Experiment 14

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

Roll No.	61
Name	V Krishnasubramaniam
Class	D10-A
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:

Introduction to Flask

Flask is a web framework that provides libraries to build lightweight web applications in python. Flask is considered as a micro framework. It aims to keep the core of an application simple yet extensible. Flask does not have built-in abstraction layer for database handling, nor does it have formed a validation support. Instead, Flask supports the extensions to add such functionality to the application.

It is developed by Armin Ronacher who leads an international group of python enthusiasts (POCCO). It is based on WSGI toolkit and jinja2 template engine. WSGI is an acronym for web server gateway interface which is a standard for python web application development. It is considered as the specification for the universal interface between the web server and web application.

Flask Environment Setup

We can install the flask by using the following command:

```
pip install flask
```

To test the flask installation, open python on the command line and type python to open the python shell. Try to import the package flask:

```
import flask
```

Basic Flask Application

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

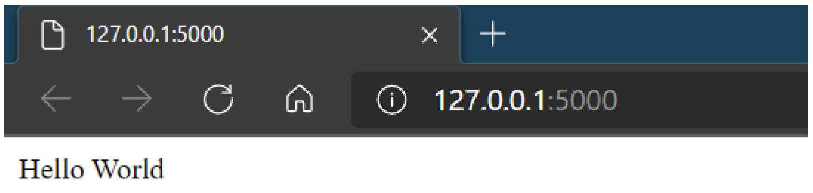
The above given Python script is executed from Python shell.

```
python app.py
```

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
```

Importing flask module in the project is mandatory. An object of Flask class is our WSGI application.

Flask constructor takes the name of 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 are 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 web server is opened in 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)

All parameters are optional

Parameters	**Description**
host	Hostname to listen on. Defaults to 127.0.0.1 (localhost). Set to '0.0.0.0' to have server available externally
port	Defaults to 5000
debug	Defaults to false. If set to true, provides a debug information
options	To be forwarded to 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.

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.

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

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 in the web app.

Modern web frameworks use the routing technique to help a user remember application URLs. It is useful to access the desired page directly without having to navigate from the home page.

The route() decorator in Flask is used to bind URL to a function.

Example:

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

Here, 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.

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. A decorator's purpose is also served by the following too.

Example:

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

Parameters in Routing:

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.

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/Debuggers is entered as a URL in the browser, 'Debuggers' will be supplied to hello() function as argument.

Example:

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

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

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

Save the above script as hello.py and run it from Python shell. Next, open the browser and enter URL http://localhost:5000/hello/Debuggers.

The following output will be displayed in the browser.

```
Hello Debuggers!
```

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 in the following table.

SN	Method	Description
1	GET	It is the most common method which can be used to send data in the unencrypted form to the server.
2	HEAD	It is similar to the GET but used without the response body.
3	POST	It is used to send the form data to the server. The server does not cache the data transmitted using the post method.
4	PUT	It is used to replace all the current representation of the target resource with the uploaded content.
5	DELETE	It is used to delete all the current representation of the target resource specified in the 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.

POST 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.

Flask Templates

It is possible to return the output of a function bound to a certain URL in the form of HTML. For instance, in the following script, hello() function will render ‘Hello World’ with <h1> tag attached to it.

Example:
```
@app.route('/')
def index():
        return '<html><body><h1>Hello World</h1></body></html>'
```

However, 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.

Example:
```
from flask import Flask
app = Flask(__name__)
@app.route('/')
def index():
  return render_template(‘hello.html’)
if __name__ == '__main__':
  app.run(debug = True)
```

Flask will try to find the HTML file in the templates folder, in the same folder in which this script is present.

```
Application folder
|-        Hello.py
|-        templates
          |-        hello.html
```

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 of a template engine, some kind of data source and a template processor.

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

The following code is saved as hello.html in the templates folder.

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

Next, run the following script from Python shell.

```
from flask import Flask, render_template
app = Flask(__name__)
@app.route('/hello/<user>')
def hello_name(user):
  return render_template('hello.html', name = user)
if __name__ == '__main__':
  app.run(debug = True)
```

As the development server starts running, open the browser and enter URL as −
http://localhost:5000/hello/debuggers

The variable part of URL is inserted at {{ name }} place holder. Hence the output on the webpage will be:
Hello Debuggers!

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 statement 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.

Example:

```
from flask import Flask, render_template
app = Flask(__name__)
@app.route('/hello/<int:score>')
def hello_name(score):
  return render_template('hello.html', marks = score)
if __name__ == '__main__':
  app.run(debug = True)
```

HTML template script of hello.html is as follows −

```
<!doctype html>
<html>
```

```
  <body>
    {% if marks>50 %}
      <h1> Your result is pass!</h1>
    {% else %}
      <h1>Your result is fail</h1>
    {% endif %}
  </body>
</html>
```

Note that the conditional statements if-else and endif are enclosed in delimiter {%..%}.

Run the Python script and visit URL http://localhost/hello/60 and then http://localhost/hello/30 to see the output of HTML changing conditionally.

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 static folder in your package or next to your module and it will be available at /static on the application.

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 OnClick event of HTML button in index.html, which is rendered on '/' URL of the Flask application.

