EXPERIMENT NO. 14

Study experiment on implementing a Basic Flask Application to build a Simple REST API.

Roll No.	01
Name	Aamir Ansari
Class	D10A
Subject	Python Lab
LO Mapped	LO1: Understand the structure, syntax, and semantics of the Python language. LO6: Design and Develop cost-effective robust applications using the latest Python trends and technologies.

Aim: Study experiment on implementing a Basic Flask Application to build a Simple REST API.

Introduction:

Web Framework: Web Application Framework or simply Web Framework represents a collection of libraries and modules that enables a web application developer to write applications without having to bother about low-level details such as protocols, thread management etc.

Flask: Flask is a web application framework written in Python. It is developed by Armin Ronacher, who leads an international group of Python enthusiasts named Pocco. Flask is based on the Werkzeug WSGI toolkit and Jinja2 template engine. Both are Pocco projects.

WSGI: Web Server Gateway Interface (WSGI) has been adopted as a standard for Python web application development. WSGI is a specification for a universal interface between the web server and the web applications.

Werkzeug: It is a WSGI toolkit, which implements requests, response objects, and other utility functions. This enables building a web framework on top of it. The Flask framework uses Werkzeug as one of its bases.

Jinja2: Jinja2 is a popular templating engine for Python. A web templating system combines a template with a certain data source to render dynamic web pages.

REST API: REST stands for REpresentational State Transfer and is an architectural style used in modern web development. It defines a set or rules/constraints for a web application to send and receive data.

Flask is a popular micro framework for building web applications. Since it is a micro-framework, it is very easy to use and lacks most of the advanced functionality which is found in a full-fledged framework. Therefore, building a REST API in Flask is very simple. There are two ways of creating a REST API in Flask:

1. **Using Flask without any external libraries**: Here, there are two functions: One function to just return or print the data sent through GET or POST and another function to calculate the square of a number sent through GET request and print it.

2. **Using flask_restful library**: In flask_restful, the main building block is a resource. Each resource can have several methods associated with it such as GET, POST, PUT, DELETE, etc. Each resource is a class that inherits from the Resource class of

flask_restful. Once the resource is created and defined, we can add our custom resource to the api and specify a URL path for that corresponding resource.

Flask Environment Setup: We can now install the flask by using the following command.

```
$ pip install flask
```

However, we can install the flask using the above command without creating the virtual environment. To test the flask installation, open python on the command line and type python to open the python shell.

```
from flask import Flask
app = Flask(__name__)
@app.route('/')
def hello_world():
        return 'Hello World'
if __name__ == '__main__':
        app.run()
```

Importing the flask module in the project is mandatory. An object of Flask class is our WSGI application. Flask constructor takes the name of the current module (__name__) as argument. The **route()** function of the Flask class is a decorator, which tells the application which URL should call the associated function.

```
app.route(rule, options)
```

The rule parameter represents URL binding with the function. The options is a list of parameters to be forwarded to the underlying Rule object. In the above example, '/' URL is bound with **hello_world()** function. Hence, when the home page of a web server is opened in the browser, the output of this function will be rendered. Finally the **run()** method of Flask class runs the application on the local development server.

```
app.run(host, port, debug, options)
```

A message in Python shell informs you that

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

Open the above URL (localhost:5000) in the browser. ‘Hello World’ message will be displayed on it. All parameters are optional.

1. **Host**: Hostname to listen on. Defaults to 127.0.0.1 (localhost). Set to ‘0.0.0.0’ to have server available externally
2. **Port**: Defaults to 5000
3. **Debug**: Defaults to false. If set to true, provides a debug information
4. **Options**: To be forwarded to the underlying Werkzeug server.

Debug mode: A Flask application is started by calling the **run()** method. However, while the application is under development, it should be restarted manually for each change in the code. To avoid this inconvenience, enable debug support. The server will then reload itself if the code changes. It will also provide a useful debugger to track the errors if any, in the application.

```
app.debug = True
app.run()
app.run(debug = True)
```

The Debug mode is enabled by setting the debug property of the application object to True before running or passing the debug parameter to the **run()** method.

