

Práctica Microservicios y Contenedores

José Luis Martínez Espíritu

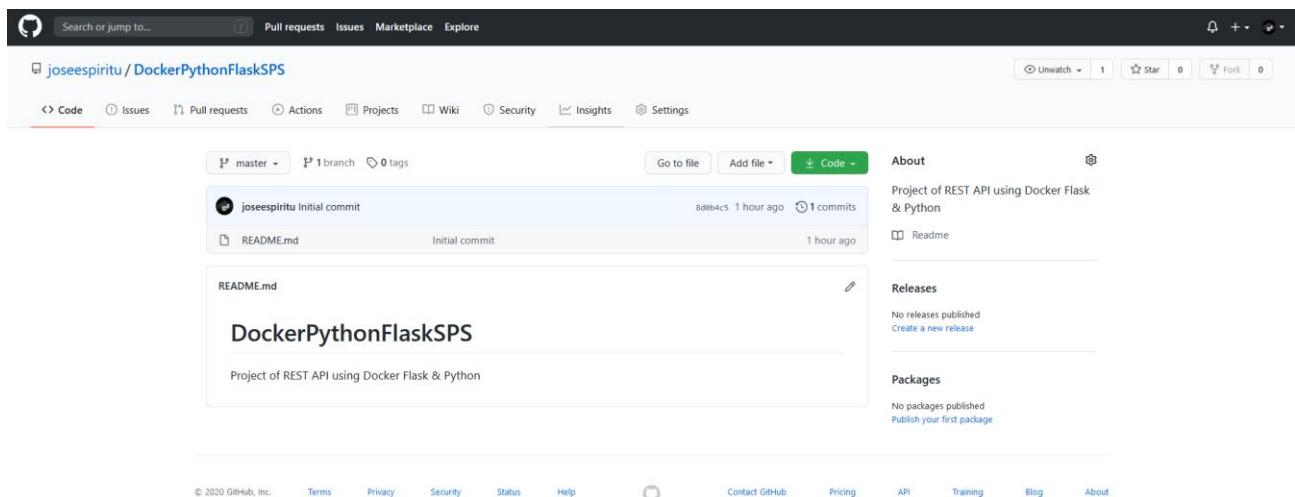
Documentación – septiembre 2020

Tecnologías a usar:

- Python 3.8.3
- Flask
- JSON
- Docker 19.03.12
- Github
- Insomnia (REST client) 2020.3.3
- MongoDB

En esta práctica se usará el formato de salida JSON el cual se considera la transición del formato XML, se considera de mejor estructura porque resulta más fácil acceder a la información, su sintaxis es básica y a la hora de procesar mucha información es más rápido, además de que tiene similitudes con bases de datos NoSQL como lo es MongoDB

Para empezar este proyecto se empezará creando un repositorio en Github el cual tendrá como nombre “**DockerPythonFlaskSPS**”, el cual contendrá todo el código y este documento como referencia (<https://github.com/joseespiritu/DockerPythonFlaskSPS>)



Este repositorio lo clonaremos dentro de una carpeta que se ubicara en el escritorio llamada “**SPS/ DockerPythonFlaskSPS**”, una vez finalizado se necesita que la computadora tenga instalado Python en su versión 3.8.3, Docker Desktop for Windows versión 2.3.0 y Docker versión 19.03.12, Flask e Insomnia versión 2020.3.3

Como editor de texto se utilizará Visual Studio Code con los plugins correspondientes a Python para garantizar su funcionamiento

Se creará un entorno virtual de Python, para ello accedemos desde consola al directorio donde se ubica el proyecto y ejecutamos el comando “pip install virtualenv” y se instalara la librería para la creación de entornos virtuales con python

```
TERMINAL  PROBLEMS  OUTPUT  DEBUG CONSOLE  1: cmd  +  [ ]  [ ]  ^  x

C:\Users\Jose Luis Espiritu\Desktop\SPS\DockePythonFlaskSPS>pip install virtualenv
Requirement already satisfied: virtualenv in c:\users\jose luis espiritu\appdata\local\programs\python\python38-32\lib\site-packages (20.0.31)
Requirement already satisfied: six<2,>=1.9.0 in c:\users\jose luis espiritu\appdata\local\programs\python\python38-32\lib\site-packages (from virtualenv) (1.15.0)
Requirement already satisfied: filelock<4,>=3.0.0 in c:\users\jose luis espiritu\appdata\local\programs\python\python38-32\lib\site-packages (from virtualenv) (3.0.12)
Requirement already satisfied: distlib<1,>=0.3.1 in c:\users\jose luis espiritu\appdata\local\programs\python\python38-32\lib\site-packages (from virtualenv) (0.3.1)
Requirement already satisfied: appdirs<2,>=1.4.3 in c:\users\jose luis espiritu\appdata\local\programs\python\python38-32\lib\site-packages (from virtualenv) (1.4.4)

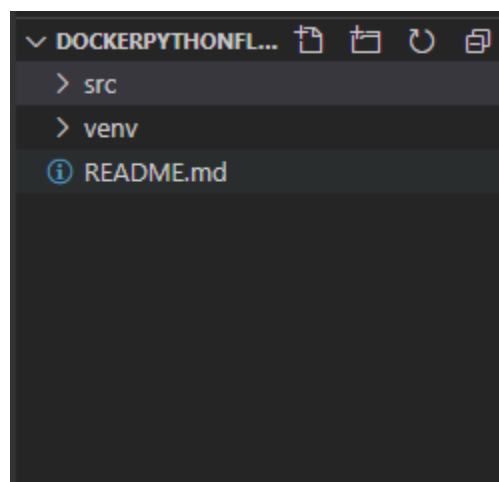
C:\Users\Jose Luis Espiritu\Desktop\SPS\DockePythonFlaskSPS>
```

Una vez instalada la librería necesaria para la creación de entornos virtuales, procedemos a crear el entorno virtual para nuestro proyecto, utilizando el comando “virtualenv venv”

```
C:\Users\Jose Luis Espiritu\Desktop\SPS\DockePythonFlaskSPS>virtualenv venv
created virtual environment CPython3.8.3.final.0-32 in 3761ms
creator CPython3Windows(dest=C:\Users\Jose Luis Espiritu\Desktop\SPS\DockePythonFlaskSPS\venv, clear=False, global=False)
seeder FromAppData(download=False, pip=bundle, setuptools=bundle, wheel=bundle, via=copy, app_data_dir=C:\Users\Jose Luis Espiritu\AppData\Local\pypa\virtualenv)
added seed packages: pip==20.2.2, setuptools==49.6.0, wheel==0.35.1
activators BashActivator,BatchActivator,FishActivator,PowerShellActivator,PythonActivator,XonshActivator

C:\Users\Jose Luis Espiritu\Desktop\SPS\DockePythonFlaskSPS>
```

En el directorio del proyecto se tendrá una carpeta llamada “venv” que contiene el entorno virtual, además se creara una nueva carpeta llamada “src” la cual contendrá todos los archivos necesarios para la aplicación



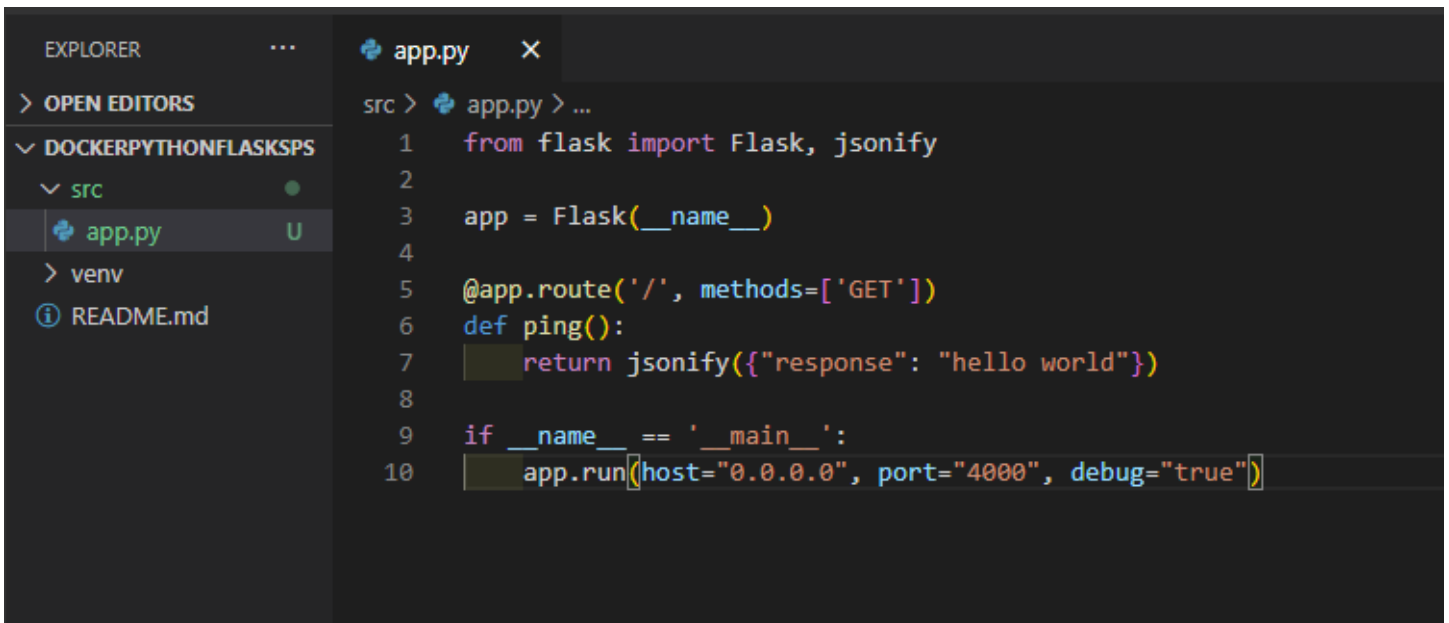
Para activar el entorno virtual de Python dentro de consola necesitamos ejecutar el comando “venv\Scripts\Activate” o simplemente acceder al archivo Activate.bat que se encuentra en la ruta anteriormente mencionada, para estar seguros de que el entorno virtual se activó se mostrara lo siguiente

```
C:\Users\Jose Luis Espiritu\Desktop\SPS\DockerPythonFlaskSPS>venv\Scripts\Activate  
(venv) C:\Users\Jose Luis Espiritu\Desktop\SPS\DockerPythonFlaskSPS>[]
```

