

28 de agosto de 2022

José Alberto Alonso González

Práctica REST API y Contenedores

En el presente trabajo describiré el objetivo, marco teórico-práctico y puntos a resolver para la práctica presentada **a SPSolutions** como prueba de conocimientos respecto a arquitectura REST API y contenedores virtuales, así como de elementos prácticos de diagramación, elementos de Frontend y elementos de Backend relacionados a la estructura y diseño de una aplicación de despliegue en una página web.

Objetivo:

En la siguiente práctica se busca demostrar las capacidades y habilidades en relación a la interacción con arquitecturas REST API, el consumo de API's, contenedores y distintos lenguajes de programación, tratando de facilitar al lector y usuario, el grado de conocimiento que poseo (yo José Alberto Alonso González) de las tecnologías que se describen a continuación.

Para una mejor claridad en la redacción y seguimiento, se desglosarán uno por uno cada punto asociado en la práctica, no sin antes explicar la lógica del flujo de datos mediante la diagramación.

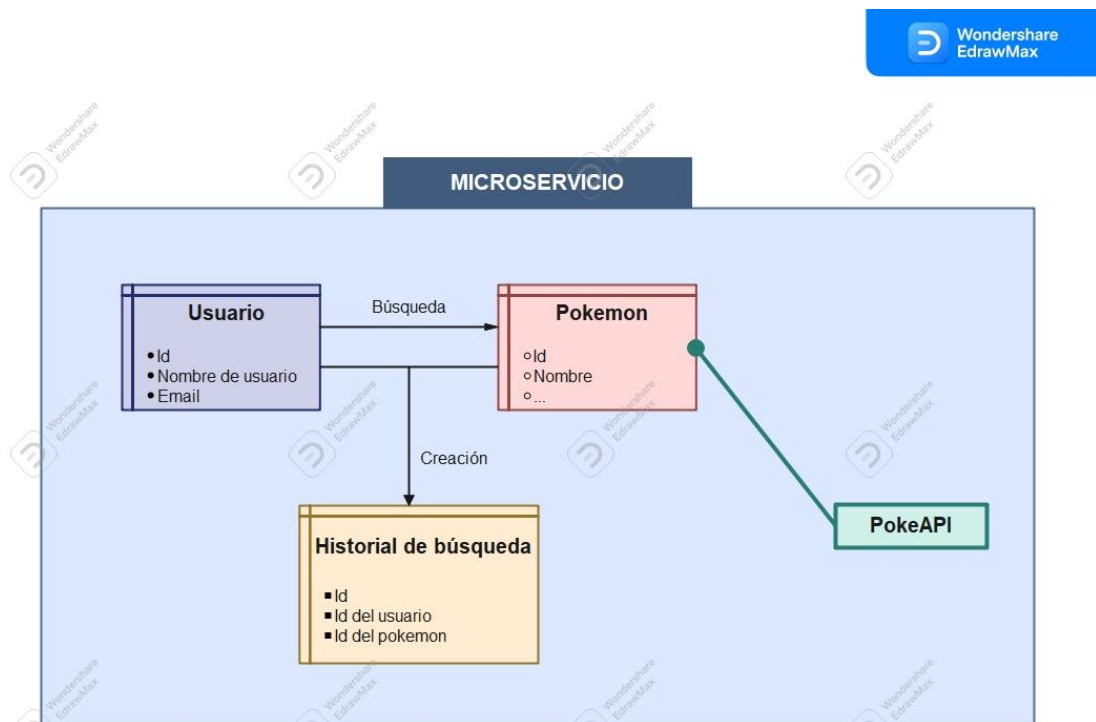


Diagrama elaborado mediante "EdrawMax"

El diagrama anterior se describirá de forma descendente de izquierda a derecha:

- ❖ Posterior a la creación de una base de datos se generaron las siguientes tablas para el almacenamiento de datos.
- ❖ La tabla Usuario recibirá la información de registro de los usuarios que entrarán a la aplicación generada, dando como datos un nombre y un correo electrónico, asociando este par de recursos a un id de *Primary Key*.

- ❖ Se creará una API para la búsqueda y consumo de recursos de la información de Pokemon expuesta por la página de internet PokeAPI.
- ❖ Retorna un objeto con los recursos específicos a consumir por la API.
- ❖ Mediante la aplicación se generará una tabla derivada que cargará consigo a la base de datos el apareamiento de la información del usuario y su relación de búsqueda de pokemons, dando como resultado el nexo mediante el id del usuario y el id del pokemon buscado por dicho usuario.

Ahora bien, una vez asentadas las bases del flujo de los recursos, se dio inicio a la creación de la aplicación web y sus API's correspondientes.

Pasos de la práctica

- 1) Elige el lenguaje de programación que más te guste.
 - Para dicho punto la elección fue Python con el framework Flask.
- 2) Define mensajes de entrada y salida para un API de tipo REST.
 - Este punto se verá postergado para casi el final debido a que la intención en esta práctica será el crear una aplicación web funcional con templates y formato, por tanto, en este punto no se podrá probar aún la API en un cliente REST (Insomnia en mi caso), por que lo que regresaría sería un html dinámico que necesitaría datos por detrás.
 - Sin embargo, para satisfacer exitosamente este punto, al final de la creación de la API en una página web se creó un apartado funcional con un cliente, pero sin que se entregue un html dinámico solo la respuesta normal, siendo el ingreso de los datos de usuario y el pokemon que el cliente REST buscaría.
 - La evidencia relacionada a este punto se muestra casi al final cuando la página web ya es funcional, justamente en el paso 5.
- 3) Internamente en la lógica de tu desarrollo, consumir alguna de las siguientes API's externas y desplegar los resultados de ese consumo.
 - La opción elegida fue la "PokeAPI", para tratar de mostrar la información de los pokemon y satisfacer este punto de la práctica.
- 4) Despliega el API REST en tu máquina.
 - Para satisfacer este punto se realizó el siguiente desarrollo:

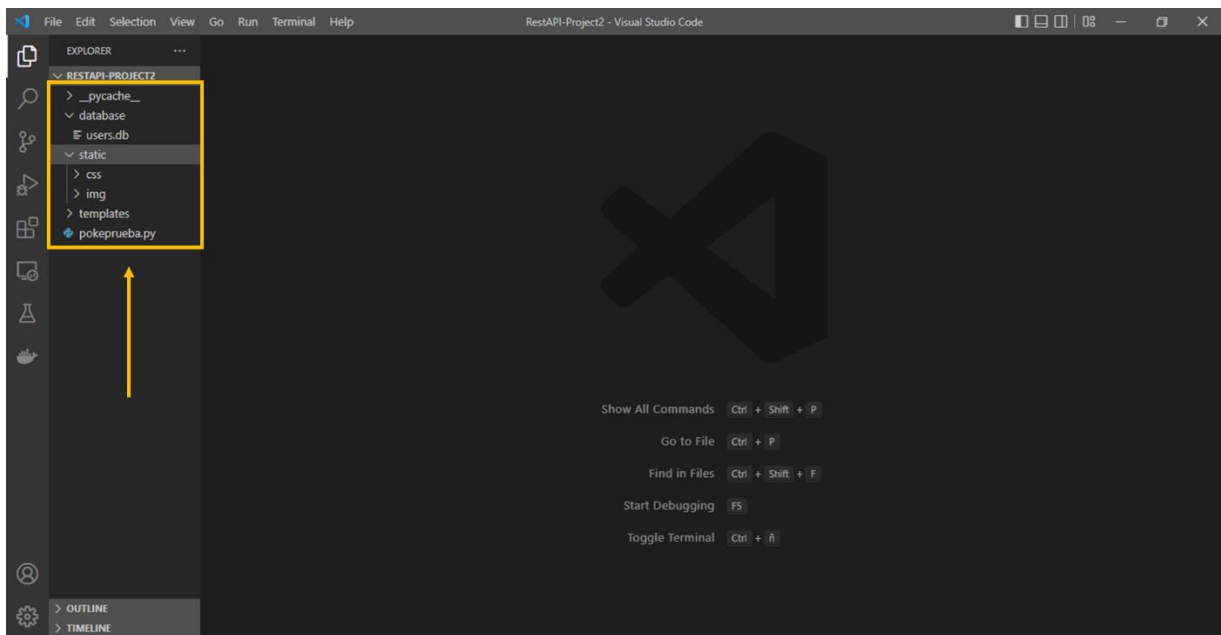
Al programa de ejecución en Python lo llamaremos “pokeprueba.py” y la interfaz de la página web contará con las siguientes rutas, junto con sus métodos asociados:

- ✓ / - (GET)
- ✓ /pokedex - (POST)
- ✓ /pokedex/ - (POST)

El siguiente paso es la creación de una base de datos en sqlite3, la cual contendrá 2 tablas con las siguientes columnas:

- Tabla: User
 - I. Id
 - II. Username
 - III. Email
- Tabla: Pokemon_History
 - I. Id
 - II. User_id
 - III. Pokemon_id

Asimismo, se crean carpetas para los archivos estáticos (imágenes en carpeta “img” y archivos css en carpeta “css”) y una más para los archivos html (carpeta “tempaltes”).



Creación de archivos

Se configura la base de datos y se crean columnas, con la siguiente sentencia de código.

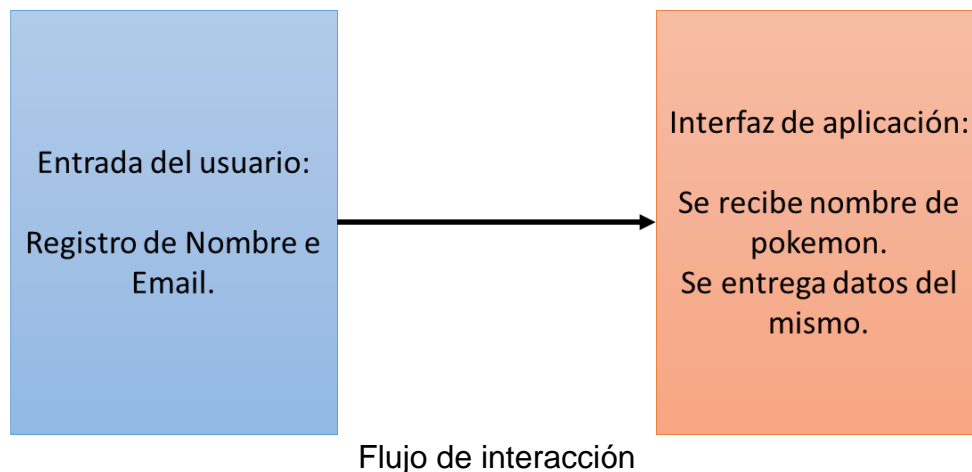
```
#Creación de Base de datos con tablas Usuario y Pokemon_History
#####
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///database/users.db'
db = SQLAlchemy(app)

class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(200))
    email = db.Column(db.String(200))

class Pokemon_History(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    user_id = db.Column(db.Integer)
    pokemon_id = db.Column(db.Integer)
#####
```

Porción de código en donde se generan las columnas y objetos asociados a las tablas en la base de datos.

Ahora bien, respecto al funcionamiento de la aplicación tendrá la siguiente estructura.



Respecto a la entrada del usuario se relacionará al template “home.html”, el cual estará directamente estilizado por el archivo css “style1.css”, por su parte la interfaz de aplicación estará relacionada de igual forma a un template y una estilización con

los archivos “index.html” y “style.css” respectivamente; dando la siguiente relación de archivos.

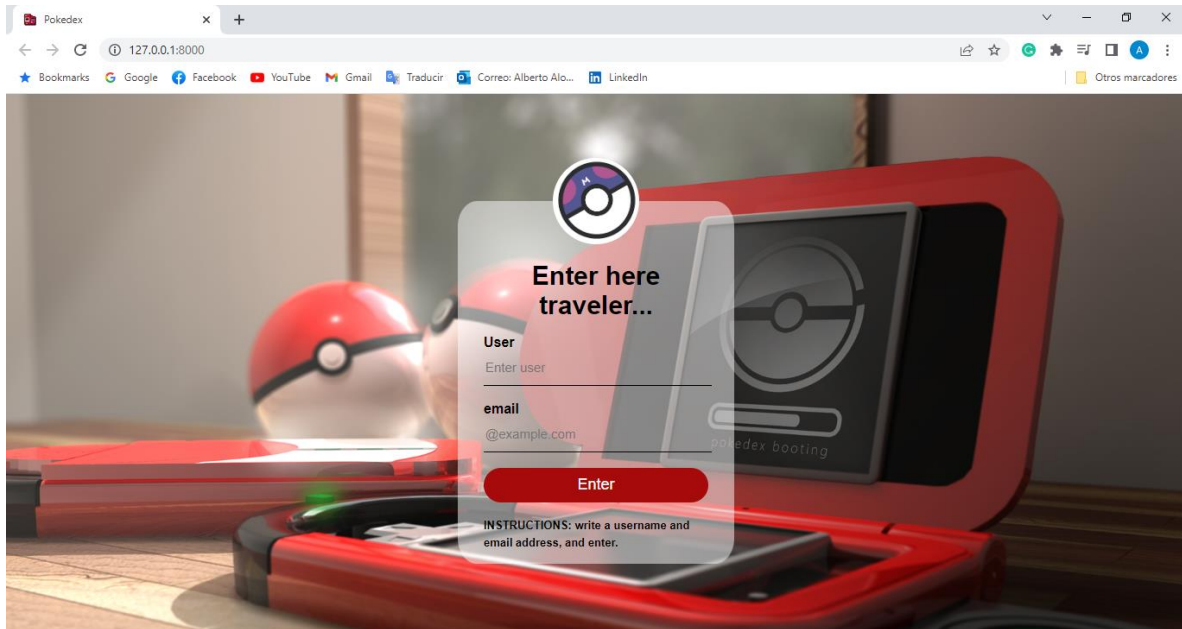
- Entrada de usuario:
 - home.html
 - style1.css
- Interfaz de aplicación:
 - index.html
 - style.css

Cabe mencionar que estos fueron estilizados en sus fuentes de letra con ayuda de “Google fonts”.

