



# Construindo nossa imagem

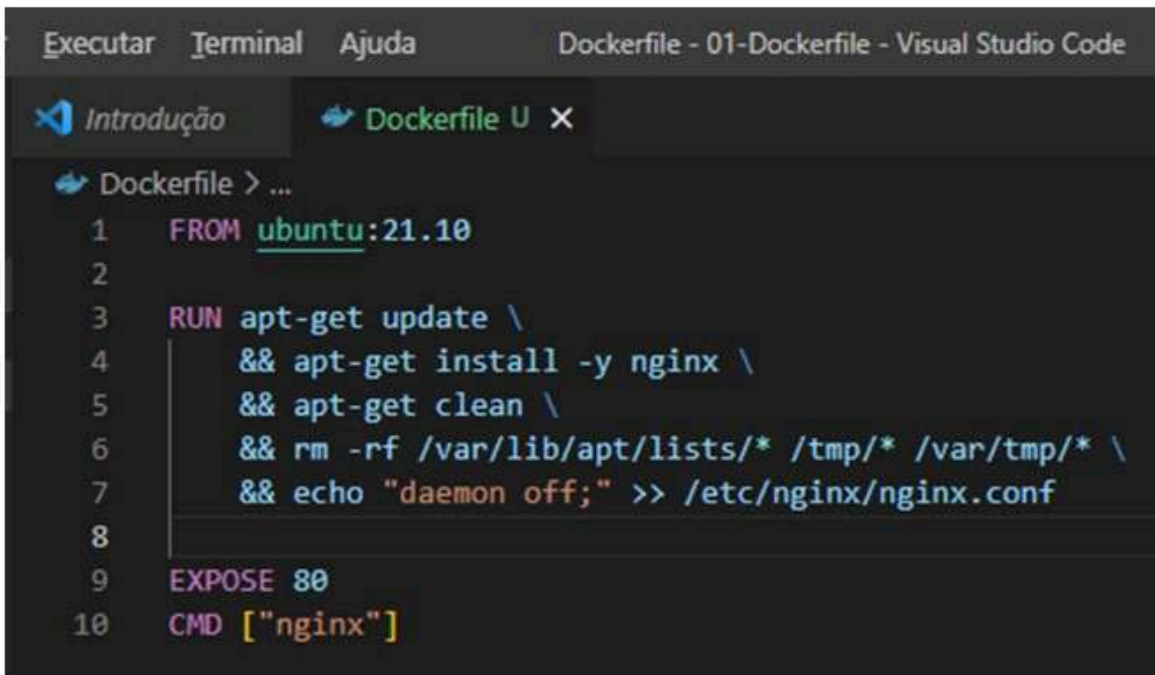
**P**egar uma imagem já pronta no DockerHub e subir um container é moleza, mas essas imagens não surgiram do nada, alguém os construíram!

É aqui que entra o docker build, uma ferramenta do Docker para construirmos nossas próprias imagens com o que queremos expor para nós subirmos containers ou quem sabe para o mundo inteiro poder subir containers à partir de uma imagem sua! Um pedacinho de você para contribuir com o mundo, quem sabe?

Bom, aspirações globais à parte, vamos começar pequeno! A primeira coisa a fazermos é entender como essas imagens são criadas e o primeiro item a olharmos com carinho é: afinal de contas o que é o tal do Dockerfile?

O Dockerfile é um arquivo que tem dentro uma “receita de bolo”, ou seja, um script passo-a-passo que diz como construir a imagem. E o arquivo é assim mesmo, ele se chama “Dockerfile” com o “D” maiúsculo e o resto minúsculo, sem extensão nenhuma (não tem “.txt”, nem “.yaml”, nem nada), e tem que ser assim, ok? É o formato padrão para que o docker build consiga lê-lo e fazer as instruções conforme a “receita de bolo”.