```
from flask import Flask, render_template
app = Flask(__name__)
@app.route("/")
def index():
  return render_template("index.html")
if __name__ == '__main__':
  app.run(debug = True)
```

The HTML script of index.html is given below.

```
<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>
```

hello.js contains sayHello() function.

```
function sayHello() {
  alert("Hello World")
}
```

Request 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.

We have already seen that the http method can be specified in URL rule. 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.

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.

The results() 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.

Given below is the Python code of application −

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

@app.route('/')
def student():
  return render_template('student.html')

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

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

Given below is the HTML script of student.html.

```
<html>
  <body>
    <form action = "http://localhost:5000/result" method = "POST">
      <p>Name <input type = "text" name = "Name" /></p>
      <p>Physics <input type = "text" name = "Physics" /></p>
      <p>Chemistry <input type = "text" name = "chemistry" /></p>
      <p>Maths <input type ="text" name = "Mathematics" /></p>
      <p><input type = "submit" value = "submit" /></p>
    </form>
  </body>
</html>
```

Code of template (result.html) is given below −

```
<!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>
```

Cookies

A cookie is stored on a client's computer in the form of a text file. Its purpose is to remember and track data pertaining to a client's usage for better visitor experience and site statistics.

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 object. Use make_response() function to get response object from return value of a view function. After that, use the set_cookie() function of 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.

In the following Flask application, a simple form opens up as you visit '/' URL.

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

This HTML page contains one text input.

```
<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 browser.

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

Sessions

Like Cookie, Session data is stored on client. 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. 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.

For example, 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 following code is a simple demonstration of session works in Flask. URL ‘/’ simply prompts user to log in, as session variable ‘username’ is not set.

```
@app.route('/')
def index():
  if 'username' in session:
    username = session['username']
      return 'Logged in as ' + username + '<br>' + \
      "<b><a href = '/logout'>click here to log out</a></b>"
  return "You are not logged in <br><a href = '/login'></b>" + \
    "click here to log in</b></a>"
```

As user browses to ‘/login’ the login() view function, because it is called through GET method, opens up a login form.

A Form is posted back to ‘/login’ and now session variable is set. Application is redirected to ‘/’. This time session variable ‘username’ is found.

```
@app.route('/login', methods = ['GET', 'POST'])

def login():
  if request.method == 'POST':
    session['username'] = request.form['username']
    return redirect(url_for('index'))
  return '''
  <form action = "" method = "post">
    <p><input type = text name = username/></p>
    <p<<input type = submit value = Login/></p>
  </form>
  '''
```

The application also contains a logout() view function, which pops out ‘username’ session variable. Hence, ‘/’ URL again shows the opening page.

```
@app.route('/logout')
def logout():
  # remove the username from the session if it is there
  session.pop('username', None)
  return redirect(url_for('index'))
```

Run the application and visit the homepage. (Ensure to set secret_key of the application)

```
from flask import Flask, session, redirect, url_for, escape, request
app = Flask(__name__)
app.secret_key = 'any random string’
```

REST API using Flask

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.

REST API is a way of accessing the web services in a simple and flexible way without having any processing. REST technology is generally preferred to the more robust Simple Object Access Protocol (SOAP) technology because REST uses the less bandwidth, simple and flexible making it more suitable for internet usage. It’s used to fetch or give some information from a web services. All communication done via REST API used only HTTP request.

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
2. Using flask_restful library

flask_restful can be installed via the pip command: pip install flask_restful

Example:
```
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)
```

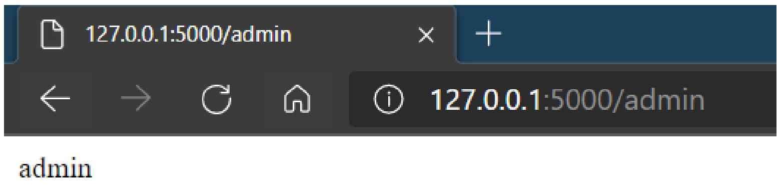
Results:

Routing with parameters:

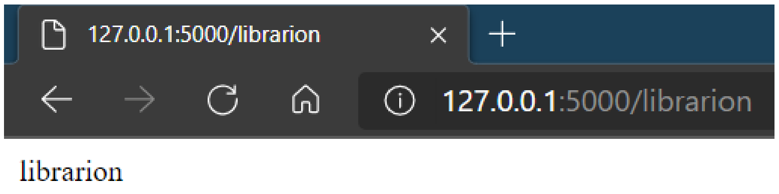
Code in run.py:

```
from flask import *

app = Flask(__name__)

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

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

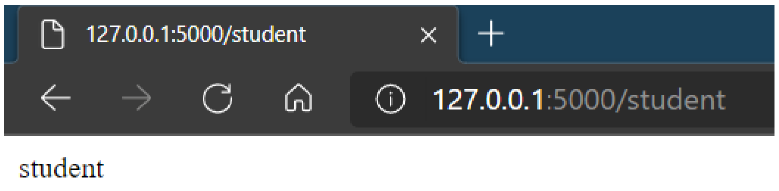
Output:



```
admin
```



librarion



student

POST HTTP method in Flask:

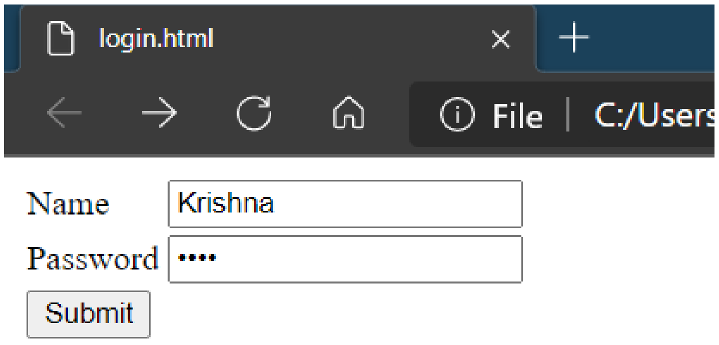
Code in run.py:

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

@app.route('/login',methods = ['POST'])
def login():
      uname=request.form['uname']
      passwrd=request.form['pass']
      if uname=="Krishna" and passwrd=="pass":
          return "Welcome %s" %uname

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

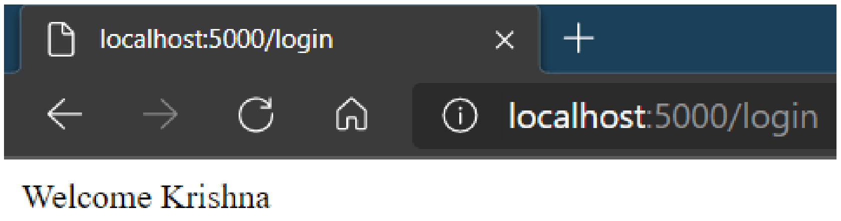
Code in login.html:

```
<html>
   <body>
      <form action = "http://localhost:5000/login" method = "post">
         <table>
        <tr><td>Name</td>
        <td><input type ="text" name ="uname"></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:



Name Krishna
Password ••••
Submit



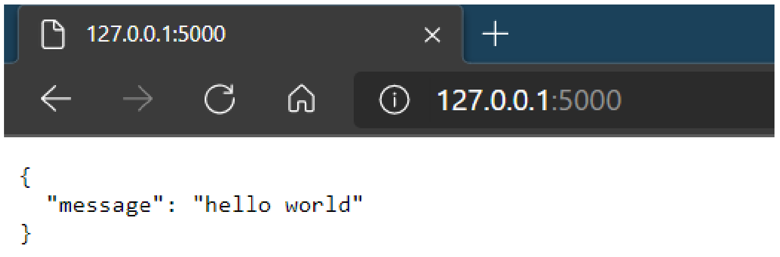
Welcome Krishna

Basic REST API app using Flask:

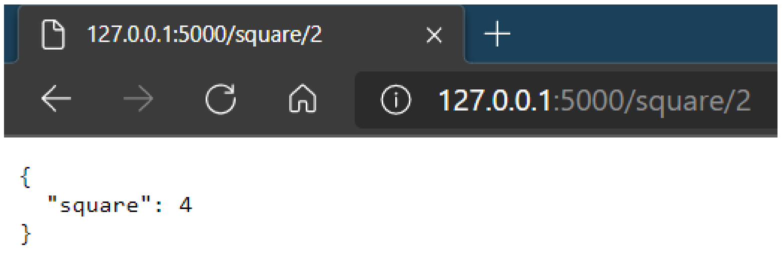
Code in run.py:

```
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)
```

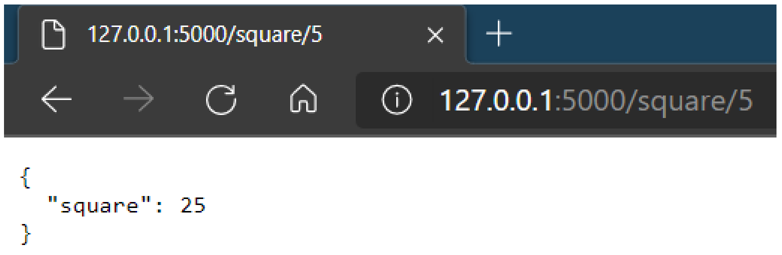
Output:



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



```
{
  "square": 4
}
```



```
{
  "square": 25
}
```

Conclusion:

Thus, we have understood the basics of web development in Python using flask framework. We learnt various ways to create routes, use HTTP methods, use templates of HTML to create modular apps in Flask, and learnt ways to use REST API in Flask by performing hands-on practical programs for each.