Respecto al código cabe mencionar que este se inicializa y manda al usuario a la interfaz de ingreso mediante la función “home()” mediante el método “GET”, retornando el template correspondiente.

```
#Ruta principal
@app.route('/', methods=['GET'])
def home():
    return render_template("home.html")
```

El resultado final de la renderización de la entrada de usuario es el siguiente:



Renderización de la entrada de usuario

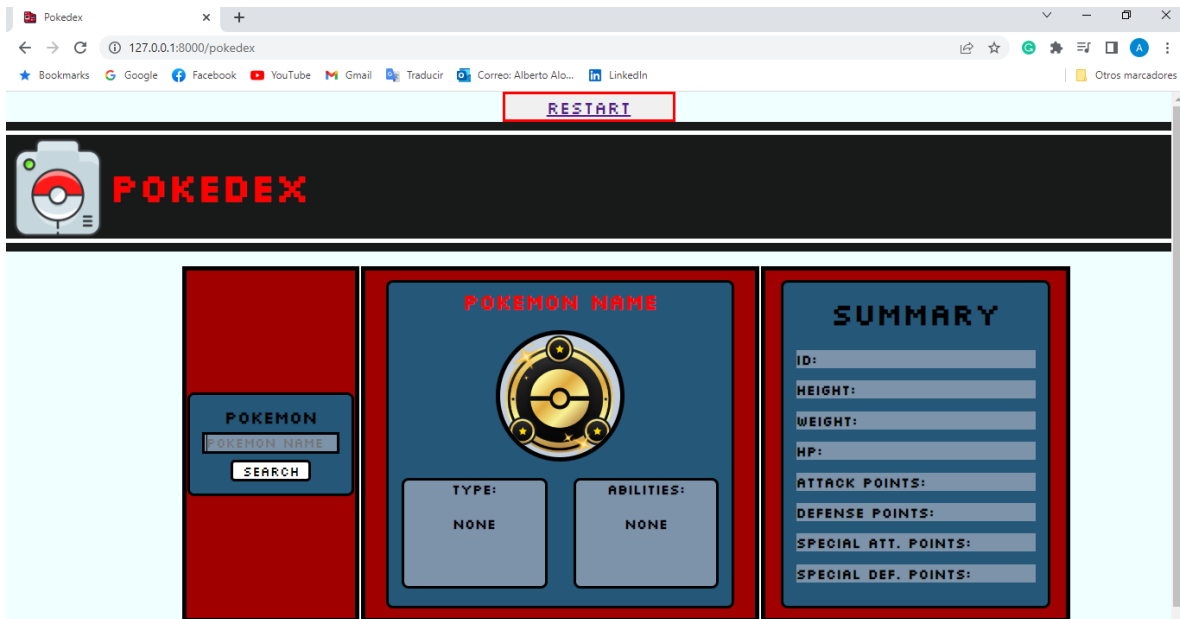
- ❖ Nota: para más información acerca de las funciones se puede consultar directamente el código y ver sus comentarios integrados en él.

Ahora bien, respecto a la interfaz de la aplicación se tuvo un desarrollo más robusto, ya que se necesitaba tener la posibilidad de mantener renderizado el template correspondiente mediante 2 fases, las cuales se enuncian a continuación junto con sus funciones y métodos asociados:

- a) Cuando se entrara por primera vez a la interfaz y se usa el método "POST" para ingresar los datos del usuario en la base de datos.

```
#Ruta para ingresar y agregar usuario
@app.route('/pokedex', methods=['POST'])
def add_user():
```

Con esta ruta se tiene la siguiente de la interfaz estática de la aplicación:



Renderización de la interfaz estática de la aplicación web.

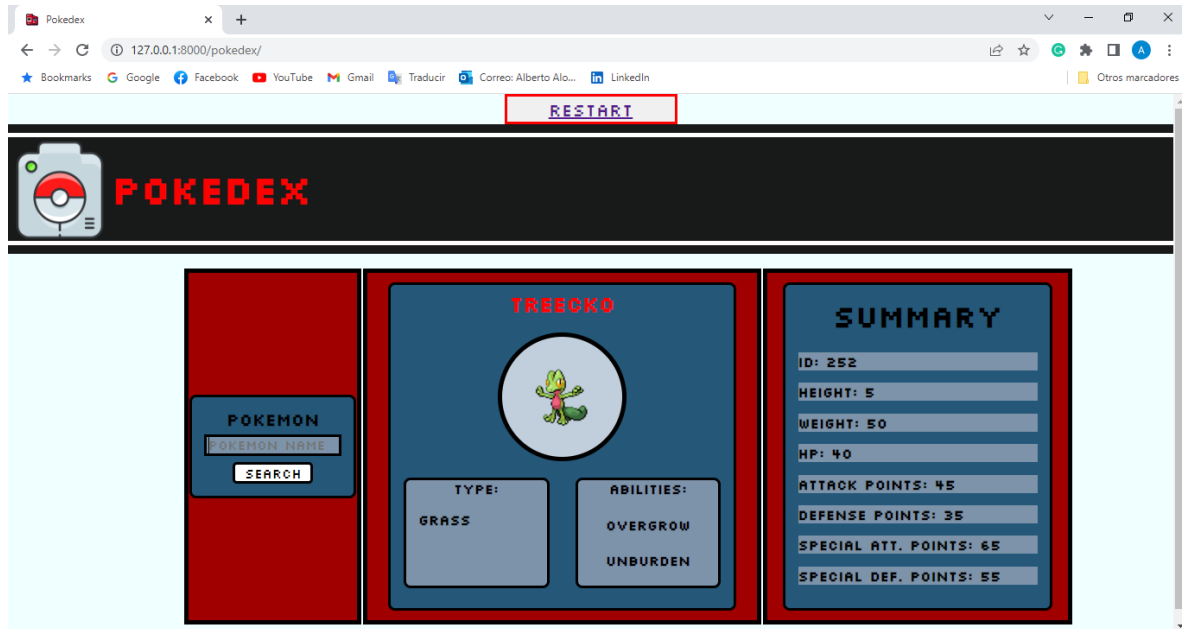
- b) Al consultar la información específica, consumo de API, lograr mantener el recurso obtenido y usar el método “POST” para ingresar los datos de la búsqueda asociada en la base de datos.

```
#Ruta para ingresar datos asociados al recurso
@app.route('/pokedex/', methods=['POST'])
def show_pokemon():
```

Es pertinente señalar que para lograr mantener el registro del usuario al momento de realizar este método POST fue necesario implementar una variable global, llamada “actual_user_id” inicializada al comienzo de la aplicación después de la creación de la base de datos.

```
# Variable global de tipo user.
actual_user_id = ""
```


Dando como resultado la consulta y renderización del pokemon deseado.



