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 afetouo 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

Revision #6

Created Sun, Aug 25, 2019 10:35 PM by razgrizone

Updated Tue, Sep 10, 2019 12:47 PM by Cainotis