Introduction to Flask - Notes

Demo project → https://chaosarium.pythonanywhere.com
Code on GitHub → https://github.com/chaosarium/flask-lecture
Docs → https://chaosarium.gitbook.io/113-flask

Flask is a Python web framework, i.e. something that lets you build web apps!

One cool thing you can then do is to make different machines talk to each other — you can even have things written in different languages talk to each

But first, how does the web work? See HTTP if not already familiar.

Files in the repo/zip file

- `basic` contains demo of Flask app setup and basic routes
- `topiclist` contains the demo project
- `docs` is the documentation source
- `starter_code` is where you can start implementing the demo project (or something else)

HTTP

- Stands for **Hypertext Transfer Protocol**
- A standard for how computers should talk to each other through internet connection, basically
- HTTP requests — whenever you want to get information, send message, etc.

An example http request (from Wikipedia):

```
GET / HTTP/1.1
Host: www.example.com
User-Agent: Mozilla/5.0
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*
/*;q=0.8
Accept-Language: en-GB,en;q=0.5
Accept-Encoding: gzip, deflate, br
Connection: keep-alive
```

Here, we see that this is a `GET` request to `www.example.com` to request data at `/` sent via `Mozilla/5.0` etc...

And here's the response

```
HTTP/1.1 200 OK
Date: Mon, 23 May 2005 22:38:34 GMT
Content-Type: text/html; charset=UTF-8
Content-Length: 155
Last-Modified: Wed, 08 Jan 2003 23:11:55 GMT
Server: Apache/1.3.3.7 (Unix) (Red-Hat/Linux)
ETag: "3f80f-1b6-3e1cb03b"
Accept-Ranges: bytes
Connection: close

<html>
  <head>
    <title>An Example Page</title>
  </head>
  <body>
    <p>Hello World, this is a very simple HTML document.</p>
  </body>
</html>
```

We see a status code `200`, date, time, content type, ..., and the data we got back, which is some `html` code.

If you open the inspect panel on your browser and go to the `Network` tab, chances are you will see http requests flying around.

(You may also see `https` somewhere. That's `http` with encryption)

In the example, we saw a `GET` request, but there are more. We'll briefly go over the two most common ones:

- `GET` is usually when you want to... duh get something
- `POST` is when you have data you want to send to the server

These requests can have some "payload". There are many places where you can include data in the request and in many different formats.

- `headers` is usually for metadata-ish key-value pairs
- `body` is where most of the data is
- `url params` is literally data embedded in the url.
  - For example, when you do a google search, you see the url looks something like this: `https://www.google.com/search?q=http&newwindow=1`. The params are:
    - `q` which has value `http`
    - `newwindow` which has value `1`

Setting up a Flask app

Create a file `app.py` (usually they call it that and it works)

Of course you import a bunch of things

```
from flask import Flask
from flask import request # now we can also use different HTTP methods
from flask import render_template # for rendering html templates
...
```

This line creates a flask app

```
app = Flask(__name__)
```

And this lets you run the app when you run `python app.py` in terminal

```
if __name__ == '__main__':
    # here, 127.0.0.1 is the IP address for localhost, and port can be though
of the channel at this address?
    app.run(host='127.0.0.1', port = 5000)
    # If you want your app to be available publically, you change the host to
0.0.0.0. Then people in your local network should be able to access your app
via your computer's IP
    # Note that your computer probably doesn't have a public IP, so someone
in, California, for example, won't be able to access your app (unless they go
on CMU VPN(?))
    # If you want your app to be made public everywhere, you need a public
IP.
```

But wait, we just created an app that doesn't "listen" to anything. We need to define functions so that it handles http requests like the one we saw earlier.

Here's the code for a function that listens at `/` and responds by sending hello world.

```
@app.route("/")
def root_route():
    return "Hello, World!"
```

Flask uses some sort of function decorator. We already said `app = Flask(__name__)`, so `app.route("...")` is creating a route for the app. And `"/"` just means root URL. Flasks makes it so that return sends our response. In this case we're just sending text.

Putting it together, we have:

```
from flask import Flask
from flask import request
from flask import render_template

app = Flask(__name__)

@app.route("/")
def root_route():
    return "Hello, World!"

if __name__ == '__main__':
    app.run(host='127.0.0.1', port = 5000)
```

More routes

We can do more than just sending back hello world — we can get data from the request, do something with it, and send back something fancy!

We can have non-root url

```
@app.route('/projects')
def projects():
    return 'The project page'
```

We can capture path pattern in url

```
@app.route('/user/<username>')
def user_profile(username):
    print(username)
    return f'hmm, see console'
```

We can specify the type. Also we are returning some html here
@app.route('/fact/<int:n>')

```
def fact_page(n):
    return f'<p style="overflow-wrap: anywhere;">{n}! = {fact(n)}</p>'
```

We can do something different depending on what type of request we got

```
@app.route('/getorpost', methods=["GET", "POST"])
def getorpost():
    if request.method == 'POST':
        return "method was POST"
    else:
        return "method was GET"
```

We can render html file. This example captures `name` from the request url and puts it in a `sayhello` template, which flask will look for in the `./templates` directory.

```
@app.route('/sayhello/<name>')
def sayhello(name):
    return render_template("sayhello.html", name = name)
```

We can return json too

```
@app.route("/api")
def api():
    return {
        "foo": "fooo",
        "bar": "barr",
        "quote": "hello world",
    }
```

