Docker Served File IO API

for back-end work and more ...

Overview

- Do you need an API to store and retrieve data for a data science web application?
- Do you need the same for an AI Architecture that you're developing?
- Do you want speed in sharing data between parts of a big application?
- Do you need to be able to store and retrieve data quickly from anywhere?
- Do you need this data storage and retrieval system to work super reliably?

If you answered yes to any of the questions above, then this talk is for you.

Together, we will:

- develop a standard Python class to store and retrieve data files;
- apply this class to a FastAPI API application;
- test this new API to make sure it works as expected; and
- serve this new API from within a Docker container.

In each step, I am trying to pass onto you some good practices that I have learned the hard way. I hope that what I will share will help you to save time in your API application development work.

There are 4 sections on the details for this work:

1. Python Module / Class Development
2. FastAPI API Development
3. Testing The API From A Python Script
4. Serving The API From A Docker Container

Python Module / Class Development

I have found that API development with Python FastAPI is much faster when I develop and test the underlying Python code first. Once that code is working as expected, I then import that to my FastAPI application. The functionality that we need for this API does NOT really need to go into a Python class. I will put each of the functions we need into a separate Python file / module that we will import into the FastAPI application. When I show you how to create the SQL API, we will have good reasons to use a class for that work.

The File IO Code Development

Even though I will present the code for our File_IO.py module in separate cells, all of these cells make up the single File_IO.py file. The File_IO.py file is also in the repository for this talk.

First we import the necessary Python modules and set an important global variable - `files_location`.

In [ ]:
```
import json
import os


files_location = "./files"
```

The next function is a convenience function. We won't use it directly in this file. We will use it later from our API file. It simply creates a directory if it doesn't exist.

In [ ]:
```
def prepare_dir(dir_name):
    if not os.path.isdir(dir_name):
        os.makedirs(dir_name)
```

The next function is also a convenience function is intended to ONLY be used in this module, BUT it could be used from other places that import this module. This function is similar to `prepare_dir` function, but this one actually looks at the file path for the file passed to the function and creates any directories under our `files_location` that do not already exist.

In [ ]:
```python
def prepare_file_dir(file_name):
    tree_list = file_name.split("/")[:-1]
    if tree_list:
        dir_path = "/".join(tree_list)
        if not os.path.isdir(dir_path):
            os.makedirs(dir_path)
```

The `load_object_from_json_file` function first builds the complete file path to where the file should be located. Then it uses the `prepare_file_dir` function to create any directories that do not already exist. Next, the logic block reads the json file using `UTF-8` encoding IF the file exists and converts the stored object in the file from a stringified JSON object to the original object. If the file does not exist, an empty object (Python dictionary) is returned. Note the use of context managers throughout. Context managers help to create clean code. By using `with`, we ensure that file alias `f` will be closed when the code within the `with` block is completed.

In [ ]:
```python
def load_object_from_json_file(file_name):
    file_name = f"{files_location}/{file_name}"
    prepare_file_dir(file_name)
    if os.path.exists(file_name):
        with open(file_name, "r", encoding="utf-8") as f:
            object = json.load(f)
    else:
        object = {}

    return object
```

The `store_object_to_json_file` function is used to store a Python dictionary as a string to a JSON file. This function is pretty much the reverse of the previous function. We build the file_name. We create any directories that need to be created. We open the JSON file for writing, and we use the `json.dump` method to store the JSON data to file. Note the following:

- `encoding="utf-8"` ensures we can store most anything
- `ensure_ascii=False` makes sure we retain the flexibility of `utf-8`
- `indent=4` makes the file easy on the eyes when reading
- `default=str` is MAGICAL in that it helps with tougher fields that could be hard to store as JSON

Finally, I make sure the `file_name` exists to check operations.

In [ ]:
```python
def store_object_to_json_file(the_dict, file_name):
    file_name = f"{files_location}/{file_name}"
    prepare_file_dir(file_name)
    with open(file_name, "w", encoding="utf-8") as f:
        json.dump(the_dict, f, ensure_ascii=False, indent=4, default=str)

    if os.path.exists(file_name):
        # print(f"Successfully stored {file_name} to {file_name}")
```

```
        return {"status": "Success"}
    else:
        # print(f"Failed to store {file_name} to {file_name}")
        return {"status": "Failure"}
```

The next two functions, `load_text_from_file` and `store_text_to_file`, are to be called when we are working with simple text files.

In [ ]:
```
def load_text_from_file(file_name):
    file_name = f"{files_location}/{file_name}"
    prepare_file_dir(file_name)
    if os.path.exists(file_name):
        with open(file_name, "r", encoding="utf-8") as f:
            text = f.read()
    else:
        text = ""

    return text
```

In [ ]:
```
def store_text_to_file(text, file_name):
    file_name = f"{files_location}/{file_name}"
    prepare_file_dir(file_name)
    with open(file_name, "w", encoding="utf-8") as f:
        f.write(text)

    if os.path.exists(file_name):
        return {"status": "Success"}
    else:
        return {"status": "Failure"}
```

Finally, we may want to remove a file from the server file storage. NOTE that you could, at this point, or in the API, add additional functionality to remove:

- all files
- all of a certain type of files (*.txt, *.html, *.json, etc.)

In [ ]:
```
def remove_file(file_name):
    file_name = f"{files_location}/{file_name}"
    if os.path.exists(file_name):
        os.remove(file_name)
    else:
        return {"message": f"{file_name} does not exist"}

    if not os.path.exists(file_name):
        return {"message": f"Successfully deleted {file_name}"}
```

File IO Module Testing

I will only test a few things here for the sake of time and amount of writing. I have tested all of these before. When you create such a module with functions, or a class or classes in a module, you will of course want to test adequately.

