

## Docker.

---



---

**Nombre del alumno:** José Ulises Vallejo Sierra

**Código:** 219747905

**Sección:** D06

**Curso:** Computación Tolerante a Fallas

**Nombre del profesor:** Michel Emanuel López Franco

## Realizar un programa que utilice Docker

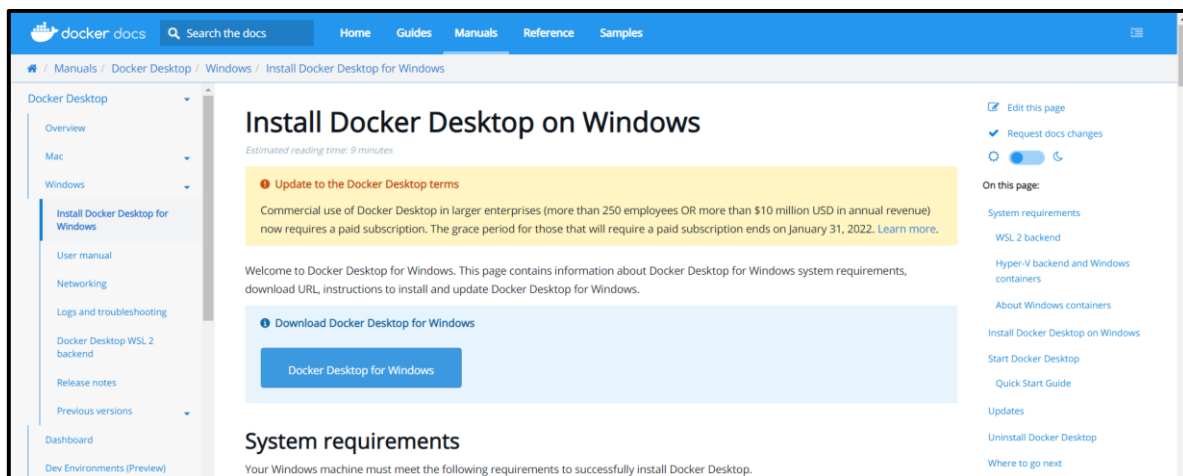
### Técnica a utilizar: Docker y Flask

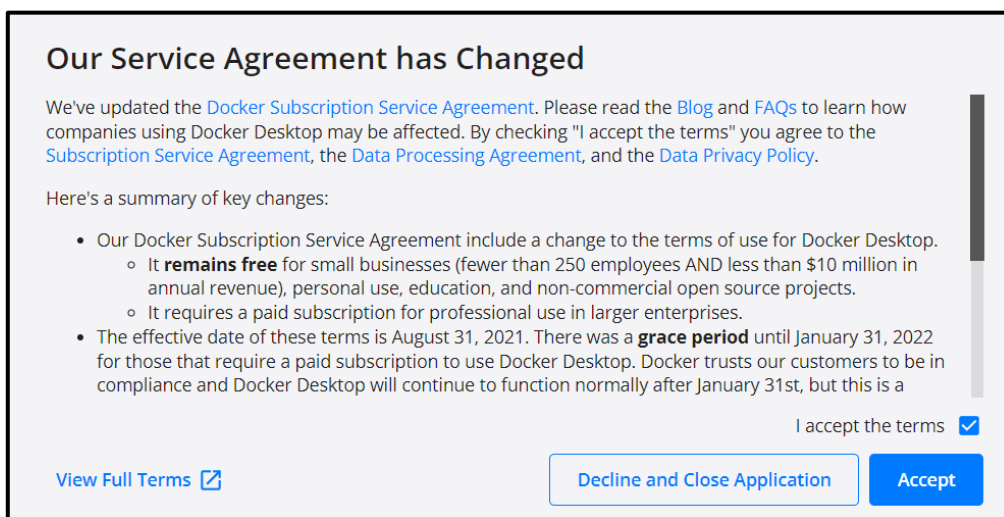
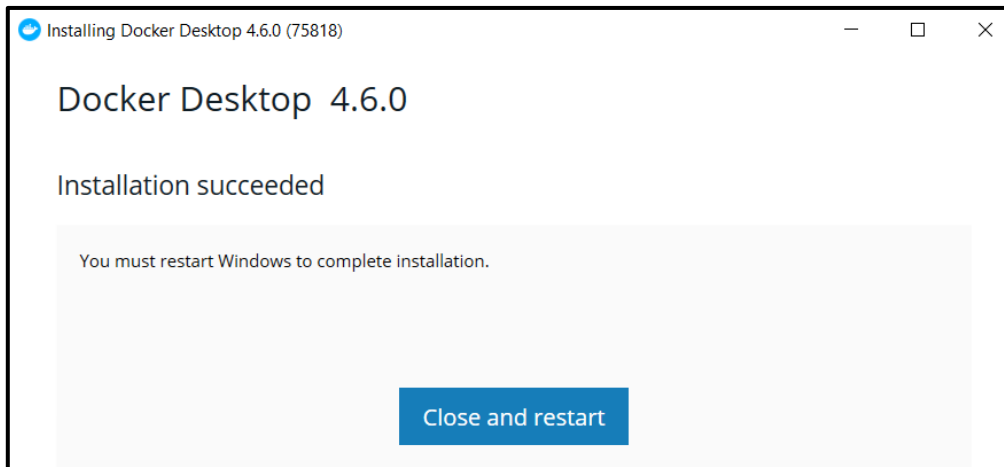
En esta ocasión se nos plantea un tema actual y novedoso que promete ser solución a diversos problemas o fallas en la computación y es una rama de la ingeniería que apoya a la tolerancia de estas fallas para mitigar los problemas presentados en las aplicaciones, pues la tecnología Docker utiliza el kernel de Linux y sus funciones, como los grupos de control y los espacios de nombre, para dividir los procesos y ejecutarlos de manera independiente. El propósito de los contenedores es ejecutar varios procesos y aplicaciones por separado para que se pueda aprovechar mejor la infraestructura y, al mismo tiempo, conservar la seguridad que se obtendría con los sistemas individuales y como dijimos, permitir que se puedan solventar errores en este caso de versiones al ya tener todas las aplicaciones programadas en el contenedor.

