Transcrição

Já trabalhamos com a imagem do **ubuntu**, **hello-world**, **dockersamples/static-site** e por fim do **node**, mas até agora não criamos a nossa própria imagem, para podermos distribuir para as outras pessoas. Então é isso que faremos nesta aula.

No começo do treinamento, foi comentado que a imagem é como se fosse uma *receita de bolo*. Então, para criarmos a nossa própria imagem, temos que criar a nossa *receita de bolo*, o **Dockerfile**, ensinando o Docker a criar uma imagem a partir da nossa aplicação, para que ela seja utilizada em outros locais.

Montando o Dockerfile

Então, no nosso projeto, devemos criar o arquivo **Dockerfile**, que nada mais é do que um arquivo de texto. Ele pode ter qualquer nome, porém nesse caso ele também deve possuir a extensão **.dockerfile**, por exemplo **node.dockerfile**, mas vamos manter o nome padrão mesmo.

Geralmente, montamos as nossas imagens a partir de uma imagem já existente. Nós podemos criar uma imagem do zero, mas a prática de utilizar uma imagem como base e adicionar nela o que quisermos é mais comum. Para dizer a imagem-base que queremos, utilizamos a palavra `FROM` mais o nome da imagem.

Como o nosso projeto precisa do Node.js, vamos utilizar a sua imagem:

```
FROM node
```

COPIAR CÓDIGO

Além disso, podemos indicar a versão da imagem que queremos, ou utilizar o `latest`, que faz referência à versão mais recente da imagem. Se não passarmos versão nenhuma, o Docker irá assumir que queremos o `latest`, mas vamos deixar isso explícito:

```
FROM node:latest
```

COPIAR CÓDIGO

Outra instrução que é comum colocarmos é quem cuida, quem criou a imagem, através do `MAINTAINER`:

```
FROM node:latest
MAINTAINER Douglas Quintanilha
```

COPIAR CÓDIGO

Agora, especificamos o que queremos na imagem. No caso, queremos colocar o nosso código dentro da imagem, então utilizarmos o `COPY`. Como queremos copiar tudo o que está dentro da pasta, vamos utilizar o `.` para copiar tudo que está na pasta do arquivo **Dockerfile**, e vamos copiar para **/var/www**, do exemplo da aula anterior:

```
FROM node:latest
MAINTAINER Douglas Quintanilha
COPY . /var/www
```

COPIAR CÓDIGO

No projeto, já temos as suas dependências dentro da pasta **node_modules**, mas não queremos copiar essa pasta para dentro do *container*, pois elas podem estar desatualizadas, quebradas, então queremos que a própria imagem instale as

dependências para nós, rodando o comando `npm install`. Para executar um comando, utilizamos o `RUN`:

```
FROM node:latest
MAINTAINER Douglas Quintanilha
COPY . /var/www
RUN npm install
```

COPIAR CÓDIGO

Agora, **deletamos a pasta node_modules**, para ela não ser copiada para o *container*. Além disso, toda imagem possui um comando que é executado quando a mesma inicia, e o comando que utilizamos na aula anterior foi o `npm start`. Para isso, utilizamos o `ENTRYPOINT`, que executará o comando que quisermos assim que o *container* for carregado:

```
FROM node:latest
MAINTAINER Douglas Quintanilha
COPY . /var/www
RUN npm install
ENTRYPOINT npm start
```

COPIAR CÓDIGO

Também podemos passar o comando como se fosse em um array, por exemplo `["npm", "start"]`, ambos funcionam.

