

# UNIT 5: PYTHON WEB FRAMEWORK: FLASK

## INSTALLATION OF FLASK AND ENVIRONMENT SETUP

1. Create a new folder
2. Rightclick open with vscode
3. Go to terminal and create virtual environment

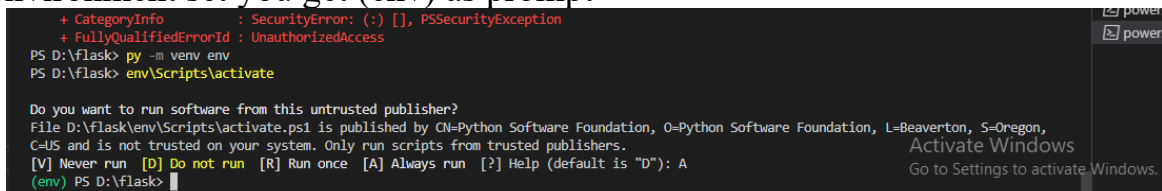
> pip install virtualenv

>virtualenv env

[isolate programs from others,because it not effect on other's environment]

If you get error like “PermissionError: [WinError 5] Access is denied: 'D:\\flask\\env’”, right click on folder and change permission from properties.

4. For Activate the environment-> env\\Scripts\\activate
  - If u get any problem to create virtual environment run-> Set-ExecutionPolicy unrestricted and Set → “A”
  - Otherwise run ->Set-ExecutionPolicy -Scope CurrentUser -ExecutionPolicy Unrestricted on vscode terminal.
  - If still you get error follow command “py –m venv env” and then “env\\Scripts\\activate”
  - If environment set you get (env) as prompt



```
+ CategoryInfo          : SecurityError: (:) [], PSSecurityException
+ FullyQualifiedErrorId : UnauthorizedAccess

PS D:\flask> py -m venv env
PS D:\flask> env\Scripts\activate

Do you want to run software from this untrusted publisher?
File D:\flask\env\Scripts\activate.ps1 is published by CN=Python Software Foundation, O=Python Software Foundation, L=Beaverton, S=Oregon, C=US and is not trusted on your system. Only run scripts from trusted publishers.
[V] Never run [D] Do not run [R] Run once [A] Always run [?] Help (default is "D"): A
(env) PS D:\flask>
```

