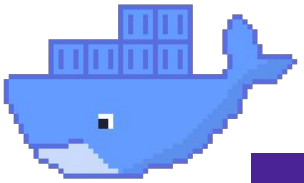
trybe

Imagens

Bloco 19 - Aula 19.2



Conceitos

- A imagem é um arquivo e o contêiner é um processo;
- Uma imagem é um pacote de sistema de arquivos que contém todas as dependências necessárias para executar um processo: arquivos de biblioteca, arquivos no sistema de arquivos, pacotes instalados, recursos disponíveis, processos em execução e módulos do kernel;

Conceitos

- Como as imagens são arquivos, elas podem ser gerenciadas por sistemas de controle de versão, melhorando a automação do contêiner e o provisionamento;
- As imagens do contêiner precisam estar disponíveis localmente ou armazenadas e mantidas em um repositório de imagens;

Docker - Comandos

- Podemos ter vários containers reproduzindo uma mesma imagem do Docker;
- Toda imagem possui sua IMAGE ID e todo container possui seu CONTAINER ID;
- Ambos são identificadores únicos desses elementos dentro do Docker e servem como referência para outras possibilidades de comando;

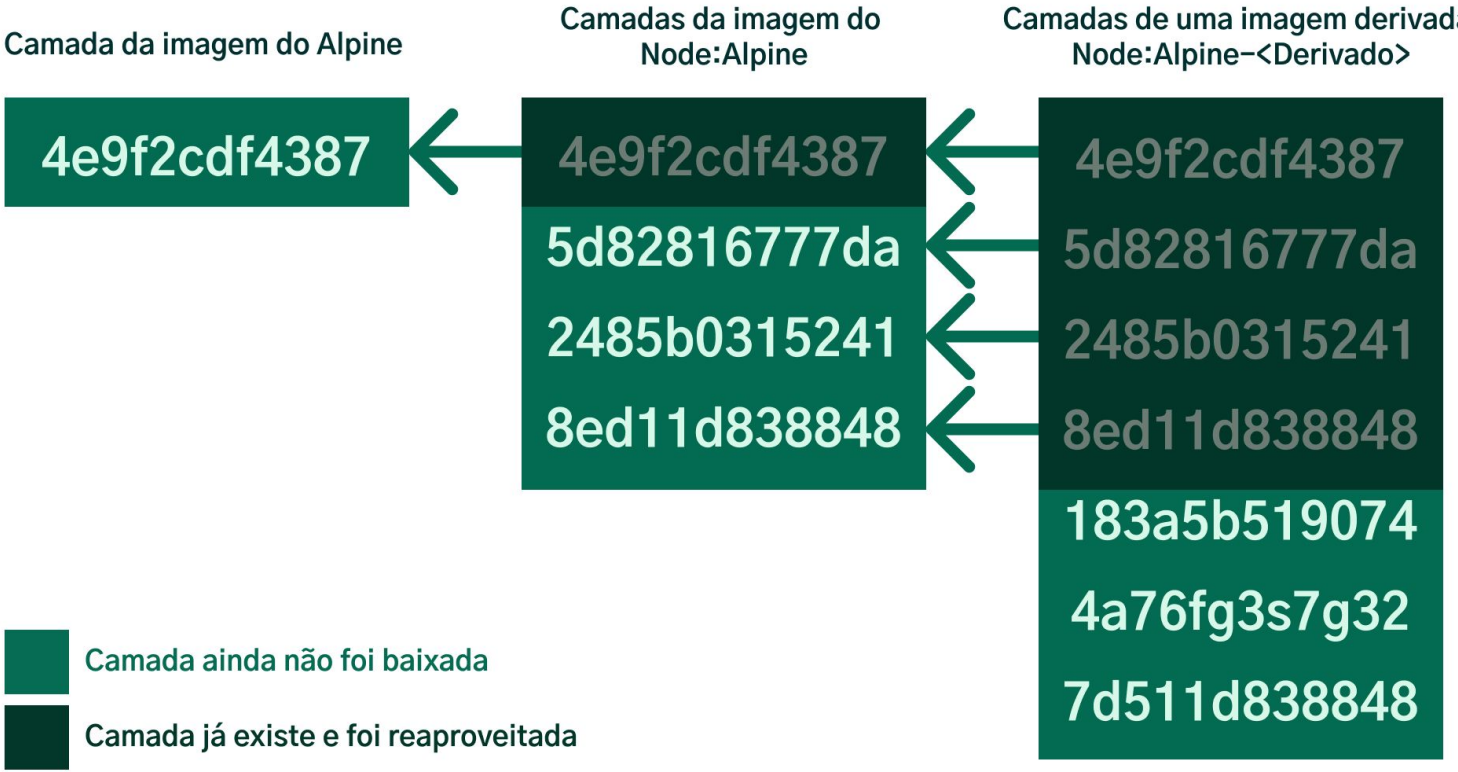
Docker - Comandos

- Para verificarmos as imagens
  - docker images
- Para verificarmos containers a partir das imagens
  - docker ps -a
- Removendo uma imagem
  - docker images
  - docker rmi -f ID

