Web Programming with Flask

Author: Denis Rinfret

The Flask Framework

Web Development One Drop at a Time

- Documentation: https://flask.palletsprojects.com/en/2.2.x/
- Recommended book: **Flask Web Development**, by Miguel Grinberg, 2018, Published by O'Reilly Media.
- *Flask* is a web development *micro*framework
  - *micro* in microframework means Flask aims to keep the core *simple* but *extensible*
  - *Flask won't make many decisions for you*, such as what database to use.
  - Those decisions that it does make, such as what templating engine to use, are easy to change
  - Everything else is *up to you*, so that Flask can be everything you need and nothing you don't

Flask Basics

First Web Site

- Flask's *hello world* program is very simple compared to other web frameworks
- We need a Python environment with the *Flask* package installed
  - first option: use this command in a Python virtual environment
    - `pip install Flask`
    - more details here: https://flask.palletsprojects.com/en/2.2.x/installation/#installation
  - second option (recommended): use PyCharm to simplify creating and updating Python virtual environments and Flask projects
    - create a new project
    - select *Flask* as the project type
    - select an existing Python virtual environment to use or create a new one
      - note: you should probably not create a new Python virtual environment for each project since you will have to reinstall all the necessary packages in each environment and use up more disk space in the process
- PyCharm will automatically create a basic Flask project for you
- In the `app.py` file, you will have code similar to the code given below

```python
from flask import Flask

app = Flask(__name__)


@app.route('/')
def hello_world():  # put application's code here
    return 'Hello World!'


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

- Code examples available in project `02_flask_basics`
  - Note that usually, a Flask project would contain only one *main* file, `app.py`, to start the development web server
  - To reduce the number of different projects, many *main* files have been grouped together for the examples
  - In real life, a project would have only 1 `app.py` file

Code Overview

- We always need to import the `Flask` class from the `flask` package
- We create a Flask `app` by creating an instance of the `Flask` class
- `@app.route('/')` is a function decorator, applying to the following function definition
  - it means that when a request to access the root page `/` is sent to the server
  - it will execute the `hello_world` function
  - this function simply returns a string containing some simple HTML
  - this string will be sent back to the client (probably a web browser) which sent the request
- If `__name__` is `__main__`, then we need to call `app.run()` to start the server
  - this will be the case when we run the `app.py` directly
- In production mode, the server is normally started differently (more on this later when discussing deployment strategies)

How to Run

- Run the server
- In your browser, go to `http://127.0.0.1:5000`
- You should see *Hello World!* in a `h1` header
- Modify the route to `@app.route('/hello')`
- Reload in browser: did something change?
- By default in PyCharm, servers do **not** run in *debug* mode
- You need to restart the server to see the changes

- Or modify the run configuration to run the server in debug mode (check the `FLASK_DEBUG` option)
- In debug mode, it will detect changes to source code files and rerun the server automatically

Returning HTML

File: `01_app.py`

Normally, we will not return simple text strings, but we will return HTML or JSON instead. We will see how to return HTML templates later.

```
from flask import Flask

app = Flask(__name__)


@app.route('/index')
def hello_world():
    return '<h1>Hello Web App!</h1>'


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

`url_for` Example

File: `02_app.py`

- Example with 2 endpoints
  - each `@app.route()` with its associated function is often called an *endpoint*
- The `index` endpoint uses a format string to create an `a` element referring to the other endpoint
- `url_for` takes a function name as its argument, and returns the route associated to it
  - in this case, it will return `/hello`
- Why not write `/hello` directly instead of calling `url_for`?
  - in this simple example, we don't need to use `url_for`
  - but in larger project, it's better to use `url_for` in case we need to move endpoints around
  - try changing the route for `hello_web_app`, and with `url_for`, you don't need to change anything else
  - without `url_for`, you would need to update every reference to it

```python
from flask import Flask, url_for

app = Flask(__name__)


@app.route('/')
def index():
    return f'<a href="{url_for("hello_web_app")}">Hello Web App</a>'


@app.route('/hello')
def hello_web_app():
    return '<h1>Hello Web App!</h1>'


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

First Template

Files: `03_app.py` and `03_index.html`

- Returning full HTML pages this way is a bad idea
  - it would take too much space in the source code
  - it's better to separate the logic code from the presentation code
- Use the `render_template` function to return the HTML code stored in an external file
- Usually, template files are stored in the `templates` folder
- By default, Flask uses the *jinja2* package as the template rendering engine

```python
from flask import Flask, render_template

app = Flask(__name__)

@app.route('/')
def index():
    return render_template("03_index.html")
```

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>First Template</title>
</head>
<body>
<h1>First Template</h1>
<ul>
    <li><a href="/hello">/hello</a></li>
```

```
    <li><a href="/goodbye">/goodbye</a></li>
</ul>
</body>
</html>
```

- This template is static, it will always be rendered the same way
- In general, templates are dynamic
  - variables can be passed to the template
  - the template can include loops, conditionals, … to make the contents more dynamic
- Templates provide a skeleton for the page, the contents can change depending on the situation
- You can pass variables to a template in this way (file `04_app.py`)

```
@app.route('/')
def index():
    return render_template('04_index.html',
                           hello=url_for('hello_web_app'),
                           goodbye=url_for('goodbye_web_app'))
```

- To get the value of a variable in a template, use the double curly brackets notation `{{ }}`
- File: `04_index.html`

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Second Template</title>
</head>
<body>
<h1>Second Template</h1>
<ul>
    <li><a href="{{ hello }}">{{ hello }}</a></li>
    <li><a href="{{ goodbye }}">{{ goodbye}} </a></li>
</ul>
</body>
</html>
```

- You can also pass a list to a template (file `05_app.py`)

```
@app.route('/')
def index():
    links = [url_for('hello_web_app'), url_for('goodbye_web_app')]
    return render_template('05_index.html', links=links)
```

- You will probably need to use a loop in the template to get all values in the list (file `05_index.html`)

- The curly bracket percent notation is used for many template constructs, such as loops and conditionals and many others

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Second Template</title>
</head>
<body>
<h1>Second Template</h1>
<ul>
    {% for link in links %}
    <li><a href="{{ link }}">{{ link }}</a></li>
    {% endfor %}
</ul>
</body>
</html>
```

Dynamic Routes

- Routes can also be dynamic
- Routes can contain *variables* (kind of)
- There must be a corresponding parameter in the endpoint function (file `06_app.py`)

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

- `<name>` denotes a variable part of the route
- By default, it should be a string not containing any special URL characters (such as `/`)
- The `name` parameter will be equal to the part of the route matching `<name>`
- The template can use the name variable as shown earlier, with the `{{ }}` notation