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

Revision #2
Created Mon, Aug 26, 2019 10:05 PM by razgrizone
Updated Tue, Sep 10, 2019 12:39 PM by Cainotis