5. Install flask-> pip install flask

Create a python file

```
from flask import Flask

app = Flask(__name__)
@app.route("/")
def hello_world():
    return "<p>Hello, World!</p>"

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

6. run→Flask run
7. To exit from virtual environment give command->”deactivate” on terminal

Explanation:

1. First we imported the **Flask** class. An instance of this class will be our WSGI application.
2. Next we create an instance of this class. The first argument is the name of the application’s module or package. `__name__` is a convenient shortcut for this that is appropriate for most cases. This is needed so that Flask knows where to look for resources such as templates and static files.
3. We then use the **route()** decorator to tell Flask what URL should trigger our function.

4. The function returns the message we want to display in the user's browser. The default content type is HTML, so HTML in the string will be rendered by the browser.

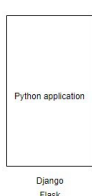
Save it as `hello.py` or something similar. Make sure to not call your application `flask.py` because this would conflict with Flask itself.

## WSGI

**WSGI**(Web Server Gateway Interface) is a specification that describes the communication between **web servers** and **Python web applications or frameworks**. It explains how a web server communicates with python web applications/frameworks and applications/frameworks can be chained for processing a request.

### HOW DOES WSGI WORK?

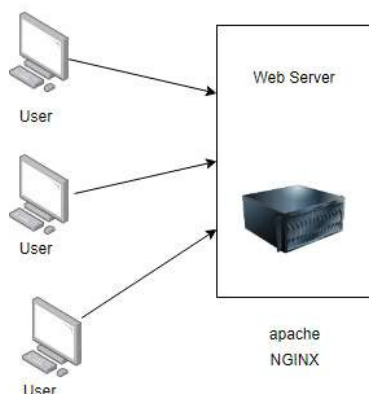
Now, let's have a look at how WSGI work. So, to obtain a clear understanding of WSGI, let us assume a case scenario where you have a web application developed in Django or Flask application as shown in the figure



The above figure represents your Django/Flask application. Since a web application is deployed in the web server. The figure below represents the web server that obtains requests from various users.

The web server which obtains requests from the users.

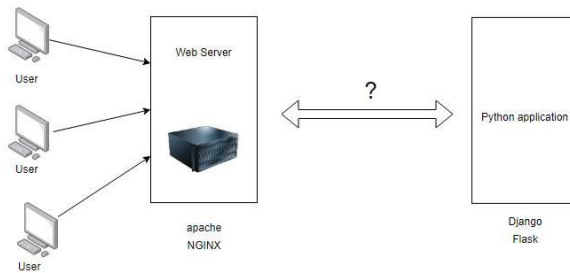
The above web server can be apache, NGINX, etc. server which is responsible for handling various static files and caching purposes. Furthermore,



you can also use the server as a load balancer if you are willing to scale multiple applications.

### HOW CAN A WEB SERVER INTERACT WITH THE PYTHON APPLICATION?

The figure representing problem in the interaction between Python application and a web server.



So, now a problem arises as a web server has to interact with a Python application.

Hence, a mediator is required for carrying out the interaction between the web servers and the Python application. So, the standard for carrying out communication between the web server and Python application is WSGI(Web Server Gateway Interface).

Now, web server is able to send requests or communicate with WSGI containers. Likewise, Python application provides a 'callable' object which contains certain functionalities that are invoked by WSGI application which are defined as per the PEP 3333 standard(pyton gateway interface standand). Hence, there are multiple WSGI containers available such as Gunicorn(Green Unicorn' is a Python WSGI HTTP Server for UNIX), uWSGI, etc.

The figure below represents the communication that is carried out between web server, WSGI, and Python application.

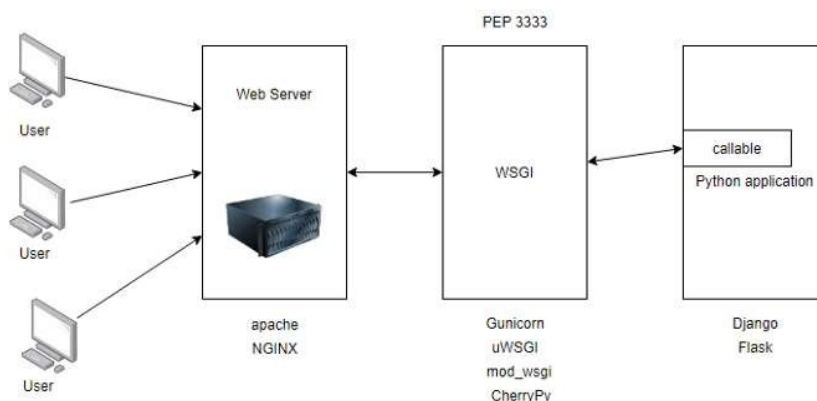
---

#### COMMUNICATION BETWEEN USER, WEB SERVER, WSGI, AND PYTHON APPLICATION.

There are multiple WSGI containers that are available today. Hence, a WSGI container is required to be installed in the project so that a web server can communicate to a WSGI container which further communicates to the Python application and provides the response back accordingly. Finally, when the web server obtains the response, it is sent back to the web browser/users.

---

#### WHY USE THE WSGI RATHER THAN DIRECTLY POINTING THE WEB SERVER TO THE DJANGO OR FLASK



#### APPLICATION?

If you directly point your web server to your application, it reduces the **flexibility** of your application. Since your web server now directly

points to your web application, you are unable to swap out web stack components. Now, let's have a look at an example to make you clear about the applicability of WSGI. For instance, today you have decided to deploy your application using Gunicorn but after some years you decide to switch from Gunicorn to `mod_wsgi`. Now, in this case, you can easily switch to `mod_wsgi` without making any changes in the application or framework that implements WSGI. Hence, WSGI provides flexibility to your application.

Another reason for using WSGI is due to its **scalability**. Once your application is live, up and running there can be thousands of requests in your application. Hence, WSGI is capable of serving thousands of requests at a time. As we know, the WSGI server is responsible for handling the requests from the web server and takes decision for carrying out the communication of those requests to an application framework's process. Here, we can divide the responsibilities among the servers for scaling web traffic.

## WEB TEMPLATE ENGINE (JINJA2)

Jinja2 is a modern day templating language for Python developers. It was made after Django's template. It is used to create HTML, XML or other markup formats that are returned to the user via an HTTP request.

Jinja is a fast, expressive, extensible templating engine. Special placeholders in the template allow writing code similar to Python syntax. Then the template is passed data to render the final document.

It includes:

- Template inheritance and inclusion.
- Define and import macros within templates.
- HTML templates can use autoescaping to prevent XSS from untrusted user input.
- A sandboxed environment can safely render untrusted templates.
- Async support for generating templates that automatically handle sync and async functions without extra syntax.
- I18N support with Babel.
- Templates are compiled to optimized Python code just-in-time and cached, or can be compiled ahead-of-time.
- Exceptions point to the correct line in templates to make debugging easier.
- Extensible filters, tests, functions, and even syntax.

Jinja's philosophy is that while application logic belongs in Python if possible, it shouldn't make the template designer's job difficult by restricting functionality too much.

## FOLDER STRUCTURE FOR A FLASK APP

A proper Flask app is going to use multiple files — some of which will be template files. The organization of these files has to follow rules so the app will work. Here is a diagram of the typical structure:

