This is how easy it is to create a REST API

Leon Wee Follow
Mar 8, 2018 · 6 min read

Learn how to create semantic REST API real quick using Python Flask

Photo by Simon Abrams on Unsplash

I was learning PHP back then, where I was taught to write codes that combine both front-end presentation and back-end logic together in the same code base. It was simple, sweet, quick to implement and no issues until I stumbled across a fairly complicated project, where the front-end is so complex that I have to separate it with the back-end in order to keep my head on it. This is where I learnt about the advantages of separation of concerns, client — server architecture, HTTP methods as well as RESTful architecture.

I started to research on frameworks and tools that can help me in achieving the separation of front-end and back-end development. I went with **Flask** as I knew a little of Python already and begin writing a REST API with it, and immediately fall in love with it.

Flask Defined

> *Flask is a microframework based on Werkzeug, Jinja2 and good intentions, and before you ask: It's BSD licensed!*
> *—Flask Official Website*

Microframework refers to a light-weight web application framework in contrast to full-stack frameworks. That means **Flask** gives only what you need essentially to create a back-end server but provides the flexibility to install any extensions to support features like database interfacing, authentication, encryption, CSRF protection and so on.

Installation

To begin with, we will need to install the required library and dependencies, we will be using **Flask-RESTful**, an extension for **Flask** which enables rapid development of REST API with minimal setup. Note that **Flask-RESTful** requires Python version 2.6, 2.7, 3.3, 3.4, 3.5 or 3.6.

```
pip install flask-restful
```

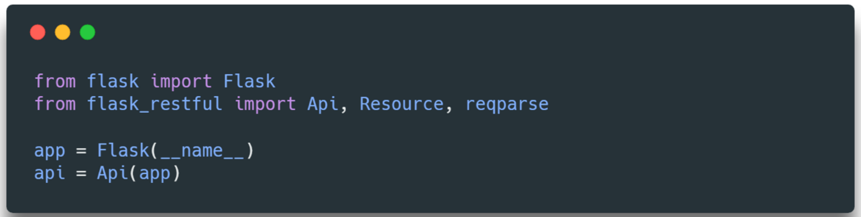
Install one of these API testing tool:

- **Postman** — This tool basically allows you to test your API endpoints, observe the responses. You can go even further to create scripts and do automated testing.
- **Insomnia** — An open source alternative to Postman. It comes with all the basic features you will need for API endpoints testing, and a better design IMO.

Implementation

We will be creating a RESTful API that is used to store users details, which will have **CRUD (Create, Read, Update, Delete)** functions, allowing us to create new user, get details of existing user, update details of existing user and delete existing user.

Let's begin by importing the required module and setting up the **Flask-RESTful** application:

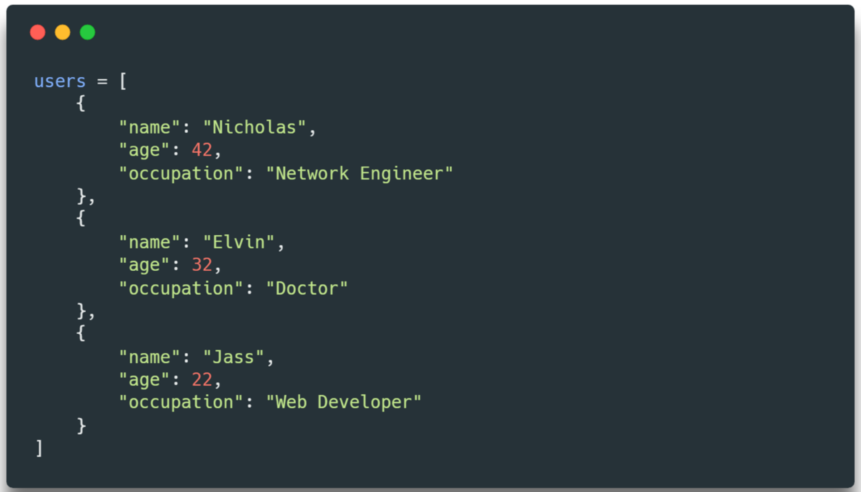


```
from flask import Flask
from flask_restful import Api, Resource, reqparse

app = Flask(__name__)
api = Api(app)
```