Una vez activado el entorno virtual ejecutamos el comando “pip install flask” para instalar el Framework de Flask

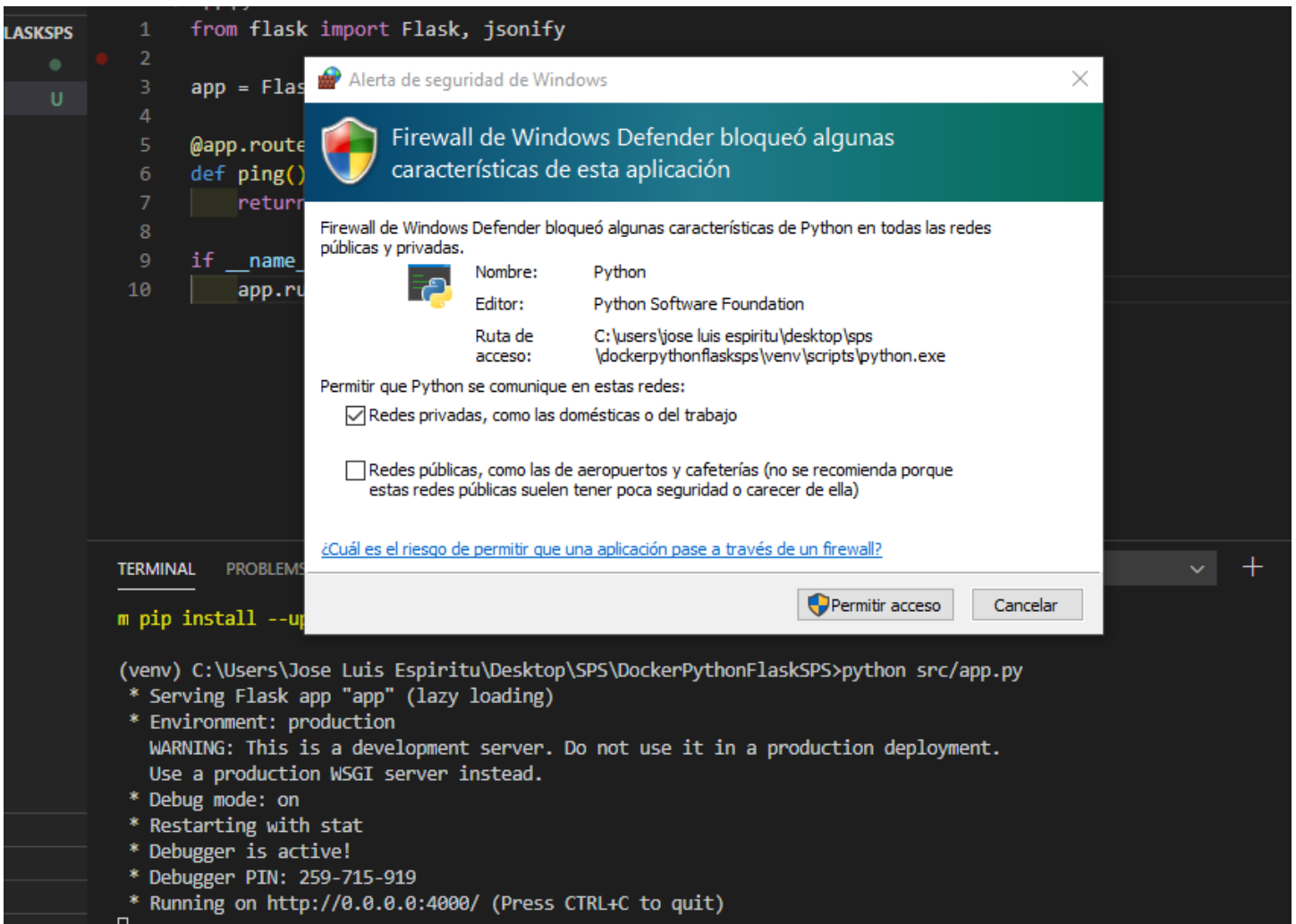
```
(venv) C:\Users\Jose Luis Espiritu\Desktop\SPS\DockerPythonFlaskSPS>pip install flask  
Collecting flask  
  Using cached Flask-1.1.2-py2.py3-none-any.whl (94 kB)  
Collecting Werkzeug>=0.15  
  Using cached Werkzeug-1.0.1-py2.py3-none-any.whl (298 kB)  
Collecting itsdangerous>=0.24  
  Using cached itsdangerous-1.1.0-py2.py3-none-any.whl (16 kB)  
Collecting Jinja2>=2.10.1  
  Using cached Jinja2-2.11.2-py2.py3-none-any.whl (125 kB)  
Collecting click>=5.1  
  Using cached click-7.1.2-py2.py3-none-any.whl (82 kB)  
Collecting MarkupSafe>=0.23  
  Using cached MarkupSafe-1.1.1-cp38-cp38-win32.whl (16 kB)  
Installing collected packages: Werkzeug, itsdangerous, MarkupSafe, Jinja2, click, flask  
Successfully installed Jinja2-2.11.2 MarkupSafe-1.1.1 Werkzeug-1.0.1 click-7.1.2 flask-1.1.2 itsdangerous-1.1.0
```

Dentro de la carpeta src del proyecto creamos un archivo llamado “app.py” el cual contendrá la clase principal para ejecutar la aplicación, el cual contendrá el siguiente código en donde se importa la librería de flask y jsonify para transformar objetos en archivos JSON, se establece la ruta por defecto, y para poder ser utilizado en un contenedor es necesario utilizar el host=”0.0.0.0”



```
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  if __name__ == '__main__':
10     app.run(host="0.0.0.0", port="4000", debug="true")
```

Para levantar el servidor de Flask es necesario ejecutar el archivo “app.py” con el comando “Python app.py”, una vez ejecutado se levantara el servidor y en algunos casos hay que permitirle el acceso a python a nuestra red



```
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  if __name__ == '__main__':
10     app.run(host="0.0.0.0", port="4000", debug="true")
```

Alerta de seguridad de Windows

Firewall de Windows Defender bloqueó algunas características de esta aplicación

Firewall de Windows Defender bloqueó algunas características de Python en todas las redes públicas y privadas.

Nombre: Python
Editor: Python Software Foundation
Ruta de acceso: C:\users\jose luis espiritu\desktop\sps\dockerpythonflasksp\venv\scripts\python.exe

Permitir que Python se comuniquen en estas redes:

☒ Redes privadas, como las domésticas o del trabajo

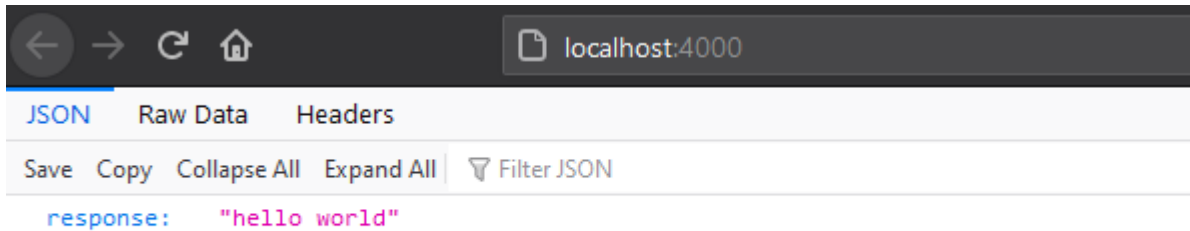
☐ Redes públicas, como las de aeropuertos y cafeterías (no se recomienda porque estas redes públicas suelen tener poca seguridad o carecer de ella)

[¿Cuál es el riesgo de permitir que una aplicación pase a través de un firewall?](#)

Permitir acceso Cancelar

```
(venv) C:\Users\Jose Luis Espiritu\Desktop\SPS\DockePythonFlaskSPS>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: 259-715-919
* Running on http://0.0.0.0:4000/ (Press CTRL+C to quit)
```

Una vez levantado el servidor desde cualquier navegador accedemos a la ruta “localhost:4000” el cual fue el puerto que se selecciono en el archivo “app.py” y mostrara lo siguiente, esto es solo una prueba de que Flask esta funcionando y recibiendo archivos tipo JSON



Posteriormente dentro del entorno virtual añadiremos las librerías correspondientes a MongoDB, ejecutando el siguiente comando “pip install Flask-PyMongo”

```
(venv) C:\Users\Jose Luis Espiritu\Desktop\SPS\DockePythonFlaskSPS>pip install Flask-PyMongo
Collecting Flask-PyMongo
  Using cached Flask_PyMongo-2.3.0-py2.py3-none-any.whl (12 kB)
Collecting PyMongo==3.3
  Using cached pymongo-3.11.0-cp38-cp38-win32.whl (376 kB)
Requirement already satisfied: Flask>=0.11 in c:\users\jose luis espiritu\desktop\spm\dockerpythonflaskspm\venv\lib\site-pack
ages (from Flask-PyMongo) (1.1.2)
Requirement already satisfied: Werkzeug>=0.15 in c:\users\jose luis espiritu\desktop\spm\dockerpythonflaskspm\venv\lib\site-p
ackages (from Flask>=0.11->Flask-PyMongo) (1.0.1)
```

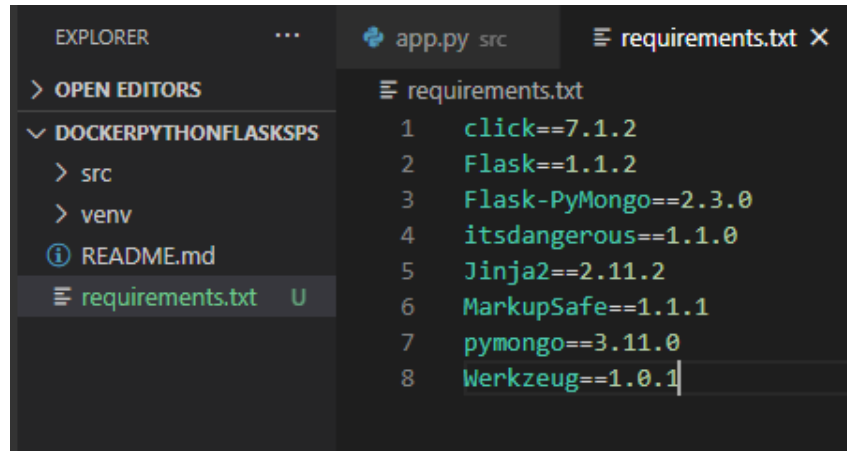
Dentro del archivo “app.py” se importan las siguientes librerías, las cuales darán las respuestas y harán peticiones a la Base de datos

```
src > app.py > ping
1  from flask import Flask, request, jsonify, Response
2  from flask_pymongo import PyMongo
3  from bson import json_util
4  from bson.objectid import ObjectId
5
6  app = Flask(__name__)
```

