Docker não irá baixar a imagem do Debian toda vez que for reconstruir sua imagem!

Se nós tivéssemos colocado para mover a pasta inteira de uma vez, toda vez que houvesse uma alteração no `index.js`, essa operação teria que ser repetida, já que não há garantias que esses dois arquivos permanecessem os mesmos.

Isso é particularmente ruim pois a próxima linha é essa:

```
RUN npm install
```

que instala arquivos da internet e potencialmente é um comando que pode demorar bastante tempo. Para evitar isso, copiamos o `index.js` **depois** de instalarmos as dependências do projeto, para que uma alteração no projeto não force todas as dependências dele a serem reinstaladas toda vez.

```
# copia o nosso app para dentro da imagem
COPY index.js .
# Esse é o comando que será rodado quando você executar o `docker run`
CMD ["npm", "start"]
```

Por fim, temos o comando `CMD`. Esse comando especifica os argumentos que serão passados para o comando especificado por `ENTRYPOINT`, que por sua vez é o comando rodado pelo `docker run`. Por padrão, o `ENTRYPOINT` é `/bin/sh -c`.

Logo, quando executamos `docker run meu-app`, Um container com a imagem `meu-app` é criado, e esse container executa `/bin/sh -c npm start`. Para entender melhor a diferença entre `ENTRYPOINT` e `CMD`, veja essa resposta no [stack overflow](#).

Ahá! Desconstruímos a nossa Dockerfile. Para saber ainda mais sobre builds, contextos de builds, e a Dockerfile, consulte a [própria documentação do Docker](#). Ela é fantástica e explica muito bem o que acontece.

Docker II: Familiarização

Self-hosted

Servidores de serviços essenciais que você pode hospedar você mesmo

Como ter sua própria VPN com OpenVPN e Docker

Nesse artigo mostraremos como você pode hospedar sua própria VPN usando os software OpenVPN e Docker.

Pré-requisitos

- Acesso à shell de um VPS com um endereço IP público
- Acesso à algum usuário do grupo `docker` na VPS
- Familiaridade com a linha de comando

Docker OpenVPN

Docker OpenVPN é um projeto que visa automatizar a configuração e *set up* do software OpenVPN usando Docker.

Como diz a própria página do GitHub

“ Out of the box stateless openvpn server docker image which starts in just a few seconds and doesn't require persistent storage

Estaremos utilizando esse software nesse tutorial.

Instalando OpenVPN

Primeiro, faça `ssh` na sua VPS.

Arquivo .env

Então, crie um diretório onde iremos armazenar algumas configurações que serão usadas posteriormente.

```
mkdir docker-openvpn && cd docker-openvpn  
touch .env
```

O arquivo `.env` terá algumas configurações básica sobre a VPN.

```
CONTAINER_NAME=openvpn  
PUBLIC_IP=127.0.0.1
```

Substitua `127.0.0.1` pelo seu endereço IP público. Se você não sabe qual é o endereço, execute o seguinte comando na shell da sua VPS:

```
curl -s https://api.ipify.org
```

Que irá imprimir o endereço IP da sua VPS. Então coloque esse endereço no arquivo `.env`.

Usando docker

Há duas formas de instalar `Docker OpenVPN`. Usando `docker`, execute os seguintes comandos:

```
source .env  
docker run --detach \  
  --cap-add=NET_ADMIN \  
  [-publish 1194:1194/udp \  
  [-env HOST_ADDR=${PUBLIC_IP} \  
  [-name ${CONTAINER_NAME} \  
  [-
```

```
alekslitvinenk/openvpn
```

O comando `source .env` irá colocar as variáveis especificadas em `.env` como variáveis de ambiente.

A flag `--detach` é para o container ser executado como um *background process*. A flag `--cap-add` adiciona *capacidades do Linux* ao *container*, nesse caso, ser administrador da rede. A flag `--env` especifica variáveis de ambiente do *container*. A flag `--name` especifica o nome do *container*. A flag `--publish` irá mapear portas do *host* para o *container* usando o protocolo especificado.