### Programa: Creación de una aplicación sencilla con Docker y Flask en python

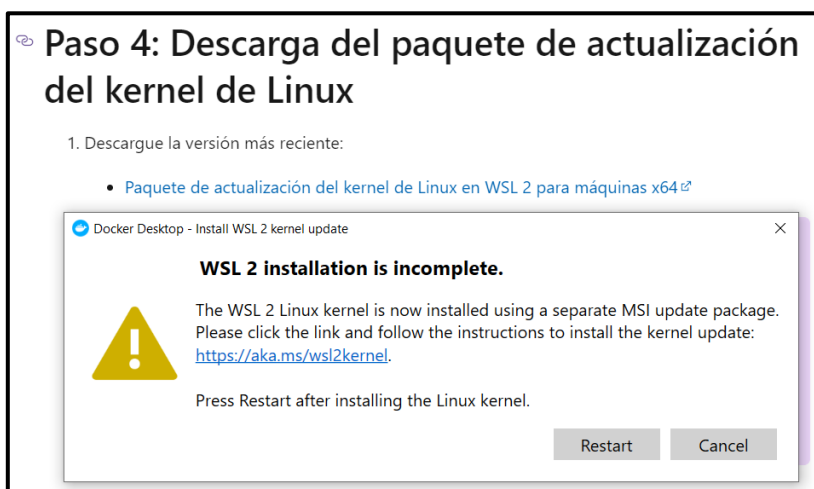
Usaremos Flask para desarrollar una aplicación sencilla en donde ejemplificaremos el uso y funcionamiento de los contenedores de Docker mediante las imágenes que crearemos pasando dicha aplicación dentro del contenedor respectivo.

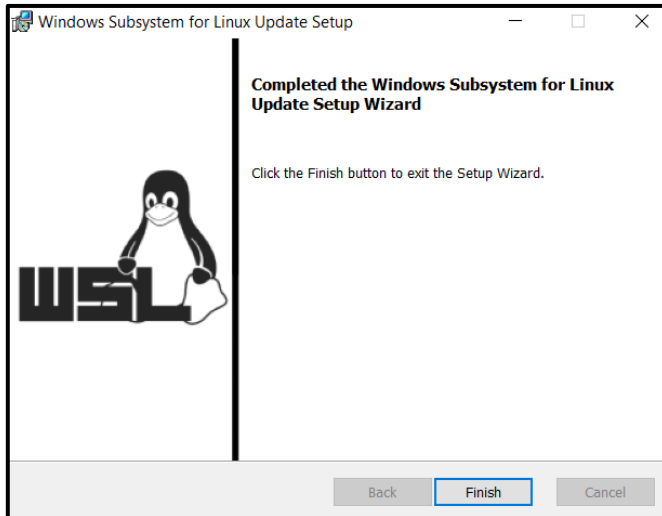
Para la instalación de Docker, vamos al sitio oficial y elegimos la versión de escritorio más reciente para Windows.