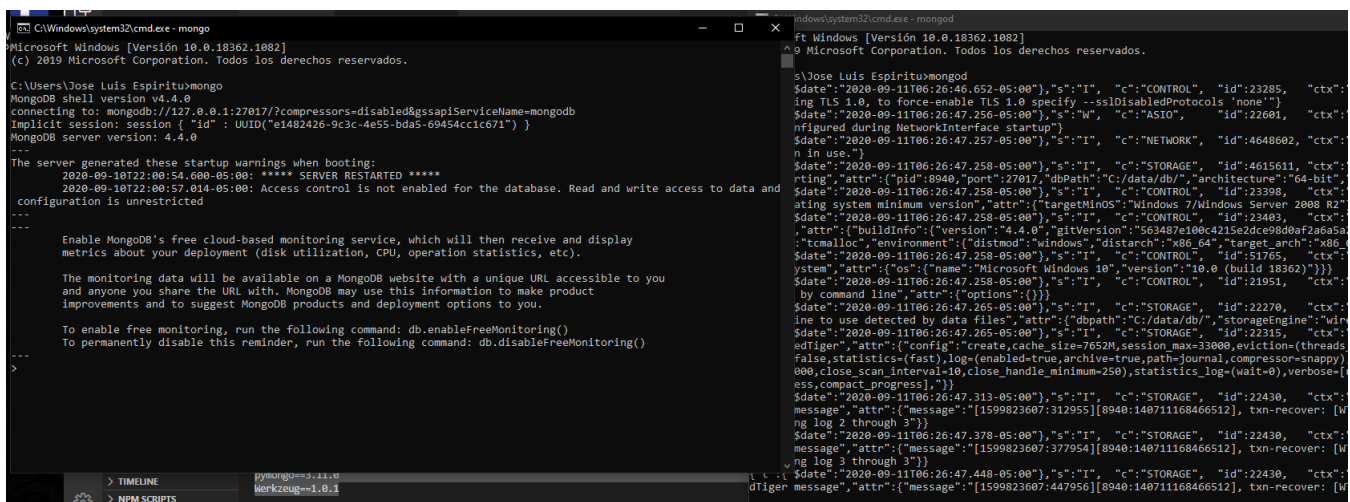
Una vez instaladas todas las librerías necesarias, procedemos a crear el archivo “requirements.txt” para que en futuras instalaciones de este proyecto se puede ejecutar con las versiones correspondientes de las librerías, para ello ejecutamos el comando “pip freeze”

```
(venv) C:\Users\Jose Luis Espiritu\Desktop\SPS\DockePythonFlaskSPS>pip freeze
click==7.1.2
Flask==1.1.2
Flask-PyMongo==2.3.0
itsdangerous==1.1.0
Jinja2==2.11.2
MarkupSafe==1.1.1
pymongo==3.11.0
Werkzeug==1.0.1
```

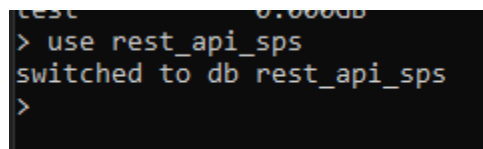
Esa lista de librerías la añadimos a un archivo llamado “requirements.txt” el cual se ubicara en la carpeta raíz del proyecto



Procedemos a la creación de la base de datos, para ello es indispensable contar con MongoDB, ejecutamos los procesos correspondientes desde la consola (mongo, mongod)



Desde la consola que esta ejecutando mongo, ingresamos el comando “use rest_api_sps” el cual creara una colección en la que se almacenaran todos los datos correspondientes



Después ingresamos los comandos “db” para estar seguros de estar usando esa base de datos e insertamos un primer registro en la colección “users” con el comando “db.users.insert({“name”:“user1”})” y mostramos todas las bases de datos con el comando “show dbs” para asegurar que la base de datos ha sido creada correctamente

```

> db
rest_api_sps
> db.users.insert({"name": "user1"})
WriteResult({ "nInserted" : 1 })
> show dbs
admin            0.000GB
config           0.000GB
local            0.000GB
pythonmongodb    0.000GB
rest_api_sps     0.000GB
test             0.000GB
>

```

Dentro del archivo “app.py” insertamos las siguientes líneas de código, las cuales establecen la configuración de Flask para que se conecte a la base de datos creada anteriormente

```

5
6 app = Flask(__name__)
7
8 app.config['MONGO_URI'] = 'mongodb://localhost/rest_api_sps'
9 mongo = PyMongo(app)
10
11 @app.route('/', methods=['GET'])
12 def ping():
13     return jsonify({"response": "hello world"})
14

```

Ahora procedemos a realizar una ruta endpoint ‘/api/sps/helloworld/v1’ la cual recibirá los datos por medio del método POST y mostrará la respuesta desde un cliente REST para lo cual usaremos “Insomnia”, ingresamos el siguiente código que recibirá un nombre de usuario y correo

```

@app.route('/api/sps/helloworld/v1', methods=['POST'])
def create_user():
    #Receiving Data
    username = request.json['username']
    email = request.json['email']

    if username and email:
        id = mongo.db.users.insert(
            {'username': username,
             'email': email}
        )
        response = {
            'id': str(id),
            'username': username,
            'email': email
        }
        return response
    else:
        return not_found()

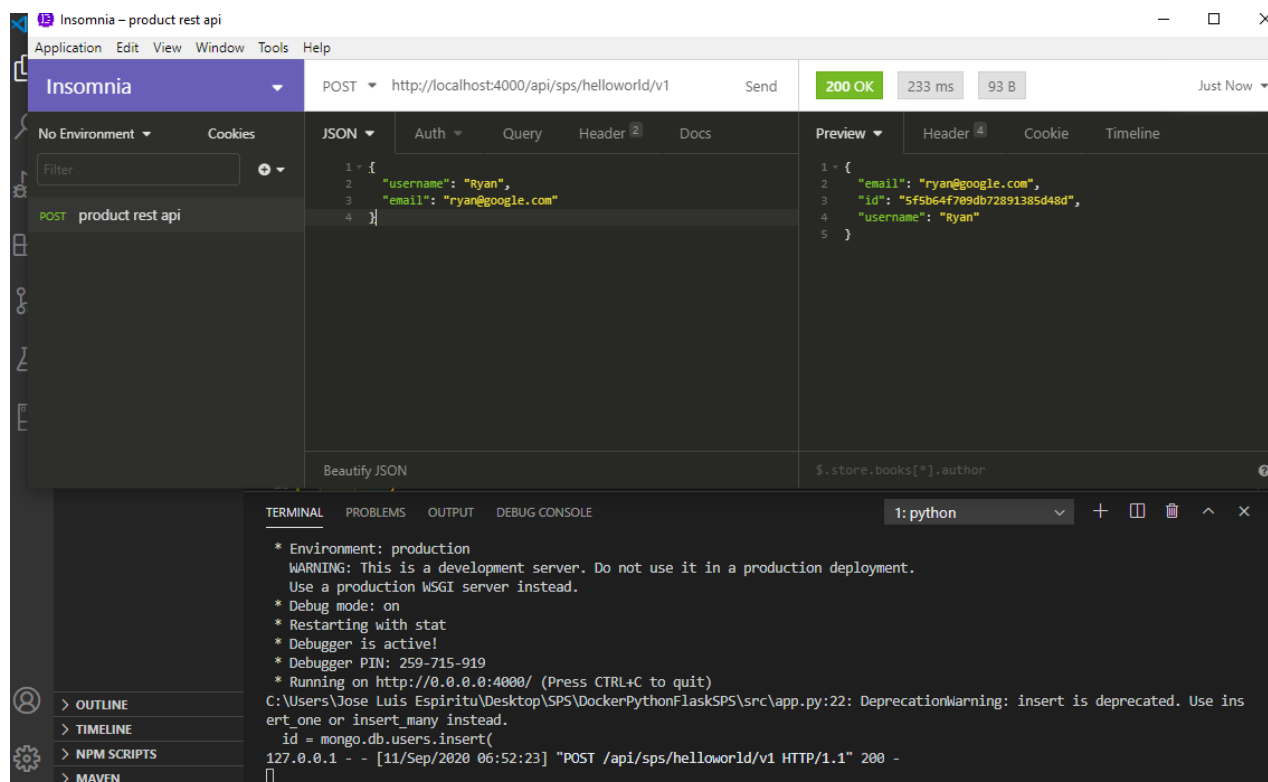
    return {'message': 'received'}

```


El código recibe los el usuario y correo desde un archivo en formato JSON, valida que esos datos existan para que puedan ser insertados en la base de datos, en caso contrario nos enviara a una función que devuelve una respuesta de que la petición no se ha hecho correctamente, para ello definimos una nueva ruta en donde se especifica el error “404” el cual hace referencia a que no se encontró la información adecuada para insertar en la base de datos

```
@app.errorhandler(404)
def not_found(error=None):
    response = jsonify({
        "message": "Resource not found: " + request.url,
        "status": 404
    })
    response.status_code = 404
    return response
```

Para probar que la inserción se hizo de manera correcta abrimos el cliente REST (Insomnia) y ejecutamos el servidor de Flask para realizar la petición, dentro de Insomnia, ingresamos a la url del servidor de Flask dentro del endpoint definido previamente, seleccionamos una petición de tipo POST, seleccionamos la opción de enviar un formato JSON, y escribimos un nombre de usuario y correo, pulsamos en el botón “SEND” y nos dará una respuesta en estatus 200 el cual muestra que la petición ha sido realizada con éxito y desde consola se puede observar que la inserción a Mongo se ha hecho correctamente



Para verificar los datos insertados en Mongo, accedemos a la consola que esta ejecutando MongoDB y escribimos el comando “db.users.find()”, el cual mostrará todos los usuarios registrados en la colección users

```
test 0.000GB
> db.users.find()
{ "_id" : ObjectId("5f5b60c0226ec86ca268e2d9"), "name" : "user1" }
{ "_id" : ObjectId("5f5b64f709db72891385d48d"), "username" : "Ryan", "email" : "ryan@google.com" }
>
```