Falta colocarmos a porta em que a aplicação executará, a porta em que ela ficará exposta. Para isso, utilizamos o `EXPOSE`:

```
FROM node:latest
MAINTAINER Douglas Quintanilha
COPY . /var/www
RUN npm install
```

```
ENTRYPOINT ["npm", "start"]
EXPOSE 3000
```

COPIAR CÓDIGO

Por fim, falta dizermos onde os comandos rodarão, pois eles devem ser executados dentro da pasta `/var/www`. Então, através do `WORKDIR`, assim que copiarmos o projeto, dizemos em qual diretório iremos trabalhar;

```
FROM node:latest
MAINTAINER Douglas Quintanilha
COPY . /var/www
WORKDIR /var/www
RUN npm install
ENTRYPOINT npm start
EXPOSE 3000
```

COPIAR CÓDIGO

Com isso, finalizamos o **Dockerfile**, baseado no comando que fizemos na aula anterior:

```
alura@alura-estudio-03:~/Desktop/volume-exemplo$ docker run -p 8080:3000 -v "$(pwd):/var/www" -w "/
```

COPIAR CÓDIGO

Resta agora criar a imagem.

Criando a imagem

Para criar a imagem, precisamos fazer o seu *build* através do comando `docker build`, comando utilizado para *buildar* uma imagem a partir de um **Dockerfile**. Para configurar esse comando, passamos o nome do **Dockerfile** através da *flag* `-f`:

```
alura@alura-estudio-03:~/Desktop/volume-exemplo$ docker build -f Dockerfile
```

COPIAR CÓDIGO

Como o nome do nosso **Dockerfile** é o padrão, poderíamos omitir esse parâmetro, mas se o nome for diferente, por exemplo **node.dockerfile**, é preciso especificar, mas vamos deixar especificado para detalharmos melhor o comando.

Além disso, passamos a *tag* da imagem, o seu nome, através da *flag* `-t`. Já vimos que para imagens não-oficiais, colocamos o nome no padrão **NOME_DO_USUARIO/NOME_DA_IMAGEM**, então é isso que faremos, por exemplo:

```
alura@alura-estudio-03:~/Desktop/volume-exemplo$ docker build -f Dockerfile -t douglasq/node
```

COPIAR CÓDIGO

E agora dizemos onde está o **Dockerfile**. Como já estamos rodando o comando dentro da pasta **volume-exemplo**, vamos utilizar o ponto (`.`);

```
alura@alura-estudio-03:~/Desktop/volume-exemplo$ docker build -f Dockerfile -t douglasq/node .
```

COPIAR CÓDIGO

Ao executar o comando, podemos perceber que cada instrução executada do nosso **Dockerfile** possui um **id**. Isso por que para cada passo o Docker cria um *container* intermediário, para se aproveitar do seu sistema de camadas. Ou seja, cada instrução gera uma nova camada, que fará parte da imagem final, que nada mais é do que a imagem-base com vários *containers* intermediários em cima, sendo que cada um desses *containers* representa um comando do **Dockerfile**.

Assim, se um dia a imagem precisar ser alterada, somente o *container* referente à instrução modificada será alterado, com as outras partes intermediárias da imagem já prontas.

Criando um container a partir da nossa imagem

Agora que já temos a imagem criada, podemos criar um *container* a partir dela:

```
docker run -d -p 8080:3000 douglasq/node
```

COPIAR CÓDIGO

Ao acessar o endereço da porta no navegador, vemos a página da nossa aplicação. No **Dockerfile**, também podemos criar variáveis de ambiente, utilizando o `ENV`. Por exemplo, para criar a variável `PORT`, para dizer em que porta a nossa aplicação irá rodar, fazemos:

```
FROM node:latest
MAINTAINER Douglas Quintanilha
ENV PORT=3000
COPY . /var/www
WORKDIR /var/www
RUN npm install
ENTRYPOINT npm start
EXPOSE 3000
```

COPIAR CÓDIGO

E no próprio **Dockerfile**, podemos utilizar essa variável:

```
FROM node:latest
MAINTAINER Douglas Quintanilha
ENV PORT=3000
COPY . /var/www
WORKDIR /var/www
RUN npm install
ENTRYPOINT npm start
EXPOSE $PORT
```

COPIAR CÓDIGO

E como modificamos o **Dockerfile**, precisamos construir a nossa imagem novamente e podemos perceber que dessa vez o comando é bem mais rápido, já que quase todas as camadas estão *cacheadas* pelo Docker.

Agora que criamos a imagem, vamos disponibilizá-la para outras pessoas. E é isso que veremos no próximo vídeo.