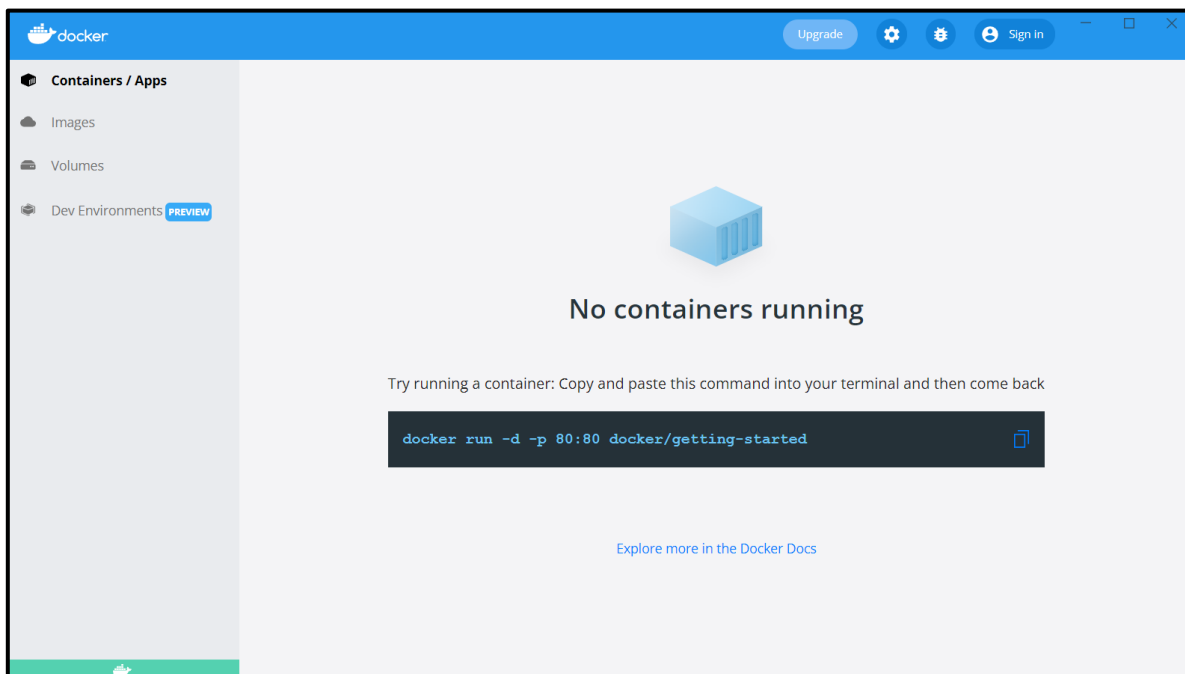



Posteriormente instalamos el Kernel update necesario para el uso de Docker y una vez instalado reiniciamos el equipo.





Ahora comprobamos que la aplicación de Docker abra correctamente y escribimos el comando para ver su versión y saber que se ha instalado correctamente.



 Símbolo del sistema

```
Microsoft Windows [Versión 10.0.19044.1586]
(c) Microsoft Corporation. Todos los derechos reservados.

C:\Users\ulise>docker --version
Docker version 20.10.13, build a224086

C:\Users\ulise>
```

Ahora creamos una carpeta llamada Docker en donde alojaremos todo el proyecto a crear y la abrimos con nuestro editor preferido (VS code en mi caso).



Abrimos la terminal del proyecto en Visual Studio para ejecutar comandos de Python y Docker e instalamos el entorno virtual mediante el paquete virtualenv.

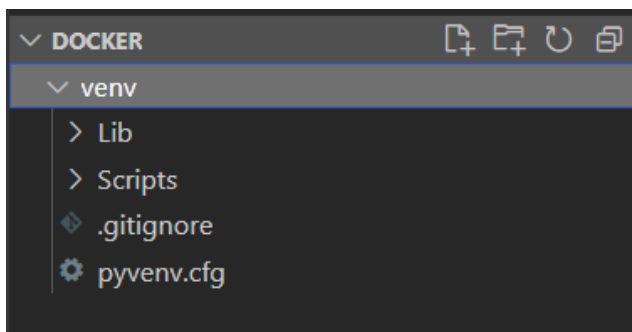
```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

(c) Microsoft Corporation. Todos los derechos reservados.

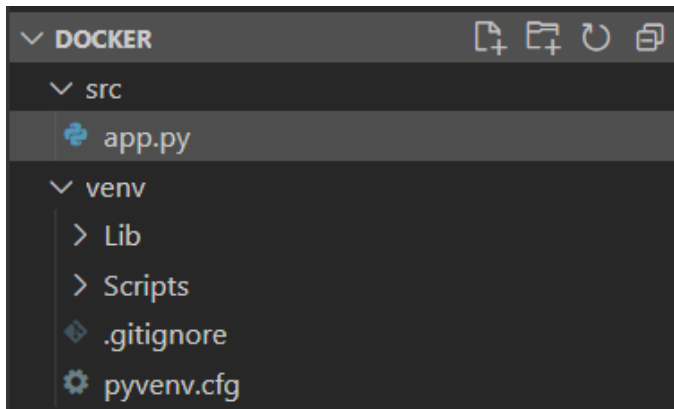
C:\Users\ulise\Escritorio\docker>pip install virtualenv
Collecting virtualenv
  Downloading virtualenv-20.13.4-py2.py3-none-any.whl (8.7 MB)
    |-----| 8.7 MB 1.3 MB/s
Collecting distlib<1,>=0.3.1
  Downloading distlib-0.3.4-py2.py3-none-any.whl (461 kB)
    |-----| 461 kB 6.4 MB/s
```

Creamos una carpeta llamada venv que será la que contenga toda la configuración de mi entorno virtual el cual ya tiene la carga de los archivos necesarios para el entorno.

```
C:\Users\ulise\Escritorio\docker>virtualenv venv
created virtual environment CPython3.9.2.final.0-64 in 8019ms
creator CPythonWindows(dest=C:\Users\ulise\Escritorio\docker\venv, clear=False, no_vcs_ignore=False, global=False)
seeder FromAppData(download=False, pip=bundle, setuptools=bundle, wheel=bundle, via=copy, app_data_dir=C:\Users\ulise\AppData\Local\pypa\virtualenv)
added seed packages: pip==22.0.4, setuptools==60.10.0, wheel==0.37.1
activators BashActivator,BatchActivator,FishActivator,PowerShellActivator,PythonActivator
```



Creamos además una nueva carpeta llamada src donde alojaremos todo el código del entorno y la aplicación, por lo que dentro creamos un archivo llamado app.py