Creamos un segundo registro

The screenshot shows a REST client interface. The top bar indicates a POST request to `http://localhost:4000/api/sps/helloworld/v1` with a status of **200 OK**, a response time of **218 ms**, and a body size of **91 B**. The left pane shows the JSON body of the request:

```
1 {
2   "username": "Jose",
3   "email": "jose@jose.com"
4 }
```

The right pane shows the JSON response body:

```
1 {
2   "email": "jose@jose.com",
3   "id": "5f5b66ee09db72891385d48e",
4   "username": "Jose"
5 }
```

Dentro de “app.py” definimos una nueva ruta, esta vez utilizando el método GET la cual nos devolverá todos los usuarios que se encuentran dentro de la base de datos

```
@app.route('/api/sps/helloworld/v1', methods=['GET'])
def get_users():
    users = mongo.db.users.find()
    response = json_util.dumps(users)
    return Response(response, mimetype='application/json')
```

El cliente REST nos muestra lo siguiente en el cual se pueden observar los usuarios insertados anteriormente

The screenshot shows the Insomnia REST client interface. The top bar indicates a GET request to `http://localhost:4000/api/sps/helloworld/v1` with a status of **200 OK**, a response time of **231 ms**, and a body size of **252 B**. The left pane shows the JSON body of the response:

```
1 {
2   "username": "Jose",
3   "email": "jose@jose.com"
4 }
```

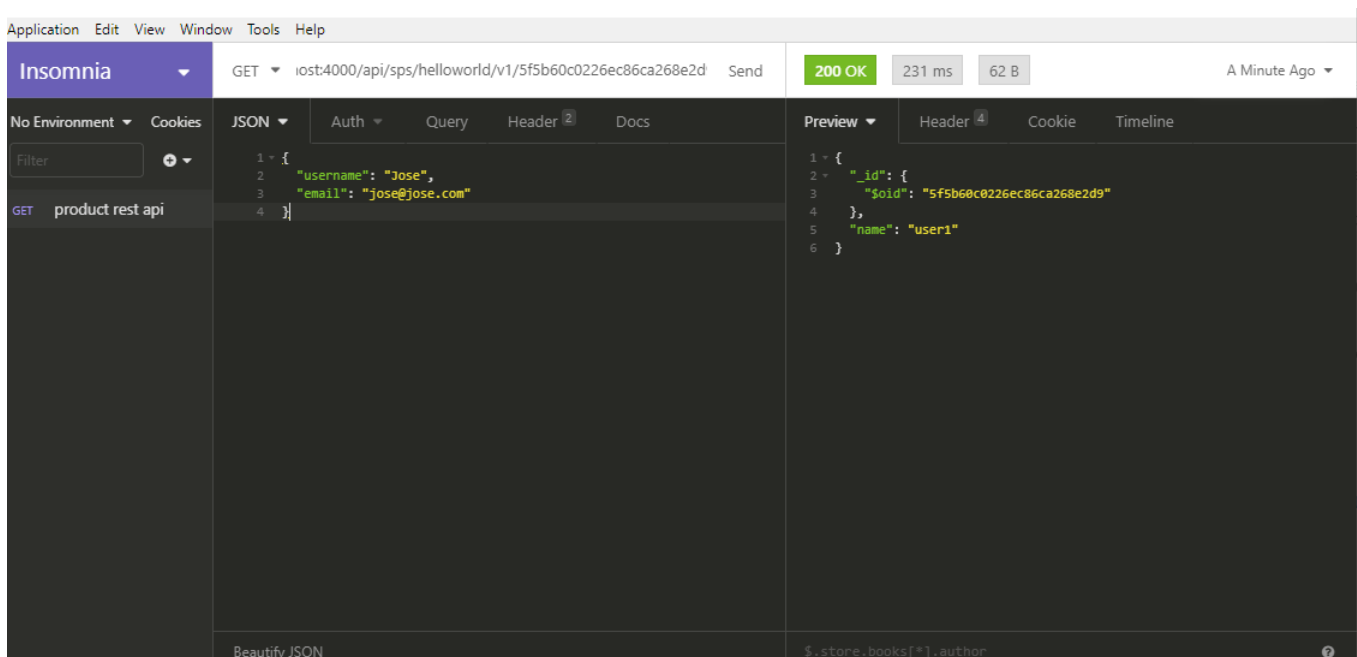
The right pane shows the JSON response body, which is a list of three users:

```
1 [
2   {
3     "_id": {
4       "$oid": "5f5b60c0226ec86ca268e2d9"
5     },
6     "name": "user1"
7   },
8   {
9     "_id": {
10      "$oid": "5f5b64f709db72891385d48d"
11    },
12    "username": "Ryan",
13    "email": "ryan@google.com"
14  },
15  {
16    "_id": {
17      "$oid": "5f5b66ee09db72891385d48e"
18    },
19    "username": "Jose",
20    "email": "jose@jose.com"
21  }
22 ]
```

Para buscar un usuario específico dentro de “app.py” escribiremos el siguiente código, el cual dentro de la ruta predefinida pasando como parámetro por la URL nos solicitará el id del usuario registrado

```
@app.route('/api/sps/helloworld/v1/<id>', methods=['GET'])
def get_user(id):
    user = mongo.db.users.find_one({'_id': ObjectId(id)})
    response = json_util.dumps(user)
    return Response(response, mimetype='application/json')
```

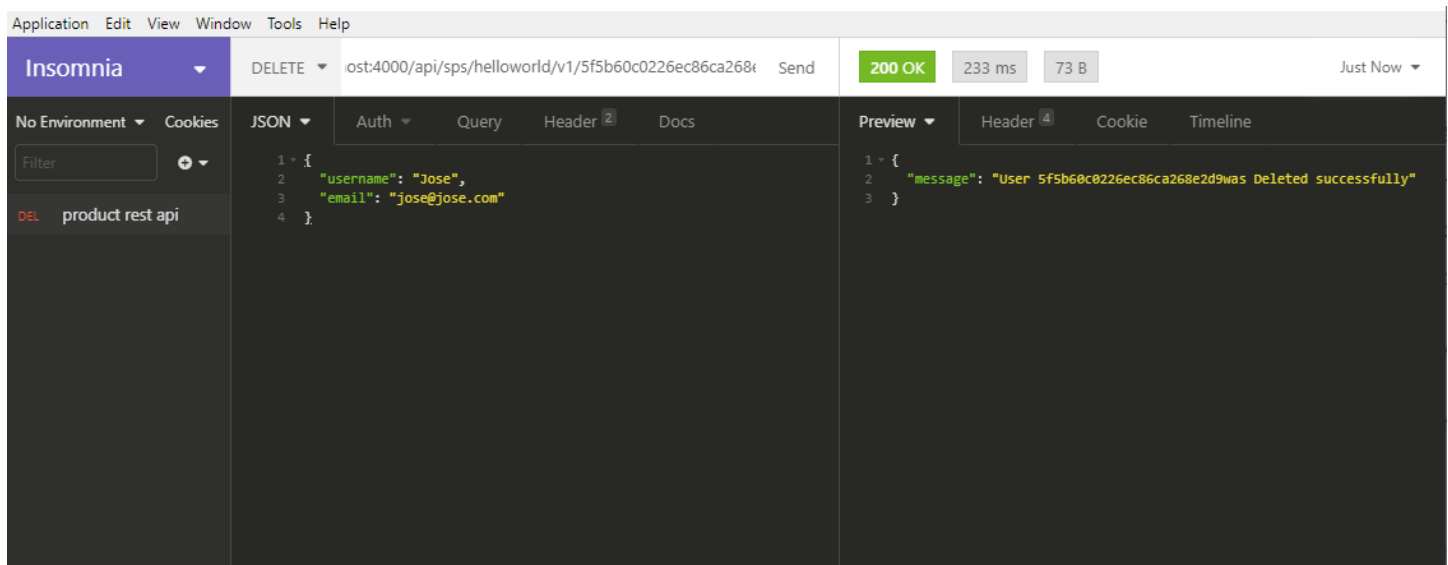
Desde Insomnia nos dirigimos a la ruta utilizando el método GET especificando como parámetro el Id que genera mongoDB por defecto, en este caso se obtendrá el registro del primer usuario insertado en la base de datos, y nos devuelve registro del usuario



Para eliminar un usuario dentro del archivo “app.py” escribimos el siguiente código, el cual pide como parámetro el id del usuario registrado para que lo pueda eliminar y mostrara un mensaje de que el usuario (id_usuario) ha sido borrado correctamente

```
@app.route('/api/sps/helloworld/v1/<id>', methods=['DELETE'])
def delete_user(id):
    mongo.db.users.delete_one({'_id': ObjectId(id)})
    response = jsonify({'message': 'User ' + id + 'was Deleted successfully'})
    return response
```

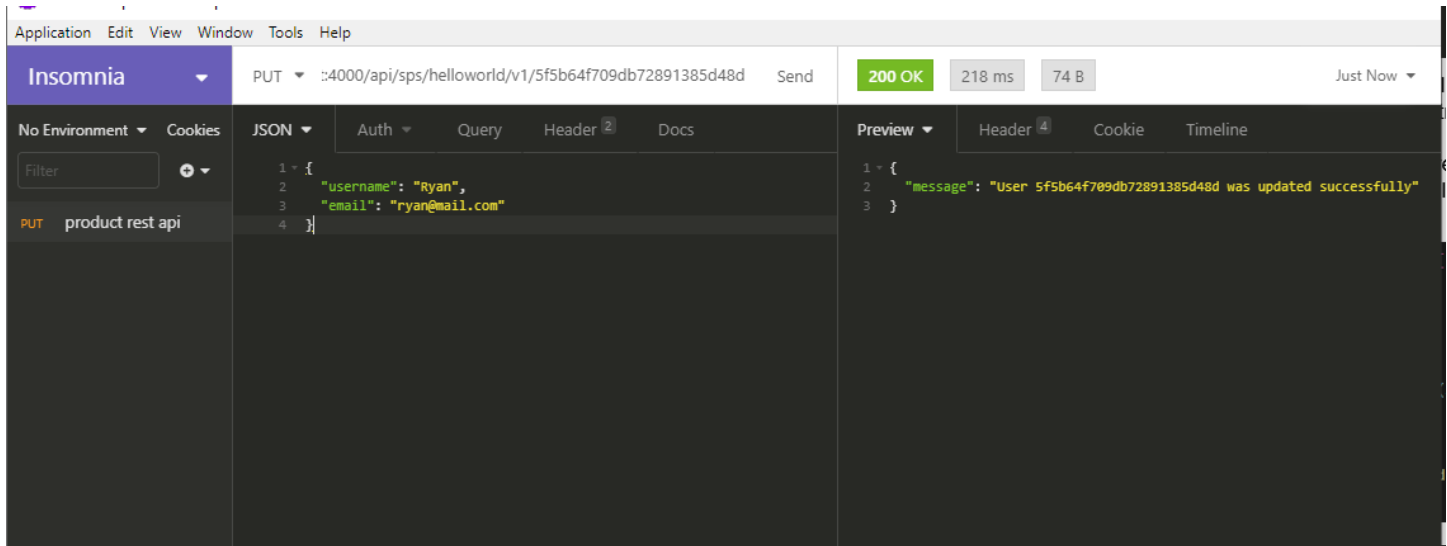
Pasando a Insomnia, eliminaremos el primer registro creado y consultado anteriormente, pasando como parámetro su id y ejecutando el método HTTP “DELETE”