```
my-flask-app
├── static/
└── └── css/
```

```

|   └─ main.css
└─ templates/
    |   └─ index.html
    |   └─ student.html
└─ data.py
└─ students.py

```

1. Everything the app needs is in one folder, here named MY-FLASK-APP.
2. That folder contains two folders, specifically named STATIC and TEMPLATES.
  - The STATIC folder contains **assets** used by the templates, including CSS files, JavaScript files, and images. In the example, we have only one asset file, MAIN.CSS. Note that it's inside a CSS folder that's inside the STATIC folder.
  - The TEMPLATES folder contains only templates. These have an **.html** extension. As we will see, they contain more than just regular HTML.
3. In addition to the STATIC and TEMPLATES folders, this app also contains **.py** files. Note that these must be OUTSIDE the two folders named STATIC and TEMPLATES.

#### CREATING THE FLASK CLASS OBJECT

After import flask library file.

Create object using `app=Flask(__name__)`

#### CREATING AND HOSTING FIRST BASIC FLASK APP.

```

from flask import Flask

app = Flask(__name__)
@app.route("/")
def hello_world():
    return "<p>Hello, World!</p>"

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

```

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** 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 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

Sr.No.	Parameters & Description
1	<b>Host</b> Hostname to listen on. Defaults to 127.0.0.1 (localhost). Set to ‘0.0.0.0’ to have server available externally
2	<b>Port</b> Defaults to 5000, we can change by passing parameter port=value
3	<b>Debug</b> Defaults to false. If set to true, provides a debug information
4	<b>Options</b> To be forwarded to underlying Werkzeug server.

The above given **Python** script is executed from Python shell.

Python Hello.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**’ message will be displayed on it.

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

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. For 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 representation –

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

## FLASK – VARIABLE RULES(APP 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/TutorialsPoint** is entered as a URL in the browser, **'TutorialPoint'** will be supplied to **hello()** function as argument.

```
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/TutorialsPoint**.

The following output will be displayed in the browser.

Hello TutorialPoint!

In addition to the default string variable part, rules can be constructed using the following converters –

Sr.No.	Converters & Description
1	<b>Int</b> - accepts integer
2	<b>Float</b> - For floating point value
3	<b>Path</b> - accepts slashes used as directory separator character

In the following code, all these constructors are used.

```
from flask import Flask  
app = Flask(__name__)  
  
@app.route('/blog/<int:postID>')  
def show_blog(postID):  
    return 'Blog Number %d' % postID
```



```
@app.route('/rev/<float:revNo>')
def revision(revNo):
    return 'Revision Number %f' % revNo

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

Run the above code from Python Shell. Visit the URL **http://localhost:5000/blog/11** in the browser.

The given number is used as argument to the **show\_blog()** function. The browser displays the following output –

Blog Number 11

## FLASK – URL BUILDING

### THE ADD\_URL\_RULE() FUNCTION

There is one more approach to perform routing for the flask web application that can be done by using the `add_url()` function of the Flask class. The syntax to use this function is given below.

```
add_url_rule(<url rule>, <endpoint>, <view function>)
```

This function is mainly used in the case if the view function is not given and we need to connect a view function to an endpoint externally by using this function.

Example:

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

def about():
    return "This is about page";

app.add_url_rule("/about","about",about)

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

### THE URL\_FOR() FUNCTION

It 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.

```
from flask import Flask, redirect, url_for
app = Flask(__name__)

@app.route('/admin')
def hello_admin():
    return 'Hello Admin'
```



```

@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))

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

```

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.

Save the above code and run from Python shell.

Open the browser and enter URL as – **http://localhost:5000/user/admin**

The application response in browser is –

Hello Admin

Enter the following URL in the browser – **http://localhost:5000/user/ami**

The application response now changes to –

Hello ami as Guest

## FLASK – HTTP METHODS

Http protocol is the foundation of data communication in world wide web. Different methods of data retrieval from specified URL are defined in this protocol.

The following table summarizes different http methods –

Sr.No.	Methods & Description
1	<b>GET</b> -Sends data in unencrypted form to the server. Most common method.
2	<b>HEAD</b> -Same as GET, but without response body
3	<b>POST</b> -Used to send HTML form data to server. Data received by POST method is not cached by server.
4	<b>PUT</b> -Replaces all current representations of the target resource with the uploaded content.
5	<b>DELETE</b> -Removes all current representations of the target resource given by a URL

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

In order to demonstrate the use of **POST** method in URL routing, first let us create an HTML form and use the **POST** method to send form data to a URL.

Save the following script as login.html