Vamos olhar para um Dockerfile rápido que criei aqui, por enquanto não se preocupem muito com o conteúdo em si, o importante é focarmos no que cada comando representa/faz:



```
Executar Terminal Ajuda Dockerfile - 01-Dockerfile - Visual Studio Code
Introdução Dockerfile U x
Dockerfile > ...
1 FROM ubuntu:21.10
2
3 RUN apt-get update \
4     && apt-get install -y nginx \
5     && apt-get clean \
6     && rm -rf /var/lib/apt/lists/* /tmp/* /var/tmp/* \
7     && echo "daemon off;" >> /etc/nginx/nginx.conf
8
9 EXPOSE 80
10 CMD ["nginx"]
```

Vamos entender o que vai acontecer se pedirmos para o Docker fazer um build (construir) uma imagem à partir desta “receita”:

```
FROM ubuntu:21.10
```

Toda imagem tem uma base à qual se sustentar, vamos dizer que essa base possui o mínimo necessário para o container rodar quando for criado um container à partir da imagem que estamos construindo. Neste caso a “base” usada é uma versão bem light do Ubuntu Linux, versão 21.10. Se estivéssemos montando uma imagem com uma aplicação em Java, por exemplo, poderíamos usar outra base como a “maven:3.8.5-openjdk-17-slim” que já vem com um mínimo necessário para rodar Java, mas veremos isso mais pra frente.

Importante lembrar que todo Dockerfile deveria iniciar com o FROM, indicando qual é a “base” da imagem a ser construída.

```
RUN apt-get update \
    && apt-get install -y nginx \
    && apt-get clean \
    && rm -rf /var/lib/apt/lists/* /tmp/* /var/tmp/* \
    && echo "daemon off;" >> /etc/nginx/nginx.conf
```

Depois da “base” indicada, podemos executar comandos como este acima. Então pensa assim: imagine que você esteja em uma versão mínima de um Ubuntu Linux, você poderia executar um comando para atualizar os pacotes que tem no Ubuntu através do comando “apt-get update” e instalar um pacote específico como o “nginx” por exemplo através do comando “apt-get install -y nginx”. Bom, é exatamente isso que estamos fazendo nesse comando acima, só que está sendo feito de uma vez várias coisas, por enquanto não precisaremos entrar no detalhe, mas o que precisa ser conhecido é que o comando RUN no script do Dockerfile executa instruções dentro da “base”, é como se eu entrasse dentro do Ubuntu dessa base e digitasse esses comandos para fazer algo. O comando RUN só é executado enquanto estivermos construindo a imagem, lembre-se disso!

```
EXPOSE 80
```

Como neste exemplo estamos criando uma imagem com o nginx só que customizando o que a imagem vai ter, nós podemos expor uma porta para nos comunicarmos com o servidor web do nginx. O nginx é um servidor web bastante enxuto, que você pode usar para rodar um site simples dentro dele, para acessarmos esse servidor web, o fazemos através de uma porta de comunicação. Por enquanto não precisa se preocupar com o que é exatamente uma porta de comunicação, veremos isso mais pra frente. Esse comando EXPOSE do script Dockerfile faz essa exposição de porta.

```
CMD ["nginx"]
```