Para actualizar el registro de un usuario, dentro del archivo “app.py” escribimos el siguiente código, el cual pide como parámetro dentro de la URL el id del usuario del cual se va actualizar algún dato, ya sea su nombre o su correo, así como un archivo JSON que contenga los valores a actualizar en el registro utilizando el método HTTP “PUT”, además nos mostrará un mensaje de que el usuario (id_usuario) se ha actualizado correctamente

```
5 @app.route('/api/sps/helloworld/v1/<id>', methods=['PUT'])
6 def update_user(id):
7     username = request.json['username']
8     email = request.json['email']
9
10    if username and email:
11        mongo.db.users.update_one({'_id': ObjectId(id)}, {'$set': {
12            'username': username,
13            'email': email
14        }})
15        response = jsonify({'message': 'User ' + id + ' was updated successfully' })
16        return response
17
```

Dentro de Insomnia actualizaremos el correo del usuario “Ryan” con id: 5f5b64f709db72891385d48d, pasando de ryan@google.com a ryan@mail.com”



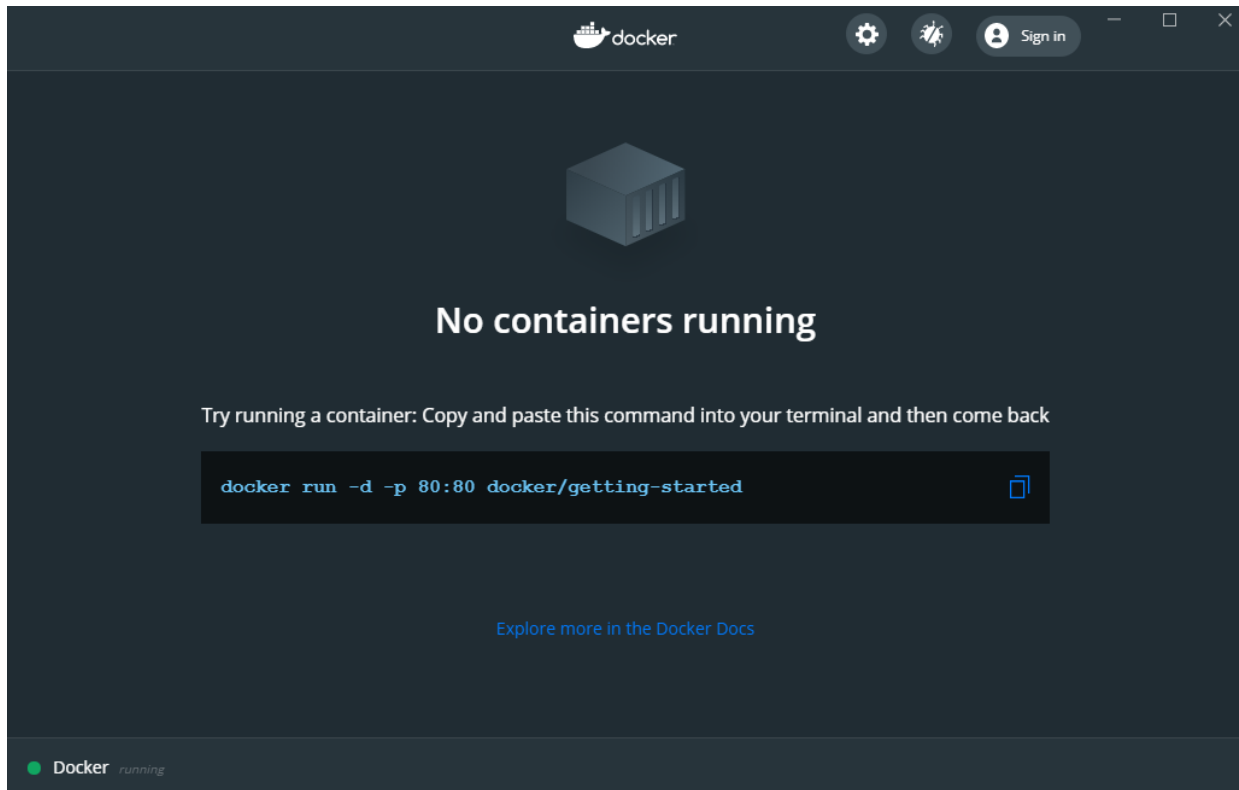
Verificamos mediante el método GET para obtener todos los usuarios registrados

```
1 [
2   {
3     "_id": {
4       "$oid": "5f5b64f709db72891385d48d"
5     },
6     "username": "Ryan",
7     "email": "ryan@mail.com"
8   },
9   {
10    "_id": {
11      "$oid": "5f5b66ee09db72891385d48e"
12    },
13    "username": "Jose",
14    "email": "jose@jose.com"
15  }
16 ]
```

Y en efecto el email de Ryan a cambiado de de ryan@google.com a ryan@mail.com”

Contenedor Docker

Para realizar el contenedor de Docker, utilizare Docker desktop for Windows 10, el cual al instalar e iniciarlo verificaremos que el servicio este ejecutándose

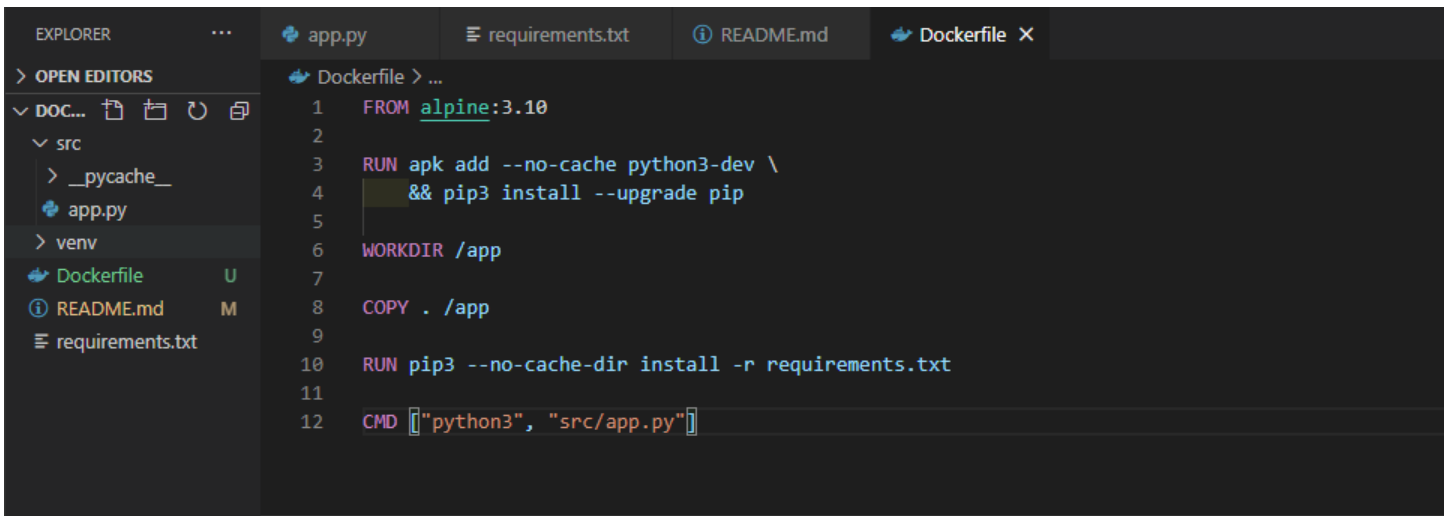


Dentro de la consola detenemos el servidor de Flask con las teclas “CTRL + C”

```
* Detected change in 'C:\\Users\\Jose Luis Espiritu\\Desktop\\SPS\\DockerPythonFlaskSPS\\src\\app.py', reloading
* Restarting with stat
* Debugger is active!
* Debugger PIN: 259-715-919
* Running on http://0.0.0.0:4000/ (Press CTRL+C to quit)
127.0.0.1 - - [11/Sep/2020 17:36:19] "GET /api/sps/helloworld/v1 HTTP/1.1" 200 -
127.0.0.1 - - [11/Sep/2020 17:38:15] "PUT /api/sps/helloworld/v1/5f5b64f709db72891385d48d HTTP/1.1" 200 -
127.0.0.1 - - [11/Sep/2020 17:39:18] "GET /api/sps/helloworld/v1 HTTP/1.1" 200 -
```

```
(venv) C:\\Users\\Jose Luis Espiritu\\Desktop\\SPS\\DockerPythonFlaskSPS>
```

En la carpeta raíz del proyecto se crea el archivo “Dockerfile” el cual contendrá el software necesario para la ejecución de la aplicación, en este caso se usara alpine la cual es un sistema operativo minimalista con distribución Linux, después con el comando RUN se procede a instalar el software necesario para la ejecución de la aplicación, se instala python3/dev que ya incluye la librería pip para instalar los requerimientos, se actualiza pip , después se crea una carpeta de trabajo denominada WORDIR que tendrá el nombre de “/app”, mediante el comando COPY procede a copiar todo el contenido del proyecto dentro de la carpeta de trabajo, después con el comando RUN se ejecuta pip3 para instalar el archivo “requirements.txt” que se creo anteriormente y finalmente mediante el comando CMD procede a ejecutarse Python y la aplicación dentro del contenedor



```
EXPLORER  ...  app.py  requirements.txt  README.md  Dockerfile X
> OPEN EDITORS
DOC...  src  __pycache__  app.py  venv  Dockerfile  README.md  requirements.txt
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 procedemos a ejecutar el comando “docker build -t flaskapp .” el cual creara una imagen con la aplicación REST API creada anteriormente, el nombre de la imagen será “flaskapp”, el proceso tarda un poco