In [ ]:
```
prepare_dir(files_location)
```

When I ran the above function, the `files` directory was created as expected.

In [ ]:
```
the_D = load_object_from_json_file("first_file.json")
the_D
```

Out[ ]:
```
{}
```

When running the `load_object_from_json_file` function, we obtained an empty dictionary as expected, since there is no `first_file.json` file yet.

In [ ]:
```
the_D["key_1"] = 1.0
the_D["key_2"] = 2.0

store_object_to_json_file(the_D, "first_file.json")
```

Out[ ]:
```
{'status': 'Success'}
```

I created two key value pairs for `the_D` and stored them using `store_object_to_json_file`. The function let us know that we stored data to `first_file.json` successfully. If you are running this notebook yourself, you will also see `first_file.json` in the `files` subdirectory. Look at the file in your integrated development environment (IDE). I am using VS Code, and I am running this notebook in VS Code. However, let's also look at the contents by repeating the operations of a previous cell.

In [ ]:
```
the_D = load_object_from_json_file("first_file.json")
the_D
```

Out[ ]:
```
{'key_1': 1.0, 'key_2': 2.0}
```

Nice. Those few tests worked well. I leave it to you to test the other functions.

Please also NOTE that a file named `File_IO_Usage.py` was created to replicate the above testing in a single Python script. I encourage you to at least look at it. To test it, I would suggest first deleting the `files` subdirectory so that you can test from scratch.

FastAPI API Development

The next step is to create a FastAPI application. Thanks to doing the module work, we can make FastAPI code VERY lean.

As before, I will show all the code contiguously, but I will do so in separate cells with comments in between. This section only covers the code development for the FastAPI API. The name of the file containing all the FastAPI code is `File_IO_API.py`.

The Code Development

The first code cell covers the imports that we need. I trust the import of the FastAPI class from fastapi is clear. We will use that soon. The CORSMiddleware import is new. We need this so that we can tell FastAPI the web pages / apps that are allowed to use it. Then from typing and pydantic, we import List and BaseModel, respectively. These are useful data type classes, and their power will soon become evident. We will want access to our `File_IO` module, thus we import and alias that to `file_io`. The json and os imports will be explained when necessary below.

In [ ]:
```
from fastapi import FastAPI
from fastapi.middleware.cors import CORSMiddleware
from typing import List
from pydantic import BaseModel

import File_IO as file_io
import json
import os
```

Next, we will create our first Pydantic data object. NOTE that the data object is contrived for the purpose of illustration. This first object is storage for each client keeping with the scheme from our first exercise at Data Hour.

In [ ]:
```
class Client_Storage_Object(BaseModel):
    File_Name: str
    Client_Name: str
    Client_Email: str
    Client_Phone: str
    Client_Company: str
    Client_ID: str
    Agent_Name: str
    Agent_Email: str
    Agent_Phone: str
```

Let's create one more for illustration purposes on how it can affect the API endpoint development.

In [ ]:
```
class Company_Storage_Object(BaseModel):
    File_Name: str
    Rep_Name: str
    Rep_Email: str
    Rep_Phone: str
```

The next part is ESSENTIAL!

In [ ]:
```
app = FastAPI()
```

We talked about `file_location` previously. It is being set here as a global variable so that we can just use its value throughout the remaining code. The `origins` holds the

URL, or domain names, of those web pages that are allowed to use this API via JavaScript fetch calls. Those origins are then added to the `app.add_middleware` tuple.

In [ ]:
```
files_location = "./files"

origins = [
    "http://127.0.0.1:3000", # Outlook Add-In
    "http://127.0.0.1:5500"  # VS Code Live Server
]

app.add_middleware(
    CORSMiddleware,
    allow_origins=origins,
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)
```

Next, we will create our first endpoint - `/list_files`. This could also be called an HTTP method. Note the use of a Python decorator. These are one of those beautiful Pythonic things. I also love context managers as previously discussed. Decorators are an elegant way to create a wrapper around a function so that it can be used in a different fashion. I highly encourage you to study them on your own as you have the chance.

Note that we declare a function name after the decorator. By personal convention, I like to use the HTTP Method followed by a clear set of words explaining the nature of this function. Next, we use the `prepare_dir` function from the `File_IO` module.

In [ ]:
```
@app.get("/list_files")
def get_json_file_list():
    file_io.prepare_dir(f"{files_location}")
    file_list = os.listdir(f"{files_location}")

    return file_list
```

Begin Testing Now From The Docs

Since we have one endpoint now, let's start testing WHILE we develop the rest of the code. I will open up a command window. I like ConEMU when I have to work in Windows. If you are fortunate enough to be working in Linux, just use your Linux Terminal window. From the command line, after you've activated your Python Virtual Environment that you want to use, run

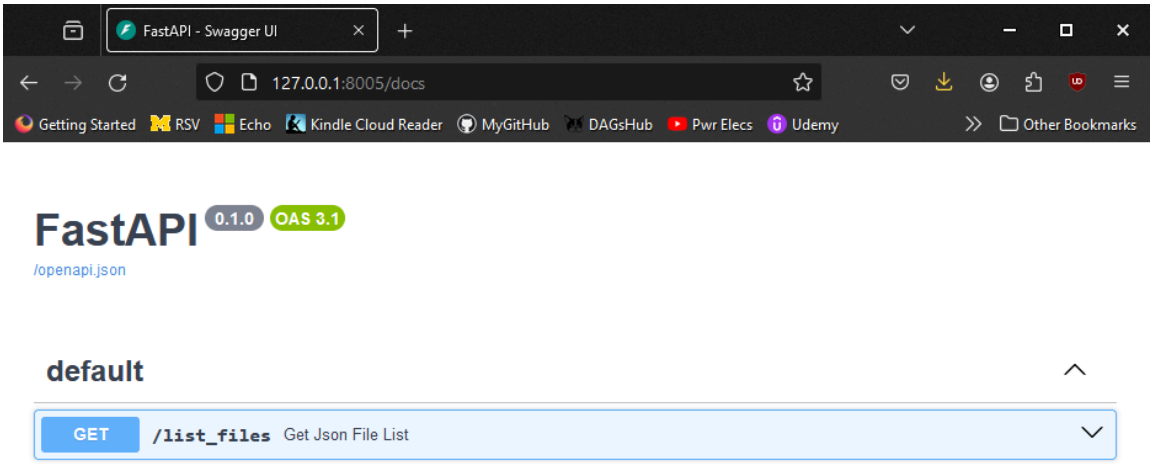
`uvicorn File_IO_API:app --port 8005 --reload`
This will start the FastAPI server on port 8005. Note that we use the name of the FastAPI Python Script file MINUS the `.py` part. If we hadn't declared the port number with `--port 8005`, uvicorn would have used the default of 8000. I like to declare the port,