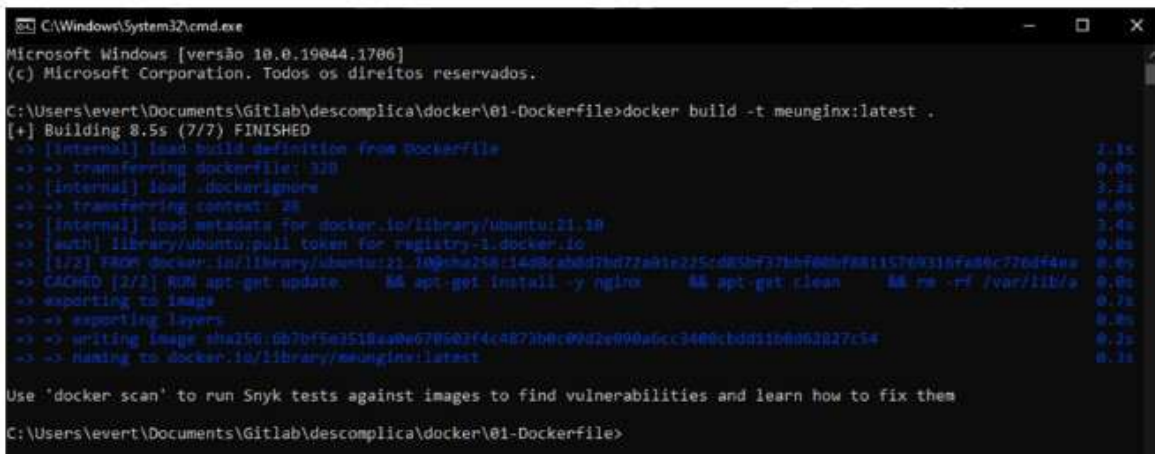
O comando CMD é parecido com o comando RUN, mas com uma diferença essencial: ele executa algo da “base” (nosso Ubuntu), só que somente quando um container é iniciado através dessa imagem que foi construída por nós, ou seja, enquanto o RUN só roda enquanto estamos construindo a imagem, o CMD só irá

rodar quando um container iniciar, ele não é executado enquanto estivermos construindo nossa imagem.

Há outros tipos de comando que podem ser utilizados no Dockerfile, por ora o intuito é falar do Dockerfile de um modo geral para entendermos o mecanismo de como ele instrui o build do Docker para construir uma imagem, mais pra frente veremos outros Dockerfile mais específicos.

Com o Dockerfile entendido (por enquanto algo simples), o seguinte comando cria uma imagem à partir das instruções desse Dockerfile:

`docker build -t meunginx:latest .`



```
C:\Windows\System32\cmd.exe
Microsoft Windows [versão 10.0.19044.1706]
(c) Microsoft Corporation. Todos os direitos reservados.

C:\Users\event\Documents\Gitlab\descomplica\docker\01-Dockerfile>docker build -t meunginx:latest .
[+] Building 8.5s (7/7) FINISHED
=> [internal] load build definition from Dockerfile 2.1s
=> => transferring dockerfile: 32B 0.0s
=> [internal] load .dockerignore 3.2s
=> => transferring context: 2B 0.0s
=> [internal] load metadata for docker.io/library/ubuntu:21.10 3.4s
=> [auth] library/ubuntu:pull token for registry-1.docker.io 0.0s
=> [1/2] FROM docker.io/library/ubuntu:21.10@sha256:1408cabdd7bd77a01e225cd85bf37bbf00bf88115769310f80c776d44ae 0.0s
=> CACHED [2/2] RUN apt-get update && apt-get install -y nginx && apt-get clean && rm -rf /var/lib/a 0.0s
=> exporting to image 0.7s
=> => exporting layers 0.0s
=> => writing image sha256:0b7bffa2518aa0e670502f4c4873b0c09d2e090a0cc3400cbdd31b0d0d327c34 0.2s
=> => naming to docker.io/library/meunginx:latest 0.1s

Use 'docker scan' to run Snyk tests against images to find vulnerabilities and learn how to fix them
C:\Users\event\Documents\Gitlab\descomplica\docker\01-Dockerfile>
```

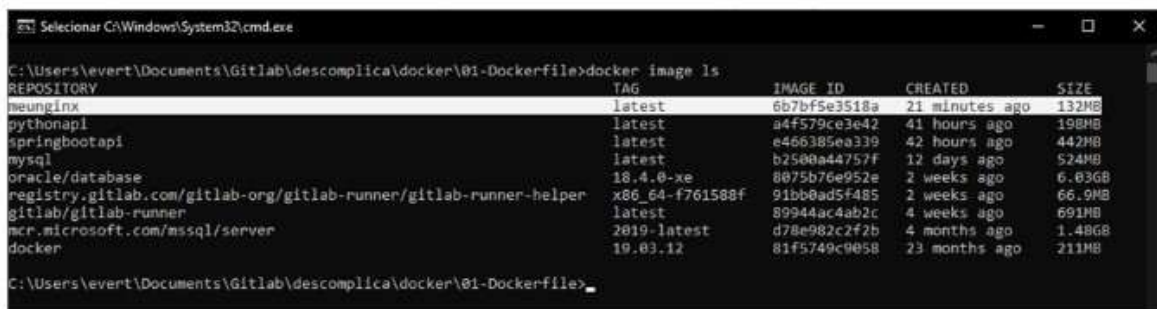
O comando “docker build” instrui ao Docker para ele ler o Dockerfile e, à partir das instruções, construir a imagem.

