Full Stack Development
Sesi 7

FLASK: BASIC

Flask

Flask is a micro web framework written in Python.

It is classified as a microframework because it does not require particular tools or libraries. It has no database abstraction layer, form validation, or any other components where pre-existing third-party libraries provide common functions.

Setting Up

Use a virtual environment to manage the dependencies for your project, both in development and in production.

Python comes bundled with the venv module to create virtual environments.

Create a project folder and a hacktiv8 folder within:

```
$ mkdir myproject
$ cd myproject
$ python3 -m venv hacktiv8
```

```
Windows: venv\Scripts\activate
Mac/Linux: . venv/bin/activate
```

Install Flask

Within the activated environment, use the following command to install Flask:

```
$ pip install Flask
```

Hello World

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

@app.route('/')
def hello_world():
    return 'Hello, World!'

if __name__ == '__main__':
    app.run()
```

What did that code do?

- First we imported the Flask class. An instance of this class will be our WSGI application.
- Next we create an instance of this class. The first argument is the name of the application’s module or package. __name__ is a convenient shortcut for this that is appropriate for most cases. This is needed so that Flask knows where to look for resources such as templates and static files.
- We then use the route() decorator to tell Flask what URL should trigger our function.
- The function returns the message we want to display in the user’s browser. The default content type is HTML, so HTML in the string will be rendered by the browser.

Save it as helloH8.py or something similar. Make sure to not call your application flask.py because this would conflict with Flask itself.

HTML Escaping

When returning HTML (the default response type in Flask), any user-provided values rendered in the output must be escaped to protect from injection attacks. HTML templates rendered with Jinja, introduced later, will do this automatically.

escape(), shown here, can be used manually. It is omitted in most examples for brevity, but you should always be aware of how you're using untrusted data.

```
from markupsafe import escape

@app.route("/<name>")
def hello(name):
    return f"Hello, {escape(name)}!"
```

If a user managed to submit the name <script>alert("bad")</script>, escaping causes it to be rendered as text, rather than running the script in the user's browser.

<name> in the route captures a value from the URL and passes it to the view function. These variable rules are explained below.

Routing

Modern web applications use meaningful URLs to help users. Users are more likely to like a page and come back if the page uses a meaningful URL they can remember and use to directly visit a page.

Use the route() decorator to bind a function to a URL.

```
@app.route('/')
def index():
    return 'Index Page'

@app.route('/hello')
def hello():
    return 'Hello, World'
```

You can do more! You can make parts of the URL dynamic and attach multiple rules to a function.

Variable Rules

You can add variable sections to a URL by marking sections with <variable_name>. Your function then receives the <variable_name> as a keyword argument. Optionally, you can use a converter to specify the type of the argument like <converter:variable_name>.

```
from markupsafe import escape

@app.route('/user/<username>')
def show_user_profile(username):
    # show the user profile for that user
    return f'User {escape(username)}'

@app.route('/post/<int:post_id>')
def show_post(post_id):
    # show the post with the given id, the id is an integer
    return f'Post {post_id}'

@app.route('/path/<path:subpath>')
def show_subpath(subpath):
    # show the subpath after /path/
    return f'Subpath {escape(subpath)}'
```

HTTP Methods

Web applications use different HTTP methods when accessing URLs. You should familiarize yourself with the HTTP methods as you work with Flask. By default, a route only answers to GET requests. You can use the methods argument of the route() decorator to handle different HTTP methods.

```
from flask import request

@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        return do_the_login()
    else:
        return show_the_login_form()
```

If GET is present, Flask automatically adds support for the HEAD method and handles HEAD requests according to the HTTP RFC. Likewise, OPTIONS is automatically implemented for you.

Static Files

Dynamic web applications also need static files. That’s usually where the CSS and JavaScript files are coming from. Ideally your web server is configured to serve them for you, but during development Flask can do that as well. Just create a folder called static in your package or next to your module and it will be available at /static on the application.

To generate URLs for static files, use the special 'static' endpoint name:

```
url_for('static', filename='style.css')
```

The file has to be stored on the filesystem as static/style.css.

Rendering Templates

Generating HTML from within Python is not fun, and actually pretty cumbersome because you have to do the HTML escaping on your own to keep the application secure. Because of that Flask configures the Jinja2 template engine for you automatically.

To render a template you can use the render_template() method. All you have to do is provide the name of the template and the variables you want to pass to the template engine as keyword arguments. Here's a simple example of how to render a template:

```
from flask import render_template

@app.route('/hello/')
@app.route('/hello/<name>')
def hello(name=None):
    return render_template('hello.html', name=name)
```

Flask will look for templates in the templates folder.

External References

Flask Documentation - [Visit Here]

HACKTIV8