Flask API

Contents

- Overview
- Flask
  - Creating a simple flask app
- Text Responses
  - GET
  - POST
  - PUT
  - DELETE
- JSON
- Tutorial
  - Prerequisites
  - Set-up
  - Deployment
  - Using the API
  - GET
  - POST
  - PUT
  - DELETE
- Exercises

Overview

An **A**pplication **P**rogamming **I**nterface (API) lists a set of operations which can be used by a developer.
Developers use these operations when developing applications, using an API to perform some operations reduces the overall amount of code they must write.
Using code which has already been written can be thought of as using the **D**on't **R**epeat **Y**ourself (DRY) mentality.

Flask

Flask is a micro-framework for deploying web applications.
Flask is written in python and uses python libraries to create a web application.

This module will demonstrate how to create a flask application which has no front-end, ie there will be no web page to display, but the application will still be accessible as an API

Creating a simple flask app

Any flask app which you create must have the following basic structure

```
from flask import Flask
app = Flask(__name__)

@app.route('/number', methods=['GET'])
def number():
    return 3

if __name__ == '__main__':
    app.run(port=5000, debug=True)
```

- The first two lines import the Flask function from the flask module and use it to create an object called `app`.

- The next three lines set up the condition that when someone makes a `get` request to the route `/number` the number `3` will be returned.
- The last two lines allow the app to be run on port `5000`.

Text Responses

When someone makes a request to our API, one of the responses we can send is `text`.

GET

When accessing a web page, the browser automatically sends a `GET` request to that URL. In Flask, `GET` is the default request type when the `method` variable is omitted from the route, however it is best practice to be explicit for better readability.

Since we're returning a `text response` we need to ensure that we have imported the `Response` function and specify that what we are sending is `text`.

For example:

```
from flask import Flask, Response
app = Flask(__name__)

@app.route('/get/text', methods=['GET'])
def get_text():
    return Response("Hello from the flask API", mimetype='text/plain')

if __name__ == '__main__':
    app.run(port=5000, debug=True)
```

Whenever someone makes a `GET` request on the route `/get/text`, the API will return the text response "Hello from the flask API".

POST

For our API to accept post requests, we must ensure that the `POST` method is allowed.

We are still returning a text response, but we must also use the data which is sent to the API.

Altering the API, we get:

```
from flask import Flask, Response, request
app = Flask(__name__)

@app.route('/post/text', methods=['POST'])
def post_text():
    data_sent = request.data.decode('utf-8')
    return Response("This is the data you sent to the API: " + data_sent,
mimetype='text/plain')

if __name__ == '__main__':
    app.run(port=5000, debug=True)
```

Here we also import the `request` function, this allows the API to access the data sent in the post request, the data must be sent in a variable called `data`.

PUT

Like before, we specify the request method, in this case it is `PUT`.

Here we have created an example dictionary as the target for our changes and defined the route with a dynamic URL to specifiy the item we are changing. We do not have to use a dynamic URL, however, it is useful for referencing the item to be changed directly.

```
from flask import Flask, Response, request
app = Flask(__name__)

dictionary = {'key1':'value1','key2':'value2'}

@app.route('/')
def home():
    return dictionary

@app.route('/<string:key>', methods=['PUT'])
def update(key):
    dictionary[key] = request.data.decode('utf-8')
    return Response(key + " changed to: " +request.data.decode('utf-8'))
```

DELETE

Here we allow the route to accept `DELETE` requests in the `methods` variable.

```
from flask import Flask, Response, request
app = Flask(__name__)

dictionary = {'key1':'value1','key2':'value2'}

@app.route('/')
def home():
    return dictionary

@app.route('/<string:key>', methods=['DELETE'])
def deleter(key):
    dictionary.pop(key)
    return Response(key + " deleted from dictionary")
```

JSON

Sometimes we may wish the API to respond with a `dictionary` instead of plain text, to do this we use something called **J**ava**S**cript **O**bject **N**otation (JSON). JSON is commonly used in industry.