Docker - Nome do Container

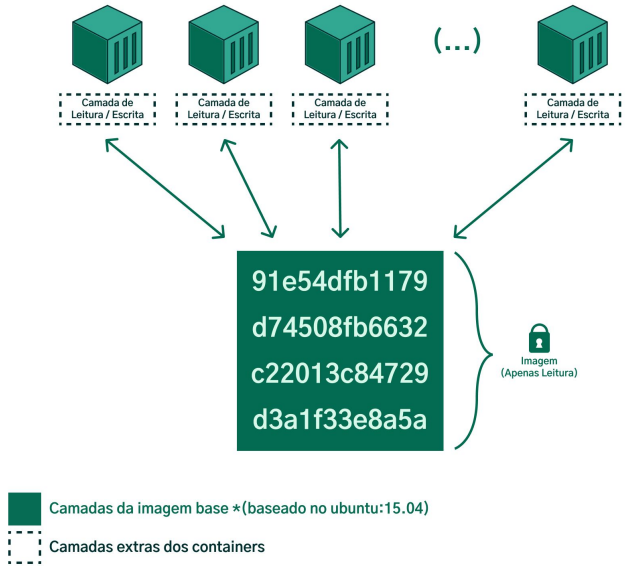
- Comandos
  - docker run --name meu_container -it ubuntu
  - echo "Teste container" > ola_mundo.txt
  - cat ola_mundo.txt
  - docker start -ai meu_container
  - cat ola_mundo.txt

Docker - Camadas



Docker - Container Layer

- Todas as vezes que criamos um container, uma camada extra (chamada frequentemente de “container layer” - camada do container) é adicionada aquela imagem para que seja possível ler e escrever nela;



Docker - Mapeamento de Portas

- O uso do parâmetro -P , ele é utilizado para que o Docker faça um mapeamento de portas automático para acesso ao container
  - docker run -d -P httpd:2.4
  - docker ps
  - http://localhost:PORTA

Docker - Mapeamento de Portas

- Podemos linkar manualmente uma porta de nosso computador com a porta estabelecida no docker
  - docker run -d -p 54321:80 httpd:2.4
- Ter cuidado com qual porta sua aplicação vai usar

Docker - DockerFile

- O Dockerfile nada mais é do que um arquivo de configuração usado pelo Docker com a descrição passo a passo do que você deseja que aconteça;
- **Vamos criar um?**
    - **A partir de um projeto em NodeJS pronto, vamos subir este projeto e rodar ele em um container no Docker?**

Docker - DockerFile - Case

- Criar um arquivo Dockerfile
- Escolher a imagem base (node)
- Deixar o projeto “pronto”
- Copiar os arquivos de nossa máquina para o Container
- Criar a Imagem
- Criar um container a partir desta Imagem
- Testar

Docker - DockerFile - FROM

- Com essa instrução, pode-se definir qual será o ponto de partida da imagem que criaremos com o nosso Dockerfile;
- É recomendado utilizar sempre uma versão específica de nossa imagem base em nossas imagens de produção;
- Outra recomendação é, sempre que possível, utilizar as versões "mínimas" da imagem;

```
FROM node:alpine
```

Docker - DockerFile - WORKDIR

- Essa instrução tem o propósito de definir o nosso ambiente de trabalho;
- Com ela, definimos onde as instruções CMD, RUN, ENTRYPOINT, ADD e COPY executarão suas tarefas, além de definir o diretório padrão que será aberto ao executarmos o container;

```
WORKDIR /usr/app
```

Docker - DockerFile - COPY E ADD

- O papel do ADD é fazer a cópia de um arquivo, diretório ou até mesmo fazer o download de uma URL de nossa máquina host e colocar dentro da imagem
- Já a instrução COPY, permite apenas a passagem de arquivos ou diretórios, diferente do ADD, que permite downloads;

```
COPY package*.json ./

COPY . .
```

Docker - DockerFile - RUN

- Ela pode ser executada uma ou mais vezes e, com ela, posso definir quais serão os comandos executados na etapa de criação de camadas da imagem;
- O RUN é comum para prepararmos a imagem para rodar nossos apps, instalando as dependências de uma aplicação;
- Atenção aos passos adicionais;

```
RUN npm install
```

