Python | Flask

Session 2

- Responses
- Templates
- Jinja

by Mohammad Amin H.B. Tehrani

www.maktabsharif.ir

Practice: Translator!

Translator

Create an Flask web app, that allows GET & POST requests and translates the sent **text** value (in request) to **TARGET** language, **FROM** language(default: auto), by **PROVIDER** (default: google)

URL Routing:
/<target_lang>/<from_lang>/<provider>
/<target_lang>/<from_lang> (provider = google)
/<target_lang> (from_lang= auto, provider = google)

Params:
text: Text to translate.

Response: Translated plain text

Hint: Run server on host='0.0.0.0' to be accessible from another devices on the private network.

Responses

Responses

Intro

The return type of flask view function must be a

- String: HTML Response
- Dict: Json Response
- Tuple: (Response data, status code)
- Response instance

Example: HTML response (String)

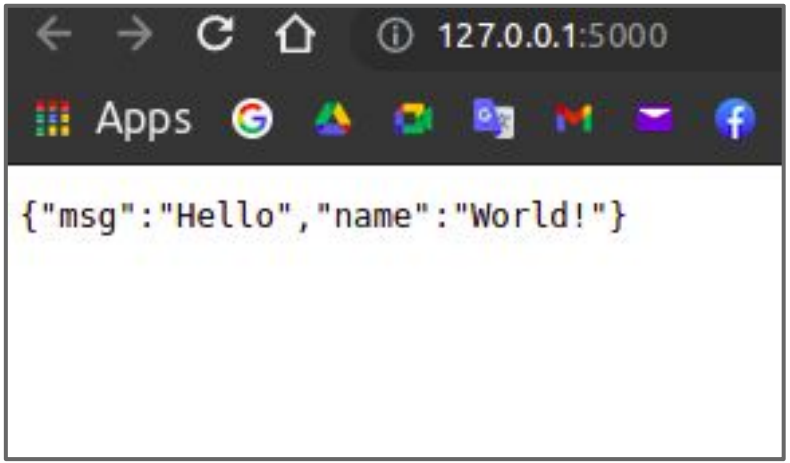
String outputs prepare a HTML Response to Client:

```
@app.route('/')
def index():
    return "<h1 style='color:red'>Hello world!</h1>"
```

Example: Json response (Dict)

You can prepare JSON response by return dict Value from view function:

```
@app.route('/')
def index():
    return {'msg': 'Hello', 'name': 'World!'}
```

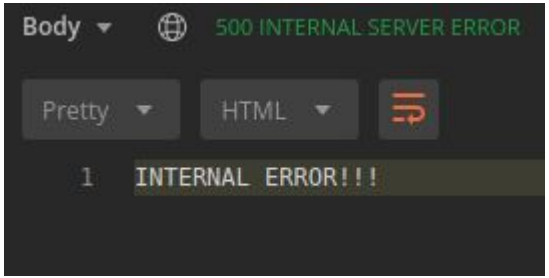


Example: Response with Status (Tuple)

Tuple response:

(response_data, status_code)

```
@app.route('/')
def index():
    return ("INTERNAL ERROR!!!", 500)
```