For JSON requests, we still must specify whether the API is accepting `GET` and `POST` requests for each root.

The difference is that we must import the function `jsonify` to send data as JSON.
For POST requests, instead of getting the data out using `request.data.decode('utf-8')`, we use `request.get_json()`.
This assumes that the data sent to the API is also in JSON form.

An example using POST:

```
from flask import Flask, request, jsonify
app = Flask(__name__)

@app.route('/post/text', methods=['POST'])
def post_text():
    data_sent = request.get_json()
    data_returned = "The data you send: " + data_sent
    return jsonify({'data':data_returned})

if __name__ == '__main__':
    app.run(port=5000, debug=True)
```

Tutorial

In this tutorial, we will deploy a flask API on an Ubuntu system, which will accept `GET`, `POST`, `PUT` and `DELETE` requests and return a `response`.

This example will be an app containing information about an individual that we can edit through the API.

Prerequisites

You need the following to be installed:

- python3
- python3-venv
- python3-pip

Set-up

To deploy this API we will need 2 files

- `app.py`
- `requirements.txt`

First let's create a new directory called flask-api to work in and the two files we need.

```
mkdir ~/flask-api && cd $_

touch app.py requirements.txt
```

Next, we will create the Flask API.
In `app.py` write the following code.

```
from flask import Flask, Response, request, jsonify
app = Flask(__name__)

information = {'name':'Someguy Somewhere','age':'32', 'occupation':'Somejob'}

@app.route('/')
@app.route('/info', methods=['GET'])
def get_text():
    # The API request will return text containing the information as a JSON
object.
    return jsonify(information)

# Here we will add functionality to add the information dictionary. The new key
is defined in the URL, and the value of the key is in the sent data. We also
want to add a check for pre-existence, so that we do not update existing entires
(we want to save that for PUT requests).
@app.route('/info/add/<string:key>', methods=['POST'])
def post_text(key):
    # adding the new key-value pair
    if key not in information:
        information[key] = request.data.decode('utf-8')
        return Response(key + " added to information with value: " +
request.data.decode('utf-8'), mimetype='text/plain')
    else:
        return Response(key + " already exists.", mimetype='text/plain')

# We will implement update functionality (PUT request) with the same URL as the
route for POST requests, but with a PUT method. Similar to before, we want to
check the dictionary for pre-existence so that we only implement changes if the
key already exists.
@app.route('/info/update/<string:key>', methods=['PUT'])
def put_text(key):
    if key in information:
        information[key] = request.data.decode('utf-8')
        return Response(key + " changed to: " + request.data.decode('utf-8'),
mimetype='text/plain')
    else:
        return Response(key + " not found.", mimetype='text/plain')

# Finally, we add a function so that if the request is DELETE, we delete that
key from the dictionary.
@app.route('/info/delete', methods=['DELETE'])
def delete_text():
    key = request.data.decode('utf-8')
    if key in information:
        information.pop(key)
        return Response(key + " deleted from information. ",
mimetype='text/plain')
    else:
        return Response(key + " not found.", mimetype='text/plain')

# Make app callabale from the command line
if __name__ == '__main__':
    app.run(port=5000, debug=True, host='0.0.0.0')
```

Now we will make the `requirements.txt`, this is a list of `pip` dependencies which are needed to run the flask API.

```
Flask==1.1.1
gunicorn==20.0.4
Werkzeug==1.0.0
```

Deployment

Before we can deploy our API we need to make sure that the requirements are installed in to a virtual environment.

```
cd ~/flask-api
python3 -m venv venv
. ./venv/bin/activate
pip3 install -r requirements.txt
```

Now we can run the API with Python.

```
python3 app.py
```

The API should now be running on your machine. If you want to access your API from outside of this machine then you will need to ensure that you have port 5000 open to the Internet.