O parâmetro “-t meunginx:latest” indica o nome de “tag” que esta imagem terá, é através dela que indicamos ao Docker qual é a imagem que usaremos ao tentar criar um container. O formato é o nome do repositório antes dos dois pontos e a versão após os dois pontos. Lembrando que este parâmetro é opcional, mas altamente recomendável incluir!

Vamos ver nossa imagem criada no repositório interno do Docker que fica no computador que estamos usando o Docker, após a criação da imagem, o seguinte

comando lista todas as imagens presentes, incluindo esta recém criada:

`docker image ls`



REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
meunginx	latest	6b7bf5e3518a	21 minutes ago	132MB
pythonapi	latest	a4f579ce3e42	41 hours ago	198MB
springbootapi	latest	e466385ea339	42 hours ago	442MB
mysql	latest	b2580a44757f	12 days ago	524MB
oracle/database	18.4.0-xe	8075b76e952e	2 weeks ago	6.03GB
registry.gitlab.com/gitlab-org/gitlab-runner/gitlab-runner-helper	x86_64-f761588f	91bb0ad5f485	2 weeks ago	66.9MB
gitlab/gitlab-runner	latest	89944ac4ab2c	4 weeks ago	691MB
mcr.microsoft.com/mssql/server	2019-latest	d78e982c2f2b	4 months ago	1.48GB
docker	19.03.12	81f5749c9058	23 months ago	211MB

O comando “docker image” é para trabalharmos com imagens do Docker, o parâmetro “ls” indica que é para listar todas as imagens do Docker.

Veja que nossa imagem aparece! Está como repositório de nome “meunginx” e a versão (tag) da imagem se chama “latest”. Note o tamanho da imagem: 132Mb, ele tem esse tamanho pois depende do tamanho da base que usamos (o Ubuntu) e dos pacotes que instalamos dentro dele (nginx).

É possível criar um container à partir desta imagem recém criada, o seguinte comando é um exemplo de como criar um container simples à partir dessa imagem:

`docker run --name MeuNginx -p 9080:80 -d meunginx:latest`

O comando “docker run” instrui ao Docker que é para executar um container.

O parâmetro “--name MeuNginx” indica um apelido que estamos dando ao container, que poderá ser consultado mais tarde.

O parâmetro “-p 9080:80” indica que poderemos acessar “de fora” do container via porta 9080 e que o Docker fará a “tradução” de tudo que vier pela porta 9080 de fora, para a porta 80 de dentro do container. Haverá uma explicação mais detalhada sobre as portas, no momento não se preocupe.

O parâmetro “-d” indica que o container subirá e ficará rodando sem precisarmos estar conectados nele, é o oposto do “-it” em que o container sobe no modo interativo, em que ficamos conectados dentro do container assim que ele sobe.

O último parâmetro “meunginx:latest” é o nome da imagem e a versão que iremos usar para subir o container, que neste exemplo é a imagem recém criada.

