Python FastAPI REST APIs - WOW - It's FAST!

Figure 1: FastAPI Logo

Recently, I did a LinkedIn post on creating a Python Flask REST API. Patrick Pichler commented on that post about how much he liked FastAPI. Since I was exploring for a preferred way to create Python based REST APIs, I decided I had to try it. I am now convinced that I will go with FastAPI. I still have a lot to learn, but I wanted to share my first API with FastAPI. I think that most of you can leverage from this simple API and take it much further. I also wanted to share how I am going about learning FastAPI.

For learning FastAPI, FastAPI has great tutorials at `fastapi.tiangolo.com`. I hope that you will check them out. I went through the main one above. I am also still going through `fastapi.tiangolo.com/tutorial/`. If you are accustomed to Flask like me, this may take some getting accustomed to. However, once I was a few steps in, I was sold by their automatically created documentation pages. These documentation pages can also be used to test your APIs. That's right. You don't need to immediately write your own HTTP method call scripts. Nor do you need to use something like Postman. It's all built right in.

Prerequisites

I recommend that we first create or activate our preferred Python virtual environment. Once you are in there, you will want to do the following `pip` installations. The first one, I trust, is obvious. The second one is the package that will serve your API. Note that it's a different package than the one used by Flask. The third and last is simply the quickest way to obtain all the FastAPI goodies available.

```
!pip install fastapi
```

```
!pip install "uvicorn[standard]"
```

```
!pip install "fastapi[all]"
```

The REST API Code

I'll cover and comment on the entire code base first. I can't say what's best, but I recommend going through the tutorials at FastAPI first at least a little bit. However, if a good portion of this makes sense to you, it would be OK to start with this code. Regardless of when you look at FastAPI's learning materials, **PLEASE** do look at them. I do not regard this overview of FastAPI as a tutorial worthy of use in isolation. PLEASE study other sources too. However, even IF I had the best FastAPI tutorial around, I'd still encourage you to look at other material to gain great understanding of it.

NOTE that I am still learning FastAPI, and I still have some questions about what I've done below. However, I think this code structure is good and safe. It's only toy intro level, but it's a good starting point.

I do at least recommend that you become familiar with decorators through some web searching if they are foreign to you. An example of a decorator below is `@app.get("/items")`. I like to think of decorators as a Pythonic elegant way to have Python wrap the functions below the decorator in a standard well defined way. They will seem magic until you study how to create your own. To me, they are one of those elegant Pythonic things like context managers. I very much appreciate them.

Imports

Note that you do NOT need line 1 if using Python 3.10 or greater. **NOTE** that I am being DRY with my own Python - I am importing a library that has some reliable JSON IO tools that I've come to rely on. Feel free to go grab them, OR you can replace my JSON IO methods with your own.

```
from typing import Union

from fastapi import FastAPI
from pydantic import BaseModel

import __General_Tools_Module__ as gtm
# The above module can be found at dagshub.com/ThomIves/Python_Tools
```

Necessary Declarations

IF you've done some Python based API work in the past, the first statement should be expected. It is essential if this is completely new to you.

```
app = FastAPI()
```

I've noticed that FastAPI practitioners love using pydantic models. I can't say much about them, but I've found that in order to learn to appreciate something, you've got to use it for a while. Thus, I am trying them out in here, and they seem pretty helpful so far. I might abandon them for my own methods down the road, but I don't even know enough to know that I'd want to do that or not yet.

```
class Client(BaseModel):
    client_id: str
    name: str
    company: str
```

When building APIs, we normally use the following specific HTTP methods:

- POST: to add complete new records to our data.
- GET: to read our existing data records.
- PUT: to update existing data.
- DELETE: to delete a specific record from our data.

In OpenAPI, each of the HTTP methods are also called an "operation".

First, we create our GET method for our API. **NOTE** that I am calling a function, from my `__General_Tools_Module__` aliased as `gtm`, called `load_object_from_json_file`. Again, if you don't want to go find and use mine, what you need at this point is a function that will open a dictionary stored in a json file and that will return an empty dictionary IF that json file has not yet been created. This function only reads the contents of a `json` file and then uses the `json` module to convert the contents to a Python object.