Docker - DockerFile - EXPOSE

- Muitas pessoas pensam que o EXPOSE serve para definir em qual porta nossa aplicação rodará dentro do container, mas na verdade o propósito é servir apenas para documentação;
- Essa instrução não publica a porta efetivamente, já que o propósito dela é fazer uma comunicação entre quem escreveu o Dockerfile e quem rodará o container;

```
EXPOSE 3000
```

Docker - DockerFile - CMD

- Sempre é executado quando o container é iniciado;
- É interessante ressaltar que pode acontecer de mais de um CMD ser definido em um mesmo Dockerfile e, neste caso, apenas o último terá efeito;
- O CMD possui 2 formas: a que vimos até aqui para a execução de comandos shell e as para executáveis;

```
CMD npm start
```

Docker - DockerFile - ENTRYPOINT

- Podemos ter quantos CMD's precisarmos, mas somente o último será executado;
- O ENTRYPOINT faz exatamente a mesma coisa, porém seus parâmetros não são sobrescritos igual ao CMD;

```
FROM ubuntu:18.04
RUN apt-get update
RUN apt-get install openjdk-8-jdk -y
CMD touch arquivo-de-boas-vindas
CMD touch outro-arquivo
```

Docker - DockerFile - CMD e ENTRYPOINT

- Em ENTYRPOINT vamos ter um comando que sempre será executado quando o container iniciar;
- Já no CMD colocamos um argumento 'default', que será executado no ENTRYPOINT quando nada for especificado;

```
FROM ubuntu
ENTRYPOINT ["top", "-i"]
CMD ["-d"]
```

```
1)  docker run -it --rm --name test  top
2)  docker run -it --rm --name test  top -P
```

Docker - DockerFile - EXECUÇÃO

- Dockerfile ao final

```
FROM node:alpine

WORKDIR /usr/app

COPY package*.json ./

RUN npm install

COPY . .

EXPOSE 3000

CMD npm start

LABEL maintener="Felipe Becker Nunes
<nunesfb@gmail.com>"
```

Docker - DockerFile - EXECUÇÃO

- Para que a gente consiga de fato consolidar as instruções do Dockerfile em uma imagem, precisamos rodar o comando docker image build -t <name:tag> <origem_docker_file>

```
docker build -t nunesfb/node .
```

Docker - DockerFile - EXECUÇÃO

- Com a imagem montada, podemos verificar se está tudo certo com ela
- E podemos então criar um container a partir desta imagem personalizada

```
docker run -p 3000:3000 -d nunesfb/node

docker ps

http://localhost:3000/
```

Docker - DockerFile - LABELS

- Labels (Rótulos em português) são um mecanismo para atribuir "metadatas" (dados auxiliares) aos seus objetos Docker , como imagens e containers;
- Com o parâmetro LABEL , é possível fazer essas definições em nosso Dockerfile;
- As informações são registradas seguindo o parâmetro de "chave e valor", e caso uma chave esteja repetida, a última sobrescreverá as anteriores;

```
LABEL maintener="Felipe Becker Nunes <nunesfb@gmail.com>"

docker inspect ID

docker logs
```

Docker - DockerFile - Copiando Arquivos

- Para realizar a ação de cópia vamos utilizar o comando de Docker chamado cp
- Ele serve tanto para cópia de arquivos do container para a máquina, como também da máquina para o container
- Para copiar arquivos vamos precisar saber o id do container

```
arquivo copiado do nosso computador, máquina host, para o container de Docker
docker cp meuarquivo.js <id_do_container>:/meuarquivo.js

copiar do containaer para a máquina host
docker cp <id_do_container>:/meuarquivo.js meuarquivo.js
```

Docker - DockerFile - ENV

- Em ambientes de desenvolvimento de apps é muito importante o uso de Env ironment Variables, felizmente também podemos utilizá-las em nossos containers
- No Dockerfile , podemos definir nossas variáveis durante a criação de nossa imagem utilizando o comando ENV;

```
docker container run \

  --env myCat=fluffy \

  --env myName=johnDoe \

  <IMAGE NAME>
```

```
FROM node:alpine

RUN mkdir -p /usr/src/app

WORKDIR /usr/src/app

COPY package.json /usr/src/app/

RUN npm install

COPY . /usr/src/app

ENV PORT 3000

ENV NODE_ENV production

EXPOSE ${PORT}

CMD [ "npm","run", "start" ]
```

