Flask

Python 2.x 2.6–2.7 , Python 3.x 3.0–3.6

Flask is a Python micro web framework used to run major websites including Pintrest, Twilio, and Linkedin. This topic explains and demonstrates the variety of features Flask offers for both front and back end web development.

Examples

Files and Templates

Instead of typing our HTML markup into the return statements, we can use the render_template() function:

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

@app.route("/about")
def about():
    return render_template("about-us.html")

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=80, debug=True)
```

This will use our template file about-us.html . To ensure our application can find this file we must organize our directory in the following format:

```
- application.py
/templates
    - about-us.html
    - login-form.html
/static
    /styles
        - about-style.css
        - login-style.css
    /scripts
        - about-script.js
        - login-script.js
```

Most importantly, references to these files in the HTML must look like this:

`<link rel="stylesheet" type="text/css", href="{{url_for('static', filename='styles/about-style.css')}}">`

which will direct the application to look for about-style.css in the styles folder under the static folder. The same format of path applies to all references to images, styles, scripts, or files.

HTTP Methods

The two most common HTTP methods are **GET** and **POST** . Flask can run different code from the same URL dependent on the HTTP method used. For example, in a web service with accounts, it is most convenient to route the sign in page and the sign in process through the same URL. A GET request, the same that is made when you open a URL in your browser should show the login form, while a POST request (carrying login data) should be processed separately. A route is also created to handle the DELETE and PUT HTTP method.

```
@app.route("/login", methods=["GET"])
def login_form():
    return "This is the login form"
@app.route("/login", methods=["POST"])
def login_auth():
    return "Processing your data"
@app.route("/login", methods=["DELETE", "PUT"])
def deny():
    return "This method is not allowed"
```

To simplify the code a bit, we can import the request package from flask.

```
from flask import request

@app.route("/login", methods=["GET", "POST", "DELETE", "PUT"])
def login():
    if request.method == "DELETE" or request.method == "PUT":
        return "This method is not allowed"
    elif request.method == "GET":
        return "This is the login forum"
    elif request.method == "POST":
        return "Processing your data"
```

To retrieve data from the POST request, we must use the request package:

```
from flask import request
@app.route("/login", methods=["GET", "POST", "DELETE", "PUT"])
def login():
    if request.method == "DELETE" or request.method == "PUT":
        return "This method is not allowed"
    elif request.method == "GET":
        return "This is the login forum"
    elif request.method == "POST":
        return "Username was " + request.form["username"] + " and password was " + request.form|
```

Jinja Templating

Similar to Meteor.js, Flask integrates well with front end templating services. Flask uses by default Jinja Templating. Templates allow small snippets of code to be used in the HTML file such as conditionals or loops.

When we render a template, any parameters beyond the template file name are passed into the HTML templating service. The following route will pass the username and joined date (from a function somewhere else) into the HTML.

```
@app.route("/users/<username>)
def profile(username):
    joinedDate = get_joined_date(username) # This function's code is irrelevant
    awards = get_awards(username) # This function's code is irrelevant
    # The joinDate is a string and awards is an array of strings
    return render_template("profile.html", username=username, joinDate=joinDate, awards=awards)
```

When this template is rendered, it can use the variables passed to it from the `render_template()` function. Here are the contents of `profile.html` :

```
<!DOCTYPE html>
<html>
    <head>
        # if username
            <title>Profile of {{ username }}</title>
        # else
            <title>No User Found</title>
        # endif
    <head>
    <body>
        {% if username %}
            <h1>{{ username }} joined on the date {{ date }}</h1>
            {% if len(awards) > 0 %}
                <h3>{{ username }} has the following awards:</h3>
                <ul>
                {% for award in awards %}
                    <li>{{award}}</li>
                {% endfor %}
                </ul>
            {% else %}
                <h3>{{ username }} has no awards</h3>
            {% endif %}
        {% else %}
            <h1>No user was found under that username</h1>
        {% endif %}
        {# This is a comment and doesn't affect the output #}
    </body>
</html>
```

The following delimiters are used for different interpretations:

- `{% ... %}` denotes a statement
- `{{ ... }}` denotes an expression where a template is outputted
- `{# ... #}` denotes a comment (not included in template output)
- `{# ... ##` implies the rest of the line should be interpreted as a statement

Routing URLs

With Flask, URL routing is traditionally done using decorators. These decorators can be used for static routing, as well as routing URLs with parameters. For the following example, imagine this Flask script is running the website `www.example.com` .