because I am often running multiple APIs on the same server, and I want to control their port numbers. The entire session in my ConEMU terminal window is shown next.

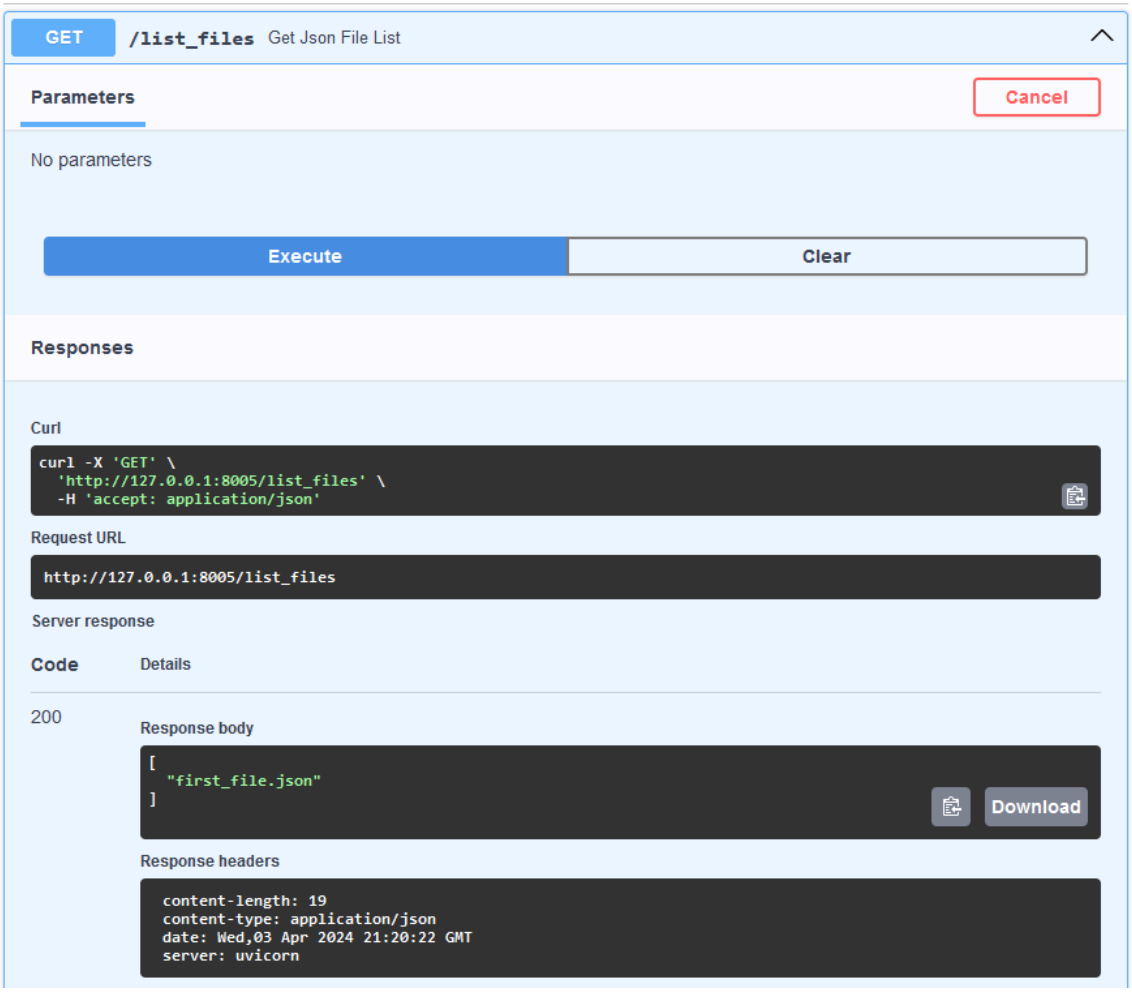
```
Microsoft Windows [Version 10.0.19045.4170]

tives@VDIDWS-A2-105
C:\Users\tives\Documents\Repos\File_IO_API_and_Docker
$ C:\Users\tives\Envs\py312std\Scripts\activate

(py312std) tives@VDIDWS-A2-105
C:\Users\tives\Documents\Repos\File_IO_API_and_Docker
$ uvicorn File_IO_API:app --port 8005 --reload
INFO:     Will watch for changes in these directories:
['C:\\Users\\tives\\Documents\\Repos\\File_IO_API_and_Docker']
INFO:     Uvicorn running on http://127.0.0.1:8005 (Press CTRL+C to
quit)
INFO:     Started reloader process [20092] using StatReload
INFO:     Started server process [23964]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
INFO:     127.0.0.1:60949 - "GET /docs HTTP/1.1" 200 OK
INFO:     127.0.0.1:60949 - "GET /openapi.json HTTP/1.1" 200 OK
```

Now NOTE that those final two lines will show up when you open your browser to a new tab and go to ...