```
(venv) C:\Users\Jose Luis Espiritu\Desktop\SP5\DockerPythonFlaskSP5>docker build -t flaskapp .
Sending build context to Docker daemon 32.16MB
Step 1/6 : FROM alpine:3.10
----> be4e4bea2c2e
Step 2/6 : RUN apk add --no-cache python3-dev && pip3 install --upgrade pip
----> Using cache
----> f9eb8745c4c1
Step 3/6 : WORKDIR /app
----> Using cache
----> 1d26387a06bb
Step 4/6 : COPY . /app
----> ebe1f965492c
Step 5/6 : RUN pip3 --no-cache-dir install -r requirements.txt
----> Running in 601d01abc64f
Collecting click==7.1.2
  Downloading click-7.1.2-py2.py3-none-any.whl (82 kB)
Collecting Flask==1.1.2
```

Ahora procederé a crear otro contenedor que contenga mongodb para hacer la conexión con la aplicación, en esta parte me tarde mas entender el funcionamiento de cada parte del contenedor, una posible herramienta es Docker compose, para ello abriré una consola nueva del sistema en la que se creara la imagen de mongo, para ello ejecutamos el comando “Docker pull mongo”

```
C:\Users\Jose Luis Espiritu>docker --version
Docker version 19.03.12, build 48a66213fe

C:\Users\Jose Luis Espiritu>docker pull mongo
Using default tag: latest
latest: Pulling from library/mongo
f08d8e2a3ba1: Pull complete
3baa9cb2483b: Pull complete
94e5ff4c0b15: Pull complete
1860925334f9: Pull complete
9d42806c06e6: Pull complete
31a9fd218257: Pull complete
5bd6e3f73ab9: Pull complete
f6ae7a64936b: Pull complete
80fde2cb25c5: Pull complete
80dd04855bec: Pull complete
38c0e96de174: Pull complete
b7256055e1ef: Pull complete
23bef11da1da: Pull complete
Digest: sha256:f8dcfaa1d5eab1fec5567ef4e7dedab97b9e41876222fba9c0588cdf312fdf8
Status: Downloaded newer image for mongo:latest
docker.io/library/mongo:latest

C:\Users\Jose Luis Espiritu>
```


Para listar las imágenes creadas se ejecuta el comando “docker images”, el cual listara todas las imágenes creadas, algunas imágenes las cree para hacer pruebas antes de realizar esta practica

```
C:\Users\Jose Luis Espiritu>docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
flaskapp	latest	76b77b10bd53	6 minutes ago	143MB
flaskapp_sps	latest	60d8b781b7cb	36 minutes ago	143MB
<none>	<none>	203c77383b8d	40 minutes ago	291MB
<none>	<none>	fd174e3a11a4	43 minutes ago	291MB
<none>	<none>	19a54cc558b3	48 minutes ago	5.57MB
<none>	<none>	15bcc548e5c6	54 minutes ago	294MB
mongo	latest	923803327a36	2 hours ago	493MB
alpine	edge	3c791e92a856	2 months ago	5.57MB
alpine	3.10	be4e4bea2c2e	4 months ago	5.58MB
alpine	3.11.2	cc0abc535e36	8 months ago	5.59MB

Pasaremos a la configuración del contenedor para que se pueda utilizar externamente con otra aplicación ejecutando el comando “docker run -p 27018:27017 --name database_2 mongo” el cual establece que el contenedor de Mongo que originalmente se conecta al puerto 27017 ahora se estará ejecutando desde el puerto 27018, después procedemos a ejecutar un cliente de mongo, y en este caso nos conectaremos a ese puerto, y listo ya tenemos nuestra base de datos ejecutándose

```
C:\Users\Jose Luis Espiritu>docker run -p 27018:27017 --name database_2 mongo
```

```
{
  "t": {
    "$date": "2020-09-12T00:54:02.865+00:00",
    "s": "I",
    "c": "CONTROL",
    "id": 23285,
    "ctx": "main",
    "msg": "Automatically
disabling TLS 1.0, to force-enable TLS 1.0 specify --sslDisabledProtocols 'none'"
  },
  "t": {
    "$date": "2020-09-12T00:54:02.868+00:00",
    "s": "W",
    "c": "ASIO",
    "id": 22601,
    "ctx": "main",
    "msg": "No TransportL
ayer configured during NetworkInterface startup"
  },
  "t": {
    "$date": "2020-09-12T00:54:02.868+00:00",
    "s": "I",
    "c": "NETWORK",
    "id": 4648601,
    "ctx": "main",
    "msg": "Implicit TCP
FastOpen unavailable. If TCP FastOpen is required, set tcpFastOpenServer, tcpFastOpenClient, and tcpFastOpenQueueSize."
  },
  "t": {
    "$date": "2020-09-12T00:54:02.869+00:00",
    "s": "I",
    "c": "STORAGE",
    "id": 4615611,
    "ctx": "initandlisten",
    "msg": "Mong
oDB starting",
    "attr": {
      "pid": 1,
      "port": 27017,
      "dbPath": "/data/db",
      "architecture": "64-bit",
      "host": "99b50e4fc2e6"
    }
  },
  "t": {
    "$date": "2020-09-12T00:54:02.869+00:00",
    "s": "I",
    "c": "CONTROL",
    "id": 23403,
    "ctx": "initandlisten",
    "msg": "Buil
d Info",
    "attr": {
      "buildInfo": {
        "version": "4.4.1",
        "gitVersion": "ad91a93a5a31e175f5cbf8c69561e788bbc55ce1",
        "opensslVersion": "OpenSSL 1.1.1 11 Sep 2018",
        "modules": [],
        "allocator": "tcmalloc",
        "environment": {
          "distmod": "ubuntu1804",
          "distarch": "x86_64"
        }
      }
    }
  },
  "t": {
    "$date": "2020-09-12T00:54:02.869+00:00",
    "s": "I",
    "c": "CONTROL",
    "id": 51765,
    "ctx": "initandlisten",
    "msg": "Oper
ating System",
    "attr": {
      "os": {
        "name": "Ubuntu",
        "version": "18.04"
      }
    }
  },
  "t": {
    "$date": "2020-09-12T00:54:02.869+00:00",
    "s": "I",
    "c": "CONTROL",
    "id": 21951,
    "ctx": "initandlisten",
    "msg": "Opti
ons set by command line",
    "attr": {
      "options": {
        "net": {
          "bindIp": "*"
        }
      }
    }
  },
  "t": {
    "$date": "2020-09-12T00:54:02.871+00:00",
    "s": "I",
    "c": "STORAGE",
    "id": 22297,
    "ctx": "initandlisten",
    "msg": "Usin
g the XFS filesystem is strongly recommended with the WiredTiger storage engine. See http://dochub.mongodb.org/core/prod
notes-filesystem",
    "tags": [
      "startupWarnings"
    ]
  },
  "t": {
    "$date": "2020-09-12T00:54:02.871+00:00",
    "s": "I",
    "c": "STORAGE",
    "id": 22315,
    "ctx": "initandlisten",
    "msg": "Open
ing WiredTiger",
    "attr": {
      "config": {
        "create": true,
        "cache_size": 483M,
        "session_max": 33000,
        "eviction": {
          "threads_min": 4,
          "threads_max": 4
        },
        "config_base": false,
        "statistics": {
          "fast": true
        },
        "log": {
          "enabled": true,
          "archive": true,
          "path": "journal",
          "compressor": "snappy"
        },
        "file_manager": {
          "close_idle_ti
me": 100000,
          "close_scan_interval": 10,
          "close_handle_minimum": 250
        },
        "statistics_log": {
          "wait": 0
        },
        "verbose": [
          "recovery_progress",
          "checkpoint
progress",
          "compact_progress"
        ]
      }
    }
  },
  "t": {
    "$date": "2020-09-12T00:54:03.533+00:00",
    "s": "I",
    "c": "STORAGE",
    "id": 22430,
    "ctx": "initandlisten",
    "msg": "Wire
dTiger message",
    "attr": {
      "message": [
        1599872043:533325
      ] [1:0x7f64f745da80],
      "txn-recover": [
        WT_VERB_RECOVERY | WT_VERB_RECOV
ERY_PROGRESS
      ]
    },
    "msg": "Set global recovery timestamp: (0, 0)"
  },
  "t": {
    "$date": "2020-09-12T00:54:03.533+00:00",
    "s": "I",
    "c": "STORAGE",
    "id": 22430,
    "ctx": "initandlisten",
    "msg": "Wire"
  }
}
```



```

C:\Users\Jose Luis Espiritu>mongo localhost:27018
MongoDB shell version v4.4.0
connecting to: mongodb://localhost:27018/test?compressors=disabled&gssapiServiceName=mongodb
Implicit session: session { "id" : UUID("3e9aad3a-a6e1-4279-9f2d-1c378c10c0bd") }
MongoDB server version: 4.4.1
---
The server generated these startup warnings when booting:
  2020-09-12T00:54:02.871+00:00: Using the XFS filesystem is strongly recommended with the WiredTiger storage engine. See http://dochub.mongodb.org/core/prodnotes-filesystem
  2020-09-12T00:54:03.568+00:00: Access control is not enabled for the database. Read and write access to data and configuration is unrestricted
---
---
  Enable MongoDB's free cloud-based monitoring service, which will then receive and display metrics about your deployment (disk utilization, CPU, operation statistics, etc).

  The monitoring data will be available on a MongoDB website with a unique URL accessible to you and anyone you share the URL with. MongoDB may use this information to make product improvements and to suggest MongoDB products and deployment options to you.

  To enable free monitoring, run the following command: db.enableFreeMonitoring()
  To permanently disable this reminder, run the following command: db.disableFreeMonitoring()
---
> show dbs
admin      0.000GB
config     0.000GB
local      0.000GB
>