We are pretending, as we did in the last post, that we are a carrier development counseling firm, and this is our first "tracer bullet" type application for maintaining data on our clients.

```
@app.get("/items")
def get_clients_object():
    data_D = gtm.load_object_from_json_file("my_clients.json")
    return {"clients dictionary": data_D}
```

The POST method is expecting a client input of type Client defined previously using pydantic. We'll see shortly how to enter this and make calls for it from both the automatically generated documentation for this API and a python script. For now, notice that: 1. we pass in a client object, 1. we open our existing json data file and assign the contents, using json, to an object variable name, 1. we add our new client to that object if we have assigned a proper non-existing client_id and store the object back to file, 1. otherwise, if the client_id already exists, we inform that data entry person that the client_id already exists.

```
@app.post("/items/client")
def add_new_client(client: Client):
    data_D = gtm.load_object_from_json_file("my_clients.json")
    if client.client_id not in data_D:
        data_D[client.client_id] = {
            "name": client.name,
            "companies": [client.company]
        }
        gtm.store_object_to_json_file(data_D, "my_clients.json")

        return {"message": "added new client",
            client.client_id: {
                "client_name": client.name,
                "client_company": [client.company]
            }}

    else:
        return {"message": f"Client ID {client.client_id} already exists"}
```

Since we are advising the carriers of our clients, we know that they will eventually seek to work at other companies. Thus, we need a PUT method to update the company work history of our clients. YES, we do need more data, but this is just a tracer bullet application for now. We will make additional improvements ASAP. For now, when needed, we pass in a client_id and a company name to add to our clients' employment history. If we try to use a non-existent client_id, we are informed of such.

```
@app.put("/users/{client_id}/items/{company}")
def update_client_records(client_id: str, company: str):
    data_D = gtm.load_object_from_json_file("my_clients.json")

    if client_id in data_D:
        data_D[client_id]["companies"] += [company]
        gtm.store_object_to_json_file(data_D, "my_clients.json")

        return {"message":
            f"Added {company} to company records for client {client_id}"}
    else:

        return {"message": f"Client ID {client_id} does not exist"}
```

If, regretfully, we lose a client, we delete them from our records. I think it would be great to have a record of deleted clients in case they come back to us. That will go on our todo list for this API's ongoing development planning.

```
@app.delete("/users/{client_id}")
def delete_a_client(client_id: str):
```

```
    data_D = gtm.load_object_from_json_file("my_clients.json")
    if client_id in data_D:
        name = data_D[client_id]["name"]
        del data_D[client_id]

        gtm.store_object_to_json_file(data_D, "my_clients.json")

        return {"message":
            f"Removed {name}, client {client_id}, from client records"}
    else:
        return {"message": f"Client ID {client_id} does not exist"}
```

Implementation

Well, that looks all wonderful in theory, but does this work? First, we have to start this script and make sure it launches on our local server using uvicorn. Then, we need to test each method. Let's test each method two different ways. Using the automatically generated documentation for this API that also allows us to execute each of our methods and using requests from Python scripts.

What does it look like when we start our API from the command line. AND how do we do that? Well, let me show you! We start our API from a command line terminal using the following line (**note** that whatever the python script name of your API is, that's what you'd put in for My_FastAPI minus the .py part).

```
uvicorn My_FastAPI:app --reload
```

If successful, your uvicorn server launch will look like the following:

```
(py38std) thom@thom-PT5610:~/DagsHub_Repos/Flask_API_Work/My_FastAPI$ uvicorn My_FastAPI:app --reload
INFO:     Will watch for changes in these directories: ['/home/thom/DagsHub_Repos/Flask_API_Work/My_Fas
INFO:     Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
INFO:     Started reloader process [754596] using watchgod
INFO:     Started server process [754598]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
```

If you are running, great! If not, go back through this document carefully, OR step through the FastAPI tutorials until you can find your issue. Many people forget to make sure their terminal is looking at the same directory that their API script is in. You may also have a typo in your

```
uvicorn <api-script-name-WITHOUT-.py>:app --reload
```

I hope it runs for you, because once it does, the rest of this will likely go pretty smoothly for you.