Flask Routing: App routing is used to map the specific URL with the associated function that is intended to perform some task. It is used to access some particular page like Flask Tutorial in the web application. The **route()** decorator in Flask is used to bind URL to a function.

```
@app.route(‘/hello’)
def hello_world():
    return ‘hello world’
```

Here, the URL /hello rule is bound to the hello_world() function. As a result, if a user visits http://localhost:5000/hello URL, the output of the hello_world() function will be rendered in the browser.

```
def hello_world():
    return ‘hello world’
app.add_url_rule(‘/’, ‘hello’, hello_world)
```

The **add_url_rule()** function of an application object is also available to bind a URL with a function as in the above example, **route()** is used.

Flask Variable Names: It is possible to build a URL dynamically, by adding variable parts to the rule parameter. This variable part is marked as <variable-name>. It is passed as a keyword argument to the function with which the rule is associated.

```
@app.route('/hello/<name>')
def hello_name(name):
    return 'Hello %s!' % name
```

In the following example, the rule parameter of **route()** decorator contains <name> variable part attached to URL **/hello**. Hence, if the **http://localhost:5000/hello/HelloWorld** is entered as a URL in the browser, 'HelloWorld' will be supplied to the **hello()** function as an argument.

Flask URL Building: The **url_for()** function is very useful for dynamically building a URL for a specific function. The function accepts the name of a function as first argument, and one or more keyword arguments, each corresponding to the variable part of URL. The following script demonstrates use of **url_for()** function.

```
@app.route('/guest/<guest>')
def hello_guest(guest):
    return 'Hello %s as Guest' % guest

@app.route('/user/<name>')
def hello_user(name):
    if name =='admin':
        return redirect(url_for('hello_admin'))
    else:
        return redirect(url_for('hello_guest',guest = name))
```

The above script has a function user(name) which accepts a value to its argument from the URL. The User() function checks if an argument received matches 'admin' or not. If it matches, the application is redirected to the hello_admin() function using url_for(), otherwise to the hello_guest() function passing the received argument as guest parameter to it.

Flask HTTP methods: HTTP is the hypertext transfer protocol which is considered as the foundation of the data transfer in the world wide web. All web frameworks including flask need to provide several HTTP methods for data communication. The methods are given below

1. **GET:** Sends data in unencrypted form to the server. Most common method.
2. **HEAD:** Same as GET, but without response body

3. **POST:** Used to send HTML form data to the server. Data received by POST method is not cached by the server.
4. **PUT:** Replaces all current representations of the target resource with the uploaded content.
5. **DELETE:** Removes all current representations of the target resource given by a URL

We can specify which HTTP method to be used to handle the requests in the **route()** function of the Flask class. By default, the requests are handled by the **GET()** method. To handle the POST requests at the server, let us first create a form to get some data at the client side from the user, and we will try to access this data on the server by using the POST request.

```
@app.route('/login',methods = ['POST', 'GET'])
def login():
    if request.method == 'POST':
        user = request.form['nm']
        return redirect(url_for('success',name = user))
    else:
        user = request.args.get('nm')
        return redirect(url_for('success',name = user))
```

After the development server starts running, open login.html in the browser, enter name in the text field and click Submit.

Flask Templates: Generating HTML content from Python code is cumbersome, especially when variable data and Python language elements like conditionals or loops need to be put. This would require frequent escaping from HTML. This is where one can take advantage of Jinja2 template engine, on which Flask is based. Instead of returning hardcode HTML from the function, a HTML file can be rendered by the **render_template()** function.

```
@app.route('/')
def index():
    return render_template('hello.html')
```

Flask will try to find the HTML file in the templates folder, in the same folder in which this script is present. The term 'web templating system' refers to designing an HTML script in which the variable data can be inserted dynamically. A web template system comprises a template engine, some kind of data source and a template processor.

```
<!doctype html>
```

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

Flask uses a jinja2 template engine. A web template contains HTML syntax interspersed placeholders for variables and expressions (in this case Python expressions) which are replaced values when the template is rendered.

