Certificación avanzada en

DevOps

Docker

UBA FCEFyN mundosE PEOPLE & BUSINESS SCHOOL

Docker login

```
$  docker login -u <USUARIO DE DOCKERHUB>
Password:
```

```
WARNING! Your password will be stored unencrypted in
~/.docker/config.json.
```

Dockerfile:

FROM: Setea la imagen base a partir de la cual vamos a trabajar

Archivo: 01/Dockerfile

```
$ cd 01
$ docker build -t local-hello-world:linux .
```


```
$ cd 01
$ docker build -t local-hello-world:linux .
```

Dockerfile:

RUN: Ejecuta comandos en una nueva capa, mientras la imagen se va creando.

ENTRYPOINT: Es el comando que se va a ejecutar siempre que e corra **docker run**

```
Archivo: 02/Dockerfile
```

```
$ cd 02
$ docker build -t wget-downloader:local .
```

```
$ docker run wget-downloader:local \
https://www.google.com/images/branding/googlelogo/1x/googlelogo_color_272x92dp.png
```

El archivo se pierde al finalizar el container!!!....

Volumenes:

Los volúmenes nos permiten enlazar un directorio local al container. De esta manera, la información perisiste al momento de eliminar el container:

```
$ docker run \
-v ${PWD}/image_download:/image_download \
wget-downloader:local \
-O /image_download/download.png \
https://www.google.com/images/branding/googlelogo/1x/googlelogo_color_272x92dp.png
```

```
$ ls image_download
```

SIEMPRE lo que sigue a la imagen es el comando. Como el entrypoint es “wget”, lo que esta en amarillo sería lo que sigue al wget… por lo tanto el comando que ejecuta es:

```
wget -O /image_download/download.png \
https://www.google.com/images/branding/googlelogo/1x/googlelogo_color_272x92dp.png
```

Dockerfile:

WORKDIR: Setea el directorio en el cual vamos a trabajar dentro del container

COPY: Copia el contenido local al container.

CMD: Comando que ejecuta por defecto, a menos que especifiquemos otra cosa. A diferencia del ENTRYPOINT, este ejecuta si o si y lo que pongamos en la linea de comandos se agrega como parámetro al comando.

```
Archivo: 03/Dockerfile
```

```
$ cd 03
$ docker build -t node-web-app .
```

```
$ docker run --name node-app -p 8080:8080 -d node-web-app
```

```
$ curl http://127.0.0.1:8080
$ docker stop node-app && docker rm node-app
```

PROBLEMA!!: Estamos pusheando a la imagen credenciales privadas

```
$ docker run --name node-app -p 8080:8080 node-web-app \
cat credentials.txt
```

```
cion super segura
42
```

```
$ docker stop node-app && docker rm node-app
```

Agregamos un archivo **.dockerignore** donde le especificamos que es lo que queremos **IGNORE** en la imagen

--rm: Elimina el container apenas este se detiene

Archivo: 04/.dockerignore

```
$ cd 04
$ docker build -t node-web-app .
```

```
$ docker run --rm --name node-app -p 8080:8080 node-web-app \
cat credentials.txt
```

De que otra manera se les ocurre ver si esta el archivo **credentials.txt** dentro?

Agregamos un archivo **.dockerignore** donde le especificamos que es lo que queremos **IGNORE** en la imagen

--rm: Elimina el container apenas este se detiene

```
Archivo: 04/.dockerignore
```

```
$ cd 04
$ docker build -t node-web-app .
```

```
$ docker run --rm --name node-app -p 8080:8080 node-web-app \
cat credentials.txt
```

TIP… pueden usar volumenes… o mirar el siguiente slide

Existe una manera de ejecutar un comando dentro del container desde nuestra consola. Esto es muy útil para si por ejemplo queremos ver como quedo la imagen.

Iniciemos el container nevamente:

```
Carpeta: 04
```

```
$ docker run --rm --name node-app -p 8080:8080 -d node-web-app
```

Vamos a abrir un intérprete de comando en el container:

```
$ docker exec -it node-app /bin/bash
```

Lo que esta diciendo es, **docker** ejecuta un **exec** interactivo y con una terminal (**-it)** en el container **node-app.** Y el comando que vas a ejecutar es **/bin/bash.** Por lo que a partir de ahora etan dentro del container. Vean en que directorio estan por defecto (Lo que pusieron como WORKDIR) y revisen que archivos están… noten como lo que pusimos en el **.dockerignore no hay nada.**

ESTO ES PARA ABRIR UNA TERMINAL; PERO PUEDE SER CUALQUIER COMANDO!!!