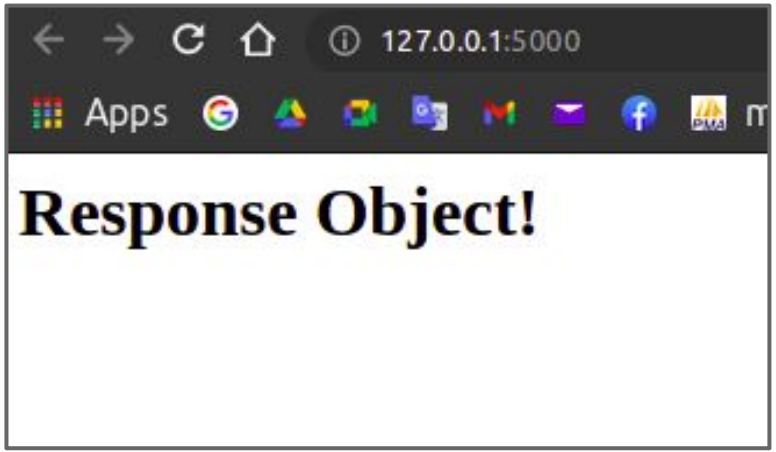


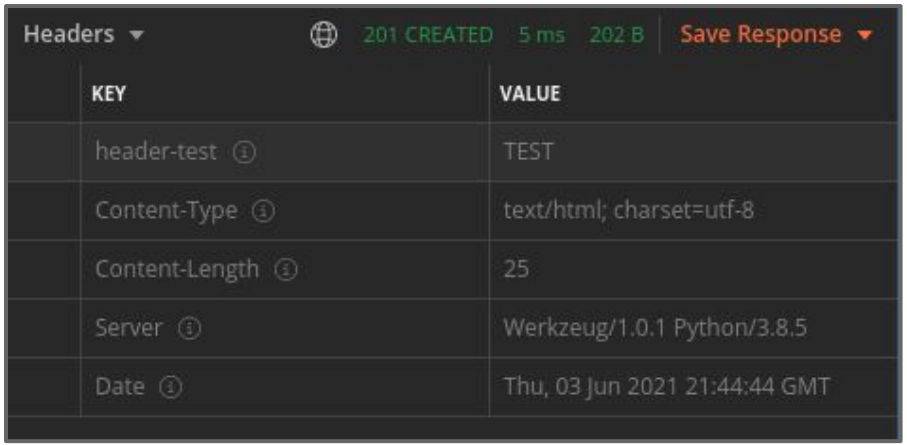

```
@app.route('/')
def index():
    return ({'error': "Invalid username"},
400)
```

Example: Response object

You can set status code, header, content-type, ... to **Response** instance.

```
@app.route('/')
def index():
    resp = Response("<h1>Response Object!</h1>", 201, headers={'h-test': 'TEST'})
    return resp
```





Templates

Intro

As you see, You can prepare a HTML response to clients from Flask view functions.
So you can response .html files content to clients.
Commonly .html files called templates. You can store your HTML files into specific Directory (Default: templates), then render the contents to clients as a HTML response.

Set **App** templates directory:

```
# Default: templates
app = Flask(__name__, template_folder='templates')
```

render_template(...) utility function

render_template(file_name, **context)

Hint: file_name should be into **templates directory.**

```
from flask import Flask, render_template

app = Flask(__name__, template_folder='templates')

@app.route('/')
def index():
    resp = render_template('index.html')
    print(resp)
    return resp
```

```
<body>
<h1>Hello world!</h1>
</body>
```

Pass data to templates

render_template(file_name, **context)

Hint: pass keyword parameters to HTML template.

```python
from flask import Flask, render_template

app = Flask(__name__, template_folder='templates')


@app.route('/<string:fname>')
def index(fname):
    resp = render_template('index.html', name=fname)
    return resp
```

```
<body>
<h1>Hello {{ name }}</h1>
</body>
```

Example

```
class User:
    def   init   (self, fname, lname, phone, email):
        self.fname = fname
        self.lname = lname
        self.phone = phone
        self.email = email
```

```
@app.route('/')
def index():
    user = User('akbar', 'babaii', '09123456789', 'akbar@gmail.com')
    resp = render_template('index.html', user=vars(user))
    return resp
```

```
<div>
    <p><b>Firstname:</b> {{ user.fname }}</p>
    <p><b>Lastname:</b> {{ user.lname }}</p>
    <p><b>Phone:</b> {{ user.phone }}</p>
    <p><b>Email:</b> {{ user.email }}</p>
</div>
```

Firstname: akbar

Lastname: babaii

Phone: 09123456789

Email: akbar@gmail.com

Jinja

Intro

A Jinja template is simply a text file. Jinja can generate any text-based format (HTML, XML, CSV, LaTeX, etc.). A Jinja template doesn't need to have a specific extension: .html, .xml, or any other extension is just fine.

A template contains variables and/or expressions, which get replaced with values when a template is rendered; and tags, which control the logic of the template. The template syntax is heavily inspired by Django and Python.