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

The jinja2 template engine uses the following delimiters for escaping from HTML.

1. {% ... %} for Statements
2. {{ ... }} for Expressions to print to the template output
3. {# ... #} for Comments not included in the template output
4. # ... ## for Line Statements

In the following example, use of conditional statements in the template is demonstrated. The URL rule to the **hello()** function accepts the integer parameter. It is passed to the hello.html template. Inside it, the value of number received (marks) is compared (greater or less than 50) and accordingly HTML is conditionally rendered.

Flask Static files: A web application often requires a static file such as a javascript file or a CSS file supporting the display of a web page. Usually, the web server is configured to serve them for you, but during the development, these files are served from the static folder in your package or next to your module and it will be available at /static on the application.

```
<html>
    <head>
        <script type = "text/javascript"
        src = "{{ url_for('static', filename = 'hello.js') }}" ></script>
    </head>
    <body>
    <input type = "button" onclick = "sayHello()" value = "Say Hello" />
    </body>
</html>
```

A special endpoint **static** is used to generate URL for static files. In the following example, a javascript function defined in hello.js is called on the OnClick event of the HTML button in index.html, which is rendered on '/' URL of the Flask application.

Flask Object: The data from a client's web page is sent to the server as a global request object. In order to process the request data, it should be imported from the Flask module. Important attributes of request object are listed below:

1. **Form**: It is a dictionary object containing key and value pairs of form parameters and their values.
2. **Args**: parsed contents of query string which is part of URL after question mark (?).
3. **Cookies**: dictionary object holding Cookie names and values.
4. **Files**: data pertaining to uploaded file.
5. **Method**: current request method.

Flask Form data to Template: The Form data received by the triggered function can collect it in the form of a dictionary object and forward it to a template to render it on a corresponding web page.

```
@app.route('/result',methods = ['POST', 'GET'])
def result():
    if request.method == 'POST':
        result = request.form
        return render_template("result.html",result = result)
```

In the following example, '/' URL renders a web page (student.html) which has a form. The data filled in it is posted to the **/result** URL which triggers the **result()** function.

```
<!doctype html>
<html>
    <body>
        <table border = 1>
            {% for key, value in result.items() %}
                    <tr>
                    <th> {{ key }} </th>
                    <td> {{ value }} </td>
                    </tr>
            {% endfor %}
        </table>
    </body>
</html>
```

The **result()** function collects form data present in request.form in a dictionary object and sends it for rendering to result.html. The template dynamically renders an HTML table of form data.

Flask Cookie: A cookie is stored on a client's computer in the form of a text file. A Request object contains a cookie's attribute. It is a dictionary object of all the cookie variables and their corresponding values a client has transmitted. In addition to it, a cookie also stores its expiry time, path and domain name of the site.

In Flask, cookies are set on response objects. Use **make_response()** function to get the response object from the return value of a view function. After that, use the **set_cookie()** function of the response object to store a cookie. Reading back a cookie is easy. The **get()** method of request.cookies attribute is used to read a cookie.

```
    <html>
        <body>
            <form action = "/setcookie" method = "POST">
                <p><h3>Enter userID</h3></p>
                <p><input type = 'text' name = 'nm'/></p>
                <p><input type = 'submit' value = 'Login'/></p>
            </form>
        </body>
    </html>
```

The Form is posted to **/setcookie** URL. The associated view function sets a Cookie name userID and renders another page.

```
    @app.route('/setcookie', methods = ['POST', 'GET'])
    def setcookie():
        if request.method == 'POST':
            user = request.form['nm']

        resp = make_response(render_template('readcookie.html'))
        resp.set_cookie('userID', user)

        return resp
```

Readcookie.html contains a hyperlink to another view function **getcookie()**, which reads back and displays the cookie value in the browser.

```
@app.route('/getcookie')
def getcookie():
    name = request.cookies.get('userID')
    return '<h1>welcome '+name+'</h1>'
```

