Conteinerização

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 conteinerização de serviços tenta resolver esse problema.

- Introdução
- Docker I: um prólogo prático
- Docker II: Familiarização
- Docker III: A Dockerfile

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 **Conteinerização**.

O que é conteinerização?

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 maquina 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

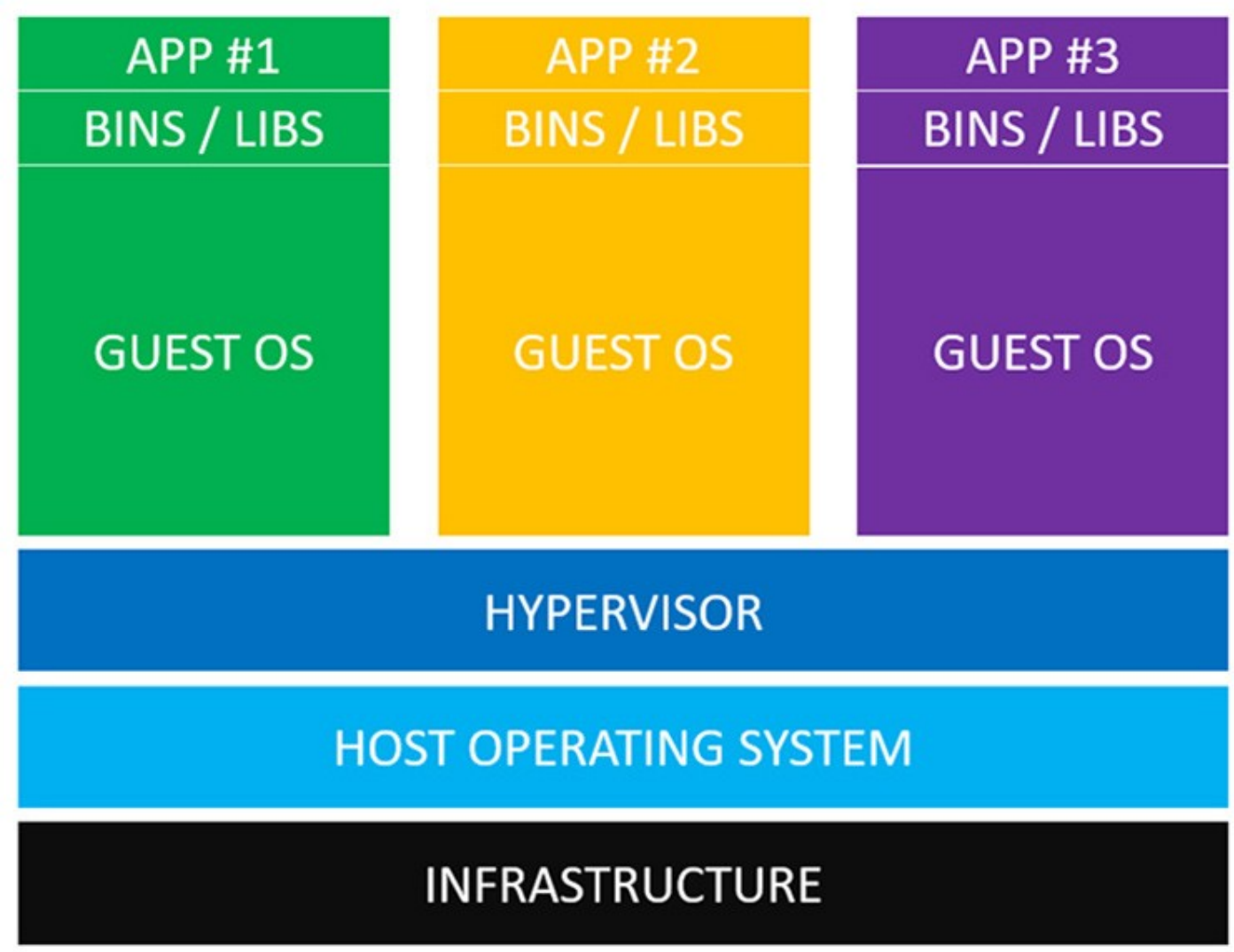
operacional rodando, junto a no mínimo 1 Hypervisor para transcrever instruções das máquinas virtuais para a máquina real.

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.

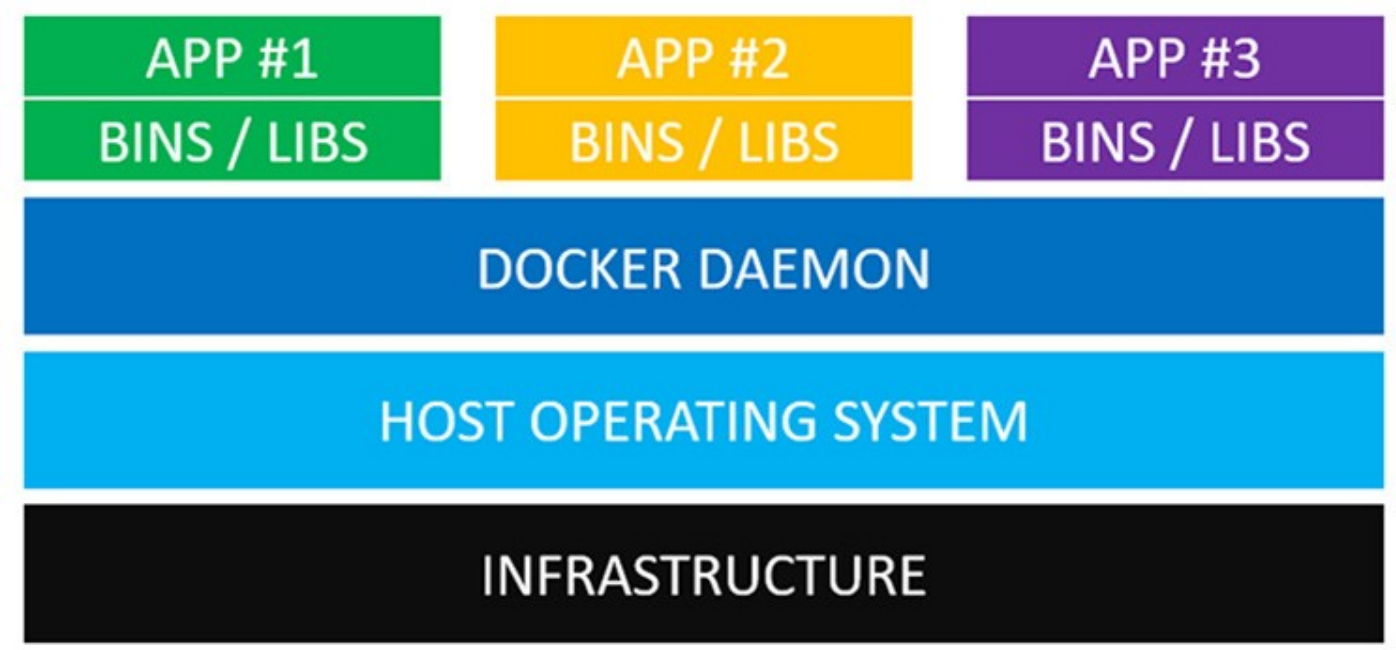
Arquitetura de VMs



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

Convencido de que conteinerização é útil? Então vamos conteinerizar 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, deveriamos 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: Conteinerizando 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

Na parte I construimos 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` construimos 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 facil 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 específicar 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 tivessemos 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 permaneceram 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