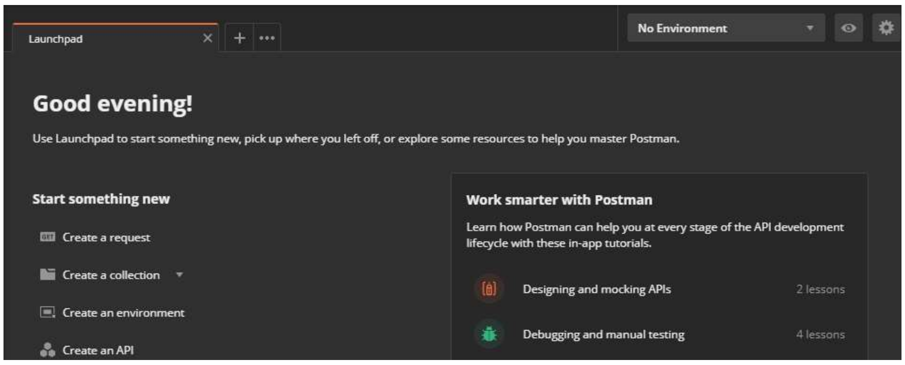
Using the API

Right now, the terminal should be hanging and telling us that our API is running on `0.0.0.0:5000`.
This is good, it means that our API is running, to interact with it we shall use a piece of software called **Postman**.

If you don't have Postman installed, you can download it from here.

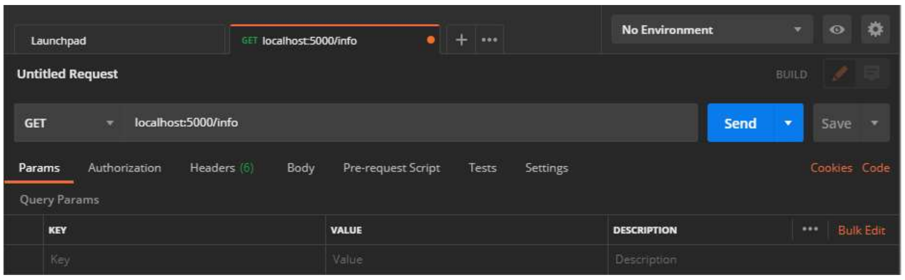
Leave the API running on this terminal and open up Postman.
Remove any notification boxes until you reach the main screen.

To create a new request, we click the + tab at the top of our main screen.


Launchpad
No Environment
Good evening!
Use Launchpad to start something new, pick up where you left off, or explore some resources to help you master Postman.
Start something new
Create a request
Create a collection
Create an environment
Create an API
Work smarter with Postman
Learn how Postman can help you at every stage of the API development lifecycle with these in-app tutorials.
Designing and mocking APIs
2 lessons
Debugging and manual testing
4 lessons

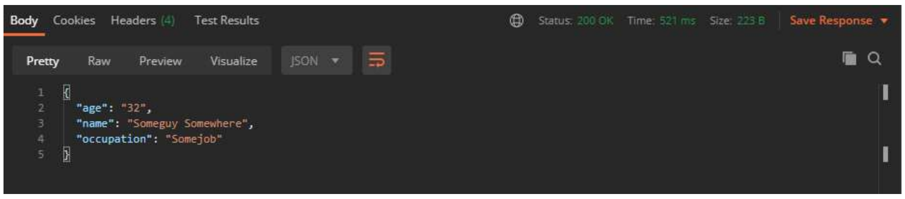
GET

Ensure the type of request is set to `GET`, then in the box which says `Enter request URL`, enter `http://[machine-ip]:5000/info`. If running the app locally, replace `[machine-ip]` with `localhost`. If running on a virtual machine then replace `[machine-ip]` with the VM's external IP.