Next, let's open the amazingly good automatically generated document page for our API using the below line.

```
http://127.0.0.1:8000/docs
```

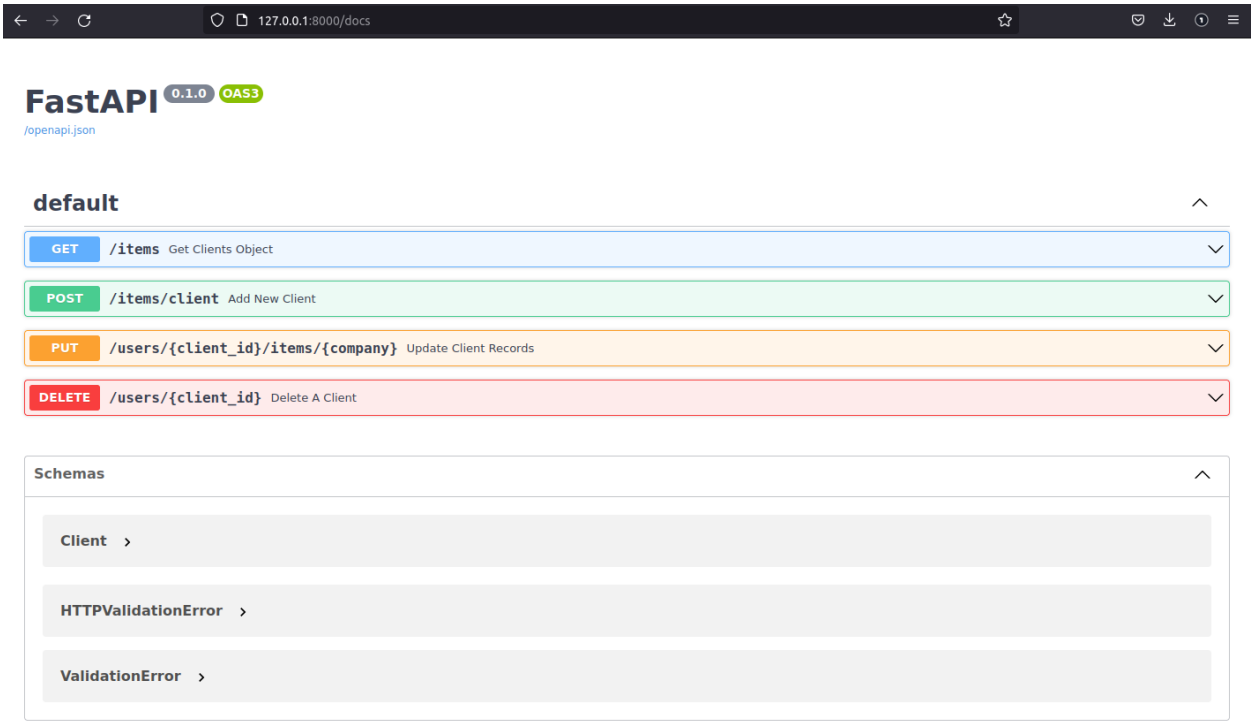

Figure 2: The Automatically Generated API Document Page

Let's NOT start out with the GET method, because that will only give us an empty dictionary, since we have not yet entered any data. Let's first enter some data by using the POST method. Expand the POST block and click on "Try it out" at the upper right of that expanded block. "Try it out" will switch to "Cancel" in case you decide you do NOT want to try it out right now. You will see a "Request body". Edit the dictionary in that "Request body" as shown in the next image.