Para utilizar el entorno virtual creado, ejecutamos desde consola el archivo actíivate.bat que se encuentra dentro de la carpeta venv y de la carpeta Scripts.

```
C:\Users\ulise\Escritorio\docker>cd venv  
C:\Users\ulise\Escritorio\docker\venv>cd Scripts  
C:\Users\ulise\Escritorio\docker\venv\Scripts>activate.bat  
(venv) C:\Users\ulise\Escritorio\docker\venv\Scripts>
```

Una vez activado el entorno virtual instalamos el framework flask y lo requerimos en el archivo app.py con las siguientes líneas de código.

```
(venv) C:\Users\ulise\Escritorio\docker\venv\Scripts>pip install flask  
Collecting flask  
  Downloading Flask-2.0.3-py3-none-any.whl (95 kB)  
    95.6/95.6 KB 1.8 MB/s eta 0:00:00  
Collecting click>=7.1.2  
  Downloading click-8.0.4-py3-none-any.whl (97 kB)  
    97.5/97.5 KB 5.8 MB/s eta 0:00:00  
Collecting Werkzeug>=2.0  
  Downloading Werkzeug-2.0.3-py3-none-any.whl (289 kB)
```

Aquí designamos cómo y donde se ejecutará la aplicación, para que sea accedida desde cualquier dirección le agregamos el host 0.0.0.0 y le asignamos el puerto 4000 dentro del localhost. Además, se ejecutará solo si es el archivo principal y le

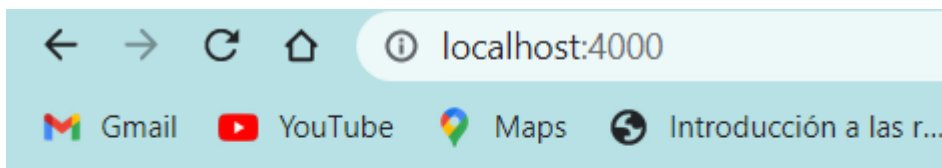
creamos la ruta principal con `@app.route` a través del método `get` y devolvemos un `jsonify` por medio de una función para convertir el objeto tipo `response` "hello world" a JSON.

```
app.py ×
src > app.py > ...
1  from flask import Flask, jsonify
2
3  app = Flask(__name__)
4
5  @app.route('/', methods=['GET'])
6  def ping():
7      return jsonify({"response": "hello world"})
8
9
10 if __name__ == '__main__':
11     app.run(host="0.0.0.0", port=4000, debug=True)
12
```

Una vez designado los parámetros para esta sencilla aplicación, la corremos desde el inicio de nuestro proyecto (carpeta Docker)

```
(venv) C:\Users\ulise\Escritorio\docker>python src/app.py
* Serving Flask app 'app' (lazy loading)
* Environment: production
  WARNING: This is a development server. Do not use it in a production deployment.
  Use a production WSGI server instead.
* Debug mode: on
* Restarting with stat
* Debugger is active!
* Debugger PIN: 128-968-638
```

Para comprobar que la aplicación arrancó correctamente dentro del Browser accedemos a `localhost:4000`

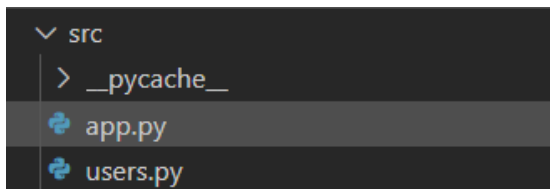


```
{
  "response": "hello world"
}
```

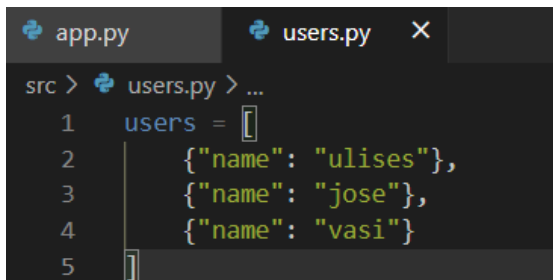
Ahora creamos una ruta extra para los usuarios con la finalidad de ejemplificar un mejor funcionamiento de Docker y los contenedores al usar un poco más de complejidad en la aplicación

```
9
10 @app.route('/users')
11 def usersHandler():
12     return jsonify({"users": users})
```

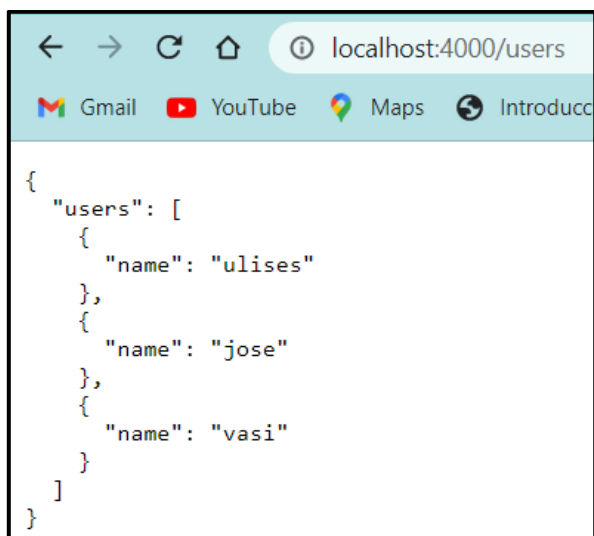
Para ello creamos un nuevo archivo llamado users.py en la carpeta src



