

Índice

Exercícios práticos de laboratórios.....	2
Hack150 – Iniciando o Docker.....	2
Hack151 – Baixando uma imagem do Linux com git pull.....	2
Hack152 – Listando as imagens disponíveis no Docker.....	2
Hack153 – Executando uma imagem do Docker “Olá Mundo”	3
Hack154 – Listando os containers disponíveis no Docker.....	3
Hack155 – Executando um container de modo interativo.....	3
Hack156 – Executando um container como deamon.....	4
Hack157 – Criando arquivo com requisitos instalados pelo pip.....	4
Hack158 – Criando um programa Python para execução containerizada.....	5
Hack159 – Criando um Dockerfile.....	5
Hack160 – Criando uma imagem com o Docker Build e Dockerfile.....	6
Hack161 – Criando e executando container do Python.....	6
Hack162 – Acessando o container via console.....	7
Hack163 – Executando comando no container via docker exec.....	7
Hack164 – Apagando containers que já estão inutilizados.....	8
Hack165 – Apagando imagens que já estão inutilizadas.....	8
Hack166 – Desafio.....	9

Exercícios práticos de laboratórios.

Cria uma pasta em seu ambiente para armazenar estes Hacks.

Hack150 – Iniciando o Docker.

1. As máquinas já tem o Docker Quick Start, executar o mesmo e ele vai levantar uma instância do docker e liberar uma interface de acesso

Hack151 – Baixando uma imagem do Linux com git pull.

1. Agora vamos baixar do repositório do container uma imagem com sistema operacional GNU/Linux da distribuição Ubuntu (Canonical) versão 18.04.

2. Acesse o console do docker e digite o comando seguir:

```
$ docker pull ubuntu:18.04
```

3. O Docker vai realizar o download da imagem do hub de imagens (repositório)

```
18.04: Pulling from library/ubuntu
38e2e6cd5626: Pull complete
705054bc3f5b: Pull complete
c7051e069564: Pull complete
7308e914506c: Pull complete
Digest: sha256:945039273a7b927869a07b375dc3148de16865de44dec8398672977e050a072e
Status: Downloaded newer image for ubuntu:18.04
```

4. Após realizado o download ele vai mostrar o status confirmando que uma nova imagem está disponível.

Hack152 – Listando as imagens disponíveis no Docker.

1. Para consultar as imagens que já temos em nosso Docker temos o comando **ls** para mostrar imagens, execute conforma a seguir

```
$ docker images ls
```

2. Serão mostradas todas as imagens existentes em seu docker e um id ou chave única chamado IMAGE ID da sua imagem.

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
ubuntu	18.04	20bb25d32758	11 days ago	87.5MB

3. Com base em uma imagem podemos iniciar agora containers.

Hack153 – Executando uma imagem do Docker “Olá Mundo”.

1. Quando executar uma imagem no Docker ele sobe a imagem executa, termina a execução e encerra as atividades, a mesma somente fica ativa se algum comando estiver em execução, diferente de uma VM.
2. Neste exemplo vamos rodar a imagem que criamos e fazer ela simplesmente escrever “Olá Mundo”, para subir o container usaremos o comando docker run

```
$ docker run --name container-ola-mundo ubuntu:18.04 "echo" "Hello World"
```

3. Será criado um container com o nome **container-ola-mundo** e simplesmente será impresso o resultado do comando echo e o container finalizado

```
Hello World
```

Hack154 – Listando os containers disponíveis no Docker.

1. Para consultar os containers que já temos em nosso Docker temos o comando **ps**, este comando mostrará apenas as máquinas em execução (que em nosso caso neste momento não temos nenhuma):

```
$ docker container ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
--------------	-------	---------	---------	--------	-------	-------

2. Para listar os container mesmo sem estarem em execução use a opção -a do comando ps.

```
$ docker container ps -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
41b8cf0881a7	ubuntu:18.04	"echo 'Hello World'"	16 minutes ago	Exited (0) 16 minutes ago		container-ola-mundo

3. Reparem que diversas informações, que usaremos na sequência estão na listagem.

Hack155 – Executando um container de modo interativo.

1. Para que acessemos o container e ele não saia de execução o modo interativo é uma forma, e pode ser realizado adicionando os parametros -it ao comando run conforma a seguir:

```
$ docker run -it --name container-ola-mundo2 ubuntu:18.04 /bin/bash
```

2. Repare que criamos container novo neste caso chamado container-ola-mundo2 (logo veremos mais detalhe sobre isso), e o terminal ficou ativo para execução de comando shell.

```
root@f1eb083c3c4e:/#
```

3. No terminal seu acesso é de root e aparece também o ID do container do Docker (que é refletido no nome da máquina)
4. Para sair digite CTRL+D ou simplesmente exit (o bash encerra e o container é finalizado)

Hack156 – Executando um container como deamon.