```

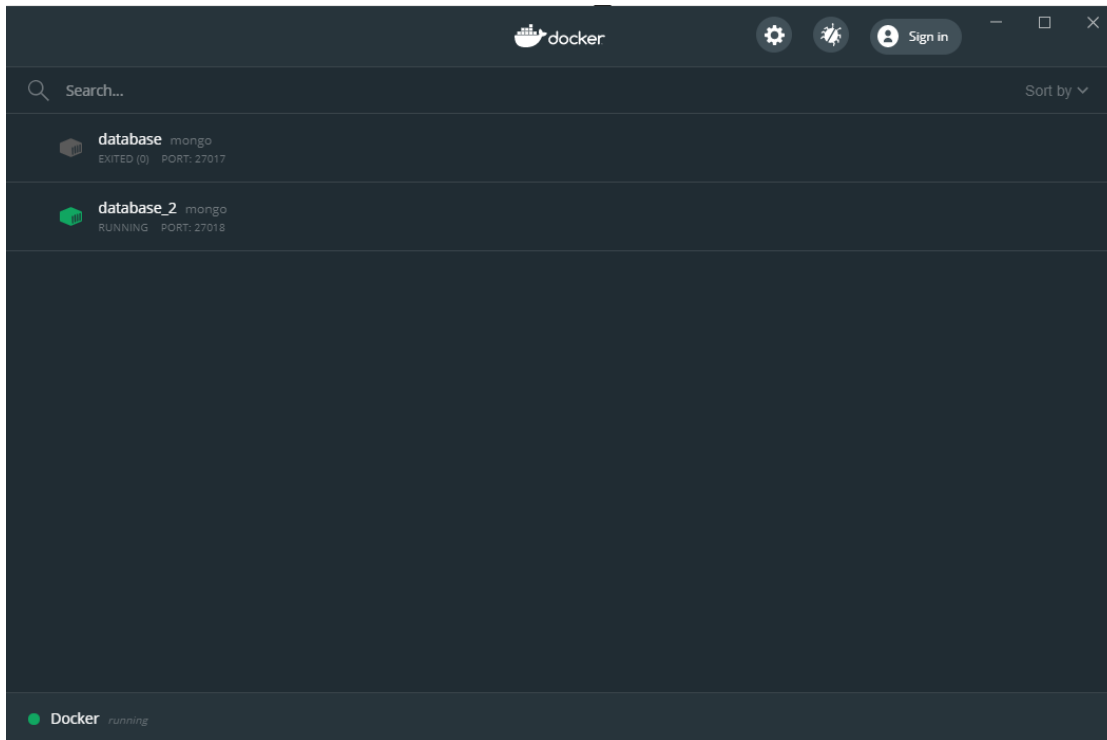
Una vez que tenemos nuestra base de datos ejecutándose, procedemos a crear la base de datos creada previamente llamada “rest_api_sps” con el comando “use rest_api_sps” y crearemos el registro de un usuario

```

---
> show dbs
admin      0.000GB
config     0.000GB
local      0.000GB
> use rest_api_sps
switched to db rest_api_sps
> db
rest_api_sps
> show dbs
admin      0.000GB
config     0.000GB
local      0.000GB
> db
rest_api_sps
> db.users.insert("name":"user_from_container")
uncaught exception: SyntaxError: missing ) after argument list :
@(shell):1:22
> db.users.insert({"name":"user_from_container"})
WriteResult({ "nInserted" : 1 })
> show dbs
admin      0.000GB
config     0.000GB
local      0.000GB
rest_api_sps 0.000GB
> db.find()
uncaught exception: TypeError: db.find is not a function :
@(shell):1:1
> db.users.find()
{ "_id" : ObjectId("5f5c1e18832b153bf62722ec"), "name" : "user_from_container" }
>

```

Nuestra base de datos esta brindando servicio, ahora procedemos a verificarlo en Docker, aquí hice una prueba antes, pero la que se está utilizando es database_2



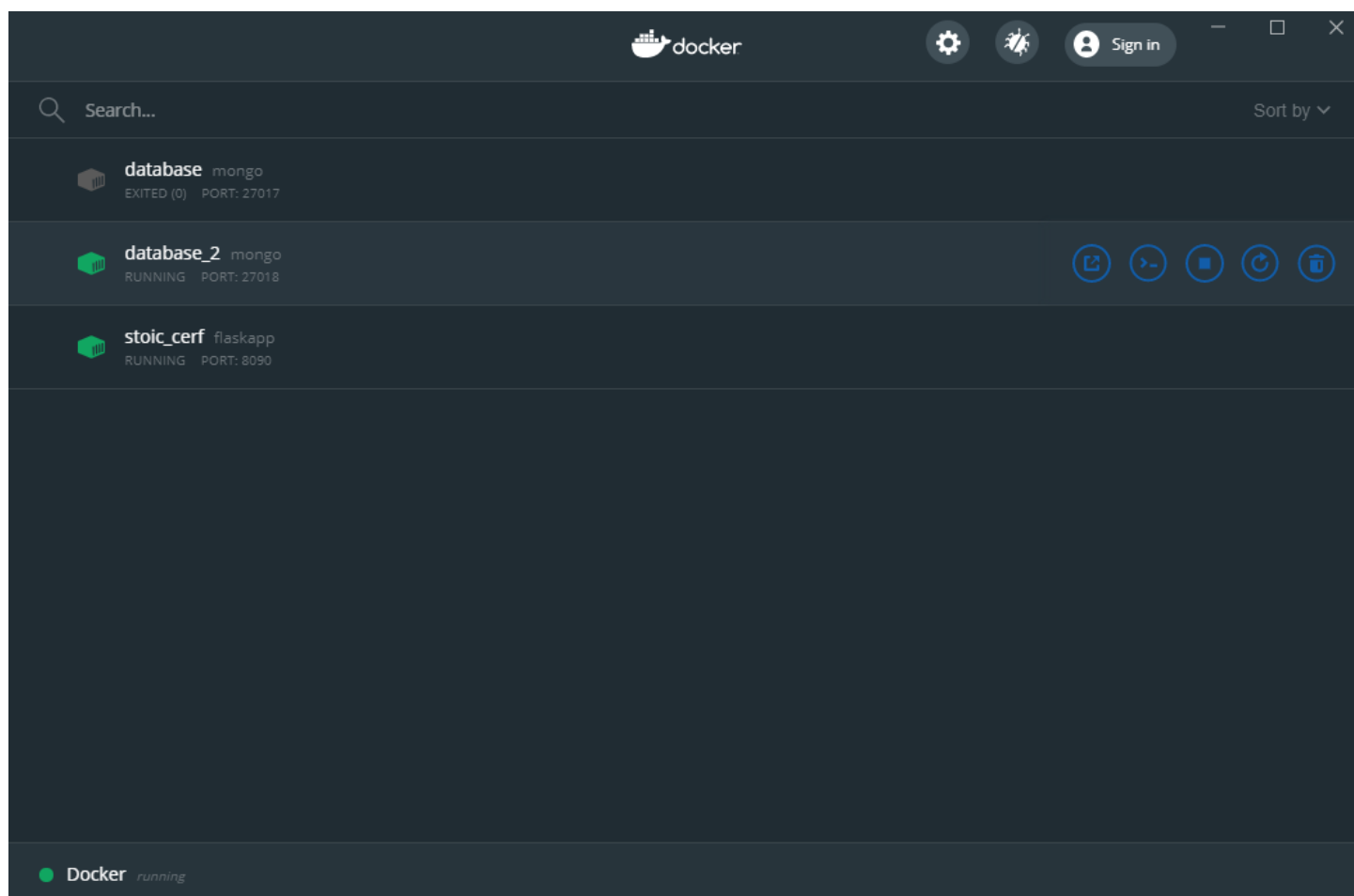
Ahora dentro de la consola de nuestra aplicación procederemos a correr la aplicación por el puerto 8090 el cual es el preestablecido en la practica para ello ejecutamos el comando “docker run -it --publish 8090:4000 -d flaskapp” el cual especifica que el puerto predeterminado de la aplicación se cambiara de 4000 al 8090

```
(venv) C:\Users\Jose Luis Espiritu\Desktop\SPS\DockerPythonFlaskSPS>docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
flaskapp             latest              76b77b10bd53       27 minutes ago     143MB
flaskapp_sps         latest              60d8b781b7cb       57 minutes ago     143MB
<none>               <none>             203c77383b8d       About an hour ago  291MB
<none>               <none>             fd174e3a11a4       About an hour ago  291MB
<none>               <none>             19a54cc558b3       About an hour ago  5.57MB
<none>               <none>             15bcc548e5c6       About an hour ago  294MB
mongo                latest              923803327a36       3 hours ago        493MB
alpine               edge                3c791e92a856       2 months ago       5.57MB
alpine               3.10                be4e4bea2c2e       4 months ago       5.58MB
alpine               3.11.2              cc0abc535e36       8 months ago       5.59MB

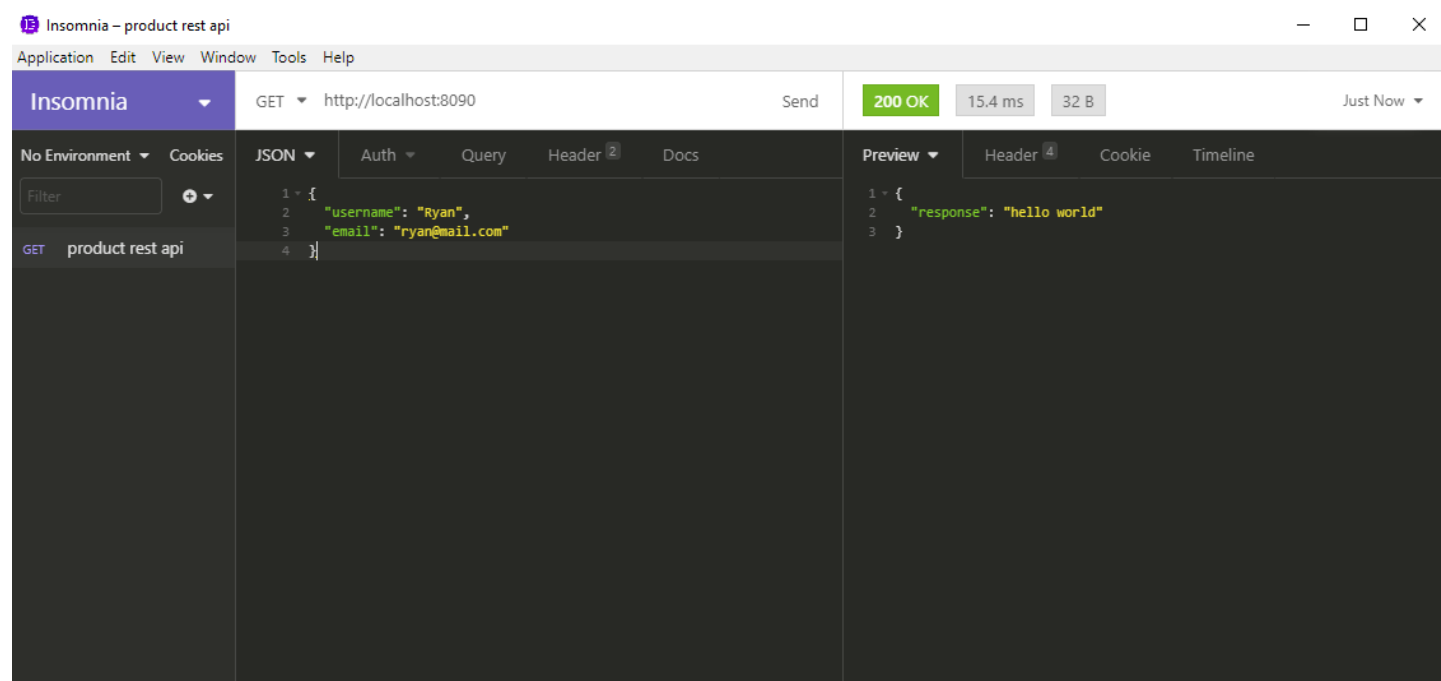
(venv) C:\Users\Jose Luis Espiritu\Desktop\SPS\DockerPythonFlaskSPS>docker run -it --publish 8090:4000 -d flaskapp
baf91ba47f0e1b64a1b477cd9ec4e9b45ae1435cb342562f164aba45a0a33bd5

(venv) C:\Users\Jose Luis Espiritu\Desktop\SPS\DockerPythonFlaskSPS>
```