Click send

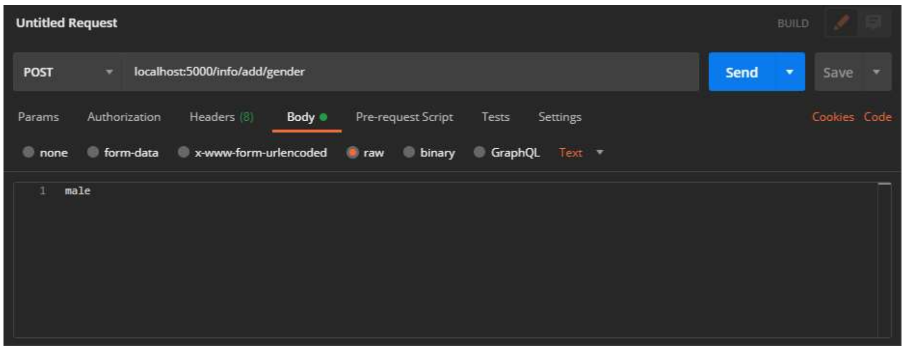
You should see the dictionary returned as a JSON



```
{
  "age": "32",
  "name": "Someguy Somewhere",
  "occupation": "Somejob"
}
```

POST

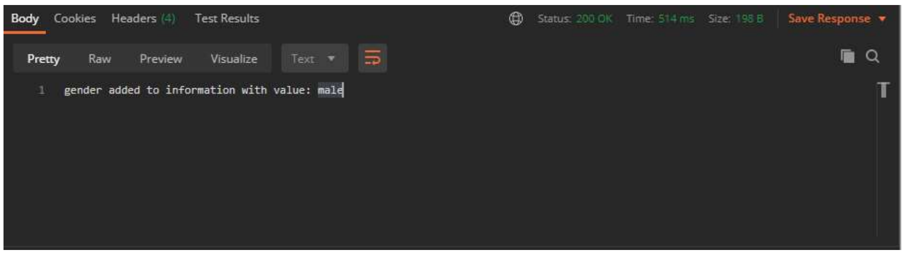
Select POST from the drop down menu and enter the url http://[machine-ip]:5000/info/add/gender. Next, navigate to the body tab and select raw and add male

!


```
localhost:5000/info/add/gender
```

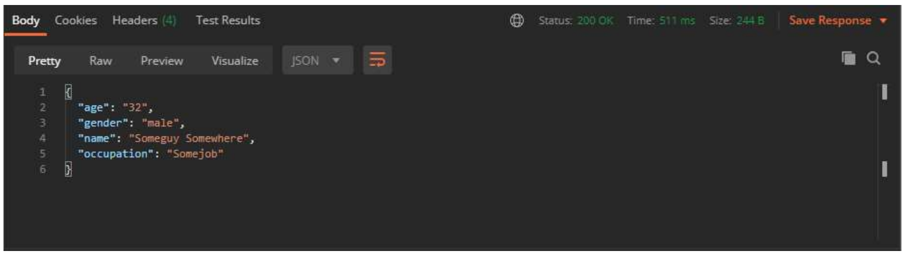
```
male
```

This should give us the response message for our POST request



```
gender added to information with value: male
```

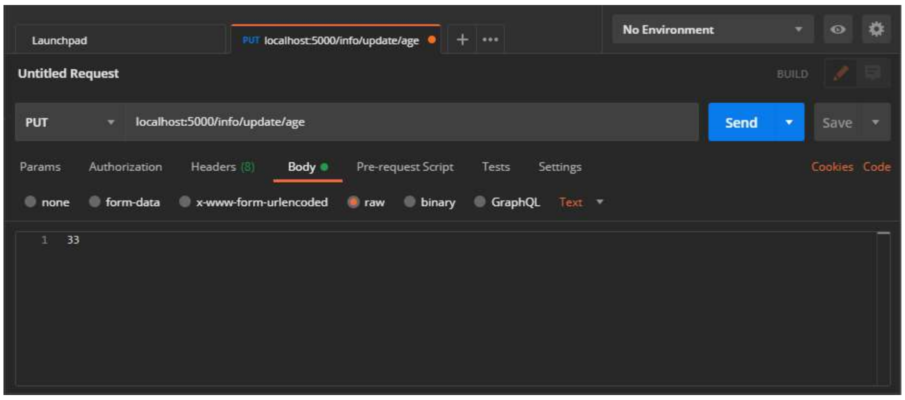
Run a GET request to the /info url to check the addition.



```
{
  "age": "32",
  "gender": "male",
  "name": "Someguy Somewhere",
  "occupation": "Somejob"
}
```