Renderización de la interfaz de consulta mediante API en la aplicación web.

Es necesario mencionar que mientras el usuario siga consultando pokemons en la misma interfaz, sin salir, se seguirán creando registros con sus datos gracias a la variable global declarada. Para crear nuevos registros de un nuevo usuario se implementó un botón de “Restart” para llevar al usuario a la primera interfaz y así entrar con un nuevo registro de nombre e email.

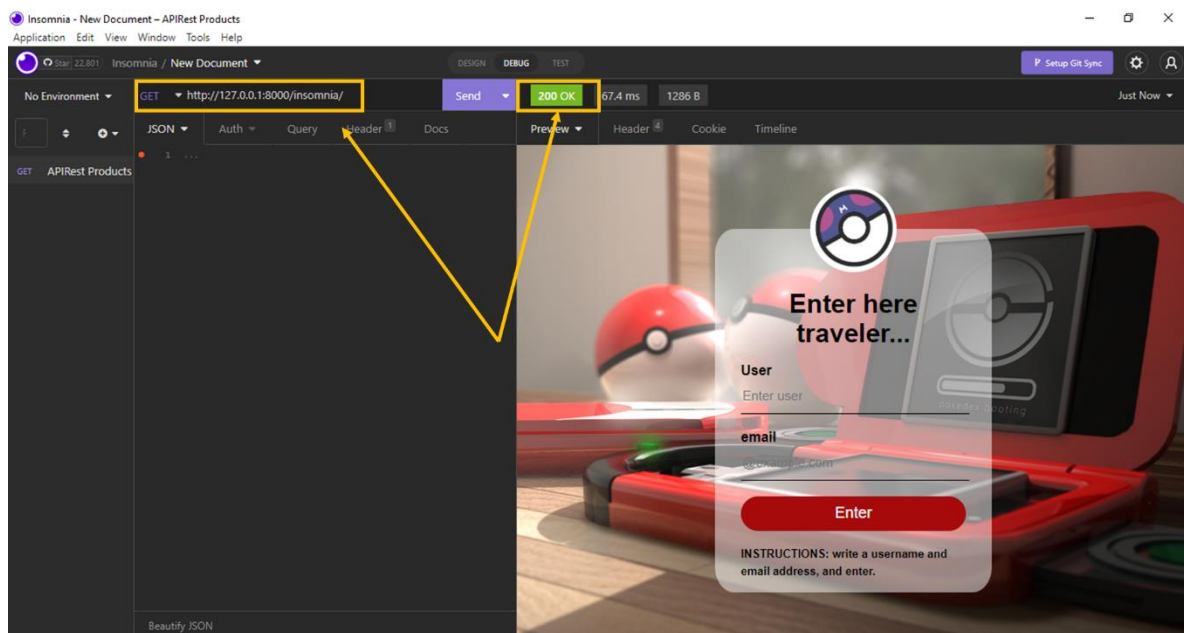
Con estas acciones implementadas fue posible realizar el flujo de datos mostrado al inicio de la práctica, pudiendo así crear registros relacionados entre usuarios y pokemons consultados, cumpliendo con ellos este punto a satisfacer.

- 5) Prueba el API con el cliente REST que más te guste y documenta.
-Adicionalmente, para este punto se realizó el siguiente desarrollo.

Se crearon 3 funciones espejo casi idénticas a las anteriores, pero dentro de estas se especificó una estructura para poder probar su funcionamiento con “Insomnia”, dando los siguientes resultados:

1. Prueba a método “GET” de la primera interfaz; ruta, función y resultado asociado.

```
#Ruta de prueba de primera interfaz - Insomnia
@app.route('/insomnia/', methods=['GET'])
def home_ins():
    return render_template("home.html")
```

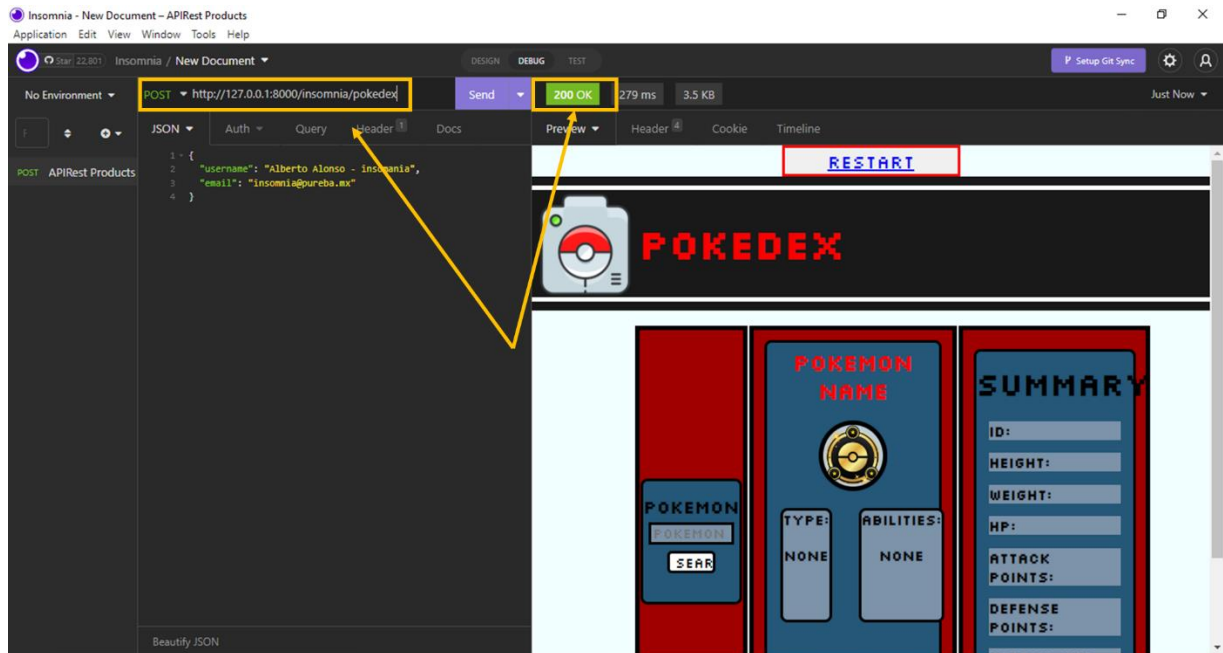


Resultado asociado a primera ruta de prueba.