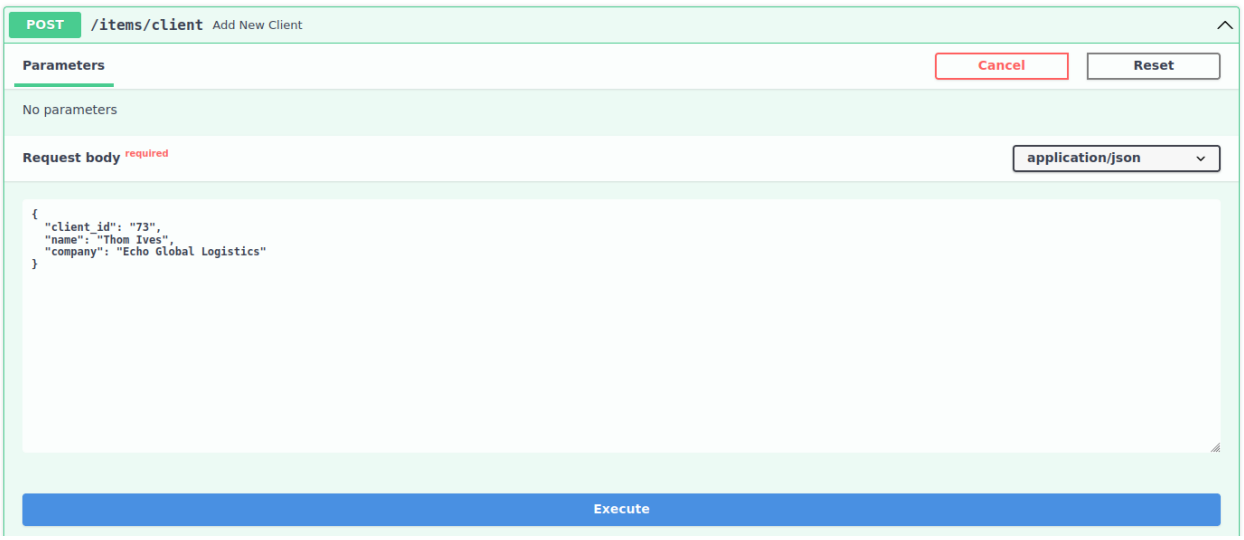

Figure 3: Edited Request Body Dictionary For First Client Entry

Now, we click on execute. We want to then check on two things. First, in this expanded POST block, scroll down a bit, and you will see the response that you formulated in your return statement if all went well. I have shown mine below.