Para ver o container recém criado, podemos usar o seguinte comando para listar todos os containers:

docker container ls

```

C:\Users\levert\Documents\Gitlab\descomplica\docker\01-DockerFile>docker run --name MeuNginx -p 9080:80 -d meunginx:latest
99a4c3c12d552965f4d0f9c1d9f01937ec91217a31c007e0b3d050e529e

C:\Users\levert\Documents\Gitlab\descomplica\docker\01-DockerFile>docker container ls
CONTAINER ID   IMAGE          COMMAND                  CREATED        STATUS        PORTS                               NAMES
99a4c3c12d55   meunginx:latest    "nginx"                  15 seconds ago    Up 10 seconds    0.0.0.0:9080->80/tcp               MeuNginx
38b5fc7dfe0a   pythonapi:latest   "uvicorn app.main:ap..." 44 hours ago     Up 15 hours     0.0.0.0:8080->8080/tcp             MeuPython
api
3f424478caef   springbootapi:latest   "java -jar /usr/loca..." 44 hours ago     Up 15 hours     0.0.0.0:8080->8080/tcp             MeuSpring
bootAPI
3d6565c08dd7   mysql:latest       "docker-entrypoint.s..." 6 days ago       Up 15 hours     0.0.0.0:3306->3306/tcp, 33060/tcp   MeuMySQL
0dad8ee4cd75   oracle/database:18.4.0-xx "/bin/sh -c 'exec $@" 6 days ago       Up 15 hours (healthy)    0.0.0.0:1521->1521/tcp, 0.0.0.0:3500->5500/tcp   MeuOracle

```

Veja que a coluna “NAMES” apresenta o apelido que demos ao container, neste exemplo “MeuNginx”.

A coluna “IMAGE” é a imagem que foi usada para subir o container, que neste exemplo é a nossa imagem recém criada chamada “meunginx:latest”.

A coluna “COMMAND” possui o comando a ser executado assim que o container é criado, é exatamente o comando daquela linha do Dockerfile que indicava “CMD [nginx]”.

A coluna “PORTS” indica em qual porta de comunicação o container aceita chamadas, colocamos para ele aceitar chamadas “de fora do container” via porta 9080, e o Docker faz a tradução “para dentro do container” via porta 80. A porta 80 “de dentro do container” é exatamente o que indicamos na linha do Dockerfile que indicava “EXPOSE 80”.

Uma parte interessante dessa parte de portas (que veremos mais no detalhe mais pra frente) é que a indicação “0.0.0.0:9080” significa que:

- 0.0.0.0 pode ser entendido como “localhost”
- Podemos fazer então uma chamada assim: localhost:9080

Vou mostrar isso na prática, vou abrir o Google Chrome (já que o nginx é um servidor web) e vou tentar chamar o container inserindo na barra de endereços “localhost:9080”:



Conseguimos criar uma imagem à partir de um Dockerfile, subir um container usando esta imagem recém criada e testamos a chama ao container.

Agora vamos adentrar em mais um detalhe importante sobre as imagens: as tais das camadas.

O tal do “Layered system” significa que o que estamos trabalhando nas imagens será um sistema em camadas, cada vez que uma imagem é utilizada essas camadas representam tudo o que a imagem tem, e o que é feito para que o container suba com o que indicamos no Dockerfile.

Para termos uma idéia de como ficaram essas camadas da nossa imagem recém criada de exemplo, podemos usar o seguinte comando para ver quais camadas tem nossa imagem:



## docker image inspect meunginx:latest

```

C:\Windows\System32\cmd.exe
Use 'docker scan' to run Snyk tests against images to find vulnerabilities and learn how to fix them
C:\Users\event\Documents\Gitlab\descomplica\docker\01-Dockerfile>docker image inspect meunginx:latest
[
  {
    "Id": "sha256:6b7bf5e3518aa0e670503f4c4873b0c09d2e990a6cc3408cbdd1b8d62827c54",
    "RepoTags": [
      "meunginx:latest"
    ],
    "RepoDigests": [],
    "Parent": "",
    "Comment": "buildkit.dockerfile.v0",
    "Created": "2022-06-05T14:24:09.434781876Z",
    "Container": "",
    "ContainerConfig": {
      "Hostname": "",
      "Domainname": "",
      "User": "",
      "AttachStdin": false,
      "AttachStdout": false,
      "AttachStderr": false,
      "Tty": false,
      "OpenStdin": false,
      "StdinOnce": false,
      "Env": null,
      "Cmd": null,
      "Image": "",
      "Volumes": null,
      "WorkingDir": "",
      "Entrypoint": null,