EXPOSE:

Informa a Docker que el container escucha en cierto puertos **en tiempo de ejecución. CUIDADO!!**, informa que **escucha;** pero no hace el forward. Es como para declarar la intención de exponer ese puerto:

```
Carpeta 05/[Dockerfile_expose | Dockerfile_noexpose ]
```

Comparen la columna **PORTS** de ambas salidas:

```
$ cd 05
$ docker build -t nginx:expose -f Dockerfile_expose .
$ docker build -t nginx:noexpose -f Dockerfile_noexpose .
$ docker run --rm -d --name ng-expose nginx:expose
$ docker run --rm -d --name ng-noexpose nginx:noexpose
$ docker ps
```

```
$ docker stop  ng-expose ng-noexpose
```

ENV: Son variables de entorno que van a vivir durante el tiempo de build y ejecución del container.
ARG: Variables que solo van a estar disponible durante el **build del container.** Estas variables son útiles si por ejemplo, el build de nuestro container requiere que compilemos algo. Esto es, por ejemplo, si la compilación se hace durante el build, y necesitamos pasarle algún argumento al compilador, **ARG** es un buen caso de uso, ya que dicho argumento no lo necesitaré más cuando el container arranque.

```
Carpeta 06/Dockerfile
```

Dado que en este ejemplo **ENV,** dentro del Dockerfile, contiene el número de puerto, esta instrucción hace que esté disponible como variable de entorno dentro de la aplicación. Es por ese motivo que dentro del archivo **index.js** esta variable que define el puerto y puede ser invocada mediante la línea: **`port = process.env.PORT || 3080`**

```
$ cd 06
$ docker build -t envapp -f Dockerfile_env .
$ docker run --rm -d --name envapp -p 3000:3000 envapp:latest
```

Ahora, si **(1)** ejecutamos una llamada; pero mas importante, ejecutamos el comando **env** dentro del container que nos trae las variables de entorno vamos a ver que esta la variable **PORT = 3000** tal cual la definimos en el **Dockerfile**

```
$ curl http://127.0.0.1:3000
App Works !!!

$ docker exec -it envapp env | grep PORT
PORT=3000
```

MULTISTAGE BUILD: En algunos casos, el build es simplemente generar un ejecutable o un simple artifact que se ejecutará. En un multistage build, se divide en dos etapas, una de build y una de build del container, utilizando el artifact previamente definido.

La primer ventaja es modularidad ya que **queda más definida la parte de build del artifact y cualquier cambio afectaría a esa sección únicamente**.

Y dos, que únicamente se conserva la última capa para generar el artifact, descartando todas las anteriores.

```
$ docker build -t artifact-app .
$ docker run --rm -d --name artifact-app -p 3000:3000 artifact-app:latest
```

Desde un navegador accedan a: **http://127.0.0.1**

Dockerfiles a analizar:

- **MySQL:** *https://github.com/mysql/mysql-docker/blob/mysql-server/5.7/Dockerfile* ¿que es ese entrypoint?.
- **Postfix:** *https://github.com/bokysan/docker-postfix/blob/master/Dockerfile* Observen como utiliza ese **${BASE_IMAGE}.**

Como subimos la imagenes a nuestro repo?:
Algo que hay que entender es que **dentro del nombre de la imagen, también se define el repositorio.** Esto es, sigue la siguiente forma:

```
url_registry/usuario/imagen:tag
```

Entonces, esto es un ejemplo de una imagen que si queremos subirla al registry (push), no la vamos a estar subiendo a Docker hub

```
REPOSITORY                                  TAG
registry.access.redhat.com/ubi8/ubi-minimal   8.2
```

registro: **registry.access.redhat.com**
usuario **dentro del registry: ubi8**
Imagen: **ubi-minimal**
Tag: **8.2**

Subamos una imagen a nuestro repo **en dockerhub** entonces:

```
$ docker images
REPOSITORY          TAG       IMAGE ID            CREATED              SIZE
artifact-app    latest        bd70dd26cfca    25 minutes ago    142MB
```

Como dijimos antes, tiene la siguiente forma

```
url_registry/usuario/imagen:tag
```

Por lo que:

a) Si no especificamos registry (de la forma **usuario/imagen:tag)**, por defecto utilizará dockerhub como registry.
b) Si no especificamos ni registry ni usuario (de la forma **imagen:tag),** por defecto utilizará mi máquina local.

Por lo que si quisiéramos subir una imagen a dockerhub, dentro de mi usuario sería de la forma

```
usuario/imagen:tag
```

Por lo que si tenemos esta imagen:

```
$ docker images
REPOSITORY          TAG       IMAGE ID            CREATED              SIZE
artifact-app    latest        bd70dd26cfca    25 minutes ago    142MB
```

Tendremos que retagearla a lo siguiente (tomando mi usuario **adrabenche para este ejemplo)**. En su caso utilizarán el correspondiente a lo que crearon dentro de Dockerhub.

```
$ docker tag artifact-app:latest adrabenche/artifact-app:latest
```

Y listo. Cabe aclarar que:

1) Tenemos que tener permisos de escritura dentro de dicho repositorio de imagen en Dockerhub.
2) Si el repositorio no existe, creará uno.

```
$ docker push adrabenche/artifact-app:latest
```

Comandos y más comandos: Estos son algunos de los comandos que utilzarán con frecuencia:

- `docker system prune`
- `docker system prune -a`
- `docker history <imagen>`
- `docker build -t <imagen> . --no-cache`
- `docker inspect <container>`
- `docker build https://github.com/<gh_user>/<gh_repo>#<image>:<tag>`
- `docker stats`
- `docker system df`

¿Que más hay?

- **Techo World with NANA:** https://youtu.be/3c-iBn73dDE
- **Pelado Nerd:** https://www.youtube.com/c/PeladoNerd
- **Free code camp:** [https://www.freecodecamp.org | https://www.youtube.com/c/freecodecamp/videos]
- Busquen ejemplos de Dockerfiles de otros repositorios!!