Lab session 9

Friday, October 9, 2020

Quick Recap:

Clients such as web browsers send *requests* to the web server, which in turn sends them to the Flask application instance. The Flask application instance needs to know what code it needs to run for each URL requested, so it keeps a mapping of URLs to Python functions. The association between a URL and the function that handles it is called a *route*.

The most convenient way to define a route in a Flask application is through the ***app.route*** decorator exposed by the application instance.

Initialization

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

Routes

```
@app.route('/')
def index():
    return '<h1>Hello Class!</h1>'
```

If the application is deployed on a server associated with the www.example.com domain name, then navigating to http://www.example.com/ in your browser would trigger index() function to run on the server. The return value of this view function is the *response* the client receives. If the client is a web browser, this response is the document that is displayed to the user in the browser window. A response returned by a view function can be a simple string with HTML content, but it can also take more complex forms, as you will see later.

Try this out - Personalized Greeting :

```
@app.route('/user/<name>')
def user(name):
    return '<h1>Hello, {}!</h1>'.format(name)
```

Dynamic Routes

If you pay attention to how some URLs for services that you use every day are formed, you will notice that many have variable sections. For example, the URL for your Facebook profile page has the format *https://www.facebook.com/<your-name>*, which includes your username, making it different for each user. Flask supports these types of URLs using a special syntax in the app.route decorator. The above example defines a route that has a dynamic component.

The portion of the route URL enclosed in angle brackets is the dynamic part. Any URLs that match the static portions will be mapped to this route, and when the view function is invoked, the dynamic component will be passed as an argument. In the preceding example, the name argument is used to generate a response that includes a personalized greeting.

Debug Mode

Flask applications can optionally be executed in *debug mode*. In this mode, two very convenient modules of the development server called the *reloader* and the *debugger* are enabled by default.

When the reloader is enabled, Flask watches all the source code files of your project and automatically restarts the server when any of the files are modified. Having a server running with the reloader enabled is extremely useful during development, because every time you modify and save a source file, the server automatically restarts and picks up the change.

To enable debug mode programmatically, use ***app.run(debug=True)***

Bonus :

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

@app.route('/')
```

```
def index():
    user_agent = request.headers.get('User-Agent')
    return '<p>Your browser is {}</p>'.format(user_agent)
```

The Jinja2 Template Engine

The key to writing applications that are easy to maintain is to write clean and well-structured code. The examples that you have seen so far are too simple to demonstrate this, but Flask view functions have two completely independent purposes disguised as one, which creates a problem.

Consider a user who is registering a new account on a website. The user types an email address and a password in a web form and clicks the Submit button. On the server, a request with the data provided by the user arrives, and Flask dispatches it to the view function that handles registration requests. This view function needs to talk to the database to get the new user added, and then generate a response to send back to the browser that includes a success or failure message. These two types of tasks are formally called *business logic* and *presentation logic*, respectively.

Moving the presentation logic into *templates* helps improve the maintainability of the application.

A template is a file that contains the text of a response, with placeholder variables for the dynamic parts that will be known only in the context of a request. The process that replaces the variables with actual values and returns a final response string is called *rendering*. For the task of rendering templates, Flask uses a powerful template engine called *Jinja2*.

Example of a template:

```
<h1>Hello, {{ name }}!</h1>
```

By default Flask looks for templates in a *templates* subdirectory located inside the main application directory.

```
from flask import Flask, render_template
# ...
@app.route('/')
def index():
    return render_template('index.html')
@app.route('/user/<name>')
def user(name):
    return render_template('user.html', name=name)
```

The function ***render_template()*** provided by Flask integrates the Jinja2 template engine with the application. This function takes the filename of the template as its first argument. Any additional arguments are key-value pairs that represent actual values for variables referenced in the template.

Variables can be modified with *filters*, which are added after the variable name with a pipe character as separator. For example, the following template shows the name variable capitalized:

```
Hello, {{ name|capitalize }}
```

Filter name	Description
safe	Renders the value without applying escaping
capitalize	Converts the first character of the value to uppercase and the rest to lowercase
lower	Converts the value to lowercase characters
upper	Converts the value to uppercase characters
title	Capitalizes each word in the value
trim	Removes leading and trailing whitespace from the value
striptags	Removes any HTML tags from the value before rendering

Control Structures

Jinja2 offers several control structures that can be used to alter the flow of the template. This section introduces some of the most useful ones with simple examples.

If:

```
{% if user %}
    Hello, {{ user }}!
{% else %}
    Hello, Stranger!
{% endif %}
```

For loop:

```
<ul>
    {% for comment in comments %}
        <li>{{ comment }}</li>
    {% endfor %}
</ul>
```

Even Macros (or Functions) can be used in Jinja:

```
{% macro render_comment(comment) %}
    <li>{{ comment }}</li>
{% endmacro %}
<ul>
    {% for comment in comments %}
        {{ render_comment(comment) }}
    {% endfor %}
</ul>
```

To make macros more reusable, they can be stored in standalone files that are then *imported* from all the templates that need them:

```
{% import 'macros.html' as macros %}
<ul>
```

```
        {% for comment in comments %}
            {{ macros.render_comment(comment) }}
        {% endfor %}
    </ul>
```

Bootstrap Integration

Bootstrap is an open-source web browser framework from Twitter that provides user interface components that help create clean and attractive web pages that are compatible with all modern web browsers used on desktop and mobile platforms.

Bootstrap is a client-side framework, so the server is not directly involved with it. All the server needs to do is provide HTML responses that reference Bootstrap's Cascading Style Sheets (CSS) and JavaScript files, and instantiate the desired user interface elements through HTML, CSS, and JavaScript code. The ideal place to do all this is in templates.