> *Note: We are importing "Flask" , "Api" and "Resource" with capital letter initials, it signifies that a class is being imported. reqparse is **Flask-RESTful** request parsing interface which will be used later on. Then we create an app using "Flask" class, "__name__" is a Python special variable which gives*

> *Python file a unique name, in this case, we are telling the app to run in this specific place.*

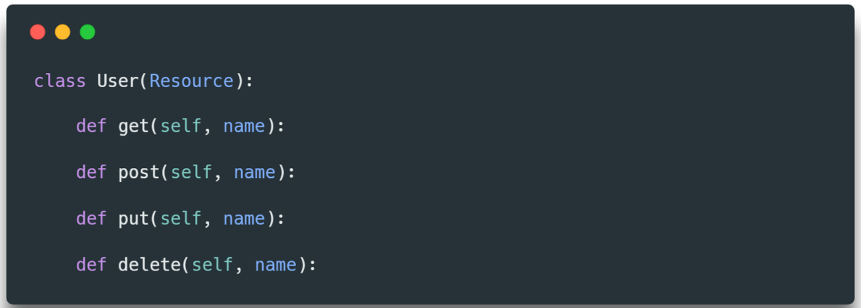
Next, we will create a lists of users using Python data structures which are lists and dictionaries to simulate a data store:



```
users = [
    {
        "name": "Nicholas",
        "age": 42,
        "occupation": "Network Engineer"
    },
    {
        "name": "Elvin",
        "age": 32,
        "occupation": "Doctor"
    },
    {
        "name": "Jass",
        "age": 22,
        "occupation": "Web Developer"
    }
]
```

> *Note: This method is used since this article is focusing in creating API, but in actual condition, the data store is usually a database.*

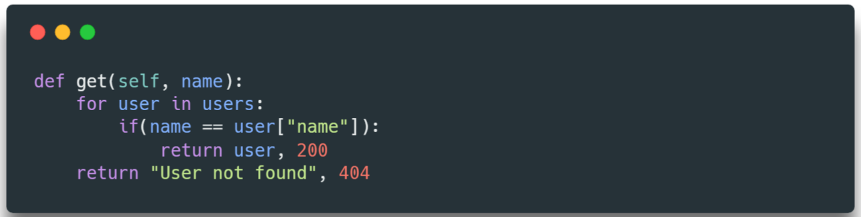
Now we will begin creating our API endpoints by defining a *User* resource class. Four functions which correspond to four HTTP request method will be defined and implemented:



```
class User(Resource):

    def get(self, name):

    def post(self, name):

    def put(self, name):

    def delete(self, name):
```

> *One of the good quality of a REST API is that it follows standard HTTP method to indicate the intended action to be performed.*

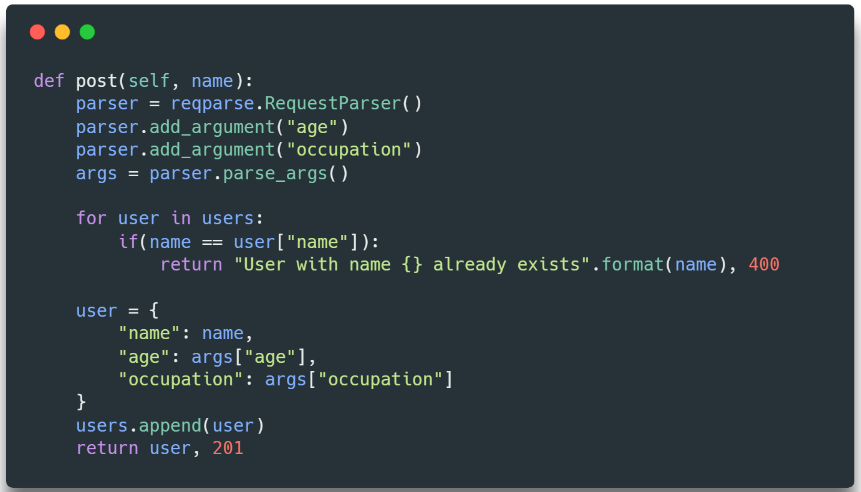
The *get* method is used to retrieve a particular user details by specifying the name:



```
def get(self, name):
    for user in users:
        if(name == user["name"]):
            return user, 200
    return "User not found", 404
```