Ingresamos una variable de usuarios con la lista de algunos usuarios y dicha variable la regresamos en la función usersHandler con jsonify



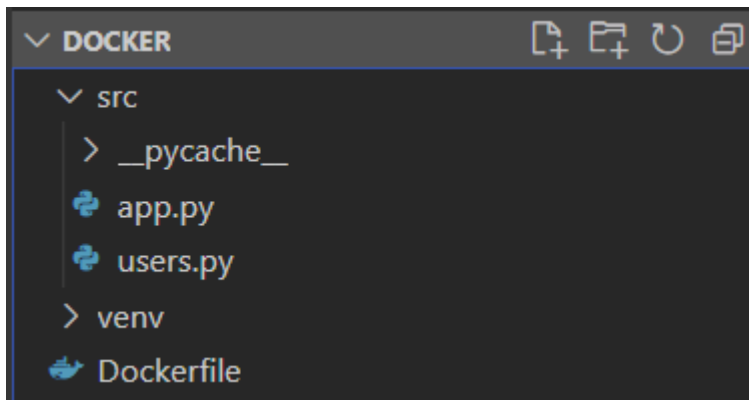
Verificamos la ruta creada y su información contenida



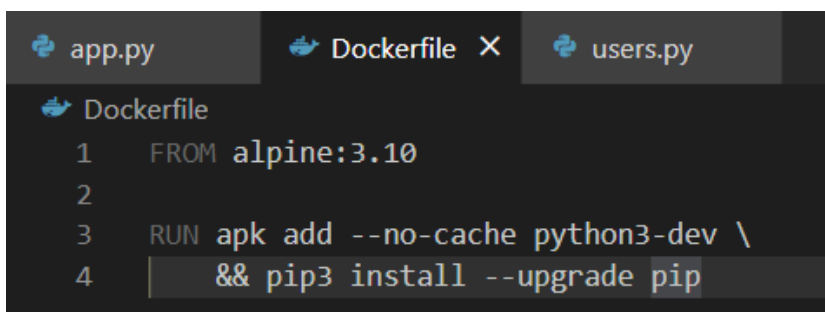


A partir de ahora haremos uso de Docker, suponiendo que queremos pasar el proyecto a otro desarrollador para continuar, pero existe el problema que dicho desarrollador no tiene Python ni los programas o versiones de programas requeridas para el proyecto, por lo que creamos los contenedores para colocar la aplicación y evitar todos estos problemas, ya que este contendrá los parámetros correctos de los programas y sus versiones requeridas para el desarrollo de la aplicación.

Para todo esto, creamos un archivo llamado Dockerfile que permitirá la creación de los contenedores por medio de imágenes Docker donde a partir de ellas se crearán los contenedores.



Dentro de Dockerfile ponemos los archivos requeridos para la creación de la aplicación, primero cargando alpine que fungirá como imagen minimalista de Docker basada en una distribución de Linux y que servirá como sistema operativo para instalar Python, pip y lo que se necesite en la aplicación con el comando RUN que precisamente ejecuta comandos.

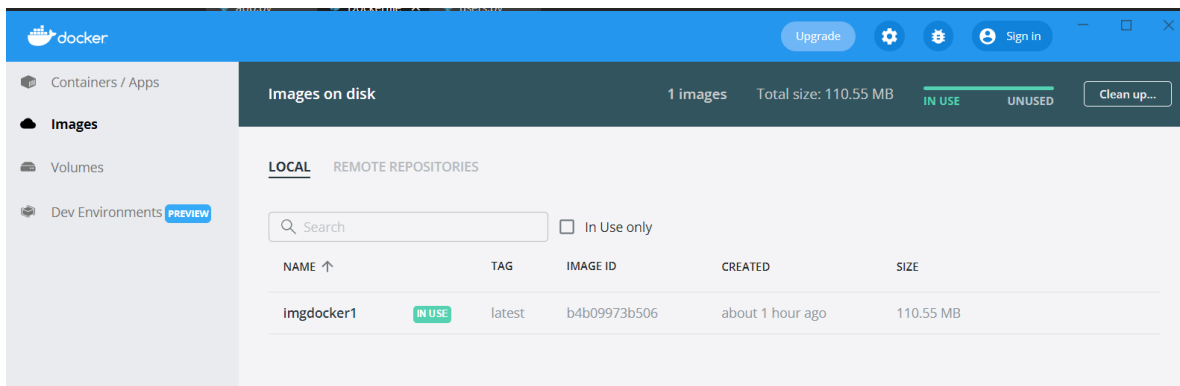


Ahora construimos la imagen conformada por los parámetros o pasos ya mencionados en Dockerfile, para ello se ejecuta el siguiente comando donde nuestra imagen se llamará imgdocker1

```
(venv) C:\Users\ulise\Escritorio\docker>docker build -t imgdocker1 -f Dockerfile .
[+] Building 1.7s (6/6) FINISHED
=> [internal] load build definition from Dockerfile
=> => transferring dockerfile: 31B
=> [internal] load .dockerignore
=> => transferring context: 2B
=> [internal] load metadata for docker.io/library/alpine:3.10
=> [1/2] FROM docker.io/library/alpine:3.10@sha256:451eee8bedcb2f029756dc3e9d73bab0e7943c1ac55cff3a4861c52a0fdd3e98
=> CACHED [2/2] RUN apk add --no-cache python3-dev && pip3 install --upgrade pip
=> exporting to image
=> => exporting layers
=> => writing image sha256:b4b09973b506adfd14b2bb839748dff7327ec97013c03a669077be11c8ea7313
=> => naming to docker.io/library/imgdocker1
```

