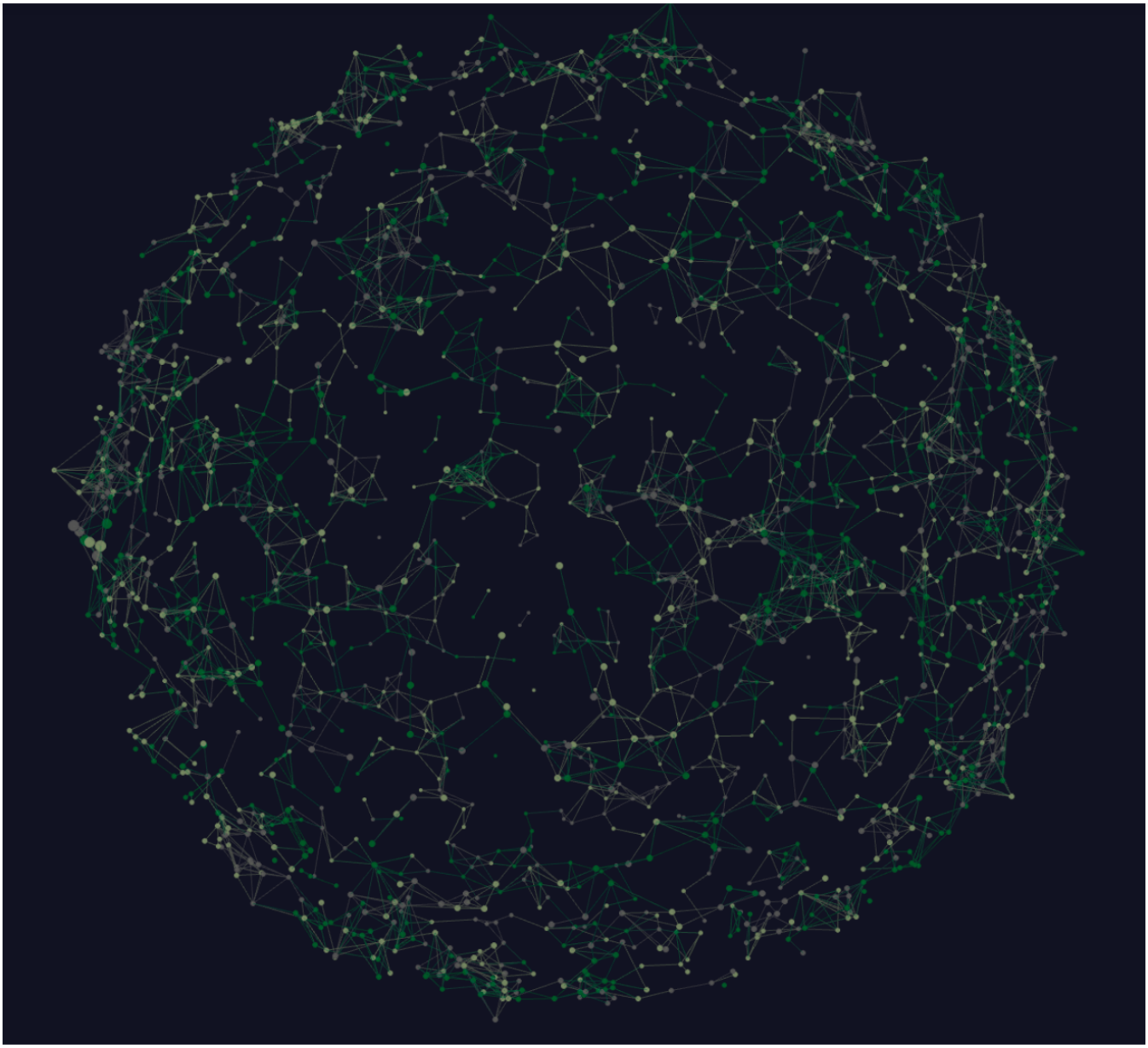
Published in Towards Data Science

Daniel Ellis Follow

Oct 26, 2020 · 4 min read ★ · Listen

Talking to Python from Javascript: Flask and the fetch API

Using Python to process data required for a dynamic web interface or visualisation.



A Sample Network — D. Ellis 2020

Within the field of data science, it is common to be required to use a selection of tools, each specific to their job. A role requiring visualisation using a web interface, but processing of a Python script, it is often better to build a bespoke visualisation in d3 or THREE.js to display it and then fetch data as required. This article covers the creation of a simple flask app that can serve data to a web interface using the Fetch API.