> *We will traverse through our users list to search for the user, if the name specified matched with one of the user in users list, we will return the user, along with **200 OK**, else return a user not found message with **404 Not Found**. Another characteristic of a well designed REST API is that it uses standard **HTTP response status code** to indicate whether a request is being processed successfully or not.*

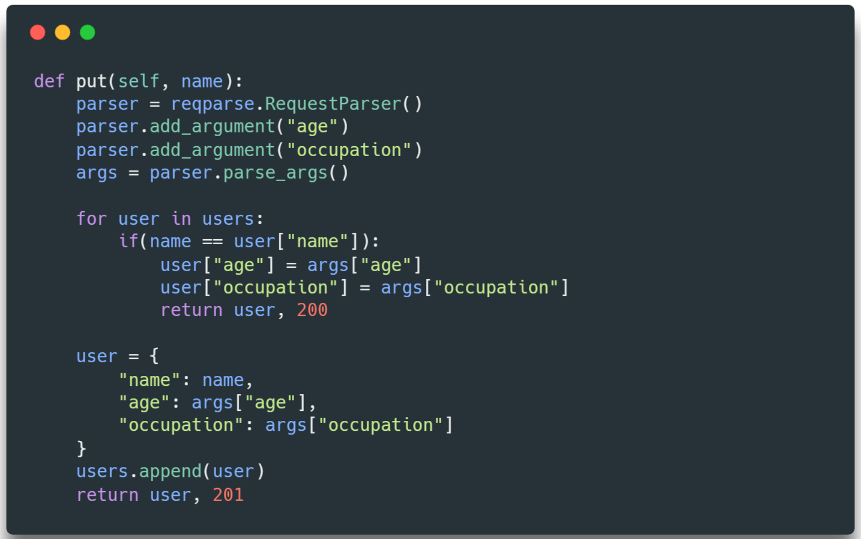
The *post* method is used to create a new user:



```
def post(self, name):
    parser = reqparse.RequestParser()
    parser.add_argument("age")
    parser.add_argument("occupation")
    args = parser.parse_args()

    for user in users:
        if(name == user["name"]):
            return "User with name {} already exists".format(name), 400

    user = {
        "name": name,
        "age": args["age"],
        "occupation": args["occupation"]
    }
    users.append(user)
    return user, 201
```

> *We will create a parser by using reqparse we imported earlier, add the age and occupation arguments to the parser, then store the parsed arguments in a variable, args (the arguments will come from request body in the form of form-data, JSON or XML). If a user with same name already exists, the API will return a message along with **400 Bad Request**, else we will create the user by appending it to users list and return the user along with **201 Created**.*

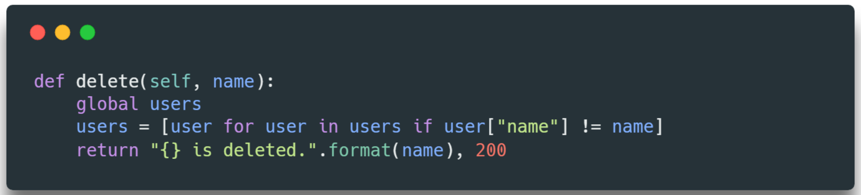
The *put* method is used to update details of user, or create a new one if it is not existed yet.



```
def put(self, name):
    parser = reqparse.RequestParser()
    parser.add_argument("age")
    parser.add_argument("occupation")
    args = parser.parse_args()

    for user in users:
        if(name == user["name"]):
            user["age"] = args["age"]
            user["occupation"] = args["occupation"]
            return user, 200

    user = {
        "name": name,
        "age": args["age"],
        "occupation": args["occupation"]
    }
    users.append(user)
    return user, 201
```