Con el comando Docker images visualizamos que se haya creado correctamente la imagen y también desde nuestra app de Docker

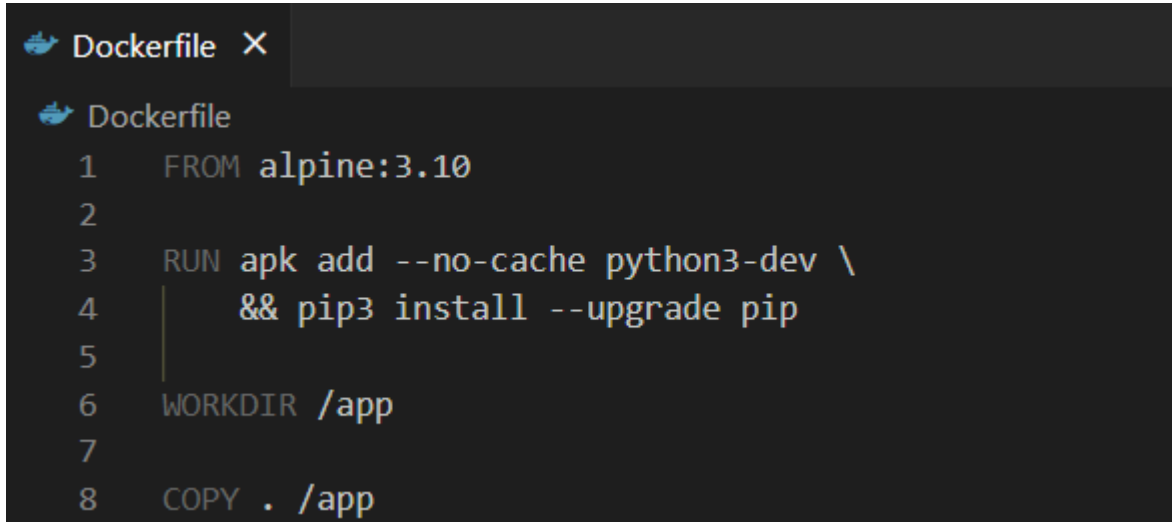
```
(venv) C:\Users\ulise\Escritorio\docker>docker images
REPOSITORY    TAG       IMAGE ID       CREATED        SIZE
imgdocker1    latest   b4b09973b506   About an hour ago   111MB
```



Ahora ejecutamos nuestra imagen creada en modo interactivo para ver como funciona el sistema creado por la imagen especificando la ejecución del Shell y escribir comandos ls o incluso crear carpetas para asegurar una interacción correcta con el sistema Linux. Consultamos también nuestras versiones de Python3 y pip que hemos instalado internamente en nuestro sistema o contenedor

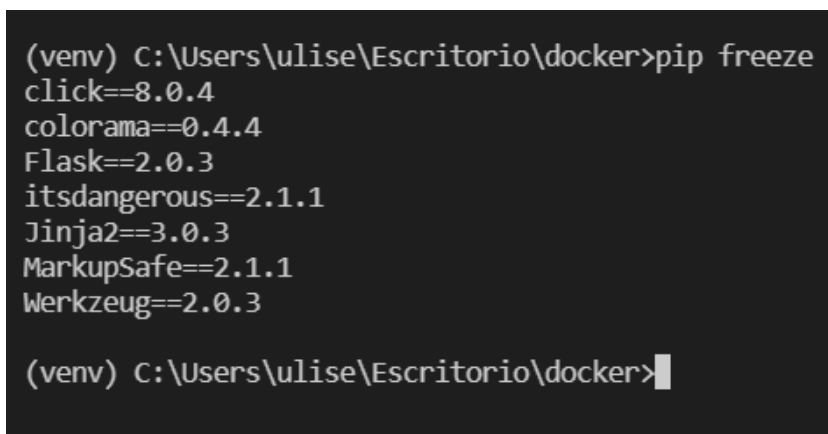
```
/ # mkdir testing
/ # ls
bin      etc      lib      mnt      proc     run      srv      testing  usr
dev      home    media    opt      root     sbin     sys      tmp      var
/ # python --version
/bin/sh: python: not found
/ # python3 --version
Python 3.7.10
/ # pip --version
pip 22.0.4 from /usr/lib/python3.7/site-packages/pip (python 3.7)
/ #
```

Ahora copiamos todos nuestros archivos desarrollados para la aplicación dentro de nuestro sistema, para ello usamos el comando WORKDIR para crear la carpeta de el directorio de trabajo en este caso llamada app en donde copiaremos los archivos creados con el comando COPY . y la carpeta creada.



```
Dockerfile X
Dockerfile
1 FROM alpine:3.10
2
3 RUN apk add --no-cache python3-dev \
4     && pip3 install --upgrade pip
5
6 WORKDIR /app
7
8 COPY . /app
```