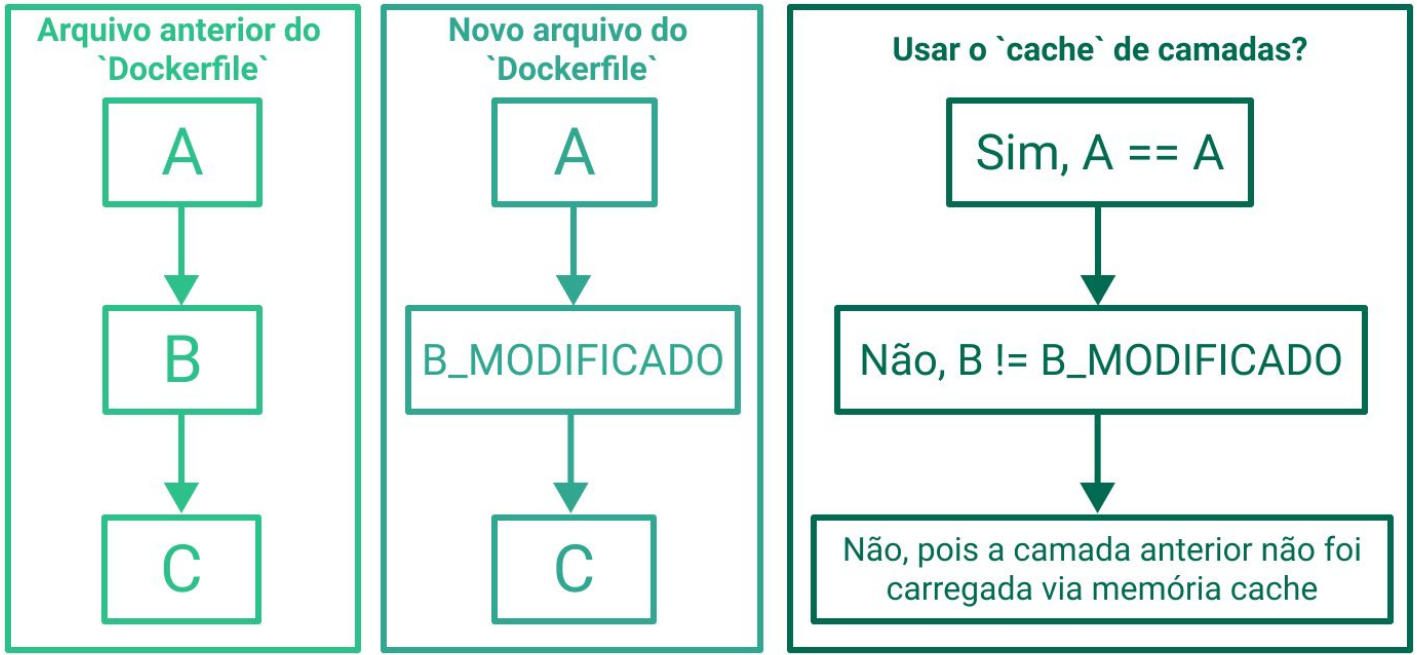
Docker - DockerFile - USER

- Com o comando USER , podemos definir qual o usuário que irá iniciar nosso app no container .
- Caso não seja definido nenhum usuário, o Docker irá utilizar o usuário root como padrão, o que não é recomendado por motivos de segurança;
- Normalmente as imagens já possuem um usuário criado para a execução de nossos apps;

```
FROM node:alpine

USER node ## definindo o usuário que realizar os comandos

WORKDIR /usr/app

COPY package.json ./

RUN npm install

CMD npm start
```

Docker - DockerFile - LAYERS/CACHE

- Estrutura exemplo



Docker - DockerFile - Imagem de um Container

- Podemos criar uma imagem a partir de um container que está rodando
  - docker create --name nginx-base -p 80:80 nginx:alpine
  - docker container start ID
  - criar um arquivo html
  - docker cp index.html nginx-base:/usr/share/nginx/html/index.html
  - teste no navegador

Docker - DockerFile - Imagem de um Container

- Podemos criar uma imagem a partir de um container que está rodando
  - crie a imagem a partir do container - docker commit nginx-base
  - verifique a imagem - docker images
  - dê um nome para a imagem - docker tag IMAGE_ID nginx-base-container
  - cria o container - docker create --name nginx-new -p 80:80 nginx-base-container:latest
  - remove o container anterior
  - inicia o novo container
  - testar

Docker - Atividade Prática

- Vamos criar uma imagem com ReactJS e rodar nosso projeto em um container a partir desta imagem personalizada?
  - FROM
  - WORKDIR
  - COPY
  - RUN
  - COPY
  - CMD

Docker - Atividade Prática

- Vamos criar uma imagem com ReactJS e rodar nosso projeto em um container a partir desta imagem personalizada?
  - Build da imagem
  - Criação do Container
  - Teste

Docker - Atividade Prática

- Vamos criar uma imagem com ReactJS e rodar nosso projeto em um container a partir desta imagem personalizada?

```
FROM node:latest
WORKDIR /app
COPY package.json ./
RUN npm install
COPY . .
CMD ["npm", "start"]
```

```
docker build -t nomedoapp:latest .
```

```
docker run --name nomealeatorio -d -p 3000:3000 nomedoapp:latest
```