Flask Session: Session is the time interval when a client logs into a server and logs out of it. The data, which is needed to be held across this session, is stored in the client browser. A session with each client is assigned a Session ID. To set a ‘username’ session variable use the statement

```
session[‘username’] = ’admin’
```

To release a session variable use **pop()** method.

```
session.pop('username', None)
```

The Session data is stored on top of cookies and the server signs them cryptographically. For this encryption, a Flask application needs a defined SECRET_KEY. Session object is also a dictionary object containing key-value pairs of session variables and associated values.

Flask Redirect and Errors: Flask class has a **redirect()** function. When called, it returns a response object and redirects the user to another target location with specific status code. Prototype of **redirect()** function is as below

```
Flask.redirect(location, statuscode, response)
```

In the above function, location parameter is the URL where response should be redirected. Status Code is sent to the browser's header, defaults to 302. Response parameter is used to instantiate response. The following status codes are standardized:

1. HTTP_300_MULTIPLE_CHOICES
2. HTTP_301_MOVED_PERMANENTLY
3. HTTP_302_FOUND
4. HTTP_303_SEE_OTHER
5. HTTP_304_NOT_MODIFIED
6. HTTP_305_USE_PROXY
7. HTTP_306_RESERVED
8. HTTP_307_TEMPORARY_REDIRECT

Flask class has abort() function with an error code.

    Flask.abort(code)

The Code parameter takes one of following values:

1. 400 − for Bad Request
2. 401 − for Unauthenticated
3. 403 − for Forbidden
4. 404 − for Not Found
5. 406 − for Not Acceptable
6. 415 − for Unsupported Media Type
7. 429 − Too Many Requests

Flask Message flashing: Flashing system of Flask framework makes it possible to create a message in one view and render it in a view function called next. A Flask module contains **flash()** method. It passes a message to the next request, which generally is a template.

    flash(message, category)

Here message parameter is the actual message to be flashed and category parameter is optional. It can be either 'error', 'info' or 'warning'. In order to remove messages from the session, template calls **get_flashed_messages().**

get_flashed_messages(with_categories, category_filter)

Both parameters are optional. The first parameter is a tuple if received messages have a category. The second parameter is useful to display only specific messages.

Flask File uploading: Handling file upload in Flask is very easy. It needs an HTML form with its enctype attribute set to **multipart/form-data**, posting the file to a URL. The URL handler fetches file from **request.files[]** object and saves it to the desired location.

Each uploaded file is first saved in a temporary location on the server, before it is actually saved to its ultimate location. Name of destination file can be hard-coded or can be obtained from filename property of **request.files[file]** object. However, it is recommended to obtain a secure version of it using the **secure_filename()** function.

```
<html>
```

```
    <body>
        <form action = "http://localhost:5000/uploader" method = "POST"
        enctype = "multipart/form-data">
            <input type = "file" name = "file" />
            <input type = "submit"/>
        </form>
    </body>
</html>
```

Click Submit after choosing the file. Form's post method invokes **/upload_file** URL. The underlying function **uploader()** does the save operation. Following is the Python code of the Flask application.

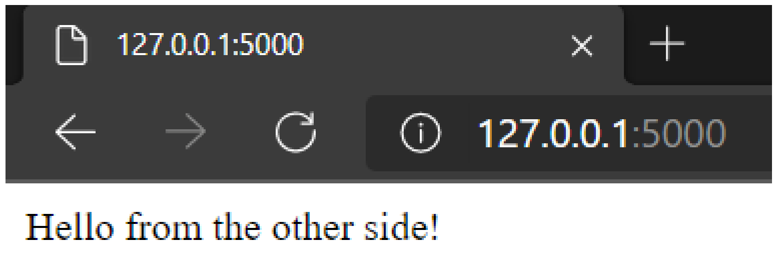
```
@app.route('/uploader', methods = ['GET', 'POST'])
def upload_file():
    if request.method == 'POST':
        f = request.files['file']
        f.save(secure_filename(f.filename))
        return 'file uploaded successfully'
```

Results:

1. **Code**:

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def hello():
    return 'Hello from the other side!'