`http://127.0.0.1:8005/docs`
which will show ...



Click on the bar for that endpoint, then click on `Try It Out`, and then click on the blue `Execute` bar. You should see the following.



This is ENCOURAGING! We can continue to develop and test like this. AND every time that you update File_IO.py or File_IO_API.py, the command terminal will communicate what FastAPI is doing.

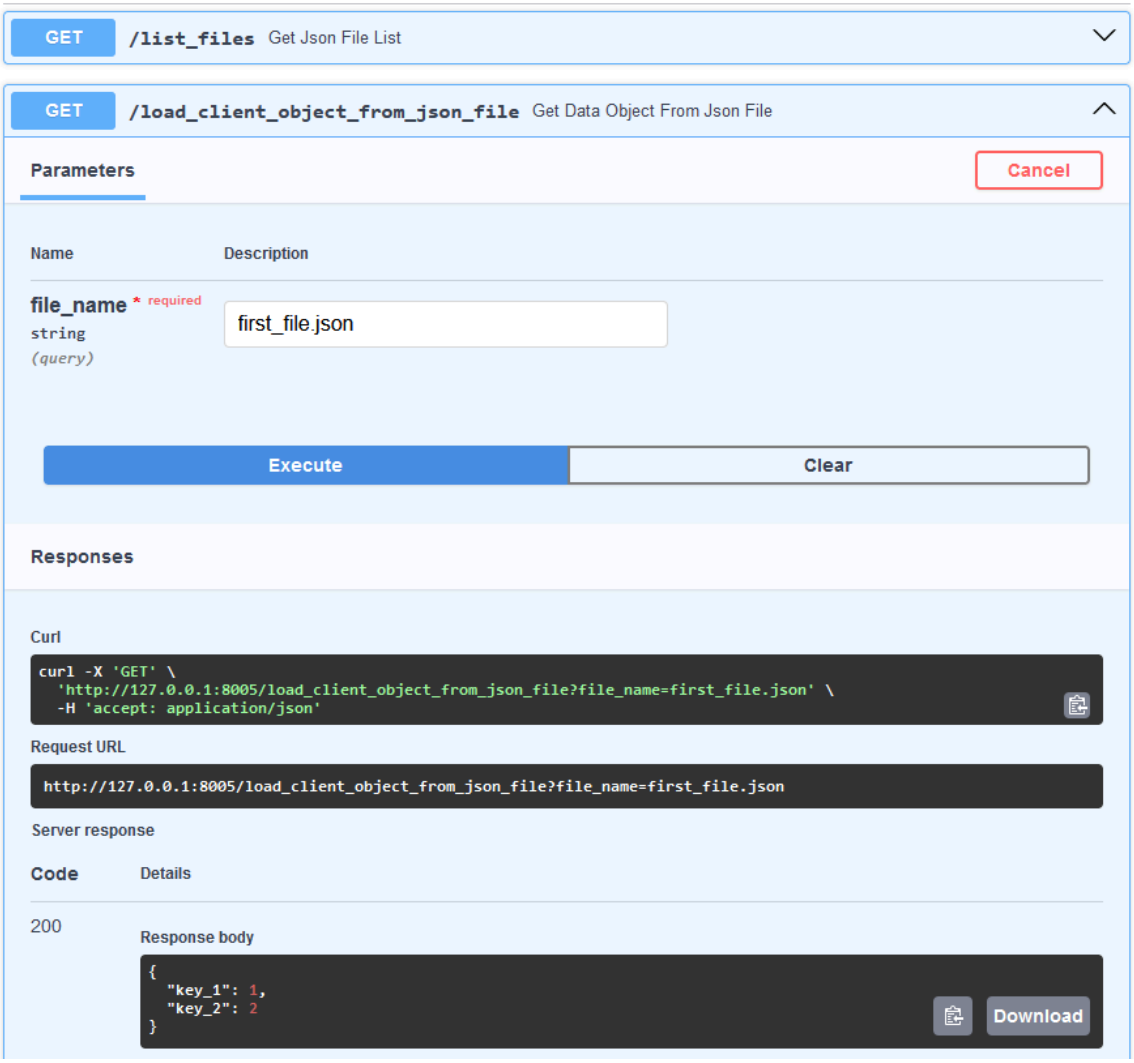
Let's write our second endpoint now.

In [ ]:
```python
@app.get("/load_client_object_from_json_file")
def get_data_object_from_json_file(file_name: str):
    data_obj = file_io.load_object_from_json_file(file_name)

    return data_obj
```

NOTE that when that was added to your File_IO_API.py file and saved, the command terminal communicated that changes were detected, and that the uvicorn server was shutdown and restarted. THUS, you will also want to reload your docs page in your browser. You will NOT see the new endpoint until you do.

We can now follow similar steps as we did before. We click on the new bar for our new endpoint in the browser. We then click on `Try It Out`, but before we click on Execute, we must enter the file name that we want our object from. Let's enter the name of our test file that we've been using - `first_file.json`. Ah! It worked! Encouraging!



OK. Now for the tricky part. We need to store data to a json file THROUGH our API. If we follow the general methods that I am about to show, it should usually go very simple for you.