PUT

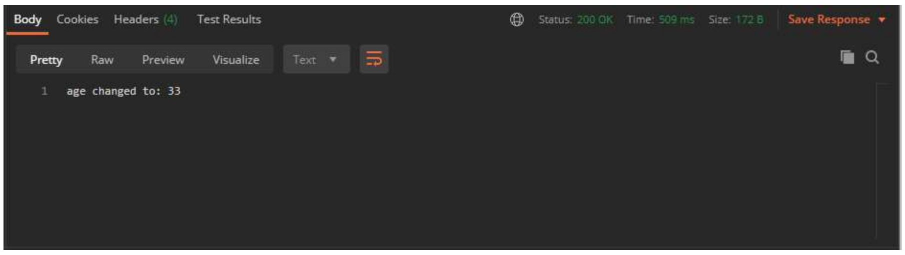
Select PUT and enter the URL http://[machine-ip]:5000/info/update/age. This time add 33 to the body



```
localhost:5000/info/update/age
```

```
33
```

Response

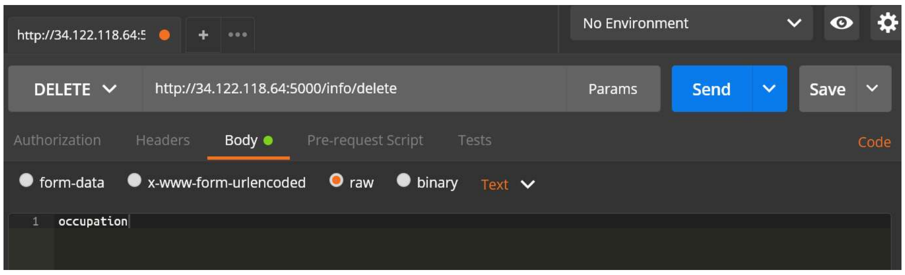


```
age changed to: 33
```

Run a GET request to the /info URL to check that the age has changed.

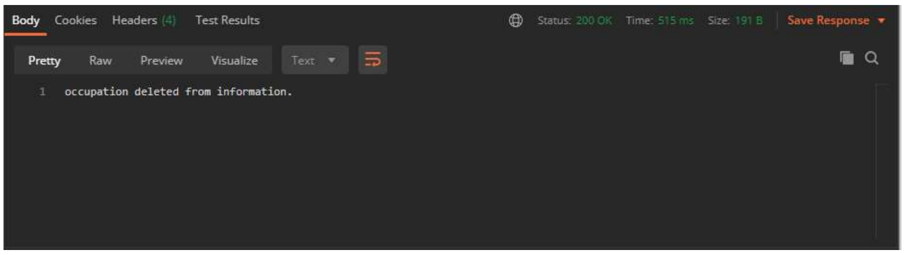
DELETE

Select `DELETE` and enter the URL `http://[machine-ip]:5000/info/delete`. This time add `occupation` to the `body` as raw text, this will delete the occupation information.



```
occupation
```

Response



```
occupation deleted from information.
```

Run a `GET` request to the `/info` URL to check that the occupation key has been removed.

Exercises

Alter the file `app.py`, to create an API that uses JSON. Test the API using Postman.

▶ Hint