Web Development with Flask

Juan F. Imbet *Ph.D.*

Flask

Overview

```python
# save this as app.py
from flask import Flask

app = Flask(__name__)

@app.route("/")
def hello_world():
    return "<p>Hello, World!</p>"
```

```
$ flask run
  * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

So what did that code do?

1. First we imported the `Flask` class. An instance of this class will be our Web Server Gateway Interface (WSGI) application.
2. 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.
3. We then use the `route()` decorator to tell Flask what URL should trigger our function.
4. 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.

Running the app

- Save it as `hello.py` or something similar. Make sure to not call your application `flask.py` because this would conflict with Flask itself.
- To run the application, use the `flask` command or `python -m flask`. You need to tell the Flask where your application is with the `--app` option.

```
$ flask --app hello run
 * Serving Flask app 'hello'
 * Running on http://127.0.0.1:5000 (Press CTRL+C to quit)
```

- As a shortcut, if the file is named `app.py` or `wsgi.py`, you don't have to use `--app`. See Command Line Interface for more details.

The Server

- This launches a very simple builtin server, which is good enough for testing but probably not what you want to use in production.
- Now head over to http://127.0.0.1:5000/, and you should see your hello world greeting.
- If another program is already using port 5000, you'll see `OSError: [Errno 98]` or `OSError: [WinError 10013]` when the server tries to start. See Address already in use for how to handle that.
- If you run the server you will notice that the server is only accessible from your own computer, not from any other in the network. This is the default because in debugging mode a user of the application can execute arbitrary Python code on your computer.

Debug Mode

- The `flask run` command can do more than just start the development server. By enabling debug mode, the server will automatically reload if code changes, and will show an interactive debugger in the browser if an error occurs during a request.

Warning

- The debugger allows executing arbitrary Python code from the browser. It is protected by a pin, but still represents a major security risk. Do not run the development server or debugger in a production environment.

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)}!"
```

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.

```python
@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>`.

Example

```python
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

- `GET` is used to request data from a specified resource. For example, a query to a search engine.
- `POST` is used to send data to a server to create/update a resource. For example, submitting a form to create a new user.

```
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()
```

HTTP Methods

You can also separate views for different methods into different functions. Flask provides a shortcut for decorating such routes with `get()`, `post()`, etc. for each common HTTP method.

```
@app.get('/login')
def login_get():
    return show_the_login_form()

@app.post('/login')
def login_post():
    return do_the_login()
```

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.
- `url_for('static', filename='style.css')` looks up the file `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.
- Templates can be used to generate any type of text file. For web applications, you'll primarily be generating HTML pages, but you can also generate markdown, plain text for emails, and anything else.

Jinja2

- Jinja2 is a modern and designer-friendly templating language for Python, modelled after Django’s (a web framework) templates. It is fast, widely used, and secure with the optional sandboxed template execution environment.

Rendering Templates

- 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:

```python
from flask import render_template

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

Rendering Templates

- Flask will look for templates in the `templates` folder. So if your application is a module, this folder is next to that module, if it's a package it's actually inside your package:

Case 1: a module:

```
/application.py
/templates
    /hello.html
```

Case 2: a package:

```
/application
    /__init__.py
    /templates
        /hello.html
```

Example Template

```
<!doctype html>
<title>Hello from Flask</title>
{% if person %}
  <h1>Hello {{ person }}!</h1>
{% else %}
  <h1>Hello, World!</h1>
{% endif %}
```

Templates are especially useful if inheritance is used. If you want to know how that works, see Template Inheritance. Basically template inheritance makes it possible to keep certain elements on each page (like header, navigation and footer).

The `request` Object

A global object.

```python
from flask import request

@app.route('/login', methods=['POST', 'GET'])
def login():
    error = None
    if request.method == 'POST':
        if valid_login(request.form['username'],
                       request.form['password']):
            return log_the_user_in(request.form['username'])
        else:
            error = 'Invalid username/password'
    # the code below is executed if the request method
    # was GET or the credentials were invalid
    return render_template('login.html', error=error)
```

Cookies

- To access cookies you can use the `cookies` attribute. To set cookies you can use the `set_cookie` method of response objects.

```python
from flask import request

@app.route('/')
def index():
    username = request.cookies.get('username')
    # use cookies.get(key) instead of cookies[key] to not get a
    # KeyError if the cookie is missing.
```

```python
from flask import make_response

@app.route('/')
def index():
    resp = make_response(render_template(...))
    resp.set_cookie('username', 'the username')
    return resp
```

APIS with JSON

- A common response format when writing an API is JSON. It’s easy to get started writing such an API with Flask. If you return a `dict` or `list` from a view, it will be converted to a JSON response.

```
@app.route("/me")
def me_api():
    user = get_current_user()
    return {
        "username": user.username,
        "theme": user.theme,
        "image": url_for("user_image", filename=user.image),
    }

@app.route("/users")
def users_api():
    users = get_all_users()
    return [user.to_json() for user in users]
```

Sessions

In addition to the `request` object there is also a second object called `session` which allows you to store information specific to a user from one request to the next. This is implemented on top of cookies for you and signs the cookies cryptographically. What this means is that the user could look at the contents of your cookie but not modify it, unless they know the secret key used for signing.

Example Sessions

```python
from flask import session

# Set the secret key to some random bytes. Keep this really secret!
app.secret_key = b'_5#y2L"F4Q8z\n\xec]/'

@app.route('/')
def index():
    if 'username' in session:
        return f'Logged in as {session["username"]}'
    return 'You are not logged in'
```

Login + Logout

```python
@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        session['username'] = request.form['username']
        return redirect(url_for('index'))
    return '''
        <form method="post">
            <p><input type=text name=username>
            <p><input type=submit value=Login>
        </form>
    '''

@app.route('/logout')
def logout():
    # remove the username from the session if it's there
    session.pop('username', None)
    return redirect(url_for('index'))
```

How to generate good secret keys

A secret key should be as random as possible. Your operating system has ways to generate pretty random data based on a cryptographic random generator. Use the following command to quickly generate a value for `Flask.secret_key` (or `SECRET_KEY`):

```
python -c 'import secrets; print(secrets.token_hex())'
'192b9bdd22ab9ed4d12e236c78afcb9a393ec15f71bbf5dc987d54727823bcbf'
```

APIs using Flask

- API with arguments

```python
from flask import Flask, request, jsonify

app = Flask(__name__)

@app.route('/api', methods=['GET'])
def api():
    name = request.args.get('name')
    return jsonify({'name': name})
```

- Call the API with `http://localhost:5000/api?name=Juan`