In [ ]:
```
@app.post("/store_client_object_to_json_file")
def post_user_object_to_json_file(storage_object: Client_Storage_Object):
    the_dict = {}

    the_dict["File_Name"] = storage_object.File_Name
    the_dict["Client_Name"] = storage_object.Client_Name
    the_dict["Client_Email"] = storage_object.Client_Email
    the_dict["Client_Phone"] = storage_object.Client_Phone
    the_dict["Client_Company"] = storage_object.Client_Company
    the_dict["Client_ID"] = storage_object.Client_ID
    the_dict["Agent_Name"] = storage_object.Agent_Name
```

```
    the_dict["Agent_Email"] = storage_object.Agent_Email
    the_dict["Agent_Phone"] = storage_object.Agent_Phone

    return file_io.store_object_to_json_file(the_dict, storage_object.File_N
```

Hey! That doesn't look too bad - right? We name an incoming data object, `storage_object`, and declare it bo be of type `Client_Storage_Object`, which was defined previously and inherited from the Pydantic BaseModel. Then, we declare an empty Python dictionary. Next, we fill that dictionary with the elements of the `storage_object`. Since we conveniently made one of the elements the file name, we can use that directly, along with our `the_dict` dictionary, and then, finally, we can use our `store_object_to_json_file` function to store our dictionary to our file name. Once we save our `File_IO_API.py` file with this new endpoint, the uvicorn server will restart, and a reload of our docs page, will show the new endpoint. Let's test it like we've done before.

Notice how when you expand the new endpoint, a schema for the data input is provided using an example value ...

```
{
  "File_Name": "string",
  "Client_Name": "string",
  "Client_Email": "string",
  "Client_Phone": "string",
  "Client_Company": "string",
  "Client_ID": "string",
  "Agent_Name": "string",
  "Agent_Email": "string",
  "Agent_Phone": "string"
}
```

Let's use the `Try it out` feature of this new endpoint. I entered ...

```
{
  "File_Name": "First_Client.json",
  "Client_Name": "Thom Ives",
  "Client_Email": "thom.ives@domain.com",
  "Client_Phone": "505.555.1234",
  "Client_Company": "Future of AI",
  "Client_ID": "1234",
  "Agent_Name": "Eye Carealot",
  "Agent_Email": "eye.carealot@domain.net",
  "Agent_Phone": "505.555.5678"
}
```.

After hitting the blue `Execute` button, I see a `Code 200` and a `Response body` of ...

```
{
  "status": "Success"
```

```
}
```

I can also see a `First_Client.json` file in my `files` subdirectory, and it's contents are ...

```
{
    "File_Name": "First_Client.json",
    "Client_Name": "Thom Ives",
    "Client_Email": "thom.ives@domain.com",
    "Client_Phone": "505.555.1234",
    "Client_Company": "Future of AI",
    "Client_ID": "1234",
    "Agent_Name": "Eye Carealot",
    "Agent_Email": "eye.carealot@domain.net",
    "Agent_Phone": "505.555.5678"
}
```

And, if you use the `load_client_object_from_json_file` endpoint, you will get back the same data structure as above.

At this point, if you are a data science / AI geek, you feel like you are having more fun than you should be allowed to have!

I'd like us to add a bit more horsepower before we move onto the next section. Let's imagine that we'd like to store a list of client objects. Let's see how we would do that.

This new endpoint will be much like the previous one, but we will make the following changes:

- declare the incoming data object to be a `List` of `Client_Storage_Object`s;
- create an empty list named clients; and
- use a for loop to:
  - append a dictionary to the clients list for each new client;
  - and assign values from each client object to each client dictionary in the list.

Finally, we store our clients list to our json file name.

In [ ]:
```
@app.post("/store_client_objects_list_to_json_file")
def post_data_object_to_json_file(storage_objects: List[Client_Storage_Objec
    clients = []

    number_of_clients = len(storage_objects)

    for client_num in range(number_of_clients):
        clients.append({})

        clients[client_num]["File_Name"] = storage_objects[client_num].File_
        clients[client_num]["Client_Name"] = storage_objects[client_num].Cli
        clients[client_num]["Client_Email"] = storage_objects[client_num].Cl
        clients[client_num]["Client_Phone"] = storage_objects[client_num].Cl
        clients[client_num]["Client_Company"] = storage_objects[client_num].
```

```
        clients[client_num]["Client_ID"] = storage_objects[client_num].Clien
        clients[client_num]["Agent_Name"] = storage_objects[client_num].Ager
        clients[client_num]["Agent_Email"] = storage_objects[client_num].Age
        clients[client_num]["Agent_Phone"] = storage_objects[client_num].Age

    response = file_io.store_object_to_json_file(clients, storage_objects[0]

    return response
```

Reload your doc page for the API in your browser tab. You will now see the new endpoint. `Try it out` and enter data in the `Request Body`. If we can enter two clients successfully into this new file, we can enter many MANY more, so let's just do 2 for time and testing sake. Make up whatever data you want, OR just use my data below.

```
[
  {
    "File_Name": "Client_List_1.json",
    "Client_Name": "Gabe Ives",
    "Client_Email": "gabe.ives@gingers.org",
    "Client_Phone": "505.555.3451",
    "Client_Company": "Costco",
    "Client_ID": "001",
    "Agent_Name": "Eye Care",
    "Agent_Email": "eye.care@alot.net",
    "Agent_Phone": "505.555.0987"
  },
  {
    "File_Name": "Client_List_1.json",
    "Client_Name": "Anna Ives",
    "Client_Email": "anna.ives@spicy.net",
    "Client_Phone": "505.555.7773",
    "Client_Company": "Wonder Land",
    "Client_ID": "002",
    "Agent_Name": "Yew Care",
    "Agent_Email": "yew.care@thanks.net",
    "Agent_Phone": "505.555.0988"
  }
]
```

After running the new endpoint, I see the new `Client_List_1.json` file in the `files` subdirectory, and the new client data is correct as entered. Also, if I use the `load_client_object_from_json_file` endpoint, I get the data back correctly.

I trust that IF you have understood everything up until this point decently well, you will be able to make MANY MORE APIs to server your needs.