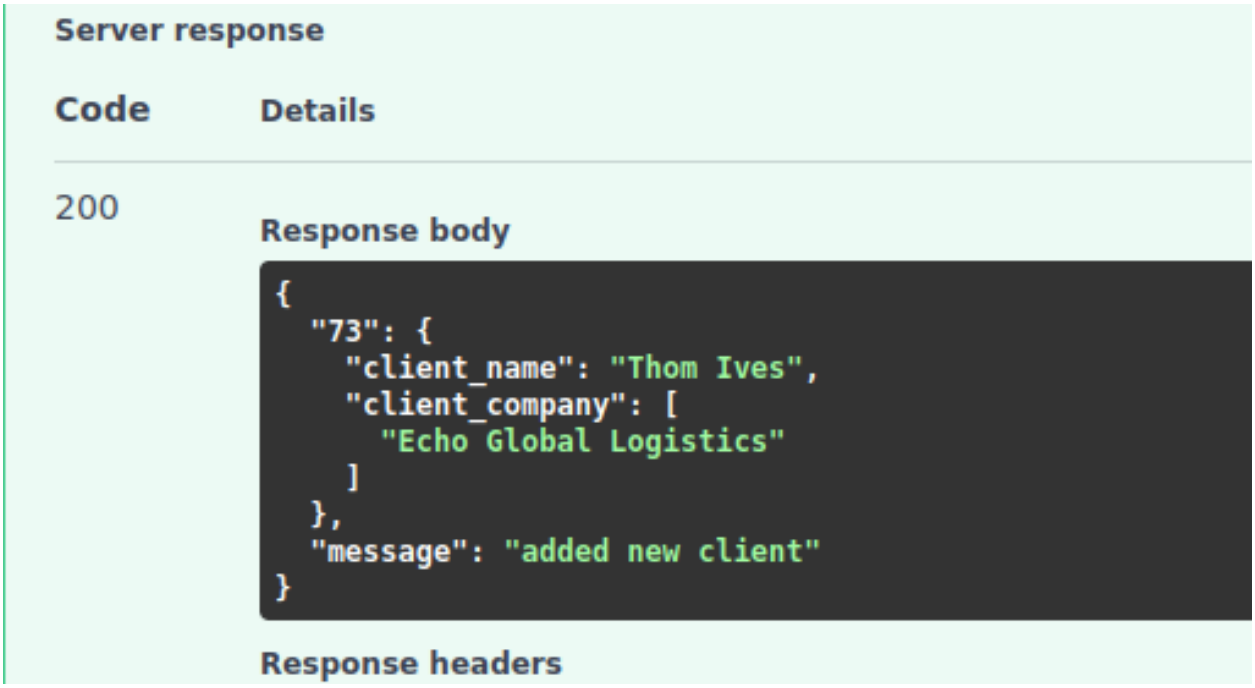

Figure 4: Correct Data Entry Response From The API's Server

Check what the response body reads if you try to execute the POST again from the doc page. You should see `{ "message": "Client ID 73 already exists" }`.

Now, let's also view the file named `my_clients.json` that was created.

```
{
    "73": {
        "name": "Thom Ives",
        "companies": [
            "Echo Global Logistics"
        ]
    }
}
```

NICE! Let's add more data now. BUT this time, let's do it with a Python script like the one shown below.

```
import requests
import json
import pprint

pp = pprint.PrettyPrinter(indent=2)

URL = "http://127.0.0.1:8000/items/"

response = requests.get(URL)
```

```
stuff = response.text
user_D = json.loads(stuff)

pp.pprint(user_D)
```

And the terminal output when we run that is the following.

```
(py38std) thom@thom-PT5610:~/DagsHub_Repos/Flask_API_Work/My_FastAPI$ python get.py
{ 'clients dictionary': { '73': { 'companies': ['Echo Global Logistics'],
                                  'name': 'Thom Ives'}}}
(py38std) thom@thom-PT5610:~/DagsHub_Repos/Flask_API_Work/My_FastAPI$
```

Nice! We did a POST with our automatically generated API documentation page. Then we did a GET with our python script. Let's do a POST with a Python script and then another GET with our API documentation page. Next is my currently **NOT working** post script.

```
import requests
import json
import pprint

pp = pprint.PrettyPrinter(indent=2)

client = {"client_id":"57","name":"JoJo Dude","company":"Franklin Building Supply"}
URL = f"http://127.0.0.1:8000/items/{client}"
print(URL)

response = requests.post(URL)

stuff = response.text
user_D = json.loads(stuff)

pp.pprint(user_D)
```

The output on the terminal for this **NOT YET WORKING** script is ...

```
(py38std) thom@thom-PT5610:~/DagsHub_Repos/Flask_API_Work/My_FastAPI$ python post.py
http://127.0.0.1:8000/items/{'client_id': '57', 'name': 'JoJo Dude', 'company': 'Franklin Building Suppl
{'detail': 'Not Found'}
(py38std) thom@thom-PT5610:~/DagsHub_Repos/Flask_API_Work/My_FastAPI$
```

Before I build the repo for this work, I will fix this script and update this document. Sorry! For now, let's add a new record using the POST block on the API's doc page again. After entering the data, remember to click on the Execute button.