Por fim, temos o nome da imagem usada: `alekslitvinenk/openvpn`.

Usando docker-compose

Eis um `docker-compose.yml` para facilitar o trabalho:

```
version: "2.0"

services:
  openvpn:
    container_name: ${CONTAINER_NAME}
    image: alekslitvinenk/openvpn
    restart: unless-stopped
    cap_add:
      - NET_ADMIN
    environment:
      - HOST_ADDR=${PUBLIC_IP}
    ports:
      - 1194:1194/udp
```

Coloque esse arquivo no mesmo diretório que o arquivo `.env`, ou seja, em `docker-openvpn/docker-compose.yml`.

Então, nesse mesmo diretório, rode o comando:

```
docker-compose up --detach
```

Novamente, a flag `detach` é para rodar a VPN no background.

Configurando o cliente

Obtendo o arquivo de configuração

Após ter configurado o servidor, precisamos obter o arquivo de configuração do OpenVPN. Para isso, usaremos `curl`.

Por padrão, essa imagem Docker não vem com `curl`. Temos que instala-lo como segue:

```
docker exec ${CONTAINER_NAME} apk add curl
```

Onde `${CONTAINER_NAME}` é o nome do *container* especificado em `.env`.

Após isso, execute:

```
docker exec ${CONTAINER_NAME} curl -s localhost:8080 > client.ovpn
```

Isso irá executar o comando `curl` dentro do *container*, que irá fazer um *request* para `localhost:8080`, o qual devolverá o arquivo de configuração do nosso OpenVPN e redirecionar a saída para o arquivo `client.ovpn`.

Atenção: Após o request ser feito, o servidor `http` que está ativo em `localhost:8080` será automaticamente desativado, o que significa que não será possível mais obter a configuração do OpenVPN. Entretanto, você pode gerar uma nova configuração usando o comando

```
docker exec ${CONTAINER_NAME} ./genclient.sh .
```

Depois disso você pode transferir o arquivo `client.ovpn` para sua máquina local usando `scp`, `rsync`, ou copiando e colando o conteúdo do arquivo caso seu terminal suporte operações de copiar-e-colar.

Usando `scp`:

```
scp username@host_address: /path/to/client.ovpn .
```

O comando acima irá usar o protocolo `ssh` para transferir o arquivo `client.ovpn` para o diretório atual na sua máquina local.

Lembre-se de substituir `username`, `host_address`, e `/path/to/client.ovpn` pelo seu usuário de nome na VPS, o endereço IP da VPS, e o caminho absoluto do arquivo salvo anteriormente.

Ativando a VPN com o Network Manager

Há inúmeros software clientes do `OpenVPN`. Essa parte do tutorial pressupõe que você está usando `network-manager`, que é um software gerenciador de rede disponível em várias distribuições Linux e BSD. Entretanto, mesmo que você não use `networkmanager`, é provável que tenha algum cliente que funciona para o seu gerenciador de redes.

Primeiro, instale os pacotes `openvpn`, `networkmanager`, e `networkmanager-openvpn`. É possível que os nomes dos pacotes sejam diferentes dependendo do seu Sistema Operacional.

Se você está usando alguma `Arch-based distro`, instale como segue:

```
pacman -S openvpn networkmanager networkmanager-openvpn
```

Após isso, e após habilitar o `networkmanager`, execute:

```
nmcli connection import type openvpn file /path/to/client.ovpn/
```

Onde `/path/to/client.ovpn/` é o caminho do arquivo `client.ovpn` na sua máquina local.

Esse comando irá adicionar uma conexão VPN na sua máquina. Para se conectar à VPN, execute o comando:

```
nmcli connection up client
```

Pronto :D, agora você tem sua própria VPN!

Teoria

A terminologia fica por aqui.