```
@app.route("/")
def index():
    return "You went to www.example.com"
```

```
@app.route("/about")
def about():
    return "You went to www.example.com/about"

@app.route("/users/guido-van-rossum")
    return "You went to www.example.com/guido-van-rossum"
```

With that last route, you can see that given a URL with /users/ and the profile name, we could return a profile. Since it would be horribly inefficient and messy to include a `@app.route()` for every user, Flask offers to take parameters from the URL:

```
@app.route("/users/<username>")
def profile(username):
    return "Welcome to the profile of " + username

cities = ["OMAHA", "MELBOURNE", "NEPAL", "STUTTGART", "LIMA", "CAIRO", "SHANGHAI"]

@app.route("/stores/locations/<city>")
def storefronts(city):
    if city in cities:
        return "Yes! We are located in " + city
    else:
        return "No. We are not located in " + city
```

The basics

The following example is an example of a basic server:

```
# Imports the Flask class
from flask import Flask
# Creates an app and checks if its the main or imported
app = Flask(__name__)

# Specifies what URL triggers hello_world()
@app.route('/')
# The function run on the index route
def hello_world():
    # Returns the text to be displayed
    return "Hello World!"

# If this script isn't an import
if __name__ == "__main__":
    # Run the app until stopped
    app.run()
```

Running this script (with all the right dependencies installed) should start up a local server. The host is `127.0.0.1` commonly known as **localhost** . This server by default runs on port **5000** . To access your webserver, open a web browser and enter the URL `localhost:5000` or `127.0.0.1:5000` (no difference). Currently, only your computer can access the webserver.

`app.run()` has three parameters, **host** , **port** , and **debug** . The host is by default `127.0.0.1` , but setting this to `0.0.0.0` will make your web server accessible from any device on your network using your private IP address in the URL. the port is by default 5000 but if the parameter is set to port `80` , users will not need to specify a port number as browsers use port 80 by default. As for the debug option, during the development process (never in production) it helps to set this parameter to True, as your server will restart when changes made to your Flask project.

```
if __name__ == "__main__":
    app.run(host="0.0.0.0", port=80, debug=True)
```

The Request Object

The `request` object provides information on the request that was made to the route. To utilize this object, it must be imported from the flask module:

```
from flask import request
```

URL Parameters

In previous examples `request.method` and `request.form` were used, however we can also use the `request.args` property to retrieve a dictionary of the keys/values in the URL parameters.

```
@app.route("/api/users/<username>")
def user_api(username):
    try:
        token = request.args.get("key")
        if key == "pA55w0Rd":
            if isUser(username): # The code of this method is irrelevant
                joined = joinDate(username) # The code of this method is irrelevant
                return "User " + username + " joined on " + joined
            else:
                return "User not found"
        else:
            return "Incorrect key"
    # If there is no key parameter
    except KeyError:
        return "No key provided"
```

To correctly authenticate in this context, the following URL would be needed (replacing the username with any username:

`www.example.com/api/users/guido-van-rossum?key=pa55w0Rd`

File Uploads

If a file upload was part of the submitted form in a POST request, the files can be handled using the `request` object:

```
@app.route("/upload", methods=["POST"])
def upload_file():
    f = request.files["wordlist-upload"]
    f.save("/var/www/uploads/" + f.filename) # Store with the original filename
```

Cookies

The request may also include cookies in a dictionary similar to the URL parameters.

```
@app.route("/home")
def home():
    try:
        username = request.cookies.get("username")
        return "Your stored username is " + username
    except KeyError:
        return "No username cookies was found")
```

Syntax

```
@app.route("/urlpath", methods=["GET", "POST", "DELETE", "PUTS", "HEAD", "OPTIONS"])
```

```
@app.route("/urlpath/<param>", methods=["GET", "POST", "DELETE", "PUTS", "HEAD",
"OPTIONS"])
```

Parameters

Remarks