2. Prueba a método “POST” para el registro del usuario; archivo json específico ruta, función y resultado asociado.

```
{"username": "Alberto Alonso - insomnia", "email": "insomnia@pureba.mx"}
```

```
#Ruta de prueba de registro de usuario - Insomnia
@app.route('/insomnia/pokedex', methods=['POST'])
def post_ins():
```



Resultado asociado a segunda ruta de prueba (“POST”).

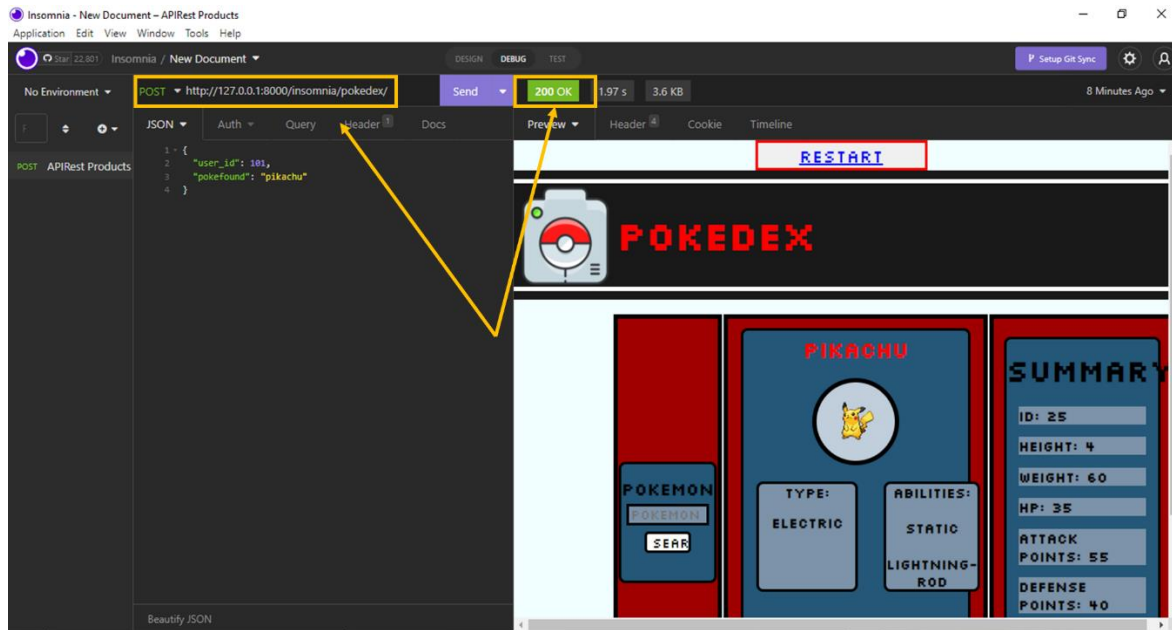
Asociado a esta prueba podemos ver el resultado guardado en nuestra base de datos, consultándolo desde sqlite3.

```
sqlite> select * from user;
1|Alberto Alonso - insomnia|insomnia@pureba.mx
sqlite> 
```

3. Prueba a método “POST” para el registro del pokemon consultado; archivo json específico ruta, función y resultado asociado

```
{"user_id": 1, "pokefound": "pikachu"}
```

```
@app.route('/insomnia/pokedex/', methods=['POST'])
def pokemon_ins():
|
```



Resultado asociado a tercera ruta de prueba (“POST”)

Asociado a esta prueba podemos ver el resultado guardado en nuestra base de datos, consultándolo desde sqlite3.

```
sqlite> select * from pokemon_history;
1|1|25
sqlite>
```

Con estos resultados comprobados se satisface este punto de la práctica.

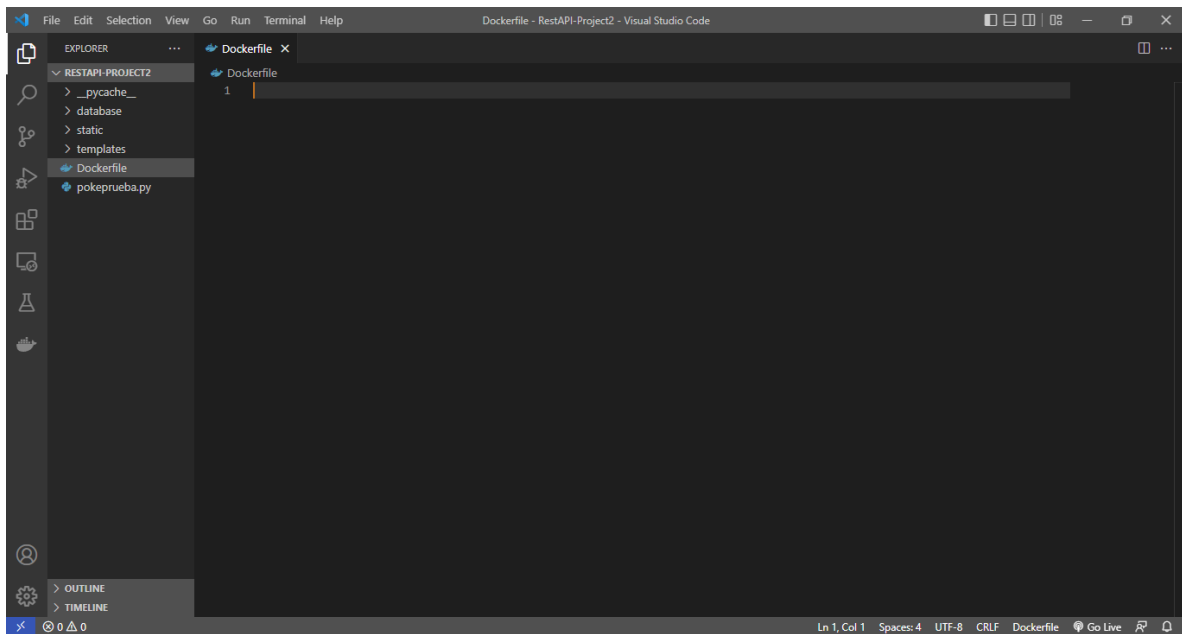
Adicionalmente, se pudiese hacer una consulta integral, para ver la relación de datos, en específico, el nombre del usuario en lugar del id del usuario.

```
sqlite> select user.username, pokemon_history.pokemon_id from user inner join pokemon_history on user.id = pokemon_history.user_id;
Alberto Alonso - insomnia|25
sqlite>
```