Syntax:

- **{% ... %}** for Statements
- **{{ ... }}** for Expressions to print to the template output
- **{# ... #}** for Comments not included in the template output

Variables

Template variables are defined by the context dictionary passed to the template.

You can mess around with the variables in templates provided they are passed in by the application. Variables may have attributes or elements on them you can access too. What attributes a variable has depends heavily on the application providing that variable.

You can use a dot (.) to access attributes of a variable in addition to the standard Python __getitem__ “subscript” syntax ([]).

The following lines do the same thing:

```
{{ foo.bar }}
{{ foo['bar'] }}
```

Variables

Template variables are defined by the context dictionary passed to the template.

You can mess around with the variables in templates provided they are passed in by the application. Variables may have attributes or elements on them you can access too. What attributes a variable has depends heavily on the application providing that variable.

You can use a dot (.) to access attributes of a variable in addition to the standard Python __getitem__ “subscript” syntax ([]).

The following lines do the same thing:

```
{{ foo.bar }}
{{ foo['bar'] }}
```

Filters

Variables can be modified by filters. Filters are separated from the variable by a pipe symbol (|) and may have optional arguments in parentheses. Multiple filters can be chained. The output of one filter is applied to the next.
For example, **{{ name|striptags|title }}** will remove all HTML Tags from variable name and title-case the output**.(title(striptags(name))).**

Filters that accept arguments have parentheses around the arguments, just like a function call. For example: **{{ listx|join(', ') }}** will join a list with commas **(str.join(', ', listx))**.

List of Built-in Filters

Statements

A control structure refers to all those things that control the flow of a program - conditionals (i.e. if/elif/else), for-loops, as well as things like macros and blocks. With the default syntax, control structures appear inside **{% ... %}** blocks.

Jinja Statements:

- **If**
- **for**
- **set**
- **macro**
- **More Jinja Statements...**

Statements: if

The if statement in Jinja is comparable with the Python if statement. In the simplest form, you can use it to test if a variable is defined, not empty and not false:

For multiple branches, elif and else can be used like in Python. You can use more complex Expressions there, too:

```
{% if kenny.sick %}
   Kenny is sick.
{% elif kenny.dead %}
   You killed Kenny!  You bastard!!!
{% else %}
   Kenny looks okay --- so far
{% endif %}
```

Statements: for

Loop over each item in a sequence. For example, to display a list of users provided in a variable called users:

```
<h1>Members</h1>
<ul>
    {% for user in users %}
        <li>{{ user.username|e }}</li>
    {% endfor %}
</ul>
```

As variables in templates retain their object properties, it is possible to iterate over containers like dict:

```
<dl>
    {% for key, value in my dict.items() %}
        <dt>{{ key|e }}</dt>
        <dd>{{ value|e }}</dd>
    {% endfor %}
</dl>
```

Routing

HTTP methods

By default, the Flask route responds to the GET requests. However, this preference can be altered by providing methods argument to route() decorator.

```
@app.route('/')
def get_view():
   return "get request!"


@app.route('/', methods=['POST'])
def post_view():
   return "POST request!"


@app.route('/', methods=['PATCH', 'PUT'])
def patch_put_view():
   return "PATCH or PUT request!"
```

```
curl -X get http://127.0.0.1:5000/
get request!
```

```
curl -X post http://127.0.0.1:5000/
POST request!
```

```
curl -X patch http://127.0.0.1:5000/
PATCH or PUT request!
```

```
curl -X put http://127.0.0.1:5000/
PATCH or PUT request!
```

Practice: User registration page

Maktab Sharif

User registration page

First create an User class with attributes like id, first_name, last_name, phone, email, ...
Then create a Flask app:

Pages:

A. **GET /:**
   -> *Response*: Registration HTML form.
B. **POST /:**
   Create User instance from incoming data
   -> *Response*: Empty or User information
C. **GET /<int:id>:**
   -> *Response*: User #id information
D. **GET /list:**
   -> *Response*: Summary List of Users

For next session

- Jinja Filters
- Jinja Statements
- Jinja Blocks
- Session in Flask
- Cookies in Flask
- Statics in Flask