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

Output:



```
Hello from the other side!
```

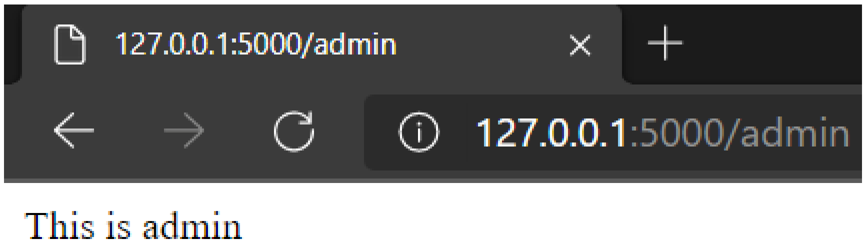
2. **Code**:

File: app.py

```
from flask import *

app = Flask(__name__)

@app.route('/')
def home():
   return 'Hello from the other side!'

@app.route('/admin')
def admin():
   return 'This is admin'

@app.route('/staff')
def staff():
   return 'This is staff'

@app.route('/student')
def student():
   return 'This is student'

@app.route('/user/<name>')
def user(name):
   if name == 'admin':
      return redirect(url_for('admin'))
   if name == 'librarion':
      return redirect(url_for('staff'))
```

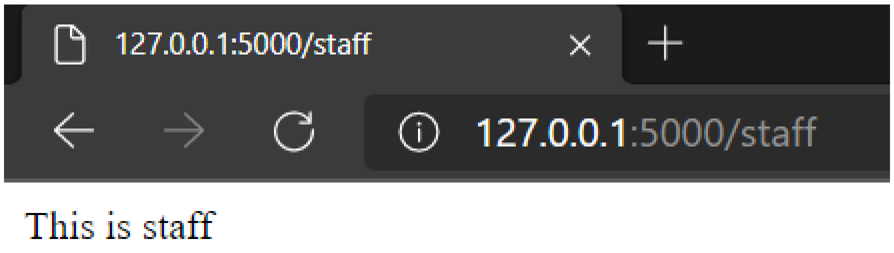
```
        if name == 'student':
            return redirect(url_for('student'))

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

Output:


127.0.0.1:5000/admin

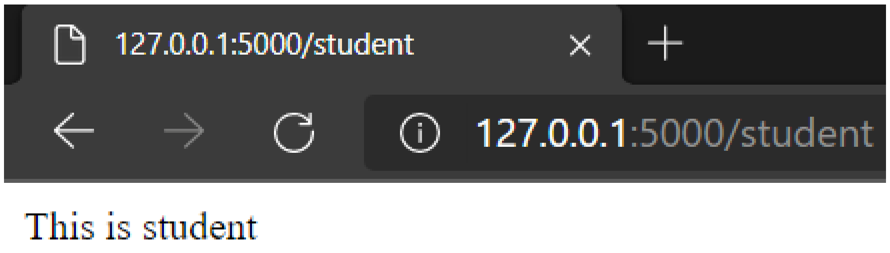
127.0.0.1:5000/admin

This is admin


127.0.0.1:5000/staff

127.0.0.1:5000/staff

This is staff


127.0.0.1:5000/student

127.0.0.1:5000/student

This is student

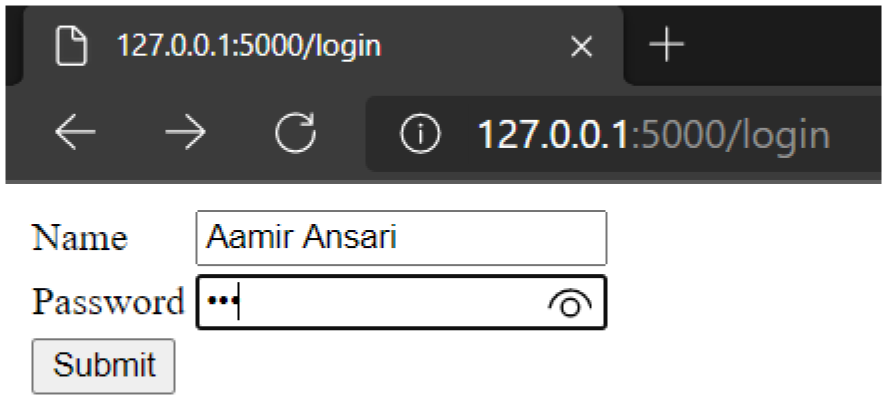
3. **Code**:
   # Adding the following code in **app.py**

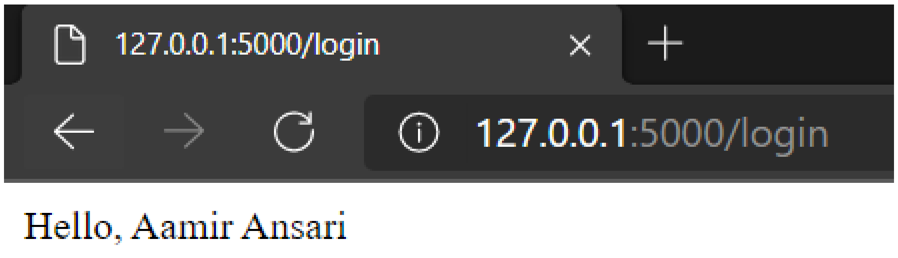
```
@app.route('/login',methods = ['GET','POST'])
def login():
   if (request.method == "POST"):
      userName=request.form['name']
      password=request.form['pass']
      if userName=="Aamir Ansari" and password=="123":
         return ("Hello, %s" %(userName))
   return render_template("login.html", title="Login")
```

File: **login.html**