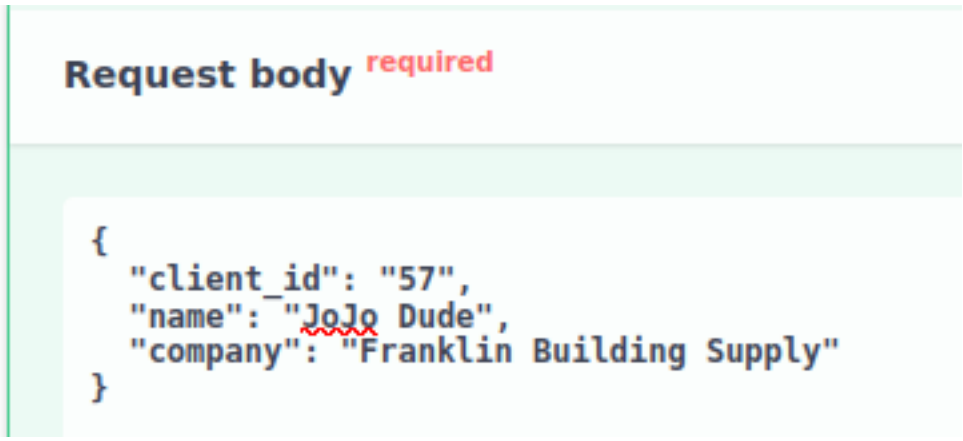

Figure 5: POSTING A New Client Using API Doc POST Block

Now, let's use the GET method from the API Doc page this time too.

Figure 6: GET From API Doc Page AFTER Entry Of Second Client

JoJo Dude, client 57, just moved over to Costco. Let's add that to his work history. We will try a Python script again.

```
import requests
import json
import pprint

pp = pprint.PrettyPrinter(indent=2)

client_id = 57
company = "Costco"
URL = f"http://127.0.0.1:8000/users/{client_id}/items/{company}"
print(URL)

response = requests.put(URL)

stuff = response.text
user_D = json.loads(stuff)

pp.pprint(user_D)
```

Phew! You can see from the output below that it worked. Nice. Just gotta figure out what's up with the client entry on the post script.

```
(py38std) thom@thom-PT5610:~/DagsHub_Repos/Flask_API_Work/My_FastAPI$ python put.py
http://127.0.0.1:8000/users/57/items/Costco
{'message': 'Added Costco to company records for client 57'}
(py38std) thom@thom-PT5610:~/DagsHub_Repos/Flask_API_Work/My_FastAPI$
```

And the below shows the contents of our updated `my_clients.json` file.

```
{
    "73": {
        "name": "Thom Ives",
        "companies": [
            "Echo Global Logistics"
        ]
    },
    "57": {
        "name": "JoJo Dude",
        "companies": [
            "Franklin Building Supply",
            "Costco"
        ]
    }
}
```

Try to ALSO use the PUT method on the API's doc page. It's be a good experience for you.

Alas, JoJo is going to leave our firm for another one. We hope he comes back. We should really move items deleted from our `my_clients.json` file to some lost clients file. It's on our development task list. For now, let's delete him from this file. After all, this is just fake data for development purposes. We'll try our own script.

```
import requests
import json
import pprint
```

```
pp = pprint.PrettyPrinter(indent=2)

client_id = 57
URL = f"http://127.0.0.1:8000/users/{client_id}"
print(URL)

response = requests.delete(URL)

stuff = response.text
user_D = json.loads(stuff)

pp.pprint(user_D)
```

When we run that, we get the following terminal output. NICE! AND, you will find that the `my_clients.json` file updated correctly.

```
(py38std) thom@thom-PT5610:~/DagsHub_Repos/Flask_API_Work/My_FastAPI$ python delete.py
http://127.0.0.1:8000/users/57
{'message': 'Removed JoJo Dude, client 57, from client records'}
(py38std) thom@thom-PT5610:~/DagsHub_Repos/Flask_API_Work/My_FastAPI$
```

In fact, let's view it using our GET Python script again. You can of course ALSO use the API's GET block and execute the GET from there.

```
(py38std) thom@thom-PT5610:~/DagsHub_Repos/Flask_API_Work/My_FastAPI$ python get.py
{ 'clients dictionary': { '73': { 'companies': ['Echo Global Logistics'],
                                  'name': 'Thom Ives'}}}
(py38std) thom@thom-PT5610:~/DagsHub_Repos/Flask_API_Work/My_FastAPI$
```

Great! I am this firms only client, but I plan to stay with them. I like the way that they code APIs using FastAPI! Just gotta figure out that POST Python Script.

Summary

We used Python FastAPI to create a REST API similar to the one that we created using Python Flask. We discovered some great new power and methods using FastAPI. We also learned that Thom needs to learn how to pass in those objects for post OR use a different method for passing in values. I'll try to fix that ASAP and update accordingly.

Until next time.