Testing The API From A Python Script

Defining The Python Functions To Call API Endpoints

Let's imagine now that we are serving our API full time from some server. For convenience, we write some Python scripts that make Python requests to the various API endpoints. Let's look at those Python scripts / functions. The following functions are defined in `File_IO_API_Usage.py`.

In [ ]:
```
import requests
import json

file_io_api_server_name_and_port = "http://127.0.0.1:8005"


def get_file_list():
    url = f"{file_io_api_server_name_and_port}/list_files"

    headers = {"Content-Type": "application/json"}
    file_io_response = requests.get(url=url, headers=headers)

    data = json.loads(file_io_response.text)

    return data

def get_client_object_from_json_file(file_name):
    url = f"{file_io_api_server_name_and_port}/load_client_object_from_json_

    headers = {"Content-Type": "application/json"}
    file_io_response = requests.get(url=url, headers=headers)

    data = json.loads(file_io_response.text)

    return data


def store_client_object_to_json_file(client_data_D):
    url = f"{file_io_api_server_name_and_port}/store_client_object_to_json_f
    headers = {"Content-Type": "application/json"}
    jsonized_data = json.dumps(client_data_D, default=str)
    file_io_response = requests.post(url=url, headers=headers, data=jsonized

    return file_io_response

def store_client_objects_list_to_json_file(client_data_D_list):
    url = f"{file_io_api_server_name_and_port}/store_client_objects_list_to_
    headers = {"Content-Type": "application/json"}
    jsonized_data = json.dumps(client_data_D_list, default=str)
    file_io_response = requests.post(url=url, headers=headers, data=jsonized

    return file_io_response
```

Using The Functions That Call API Endpoints

Now that these functions are defined that use calls to the API endpoints, let's use them.

In [ ]:

```
####### Calls To Functions ########
### Call 1
file_list = get_file_list()
type_file_list = type(file_list)
print(file_list)
print(type_file_list)
print()

### Call 2
new_client_object = {
    "File_Name": "Second_Client.json",
    "Client_Name": "Sue Ives",
    "Client_Email": "sue.ives@donkey.org",
    "Client_Phone": "505.555.3452",
    "Client_Company": "Dockeys On The Edge",
    "Client_ID": "007",
    "Agent_Name": "Death Con 5",
    "Agent_Email": "dogs.out@now.org",
    "Agent_Phone": "505.555.0985"
  }

client_storage_response = store_client_object_to_json_file(new_client_object
print(client_storage_response.text)
print()

### Call 3
client_object = get_client_object_from_json_file("Second_Client.json")
type_client_object = type(client_object)
print(client_object)
print(type_client_object)
print()

### Call 4
new_client_objects_list = [
    {
        "File_Name": "Client_List_2.json",
        "Client_Name": "David Ives",
        "Client_Email": "david.ives@mowers.net",
        "Client_Phone": "505.555.3449",
        "Client_Company": "Testers",
        "Client_ID": "003",
        "Agent_Name": "Eye Care",
        "Agent_Email": "eye.care@alot.net",
        "Agent_Phone": "505.555.0982"
    },
    {
        "File_Name": "Client_List_2.json",
        "Client_Name": "Abby Ives",
        "Client_Email": "anna.ives@spicy.net",
        "Client_Phone": "505.555.7773",
        "Client_Company": "Exercise Physiologists",
        "Client_ID": "004",
        "Agent_Name": "Beat You Up",
```

```
        "Agent_Email": "beat.up@you.net",
        "Agent_Phone": "505.555.0983"
    }
]

clients_storage_response = store_client_objects_list_to_json_file(new_client
print(clients_storage_response.text)
print()

### Call 5
client_objects = get_client_object_from_json_file("Client_List_2.json")
type_client_objects = type(client_objects)
print(client_objects)
print(type_client_objects)
print()
```

```
['Client_List_1.json', 'Client_List_2.json', 'First_Client.json', 'first_fil
e.json', 'Second_Client.json']
<class 'list'>

{"status":"Success"}

{'File_Name': 'Second_Client.json', 'Client_Name': 'Sue Ives', 'Client_Emai
l': 'sue.ives@donkey.org', 'Client_Phone': '505.555.3452', 'Client_Company':
'Dockeys On The Edge', 'Client_ID': '007', 'Agent_Name': 'Death Con 5', 'Age
nt_Email': 'dogs.out@now.org', 'Agent_Phone': '505.555.0985'}
<class 'dict'>

{"status":"Success"}

[{'File_Name': 'Client_List_2.json', 'Client_Name': 'David Ives', 'Client_Em
ail': 'david.ives@mowers.net', 'Client_Phone': '505.555.3449', 'Client_Compa
ny': 'Testers', 'Client_ID': '003', 'Agent_Name': 'Eye Care', 'Agent_Email':
'eye.care@alot.net', 'Agent_Phone': '505.555.0982'}, {'File_Name': 'Client_L
ist_2.json', 'Client_Name': 'Abby Ives', 'Client_Email': 'anna.ives@spicy.ne
t', 'Client_Phone': '505.555.7773', 'Client_Company': 'Exercise Physiologist
s', 'Client_ID': '004', 'Agent_Name': 'Beat You Up', 'Agent_Email': 'beat.up
@you.net', 'Agent_Phone': '505.555.0983'}]
<class 'list'>
```

Outstanding! Also check that the new file names showed up in your `files` subdirectory. We can now import and use these functions wherever we need. We could also use JavaScript fetch calls to bring the data into JavaScript or something like React.js.

Serving The API From A Docker Container

Let's remember WHY we want to use Docker. We can have our API served from a Docker Container so that it is ALWAYS isolated from the host system, and will always reliably serve, and will always reliably restart if the server restarts. This really is enough motivation, but we could go on. By serving our API from a container, we can also take

advantage of scaling it with a system like Kubernetes for those times our API's usage volume starts to increase.