```
<html>
        <body>
                <form action = "{{url_for('login')}}" method = "post">
                <table>
                        <tr><td>Name</td>
                        <td><input type ="text" name ="name"></td></tr>
                        <tr><td>Password</td>
                        <td><input type ="password" name ="pass"></td></tr>
                        <tr><td><input type = "submit"></td></tr>
                </table>
                </form>
          </body>
</html>
```

Output:

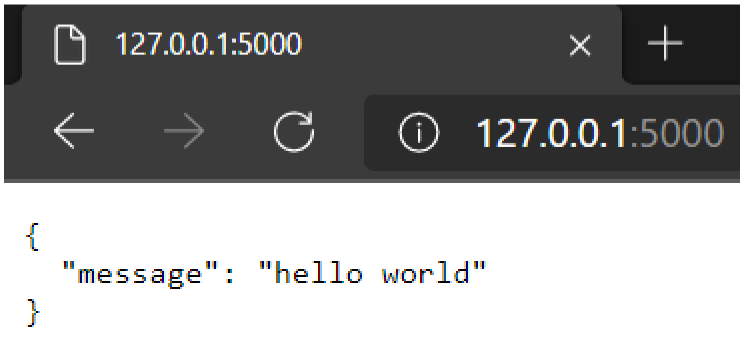



127.0.0.1:5000/login

Hello, Aamir Ansari

4. **Code**:

```
from flask import Flask, jsonify, request
from flask_restful import Resource, Api

app = Flask(__name__)
api = Api(app)

class Hello(Resource):
        def get(self):
                return jsonify({'message': 'hello world'})
        def post(self):
                data = request.get_json()
                return jsonify({'data': data}), 201

class Square(Resource):
        def get(self, num):
                return jsonify({'square': num**2})

api.add_resource(Hello, '/')
api.add_resource(Square, '/square/<int:num>')

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



```
{
  "message": "hello world"
}
```

Conclusion: Hence we have successfully learned and understood the basic concepts of web development in Python using FLASKframework. We learned to create routes, use GET and POST requests. We also learned how to use REST API using FLASK