> *If the user already exist, we will update his/her details with the parsed arguments and return the user along with **200 OK**, else we will create and return the user along with **201 Created**.*

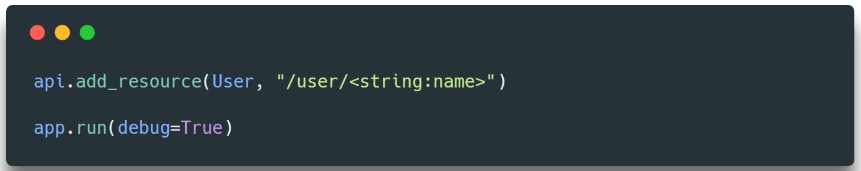
The *delete* method is used to delete user that is no longer relevant:



```python
def delete(self, name):
    global users
    users = [user for user in users if user["name"] != name]
    return "{} is deleted.".format(name), 200
```

> *By specifying users as a variable in global scope, we update the users list using list comprehension to create a list without the name specified (simulating delete), then return a message along with* ***200 OK****.*

Finally, we have done implementing all the methods in our *User* resource, we will add the resource to our API and specify its route, then run our Flask application:



```python
api.add_resource(User, "/user/<string:name>")

app.run(debug=True)
```

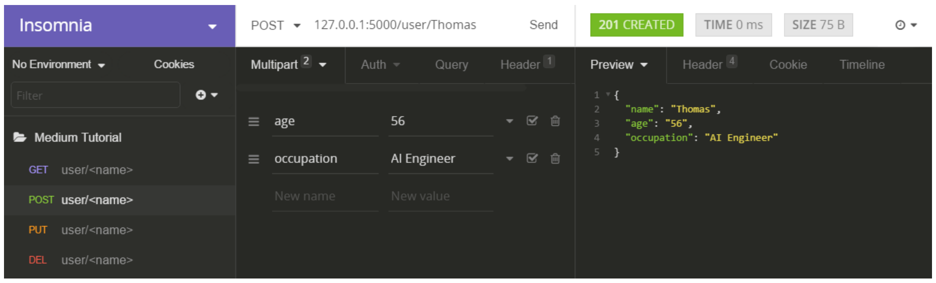
> *Note:*
> *<string:name> indicates that it is a variable part in the route which accepts any name.*
> *To specify Flask to run in debug mode enables it to reload automatically when code is updated and give us helpful warning messages if something went wrong. It is useful in development setting, but should* ***never*** *be used in production setting.*

Save the file as *app.py* and run it:

```
python app.py
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: XXX-XXX-XXX
 * Running on http://127.0.0.1:5000/
```

The complete code is available here.

Now we have our REST API up and running, so it is time to test it using one of the tools I recommended in the beginning.



POST 127.0.0.1:5000/user/Thomas

age	56
occupation	AI Engineer

201 CREATED TIME 0 ms SIZE 75 B

```
{
  "name": "Thomas",
  "age": "56",
  "occupation": "AI Engineer"
}
```

Example of creating a new user named Thomas in Insomnia

Conclusion

Thanks for reading this article, hopefully by now you have an idea on how to write a RESTful API rapidly using Flask, as well as adhere to some existing standards to design a high quality RESTful API.

If you think this tutorial was helpful, please tap the clap button 👏 several times to show your support! Your encouragement will definitely be my motivation to write more article or tutorial like this. 😉

codeburst.io

✉️ Subscribe to *CodeBurst's* once-weekly **Email Blast**, 🐦 Follow *CodeBurst* on **Twitter**, view 🗺️ **The 2018 Web Developer Roadmap**, and 🕸️ **Learn Full Stack Web Development**.