Let's first review our Dockerfile.

The Dockerfile

```
# Use the image we originally intended to use
FROM python:3.8.0-slim-buster

# Change our working directory to /code/app for simplicity
WORKDIR /app

# Simplify the pip install (we do NOT want upgrades)
RUN pip install fastapi==0.92.0
RUN pip install uvicorn==0.20.0

# Copy code in app directory to app directory under code in image
COPY ./app/File_IO.py /app/File_IO.py
COPY ./app/File_IO_API.py /app/File_IO_API.py

RUN mkdir /app/files

# Start from app directory now. Set host and port to typical
settings
CMD ["uvicorn", "File_IO_API:app", "--host", "0.0.0.0", "--port",
"8000"]
```

If the notes above each docker command in the dockerfile are not helpful enough, it is time to learn Docker. It is worth learning. Please note that the repo that this notebook is part of has a subdirectory named Docker_Notes. These explain my most commonly used Docker Commands. I hope it helps, and I wish we had more time to go into Docker details here, but, alas, we can only go over it at the level shown.

A Local Reduced Set Of The Docker Commands

The file named `Docker_Commands.txt` contains a larger set of my most commonly used commands. Some of those commands help me to push my Docker image to DockerHub, so that I can pull that image down to another server and run it conveniently. That file is worth further study and research. NOTE that I always name my images `container_name_i` where the `_i` at the end tells me that this is the image's file name. The container name is thus `container_name`. This way, I can copy and paste my `Docker_Commands.txt` file to other project directories and then replace all instances of `container_name` with the new container name for that new project. I welcome you to plagiarize and use this `Docker_Commands.txt` file for your own projects and modify it to better suit your needs.

The set of commands below are a reduced set of commands for running a Docker container locally. I can now copy the commands below, OR the commands from the `Docker_Commands.txt` file, and paste commands them into the command terminal. I'll start with the second command below.

```
### To start clean on your local machine
sudo docker image rm file_io_api_i:latest

### Build the image from your latest Dockerfile
sudo docker image build -t file_io_api_i:latest .

### Run the image with the necessary flags
sudo docker container run --name file_io_api --restart always -p
8006:8000 -d file_io_api_i

### Stop the container and remove it when you need to rerun a new
image
sudo docker container stop file_io_api
sudo docker container rm file_io_api
```

NOTE that as I was writing the previous sections to this one, I was on a Windows machine. I decided to commit and push the repo for this work and to clone it to the Linux Virtual Machine that I am now on. However, you could do the same steps using Docker For Windows.

I'll now open a command terminal and run that second command from above.

```
thom@LM-20:~/Repos/Docker_Served_File_IO_API$ sudo docker image
build -t file_io_api_i:latest .
DEPRECATED: The legacy builder is deprecated and will be removed in
a future release.
            Install the buildx component to build images with
BuildKit:
            https://docs.docker.com/go/buildx/

Sending build context to Docker daemon  10.86MB
Step 1/8 : FROM python:3.8.0-slim-buster
 ---> 577b86e4ee11
Step 2/8 : WORKDIR /app
 ---> Using cache
 ---> ac15336a4d40
Step 3/8 : RUN pip install fastapi==0.92.0
 ---> Using cache
 ---> 7d6eac04d4bd
Step 4/8 : RUN pip install uvicorn==0.20.0
 ---> Using cache
 ---> 983062d79d35
Step 5/8 : COPY ./app/File_IO.py /app/File_IO.py
 ---> 44eb135a76d0
Step 6/8 : COPY ./app/File_IO_API.py /app/File_IO_API.py
```

```
 ---> 3b530cdcaec7
Step 7/8 : RUN mkdir /app/files
 ---> Running in 58c87a449bd8
Removing intermediate container 58c87a449bd8
 ---> 5c4f861cc64e
Step 8/8 : CMD ["uvicorn", "File_IO_API:app", "--host", "0.0.0.0",
"--port", "8000"]
 ---> Running in 7acab383ef53
Removing intermediate container 7acab383ef53
 ---> 2609ce25b824
Successfully built 2609ce25b824
Successfully tagged file_io_api_i:latest
```

Looks like we will ALL need to soon learn how to "Install the buildx component to build images with BuildKit".

Next, we will run the third command from the above group of commands ...

```
sudo docker container run --name file_io_api --restart always -p
8006:8000 -d file_io_api_i
```

and note that the `-p 8006:8000` means the port 8000 in the container will be port 8006 outside the container.

Running the above command yields ...

```
thom@LM-20:~/Repos/Docker_Served_File_IO_API$ sudo docker container
run --name file_io_api --restart always -p 8006:8000 -d
file_io_api_i
b1ee757a92c2a28237c4da62a8490bc9992edbfcf47feaf849b89cbee4f98f25
```