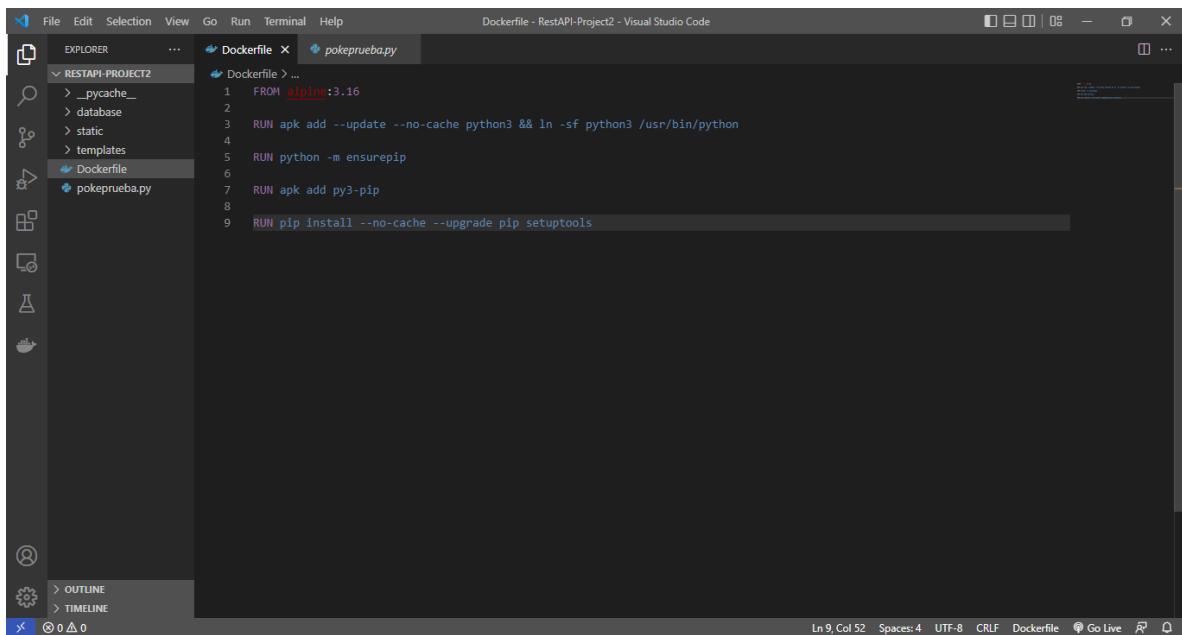
6) Elige una de las siguientes opciones para esta práctica.

- La opción elegida es la de inciso “a”, usando los contenedores de Docker como se muestra a continuación.

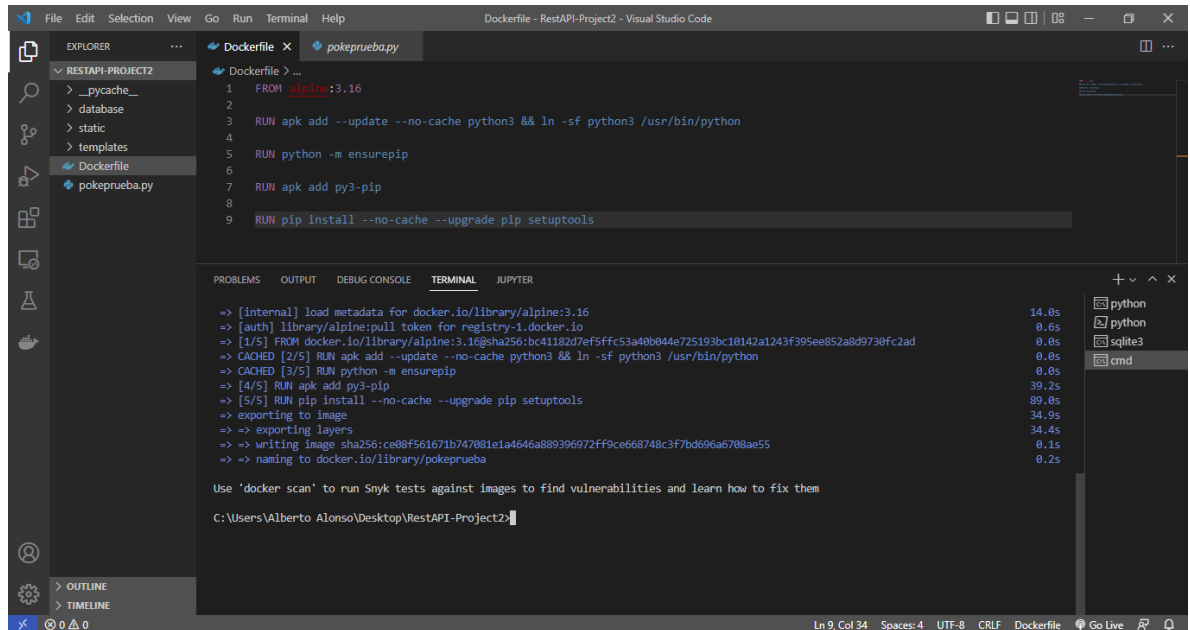
1. Se crea un archivo Dockerfile.



2. Se debe especificar un sistema Linux, en este caso se ocupará la última versión de “alpine”, y dentro de este se estaría instalando de igual forma Python. Esto haría que ya se tuviese el sistema base para ejecutar la app desde el contenedor.



3. A continuación, se crea la imagen de los programas involucrados.



The screenshot shows the Visual Studio Code interface with a Dockerfile open in the editor. The Dockerfile contains the following instructions:

```
1 FROM alpine:3.16
2
3 RUN apk add --update --no-cache python3 && ln -sf python3 /usr/bin/python
4
5 RUN python -m ensurepip
6
7 RUN apk add py3-pip
8
9 RUN pip install --no-cache --upgrade pip setuptools
```

The terminal output shows the execution of these instructions, including the installation of python3, pip, and setuptools. The output is as follows:

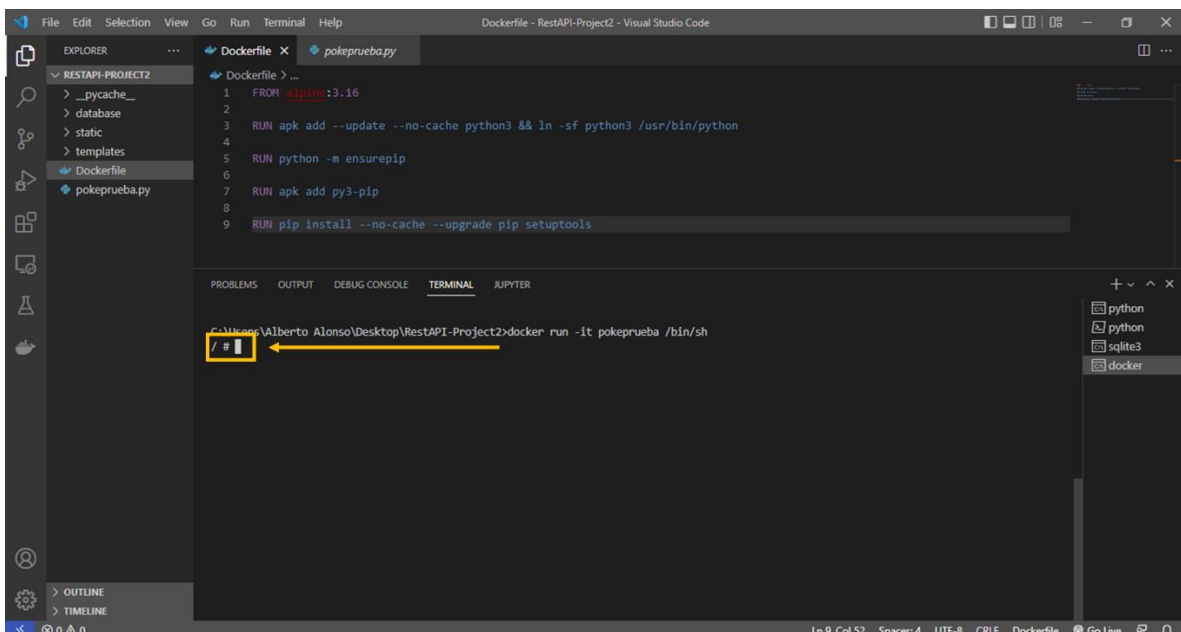
```
-> [internal] load metadata for docker.io/library/alpine:3.16
-> [auth] library/alpine:pull token for registry-1.docker.io
-> [1/5] FROM docker.io/library/alpine:3.16@sha256:bc41182d7ef5ffc53a40b844e725193bc10142a1243f395ee852a8d9730fc2ad
-> CACHED [2/5] RUN apk add --update --no-cache python3 && ln -sf python3 /usr/bin/python
-> CACHED [3/5] RUN python -m ensurepip
-> [4/5] RUN apk add py3-pip
-> [5/5] RUN pip install --no-cache --upgrade pip setuptools
-> exporting layers
-> writing image sha256:ce08f561671b747081e1a4646a889396972ff9ce68748c3f7bd696a6708ae55
-> naming to docker.io/library/pokeprueba

Use 'docker scan' to run Snyk tests against images to find vulnerabilities and learn how to fix them

C:\Users\Alberto Alonso\Desktop\RestAPI-Project2>
```

❖ Nota: asegurarse de que posterior a la creación de imagen se instalen los demás programas involucrados (pip install), en este caso, flask, flask_sqlalchemy y sqlite.

4. Posteriormente se debe ejecutar la aplicación en modo interactivo de Docker, accediendo a un cursor con el cual se puede interactuar con el sistema de “alpine” (/#).

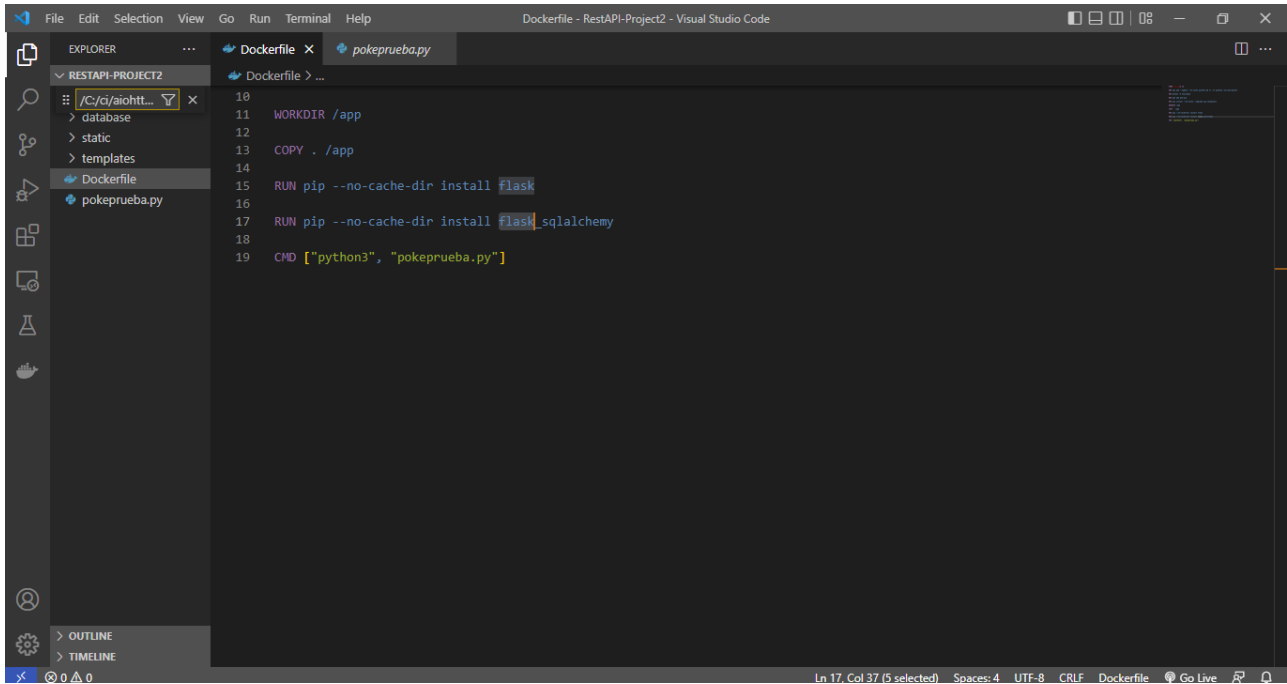


The screenshot shows the Visual Studio Code interface with the same Dockerfile open. The terminal output shows the execution of the Docker command to run the container:

```
C:\Users\Alberto Alonso\Desktop\RestAPI-Project2>docker run -it pokeprueba /bin/sh
/#
```

A yellow box highlights the prompt `/#` in the terminal, indicating that the container is running in interactive mode.