Running a Flask app

There is a way to start a server with the flask command. You also have the option of enabling debug mode.

```
flask run
```

or

```
flask --debug run
```

or you can make the app run when you run the python file by writing:

```
if __name__ == '__main__':
    app.run(host='127.0.0.1', port = 5000) # this runs it on local network
    # app.run(host='0.0.0.0', port = 5000) # this makes it public
```

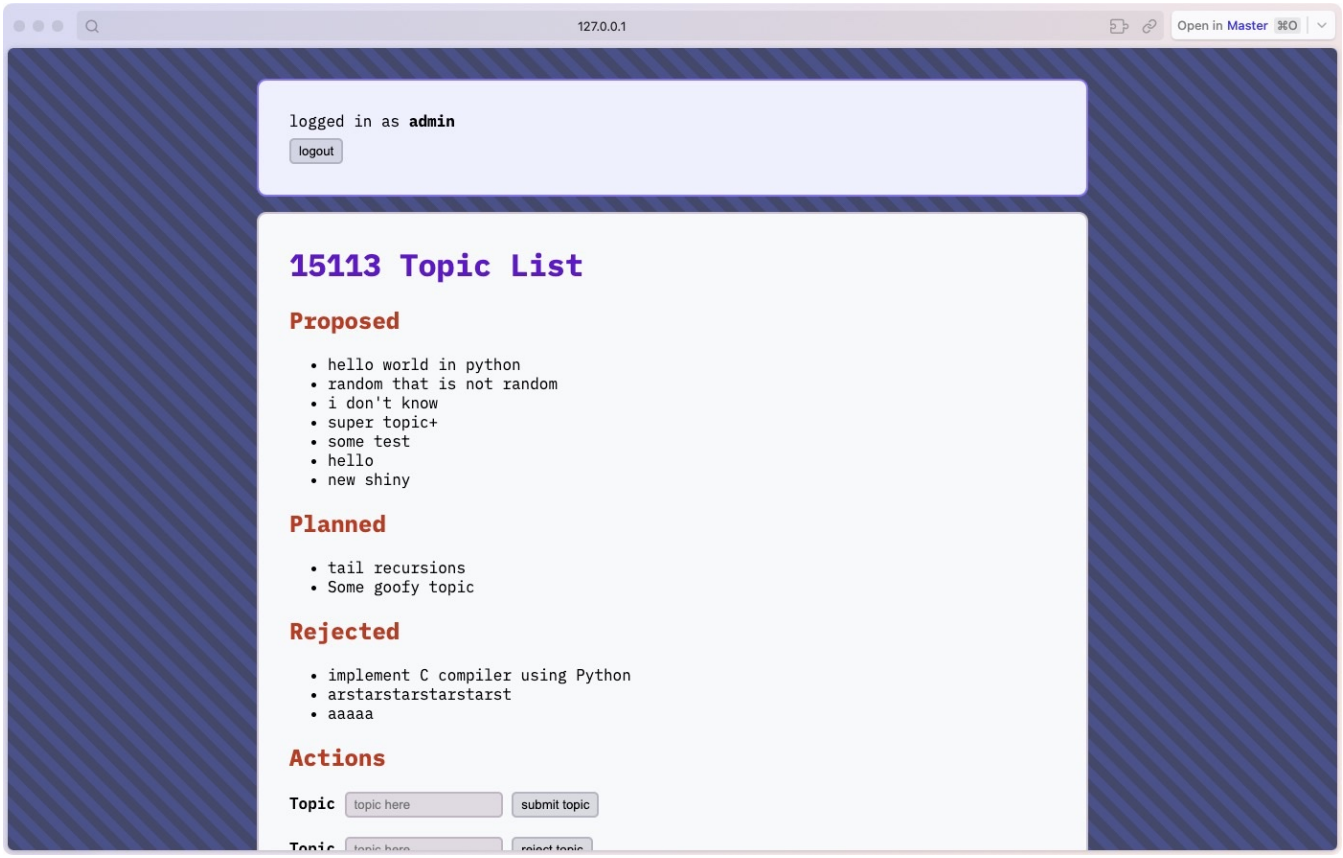
Example project — a 15113 topic list

Demo link: https://chaosarium.pythonanywhere.com

What we need:

- Some database to store data. See database
- A home page that displays data (`GET`)
- Places to update data using `POST`
    - `/add` topic
    - `/promote` topic

    - `/reject` topic
- Some interface to send `POST` requests
- UI design (maybe) (implemented on the stylish-topiclist branch)



Database

Database is a way to store data in a manageable way. By manageable it could mean structured, scalable, etc.

Technically, you can just use a dictionary to hold data, but notice what happens when you restart the app—all your data is lost.

One benefit of using a database, therefore, is that you can keep the data no matter what happens to your python process.

For the sake of this demo, we use TinyDB to keep things simple. You google for more options.

TinyDB

TinyDB stores data in json format. You can read more about the library on its website.

Deployment

PythonAnywhere

The demo app is hosted on PythonAnywhere, which is free and simple to use. To run your Flask app, do these:

1. Go to PythonAnywhere and register for an account
2. Under `Dashboard > Consoles`, you can open up a terminal to install dependencies. For our app, do `pip install tinydb`
3. Click `Web apps`, then `Add a new web app`, then `Next`
4. Select `Flask` and use `Python 3.10`
5. Set a path to where your app is on the remote file system
6. You should now see "Hello from Flask!" in the site that was just created. Start modifying the python code!

Other options

- Vercel: supports continuous deployment from git but doesn't allow writing to disk
- VPS: you'll have to set things up on a server, but it's fun