```

```

    "Labels": null
  },
  "Architecture": "amd64",
  "Os": "linux",
  "Size": 131633271,
  "VirtualSize": 131633271,
  "GraphDriver": {
    "Data": {
      "LowerDir": "/var/lib/docker/overlay2/b9bce61e17ae214d679d65fc814cbd905950188e1befdeb317c8dccee58fbdb/d
      iff",
      "MergedDir": "/var/lib/docker/overlay2/3vb0mx3r1jrw8jhace8j6m2b0/merged",
      "UpperDir": "/var/lib/docker/overlay2/3vb0mx3r1jrw8jhace8j6m2b0/diff",
      "WorkDir": "/var/lib/docker/overlay2/3vb0mx3r1jrw8jhace8j6m2b0/work"
    },
    "Name": "overlay2"
  },
  "RootFS": {
    "Type": "layers",
    "Layers": [
      "sha256:11802771315338383fcfa68b27037b7fbb464c91fdd7e79963877300466f28a0",
      "sha256:f1edd56faa6cf21dcfd8a363bc72612c58b2dfa5b82f58643eb4a1f395385704"
    ]
  },
  "Metadata": {
    "LastTagTime": "2022-06-05T19:42:23.992699615Z"
  }
}
C:\Users\event\Documents\Gitlab\descomplica\docker\01-Dockerfile>

```

O comando “docker image inspect” é justamente para vermos todas as informações da imagem, na parte das informações do tipo “layers”, estão as camadas da imagem, o parâmetro “meunginx:latest” indica de qual imagem queremos essas informações.

Então na prática esta imagem que criamos tem 2 camadas, a primeira camada é a da imagem base (o Ubuntu 21.10) que usamos. Todas instruções que colocamos no Dockerfile produzirão mais uma camada que estará “em cima” da primeira camada.



Ou seja: quanto mais builds fizermos e mais referências de imagens colocarmos, mais camadas colocamos por cima do que já tinha, quanto mais camadas maior a imagem ficará.

Aí fica a pergunta: e se nossa imagem precisar ter várias versões?

Há 2 abordagens: ou você literalmente substitui a imagem atual, tendo que destruir todos os containers atrelados a esta imagem, remover a imagem e criar a imagem novamente, ou você pode usar tags de versão de imagem.

Repare no comando que usamos:

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

O latest indica a versão da imagem, se tivermos um fluxo em que possamos construir várias imagens, podemos utilizar também o seguinte:

```
docker build -t meunginx:1.0 .
```

```
docker build -t meunginx:1.1 .
```

```
docker build -t meunginx:1.2 .
```

```
docker build -t meunginx:2.0 .
```

Geralmente a palavra “latest” significa ser “a última versão”, ou “a versão mais recente” e tudo bem usar ela para indicar que é a última versão.

Além da “latest”, você pode ir armazenado várias versões como no exemplo acima, supondo que haja alguma diferença em cada versão.

Assim ao subir um container, podemos escolher uma das versões e não obrigatoriamente a última.

Isso é útil quando você tem algo em produção e aí surgem versões mais novas, mas você não quer ir direto para a última versão (que pode ter bugs desconhecidos,

justamente por ser a última), então você pode optar por uma versão “mais estável”.

Agora vou mostrar como remover uma imagem, neste caso nossa imagem “meunginx” de teste. A primeira coisa a se fazer é parar e excluir todos os containers que estejam usando essa nossa imagem de exemplo, é recomendável sempre parar um container antes de excluí-lo, para que o container tenha a oportunidade de terminar todas as tarefas da aplicação antes de parar a aplicação. Tentar excluir o container de forma abrupta não é uma boa prática. Podemos usar o seguinte comando para parar um container:

```
docker stop MeuNginx
```

“docker stop” é o comando do Docker para parar um container, “MeuNginx” serve como parâmetro para indicar o nome do container que desejamos parar.