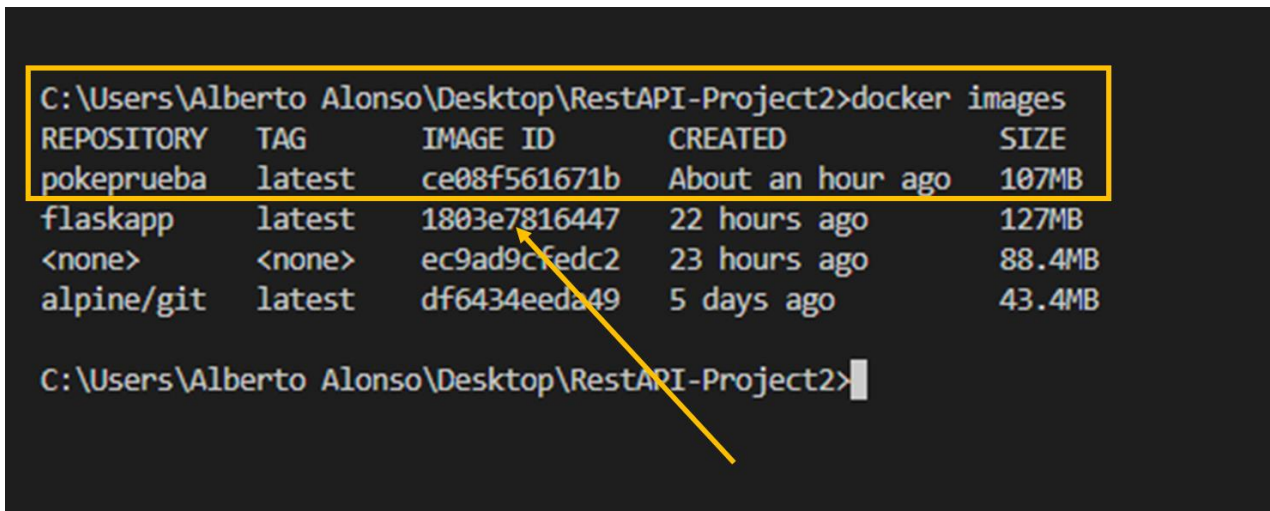
5. Se completa la imagen realizando la sentencia para que se instale “flask” y “flask_sqlalchemy”.



The screenshot shows the Visual Studio Code editor with a Dockerfile open. The file contains the following commands:

```
10  
11 WORKDIR /app  
12  
13 COPY . /app  
14  
15 RUN pip --no-cache-dir install flask  
16  
17 RUN pip --no-cache-dir install flask_sqlalchemy  
18  
19 CMD ["python3", "pokeprueba.py"]
```

6. Se comprueba la existencia de la imagen.



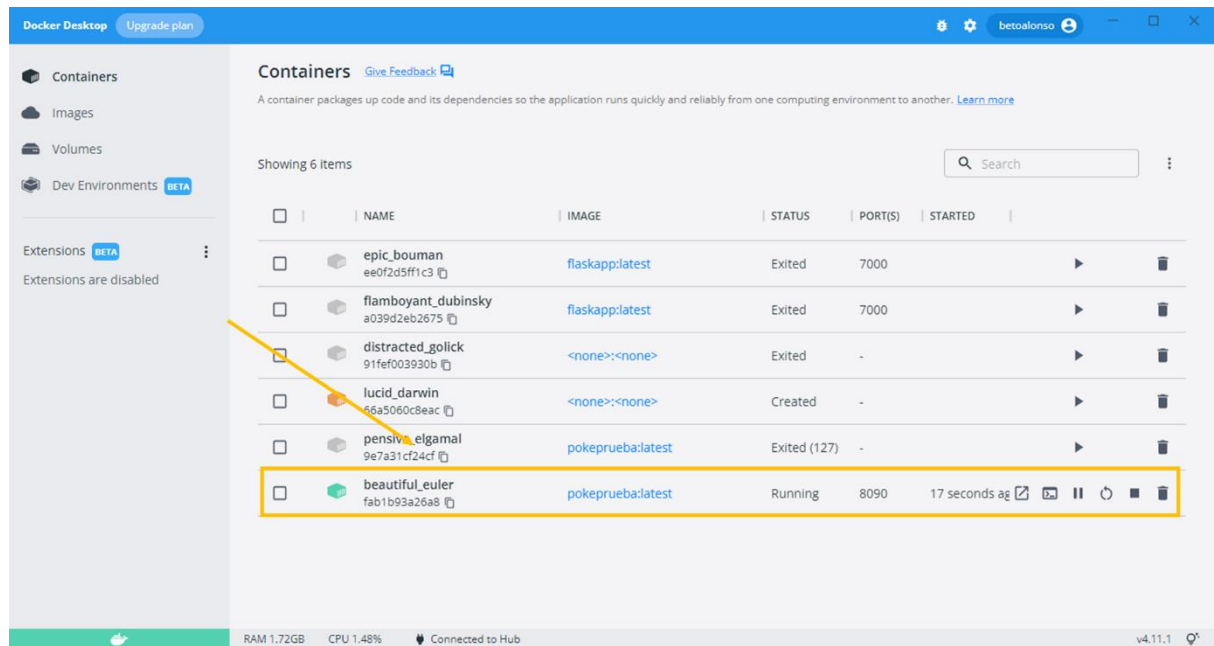
The screenshot shows a terminal window with the command `docker images` executed. The output is as follows:

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
pokeprueba	latest	ce08f561671b	About an hour ago	107MB
flaskapp	latest	1803e7816447	22 hours ago	127MB
<none>	<none>	ec9ad9cfedc2	23 hours ago	88.4MB
alpine/git	latest	df6434eeda49	5 days ago	43.4MB

7. Exponer al puerto especificado (8090) de Docker, desde el puerto local en donde se desplegó la aplicación.

```
C:\Users\Alberto Alonso\Desktop\RestAPI-Project2>docker run -it --publish 8090:8000 pokeprueba / #
```

8. Como comentario adicional desafortunadamente se especifica que se va a comentar en el código cualquier referencia a la biblioteca flask_sqlalchemy, debido a que en el sistema alpine no se reconoce esta biblioteca. Sin embargo, se adjunta prueba del contenedor levantado.




```
File Edit Selection View Go Run Terminal Help
Dockerfile - RestAPI-Project2 - Visual Studio Code

EXPLORER
RESTAPI-PROJECT2
  > __pyscache__
  > database
  > static
  > templates
  Dockerfile
  pokeprueba.py

Dockerfile
1
2
3 RUN apk add --update --no-cache python3 && ln -sf python3 /usr/bin/python
4
5 RUN python -m ensurepip
6
7 RUN apk add py3-pip
8
9 RUN pip install --no-cache --upgrade pip setuptools
10
11 WORKDIR /app
12
13 COPY . /app
14
15 RUN pip --no-cache-dir install flask
16

TERMINAL
-> naming to docker.io/library/pokeprueba

Use 'docker scan' to run Snyk tests against images to find vulnerabilities and learn how to fix them

C:\Users\Alberto Alonso\Desktop\RestAPI-Project2>docker run -it -p 8090:8000 pokeprueba
* Serving Flask app "pokeprueba"
* Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
* Running on http://127.0.0.1:8000
Press CTRL+C to quit
* Restarting with stat
* Debugger is active!
* Debugger PIN: 139-152-629
```

Comentarios adicionales:

- ❖ Se tuvo la dificultad de levantar el contenedor debido a las bibliotecas empleadas, para un futuro la expectativa es utilizar otro recurso para el manejo de las bases de datos, o solucionar el problema con los recursos empleados en esta práctica.
- ❖ Como proceso adicional, se podría realizar una validación en los registros de los usuarios para asegurar que cada uno tuviese su inicio de sesión, de la mano de esta idea también se podría implementar un token.