NOTE that the long alphanumeric value is the container ID. Let's run

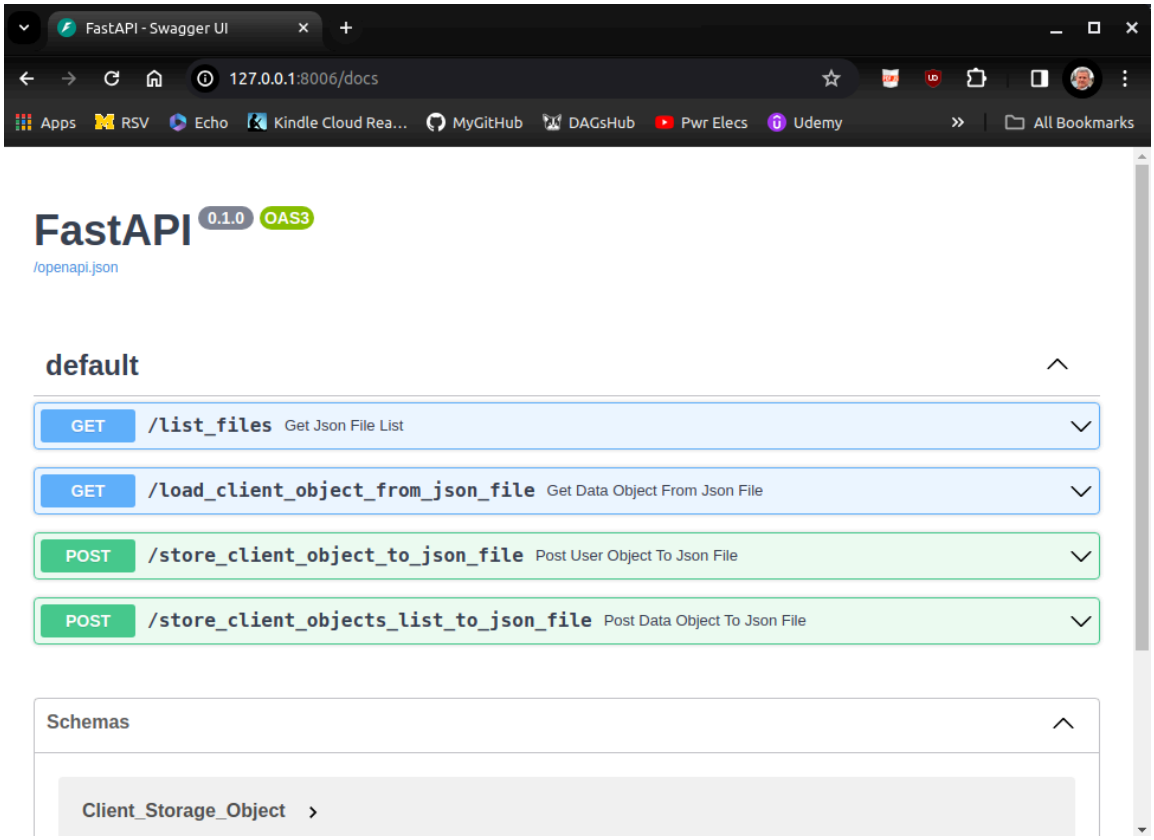
```
sudo docker container ls
```

to see a list of the running containers.

```
thom@LM-20:~/Repos/Docker_Served_File_IO_API$ sudo docker container
ls
CONTAINER ID   IMAGE          COMMAND                  CREATED
STATUS         PORTS                                      NAMES
b1ee757a92c2   file_io_api_i   "uvicorn File_IO_API…"   2 minutes
ago   Up 2 minutes   0.0.0.0:8006->8000/tcp, :::8006->8000/tcp
file_io_api
```

NOTE that the container ID has been abbreviated now. When using docker commands from the command line, you can use the container name, OR even only the first 3 alphanumeric values of the ID to specify the container.

Let's see if our API is being served by Docker. We open a browser, and point it's new tab to `http://127.0.0.1:8006/docs`. When I do this on my Linux Virtual Machine (which is running Linux Mint), I see the following page.



A Test Using File_IO_API_Usage.py

Let's rerun `File_IO_API_Usage.py`, but let's make one change first. Near the top of the file, we will do this ...

```
# file_io_api_server_name_and_port = "http://127.0.0.1:8005"
file_io_api_server_name_and_port = "http://127.0.0.1:8006"
```

When I run this file, I see the following output.

```
[]
<class 'list'>

{"status":"Success"}

{'File_Name': 'Second_Client.json', 'Client_Name': 'Sue Ives',
'Client_Email': 'sue.ives@donkey.org', 'Client_Phone':
'505.555.3452', 'Client_Company': 'Dockeys On The Edge',
'Client_ID': '007', 'Agent_Name': 'Death Con 5', 'Agent_Email':
'dogs.out@now.org', 'Agent_Phone': '505.555.0985'}
<class 'dict'>
```

```
{"status":"Success"}

[{'File_Name': 'Client_List_2.json', 'Client_Name': 'David Ives',
'Client_Email': 'david.ives@mowers.net', 'Client_Phone':
'505.555.3449', 'Client_Company': 'Testers', 'Client_ID': '003',
'Agent_Name': 'Eye Care', 'Agent_Email': 'eye.care@alot.net',
'Agent_Phone': '505.555.0982'}, {'File_Name': 'Client_List_2.json',
'Client_Name': 'Abby Ives', 'Client_Email': 'anna.ives@spicy.net',
'Client_Phone': '505.555.7773', 'Client_Company': 'Exercise
Physiologists', 'Client_ID': '004', 'Agent_Name': 'Beat You Up',
'Agent_Email': 'beat.up@you.net', 'Agent_Phone': '505.555.0983'}]
<class 'list'>
```

Cool! But why is the file list empty? Remember that THIS is the file list inside the container! It started out with NO files.

If we ask for a list of the files again, it will yield ...

```
['Client_List_2.json', 'Second_Client.json']
<class 'list'>
```

Summary

1. We developed a Python module to do File IO operations for us.
2. We then tested the functions in that module.
3. We then imported that module to our FastAPI Python script file, and created an API with endpoints using FastAPI, and we used the docs page to test as we developed each new endpoint.
4. Then we wrote Python scripts to send HTTP requests to our new API and its endpoints.
5. Finally, we served our API from a Docker container.

This API will run steadily whenever the server is running or restarts.

NOTE:

1. We did NOT make endpoints for the text file functions.
2. We also did not yet make endpoints for removing all of certain types of files, NOR all files.

I will leave those tasks and others to you for your own learning. You now have the know how to battle through the code to get that working.

I sincerely hope what I shared in this document and through this repo helps you grow your skills mightily for your career!

Until next time, Thom