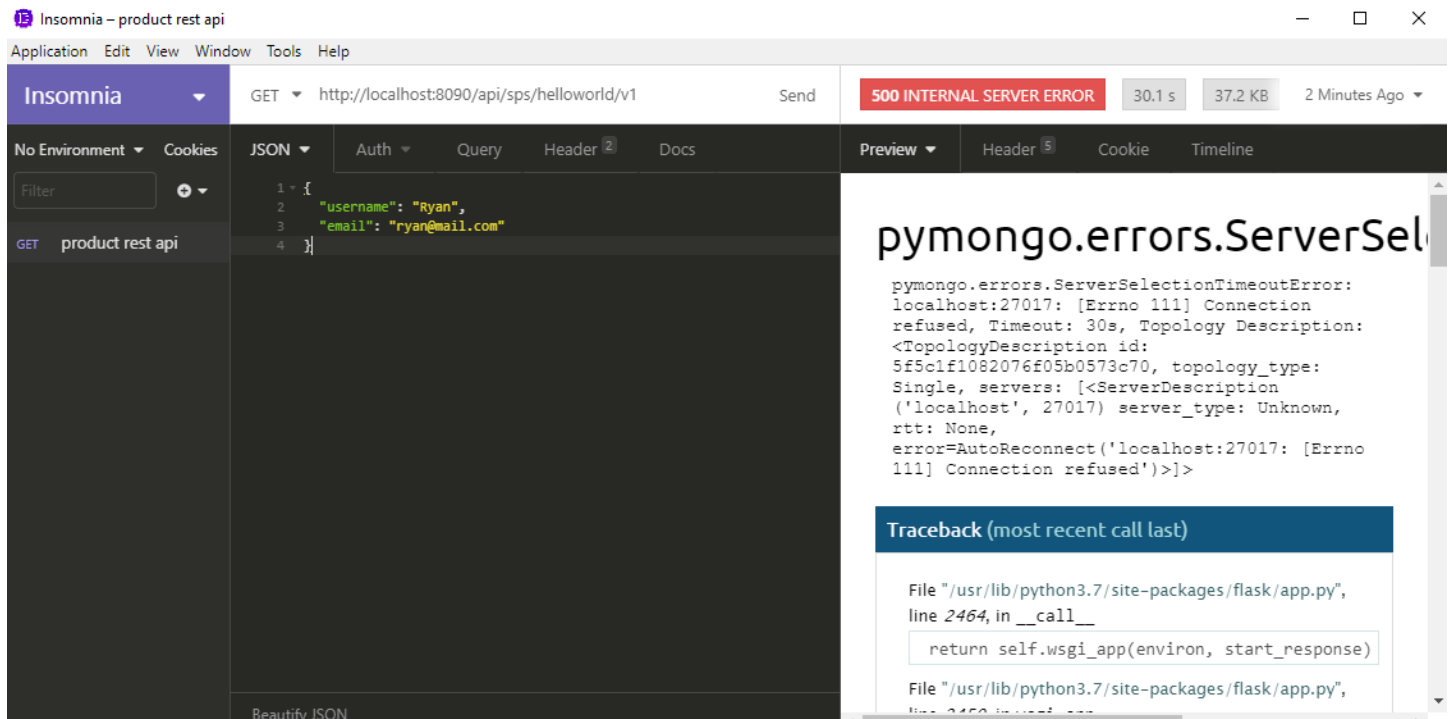
Procedemos a verificar en Docker que contenedores se están ejecutando



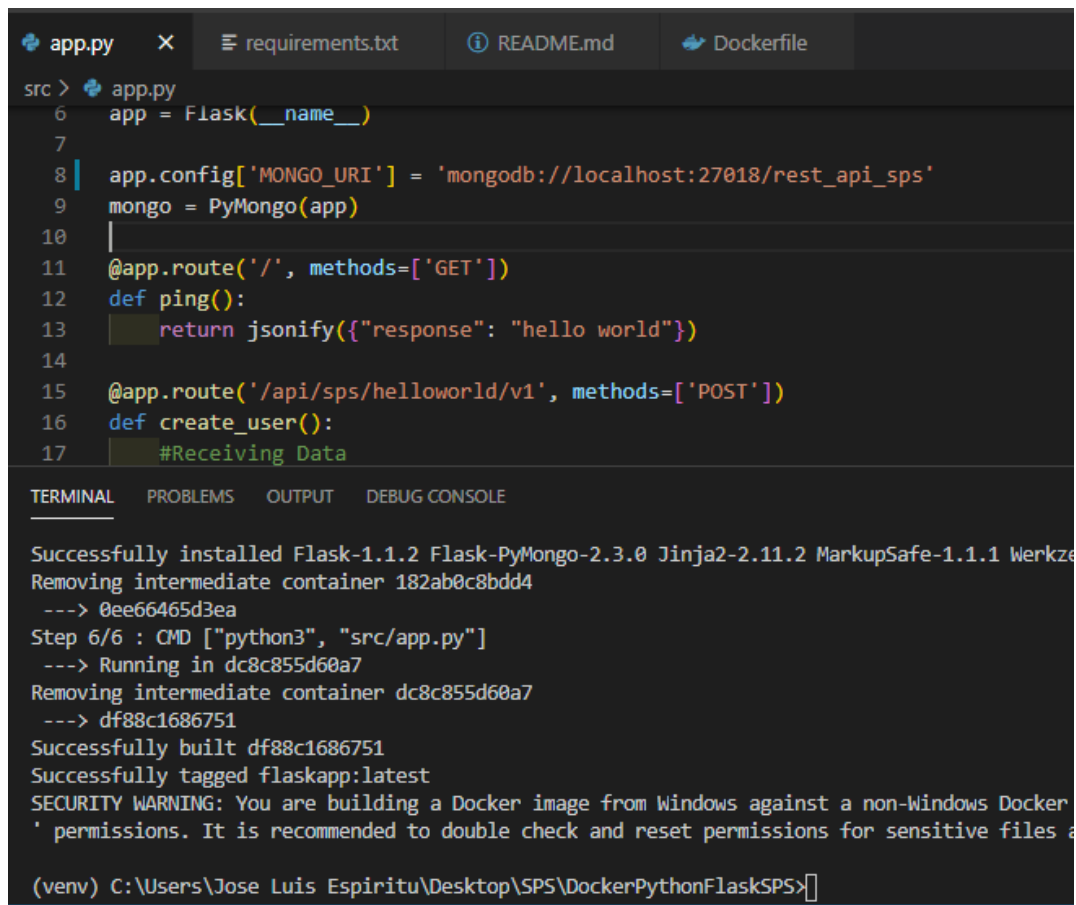
Ahora tenemos dos contenedores ejecutándose, y procedemos a probar la api con los métodos anteriormente creados, por medio del REST Client “Insomnia”



En esta parte me dio un error en el cual no se podía conectar con la base de datos



Para ello la solución es que en el archivo “app.py” tenía que especificar a que puerto se tiene que conectar, haciendo los cambios necesarios se vuelve a crear la imagen y a ejecutar esta imagen



En esta parte es donde aún no se como conectar con las bases de datos dentro de un contenedor porque me muestra un error al hacer la petición desde Insomnia

Insomnia – product rest api

Application Edit View Window Tools Help

Insomnia POST http://localhost:8090/api/sps/helloworld/v1 Send 500 INTERNAL SERVER ERROR 30 s 38.2 KB 5 Minutes Ago

No Environment Cookies JSON Auth Query Header 2 Docs Preview Header 3 Cookie Timeline

Filter

POST product rest api

```
1 {
2   "username": "Ryan",
3   "email": "ryan@mail.com"
4 }
```

Beautiful JSON

pymongo.errors.ServerSelectionTimeoutError

```
pymongo.errors.ServerSelectionTimeoutError:
localhost:27018: [Errno 111] Connection
refused, Timeout: 30s, Topology Description:
<TopologyDescription id:
5f5c2362721952089f4f4185, topology_type:
Single, servers: [<ServerDescription
('localhost', 27018) server_type: Unknown,
rtt: None,
error=AutoReconnect('localhost:27018: [Errno
111] Connection refused')>]>>
```

Traceback (most recent call last)

```
File "/usr/lib/python3.7/site-packages/flask/app.py",
line 2464, in __call__
    return self.wsgi_app(environ, start_response)
File "/usr/lib/python3.7/site-packages/flask/app.py",
line 2464, in __call__
```