Creating a Flask app

Our `app.py` file contains the data required to create a web interface. To do this we use the flask (`pip install flask`) python library. For this we can use the following template:

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

#############################
# Additional code goes here #
#############################

#########  run app  #########
app.run(debug=True)
```

Here we start by importing the required functions, the app structure and the run command to start the app.

Index.html

Having defined our flask app, we now need to create a template webpage. This can be done by placing the file `index.html` within the templates directory.

```
<body>
<h1> Python Fetch Example</h1>
<p id='embed'>{{embed}}</p>

<p id='mylog'/>
<body>
```

Since we are using this as a template we can use a react-style replacement syntax for certain keywords. In this instance `{{embed}}` is to be replaced by the `embed_example` string in the snippet below.

```
@app.route('/')
def home_page():
```

This defines the code which runs when the home page of the flask app is reached and needs to be **added between the** `imports` **and the** `app.run()` **lines within** `app.py`**.**

Running the app

Finally, we can run the app and view it using `python app.py` and navigating to `https://127.0.0.1:5000/` in a web browser to view it.

GET and POST — how do they work?

When it comes to transferring data we rely on the GET and POST functions within the fetch API. These terms are pretty self-explanatory:

- POST refers to the sending of information to a location, similar to sending a letter.
- GET refers to the retrieval of data — you know you have mail, so you go to the post office to collect (ask for) it.

Test Function

Within `app.py` we can create a URL for GET requests. The following code defines the response when the URL is called.

```
@app.route('/test', methods=['GET', 'POST'])
def testfn():

    # GET request
    if request.method == 'GET':
        message = {'greeting':'Hello from Flask!'}
        return jsonify(message)  # serialize and use JSON headers

    # POST request
    if request.method == 'POST':
        print(request.get_json())  # parse as JSON
        return 'Sucesss', 200
```

Following a GET request, we define a dictionary containing a `greeting` element and


result:

```
{
  "greeting": "Hello from Flask!"
}
```

Calling for the data from Javascript

Now we have set up the server-side of things, we can use fetch command to retrieve the data from it. To do this we can use the `fetch` promise as follows:

```
fetch('/test')
    .then(function (response) {
        return response.json();
    }).then(function (text) {
        console.log('GET response:');
        console.log(text.greeting);
    });
```

Here we run a GET request on `/test` which converts the returned JSON string into an object, and then prints the `greeting` element to the web console. As usual, the JavaScript code should be nested between `<script>` tags within the HTML document.

Requesting data from a server

Now we have a working example we can expand it to include actual data. In reality, this could involve accessing a database, decrypting some information or filtering a table.

For the purpose of this tutorial we create a data array from which we index elements:

```
######## Example data, in sets of 3 ############
data = list(range(1,300,3))
```


Within our Flask app, we can add optional arguments to the GET request — in this case, the array index we are interested in. This is specified through the additional page extension within the page URL `/getdata/<index_no>` . This argument is then passed into the page function and processed (`data[int(index_no)]`) within the return command.

```
######## Data fetch ############
@app.route('/getdata/<index_no>', methods=['GET','POST'])
def data_get(index_no):

    if request.method == 'POST': # POST request
        print(request.get_text())  # parse as text
        return 'OK', 200

    else: # GET request
        return 't_in = %s ; result: %s ;'%(index_no,
data[int(index_no)])
```

Writing the Fetch request

The fetch request remains the same with the exception of changing the GET URL to include the index of the data element we are interested in. This is done within the JS script in `index.html` .

```
var index = 33;

fetch(`/getdata/${index}`)
      .then(function (response) {
          return response.text();
      }).then(function (text) {
          console.log('GET response text:');
          console.log(text);
      });
```

This time, instead of returning an object, we return a string and parse it as such. This can then be used as needed within the code — be it to update a figure or display a message.

We have explored a method to extract data using python and serve it to a javascript code for visualisation (alternatives include web/TCP sockets and file streaming). We are able to create a server-side python code to pre-process or decrypt data and only serve the required information to the client.

This is useful if we have constantly updating data, a large (high resource) dataset, or sensitive data we can not provide to the client directly.

Accompanying sample code used within this article can be found at:

wolfiex/FlaskFetch

Using the Fetch API to communicate between flask and a served webpage Dismiss GitHub is home to over 50 million...

github.com