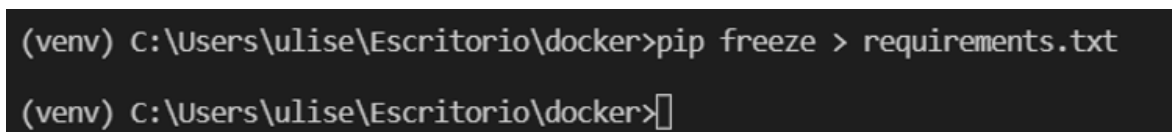
Después, para que la aplicación funcione en cualquier sistema se necesita crear un archivo que guarde todos los módulos o dependencias para poder reinstalarlos, para crear estos archivos usamos el siguiente comando.



```
(venv) C:\Users\ulise\Escritorio\docker>pip freeze
click==8.0.4
colorama==0.4.4
Flask==2.0.3
itsdangerous==2.1.1
Jinja2==3.0.3
MarkupSafe==2.1.1
Werkzeug==2.0.3

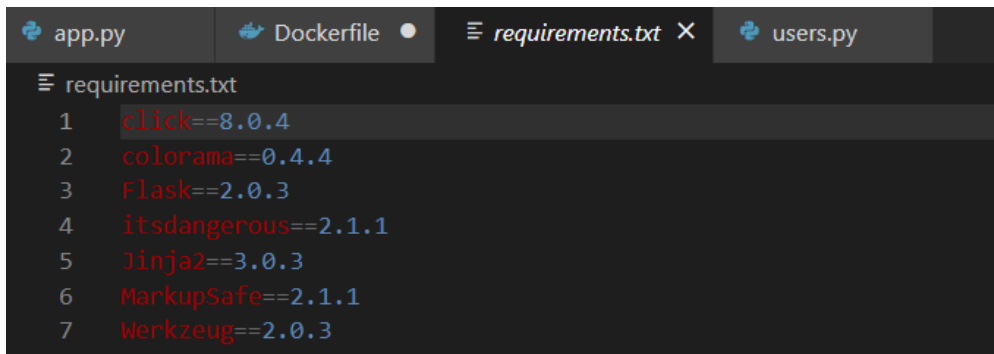
(venv) C:\Users\ulise\Escritorio\docker>
```

El cual nos arroja los paquetes que la aplicación necesita para funcionar (Flask y sus dependencias), pues estos paquetes son los que debemos guardar en un archivo requirements.txt, así que creamos dicho archivo de la siguiente manera



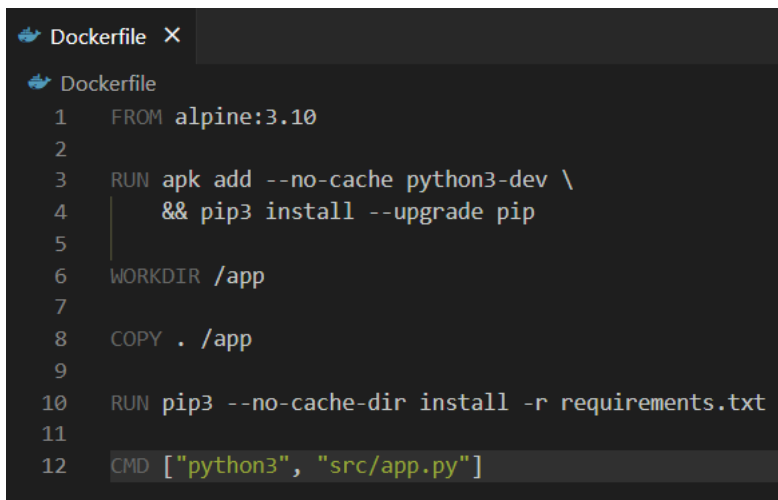
```
(venv) C:\Users\ulise\Escritorio\docker>pip freeze > requirements.txt

(venv) C:\Users\ulise\Escritorio\docker>
```



```
app.py  Dockerfile  requirements.txt  users.py
requirements.txt
1 click==8.0.4
2 colorama==0.4.4
3 Flask==2.0.3
4 itsdangerous==2.1.1
5 Jinja2==3.0.3
6 MarkupSafe==2.1.1
7 Werkzeug==2.0.3
```

Una vez teniendo este archivo copiado dentro del contenedor, instalamos los modulos contenidos ahí dentro, para ello usamos el siguiente código, que dice que intalará desde requirements.txt todos los modulos listados, posteriormente ejecutamos python3 y desde la carpeta src app.py, con ello la imagen está creado.



```
Dockerfile
Dockerfile
1 FROM alpine:3.10
2
3 RUN apk add --no-cache python3-dev \
4     && pip3 install --upgrade pip
5
6 WORKDIR /app
7
8 COPY . /app
9
10 RUN pip3 --no-cache-dir install -r requirements.txt
11
12 CMD ["python3", "src/app.py"]
```

Ahora volvemos a generar la imagen con el comando de la última vez, lo cual sobrescribirá la imagen anterior.