Logo depois podemos usar o seguinte comando para verificar se o container parou mesmo:

```
docker container ls --all
```

O parâmetro “--all” indica que queremos listar todos os containers, inclusive os parados, se não colocarmos esse parâmetro, só serão listados os containers em execução.



```
Selecionar Prompt de Comando
Microsoft Windows [versão 10.0.19044.1700]
(c) Microsoft Corporation. Todos os direitos reservados.

C:\Users\evert>docker stop MeuNginx
MeuNginx

C:\Users\evert>docker container ls --all
CONTAINER ID   IMAGE                                COMMAND                  CREATED        STATUS        PORTS
NAME
59aac2c120d5   meunginx:latest                    "nginx"                 11 minutes ago Exited (0) 10 seconds ago
MeuNginx
5a3ada9ca439   mcr.microsoft.com/mssql/server:2019-latest "/opt/mssql/bin/per..." 10 hours ago   Exited (0) 18 hours ago
MeuSQLServer
30b5fc7df0ba   pythonapi:latest                  "uvicorn app.main:ap..." 44 hours ago   Up 16 hours    0.0.0.0:8000->8000/tcp
MeuPythonAPI
9f404476caaf   springbootapi:latest              "java -jar /usr/loca..." 44 hours ago   Up 16 hours    0.0.0.0:8000->8000/tcp
MeuSpringBootAPI
3e656dc98dd7   mysql:latest                       "docker-entrypoint.s..." 6 days ago     Up 16 hours    0.0.0.0:3306->3306/tcp, 33060/tcp
MeuMySQL
9dad8ee4cd75   oracle/database:18.4.0-ee        "/bin/sh -c 'exec $@" 6 days ago     Up 16 hours (healthy) 0.0.0.0:1521->1521/tcp, 0.0.0.0:5500->5500/tcp
MeuOracle
0ea109ef14a3   gitlab/gitlab-runner:latest       "/usr/bin/dumb-init ..." 2 weeks ago    Exited (0) 13 days ago
gitlab-runner

C:\Users\evert>
```

A coluna “STATUS” do resultado nos mostra que o nosso container de exemplo não está executando. A palavra “Exited (0)” indica que o container foi parado.

Agora podemos excluir o container, o seguinte comando exclui o container:

```
docker rm MeuNginx
```

“docker rm” é o comando para excluir um container, “MeuNginx” serve como parâmetro para indicar o nome do container que desejamos excluir.

Se executarmos novamente o comando para listar todos os containers, você verá que o nosso container de exemplo “MeuNginx” sumiu, significa que foi excluído com sucesso:

```
docker container ls --all
```



```
Prompt de Comando
C:\Users\levert>docker rm MeuNginx
MeuNginx
C:\Users\levert>docker container ls --all
CONTAINER ID   IMAGE                                COMMAND                  CREATED        STATUS        PORTS
NAME
5a5ada9ca439   mcr.microsoft.com/mssql/server:2019-latest  "/opt/mssql/bin/per..."  19 hours ago   Exited (0) 19 hours ago
MeuSQLServer
3085fc7df0ba   pythonapi:latest                       "unicorn app.main:ap..."  44 hours ago   Up 16 hours   0.0.0.0:8000->8000/tcp
MeuPythonAPI
9f49447bcaef   springbootapi:latest                  "java -jar /usr/loca..."  44 hours ago   Up 16 hours   0.0.0.0:8000->8000/tcp
MeuSpringBootAPI
268585598dd7   mysql:latest                           "docker-entrypoint.s..."  6 days ago     Up 16 hours   0.0.0.0:3306->3306/tcp, 33060/tcp
MeuMySQL
9dad8ee4cd75   oracle/database:18.4.0-ee             "/bin/sh -c 'exec $@"..."  6 days ago     Up 16 hours (healthy)  0.0.0.0:1521->1521/tcp, 0.0.0.0:5500->5500/tcp
MeuOracle
9ea189ef14a3   gitlab/gitlab-runner:latest            "/usr/bin/dumb-init ..."  2 weeks ago    Exited (0) 13 days ago
gitlab-runner
C:\Users\levert>
```