1. Podemos iniciar um container e mantê-lo inicializado com o parametro -d , novamente criamos um container novo com base na imagem do ubuntu:18.04 (container-ola-mundo3).

```
$ docker run -d --name container-ola-mundo3 ubuntu:18.04 /bin/bash -c "while true; do echo ola; sleep 1; done"
```

2. Esta container vai ficar em loop eterno com o while e executando como daemon. Agora podemos usar o comando docker ps conforme a seguir , para ver os containers em execução:

```
$ docker container ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
bdee21159f45	ubuntu:18.04	"/bin/bash -c 'while..."	2 seconds ago	Up 2 seconds		container-ola-mundo3

3. No momento vamos deixar este container em execução.

Hack157 – Criando arquivo com requisitos instalados pelo pip.

1. Crie um diretório em seu computador com o nome docker e acesse este diretório.
2. Crie um arquivo onde vamos adicionar os requisitos que precisaremos (no momento somente o servidor flask de HTTP), e chame este arquivo de **requirements.txt** (este arquivo poderia se chamar com qualquer nome, referenciado no programa, não é um nome padrão como o Dockerfile)

```
flask
```

<https://github.com/marciojv/hacks-cognitives-plataforms/blob/master/docker/>

[requirements.txt](#)

Hack158 – Criando um programa Python para execução containerizada.

1. Vamos criar um programa bem simples que possa ser acessado por HTTP no container, nosso programa vai apenas escrever “Olá Mundo”. Crie o arquivo **index.py** com as instruções a seguir:

```
from flask import Flask
app = Flask(__name__)
@app.route("/")
def hello():
    return "Olá Mundo!"
if __name__ == "__main__":
    app.run(host="127.0.0.1", port=int("5000"), debug=True)
```

<https://github.com/marciojv/hacks-cognitives-plataforms/blob/master/docker/index.py>

2. O programa será aberto em uma interface de HTTP na porta 5000, caso tenha algum bloqueio em sua rede pode mudar a porta da aplicação.

Obs. Para Docker usando máquinas Windows e Mac mude o ip para 127.0.0.1

Hack159 – Criando um Dockerfile.

1. Agora vamos conteinizar uma aplicação e usaremos o DockerFile para criar uma imagem com Python e um container rodando uma aplicação simples em Python/Flash (Servidor HTTP)
2. Neste diretório crie o arquivo com o nome **Dockerfile** e adicione o conteúdo a seguir:

```
FROM python:alpine3.7
COPY . /app
WORKDIR /app
RUN pip install -r requirements.txt
EXPOSE 5000
CMD python ./index.py
```

<https://github.com/marciojv/hacks-cognitives-plataforms/blob/master/docker/Dockerfile>

Hack160 – Criando uma imagem com o Docker Build e Dockerfile.

1. Agora vamos criar a imagem com os pre-requisitos executando o DockerFile.
2. Acesse o diretório que criamos os arquivos Dockerfile, requirements.txt e index.py, (este diretório foi criado com o nome docker) , e execute o seguinte comando no mesmo , usando o console.

```
docker build --tag my-python-app .
```

3. Este comando pode demorar alguns minutos para execução , pois ele vai baixar uma imagem com Linux/Python e depois instalar o flask.

```
Removing intermediate container 06f9a89cb7c2
----> 06c23c319cc1
Step 4/6 : RUN pip install -r requirements.txt
----> Running in d2fdd06b44b
Collecting flask (from -r requirements.txt (line 1))
  Downloading https://files.pythonhosted.org/packages/7f/e7/08578774ed4536d3242b14dab4696386634607af024ea997202cd0edb4b/Flask-1.0.2-py2.py3-none-any.whl (91kB)
Collecting Werkzeug>=0.14 (from flask->-r requirements.txt (line 1))
  Downloading https://files.pythonhosted.org/packages/20/c4/12e3e56473e52375aa29c4764e70d1b0f3fa6682bef80aae04fe335243/Werkzeug-0.14.1-py2.py3-none-any.whl (322kB)
Collecting Jinja2>=2.10 (from flask->-r requirements.txt (line 1))
  Downloading https://files.pythonhosted.org/packages/7f/ff/ae04bacdfc95f27a016a7bed8e886763ba4d277a78ca76f32659228a731/Jinja2-2.10-py2.py3-none-any.whl (126kB)
Collecting click>=5.1 (from flask->-r requirements.txt (line 1))
  Downloading https://files.pythonhosted.org/packages/fa/37/45185cb5abbc30d7257104c434fe0b07e5a195a6847506c074527aa599ec/Click-7.0-py2.py3-none-any.whl (81kB)
Collecting itsdangerous>=0.24 (from flask->-r requirements.txt (line 1))
  Downloading https://files.pythonhosted.org/packages/76/ae/44b03b253d6f4ade317f32c24d100b3b35c2239807046a4c953c7b89fa49e/itsdangerous-1.1.0-py2.py3-none-any.whl
Collecting MarkupSafe>=0.23 (from Jinja2>=2.10->flask->-r requirements.txt (line 1))
  Downloading https://files.pythonhosted.org/packages/ac/7e/1b4c2e05809a4414ebce0892fe1e32c14ace86ca7d50c70f00979ca9b3a3/MarkupSafe-1.1.0.tar.gz
Building wheels for collected packages: MarkupSafe
  Building wheel for MarkupSafe (setup.py): started
  Building wheel for MarkupSafe (setup.py): finished with status 'done'
  Stored in directory: /root/.cache/pip/wheels/01/23/64/51095ea52825dc116a55f37043f49be0939bcf603de54e5cde
Successfully built MarkupSafe
Installing collected packages: Werkzeug, MarkupSafe, Jinja2, click, itsdangerous, flask
Successfully installed Jinja2-2.10 MarkupSafe-1.1.0 Werkzeug-0.14.1 click-7.0 flask-1.0.2 itsdangerous-1.1.0
Removing intermediate container d2fdd06b44b
----> ba5c0a087e4
Step 5/6 : EXPOSE 5000
----> Running in d6906f1b43ab
Removing intermediate container d6906f1b43ab
----> 96fa124fdd7a
Step 6/6 : CMD python ./index.py
----> Running in 4c4e9fa9688c
Removing intermediate container 4c4e9fa9688c
----> 21f38919e5c9
Successfully built 21f38919e5c9
Successfully tagged my-python-app:latest
```

4. Confira se a imagem foi criada com o comando docker image com parâmetro ps