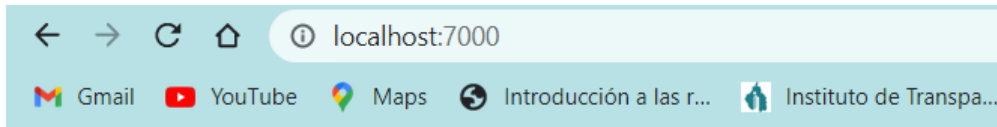
```
(venv) C:\Users\ulise\Escritorio\docker>docker build -t imgdocker1 -f Dockerfile .
[+] Building 16.5s (10/10) FINISHED
=> [internal] load build definition from Dockerfile
=> => transferring dockerfile: 249B
=> [internal] load .dockerignore
=> => transferring context: 2B
=> [internal] load metadata for docker.io/library/alpine:3.10
=> [internal] load build context
=> => transferring context: 20.24MB
```

```
(venv) C:\Users\ulise\Escritorio\docker>docker images
REPOSITORY    TAG       IMAGE ID       CREATED        SIZE
imgdocker1    latest   45b258797ed7   About a minute ago   135MB
```

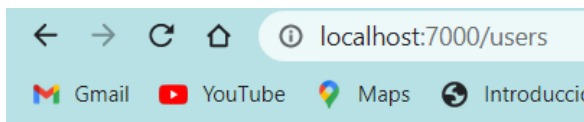
Ahora ejecutamos la aplicación del contenedor con `-publish` cambiando al puerto 7000 para que pueda funcionar fuera del container exponiendo el puerto 4000 dentro del contenedor.

```
(venv) C:\Users\ulise\Escritorio\docker>docker run -it --publish 7000:4000 imgdocker1
* Serving Flask app 'app' (lazy loading)
* Environment: production
  WARNING: This is a development server. Do not use it in a production deployment.
  Use a production WSGI server instead.
* Debug mode: on
* Running on all addresses.
  WARNING: This is a development server. Do not use it in a production deployment.
* Running on http://172.17.0.2:4000/ (Press CTRL+C to quit)
* Restarting with stat
* Debugger is active!
* Debugger PIN: 184-970-758
```

Ahora la aplicación se está ejecutando dentro del contenedor, por lo que necesitamos ingresar desde el puerto 7000 para acceder correctamente



```
{
  "response": "hello world"
}
```

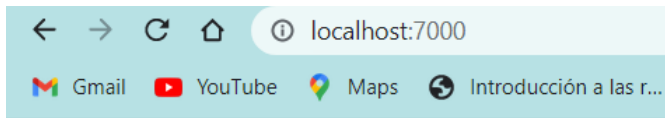


```
{
  "users": [
    {
      "name": "ulises"
    },
    {
      "name": "jose"
    },
    {
      "name": "vasi"
    }
  ]
}
```

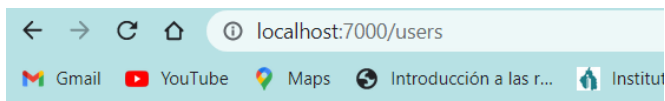
Si el cancelamos la ejecución de nuestro contenedor entonces el servidor se desconectará, para evitar eso ejecutaremos la aplicación como un proceso con el siguiente comando

```
(venv) C:\Users\ulise\Escritorio\docker>docker run -it -p 7000:4000 -d imgdocker1
175e0d18c38af1338917f4fbedbbfd485d0089fd709cd1466b5a586e16ca208e
```

Donde -d indica detach que nos permite evitar mantener a la consola ejecutando el programa, sino simplemente lo hará una vez y el programa se mantendrá ejecutando como un proceso liberando a la consola. Esto nos devuelve un ID y vemos que la app está funcionando.



```
{
  "response": "hello world"
}
```



```
{
  "users": [
    {
      "name": "ulises"
    },
    {
      "name": "jose"
    },
    {
      "name": "vasi"
    }
  ]
}
```

Y finalmente para ver el proceso escribimos el siguiente comando que nos muestra el contenedor que se está ejecutando y su ID

```
(venv) C:\Users\ulise\Escritorio\docker>docker container ls
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
175e0d18c38a	imgdocker1	"python3 src/app.py"	About a minute ago	Up About a minute	0.0.0.0:7000->4000/tcp	quizzical_feynman

Para detener el proceso usamos el siguiente comando con las primeras 3 caracteres del ID y comprobamos que el contenedor está detenido

```
(venv) C:\Users\ulise\Escritorio\docker>docker stop 175
175
```

```
(venv) C:\Users\ulise\Escritorio\docker>docker container ls
```

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

## Conclusión

Docker es una herramienta muy útil en las aplicaciones colaborativas de la actualidad, ya que permite flexibilidad y de cierta manera mitiga errores de aplicaciones y versiones no compatibles entre colaboradores, ya que permite mediante imágenes predefinidas y personalizadas por el desarrollador, la creación de contenedores que tendrán los módulos y sus dependencias que se requieran para la creación de alguna aplicación.

La ventaja de esto, es que Docker se considera portátil ya que siempre se ejecutan las aplicaciones exactamente de la misma manera y en el mismo orden en cualquier sistema, que como he mencionado lo define el programador a su gusto y necesidades; en este documento se ejemplificó el funcionamiento de Docker mediante Python para crear una aplicación muy sencilla con ayuda de Flask.

## Bibliografías

- Code, F. [FaztCode]. (2019, agosto 20). Docker & Python Flask. Contenedores con Python. Youtube.  
<https://www.youtube.com/watch?v=YENw-bNHZwg>
- ¿Qué es Docker? (s/f). Redhat.com. Recuperado el 22 de marzo de 2022, de <https://www.redhat.com/es/topics/containers/what-is-docker>

## LINK AL REPOSITORIO:

[https://github.com/UlisesVallejo/RestAPI\\_Docker.git](https://github.com/UlisesVallejo/RestAPI_Docker.git)