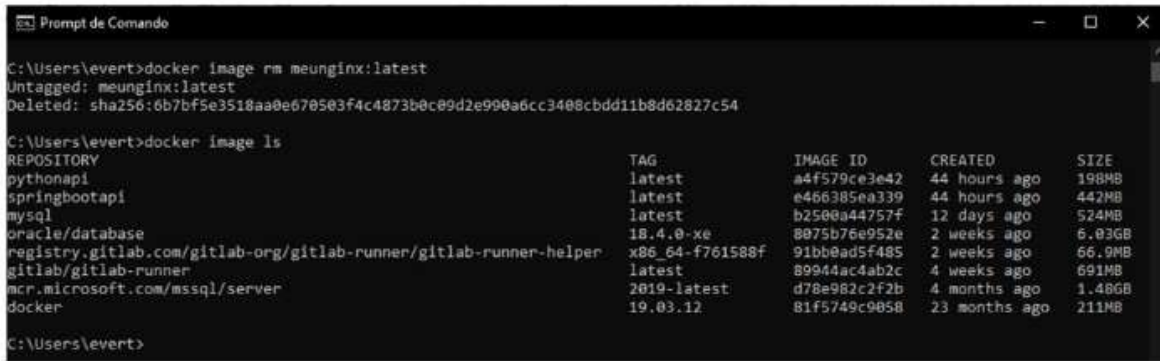
Agora sim podemos excluir a imagem que criamos, conseguimos isso através do seguinte comando:

```
docker image rm meunginx:latest
```

“docker image” é o comando do Docker para trabalharmos com imagens, o parâmetro “rm” indica que é para excluir uma imagem e o parâmetro “meunginx:latest” indica qual imagem e tag que é para excluir.

Para verificar se a imagem foi excluída, podemos executar o seguinte comando que lista as imagens disponíveis no repositório local do Docker:

```
docker image ls
```



```
C:\Users\event>docker image rm meunginx:latest
Untagged: meunginx:latest
Deleted: sha256:6b7bf5e3518aa0e670503f4c4873b0c09d2e990a6cc3408cbdd11b8d62827c54

C:\Users\event>docker image ls
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
pythonapi	latest	a4f579ce3e42	44 hours ago	198MB
springbootapi	latest	e466385ea339	44 hours ago	442MB
mysql	latest	b2500a44757f	12 days ago	524MB
oracle/database	18.4.0-xe	8075b76e952e	2 weeks ago	6.03GB
registry.gitlab.com/gitlab-org/gitlab-runner/gitlab-runner-helper	x86_64-f761588f	91bb0ad5f485	2 weeks ago	66.9MB
gitlab/gitlab-runner	latest	89944ac4ab2c	4 weeks ago	691MB
mcr.microsoft.com/mssql/server	2019-latest	d78e982c2f2b	4 months ago	1.40GB
docker	19.03.12	81f5749c9058	23 months ago	211MB

```
C:\Users\event>
```

Pronto! Até aqui vimos um exemplo prático de como construir uma imagem do zero, como usar a imagem para subir um container, como funcionam as camadas de uma imagem, como trabalhar com as tags e como remover uma imagem do repositório local.

### Atividade Extra

Para se aprofundar no assunto desta aula leia o capítulo 8 da bibliografia de referência: [VITALINO, J. F. N.; CASTRO, M. A. N. Descomplicando o docker. 2.ed. Brasport: 2018](#)

Como bibliografia complementar com maior detalhamento sobre imagens e camadas, leia o capítulo 6, sub capítulo “Images and layers” da seguinte referência: POULTON, Nigel. Docker Deep Dive: Zero to Docker in a single book (English Edition). May 2020 ed. Nigel Poulton: 2020

## Referência Bibliográfica

- VITALINO, J. F. N.; CASTRO, M. A. N. Descomplicando o docker. 2.ed. Brasport: 2018.

**Ir para exercício**