```
docker image ps
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
my-python-app	latest	21f38919e5c9	About a minute ago	91.7MB
python	alpine3.7	00be2573e9f7	4 days ago	81.3MB
ubuntu	18.04	20bb25d32758	12 days ago	87.5MB

5. Veja na imagem que ele fez o download da imagem chamada python:alpine3.7 e com base nesta criou uma outra imagem chamada **my-python-app** resultado do nome que passamos na linha de comando.

Hack161 – Criando e executando container do Python

1. Agora vamos criar um container com com as dependências e python baseado na imagem previamente criada.

```
docker run -tid --name python-app -p 5000:5000 my-python-app
```

2. Com um browser de sua máquina acesse <http://localhost:5000> e será simplesmente escrito Olá Mundo.



3. Com isso seu aplicativo Python está na máquina local mas sua execução em um container.

Hack162 – Acessando o container via console

1. Para acessar o container podemos usar o comando docker exec.

```
docker exec -ti python-app /bin/sh
```

2. Você acessará no console bash do container, no diretório raiz da aplicação configurada (/app)

Hack163 – Executando comando no container via docker exec

1. Podemos tanto acessar o console do container como executar comandos no ambiente chamando com o docker exec

```
docker exec -ti python-app /bin/sh -c 'top -b -n 1'
```

2. O resultado retorna para nosso console de chamadas


```
Mem: 7910620K used, 174076K free, 894840K shrd, 493996K buff, 1269964K cached
CPU: 15% usr 2% sys 0% nic 82% idle 0% io 0% irq 0% irq
Load average: 0.88 1.07 1.18 4/1175 125
  PID PPID USER STAT VSZ %VSZ CPU %CPU COMMAND
   10   9 root  S   95344 1% 0 0% /usr/local/bin/python ./index.py
    9    1 root  S   94552 1% 1 0% python ./index.py
    1    0 root  S   1564 0% 0 0% /bin/sh -c python ./index.py
   118   0 root  S   1564 0% 0 0% /bin/sh -c top -b -n 1
   125  118 root  R   1504 0% 0 0% top -b -n 1
```

3. Apesar do exemplo do top deixar claro que podemos chamar qualquer comando no Container, temos um comando mais interessante para monitoramento se for o caso o docker stats mostra todos os estados dos containers

```
docker stats
```

4. Será apresentada uma tela com os recursos sendo consumidos pelo docker nos containers.

CONTAINER ID	NAME	CPU %	MEM USAGE / LIMIT	MEM %	NET I/O	BLOCK I/O	PIDS
6415d1700681	python-app5	1.83%	34.48MiB / 7.71GiB	0.44%	10.4kB / 0B	4.92MB / 0B	4

Hack164 – Apagando containers que já estão inutilizados

1. Nos testes que vamos fazendo um comando errado pode criar um container mesmo que haja algum erro na igagem ou inicialização etc , e temos de removê-los para uma boa manutenção do docker. O comando a seguir elimina os containers “mortos”

```
docker rm -v $(docker ps -a -q -f status=exited)
```

2. Todas os containers que não tiverem mais em uso serão eliminados, ficarão ativas somente nosso container que está com o while no shell e a com python/flash

Hack165 – Apagando imagens que já estão inutilizadas.

1. Imagens não utilizadas podem ser eliminada.

```
docker rmi $(docker images -f dangling=true -q)
```


Hack166 – Desafio.

1. Cria uma imagem que contenha o Python e as bibliotecas do Watson e cria uma imagem que executa algum serviço cognitivo ao iniciar o container.