```
<html>
  <body>
    <form action = "http://localhost:5000/login" method = "post">
      <p>Enter Name:</p>
      <p><input type = "text" name = "nm" /></p>
      <p><input type = "submit" value = "submit" /></p>
    </form>
  </body>
</html>
```

Now enter the following script in Python shell.

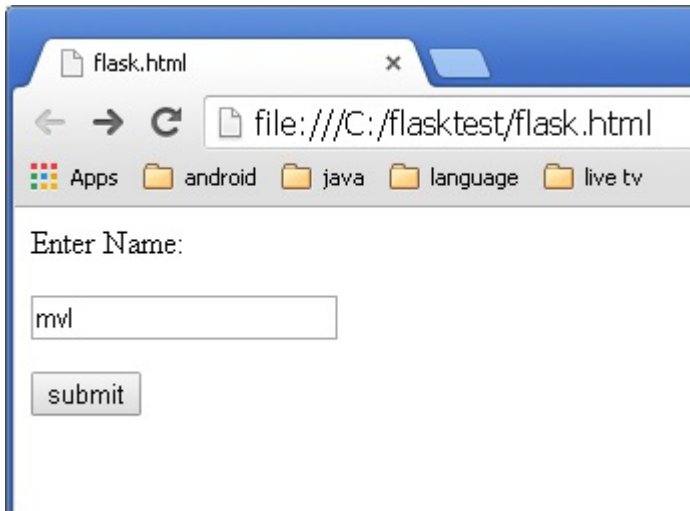
```
from flask import Flask, redirect, url_for, request
app = Flask(__name__)

@app.route('/success/<name>')
def success(name):
    return 'welcome %s' % name

@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))

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

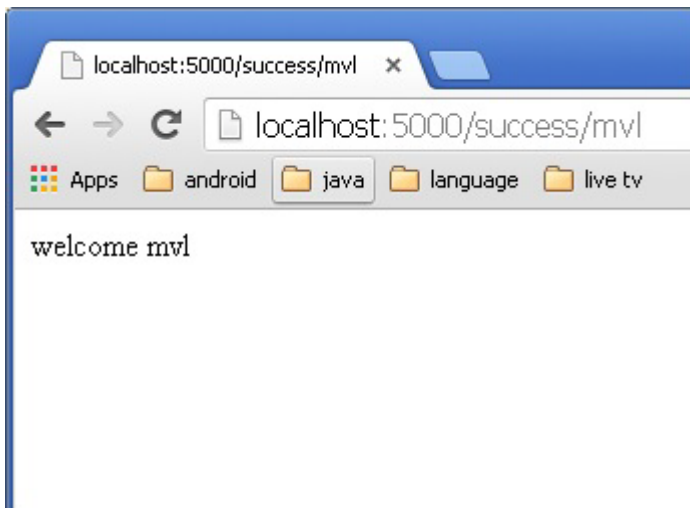
After the development server starts running, open **login.html** in the browser, enter name in the text field and click **Submit**.



Form data is POSTed to the URL in action clause of form tag.

**http://localhost/login** is mapped to the **login()** function. Since the server has received data by **POST** method, value of 'nm' parameter obtained from the form data is obtained by –  
`user = request.form['nm']`

It is passed to **'/success'** URL as variable part. The browser displays a **welcome** message in the window.



Change the method parameter to **'GET'** in **login.html** and open it again in the browser. The data received on server is by the **GET** method. The value of 'nm' parameter is now obtained by –

`User = request.args.get('nm')`

Here, **args** is dictionary object containing a list of pairs of form parameter and its corresponding value. The value corresponding to 'nm' parameter is passed on to **'/success'** URL as before.

## FLASK – TEMPLATES

(Embedding Python statement in HTML )

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.

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

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

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

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 hardcoded HTML from the function, a HTML file can be rendered by the **render\_template()** function.

```
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/mvl**

The **variable** part of URL is inserted at **{{ name }}** place holder.



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

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

The Python Script is as follows –

```
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.

The Python loop constructs can also be employed inside the template. In the following script, the **result()** function sends a dictionary object to template **results.html** when URL **`http://localhost:5000/result`** is opened in the browser.

The Template part of **result.html** employs a **for loop** to render key and value pairs of dictionary object **result** as cells of an HTML table.

Run the following code from Python shell.

```

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

@app.route('/result')
def result():
    dict = {'phy':50,'che':60,'maths':70}
    return render_template('result.html', result = dict)

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

```

Save the following HTML script as **result.html** in the templates folder.

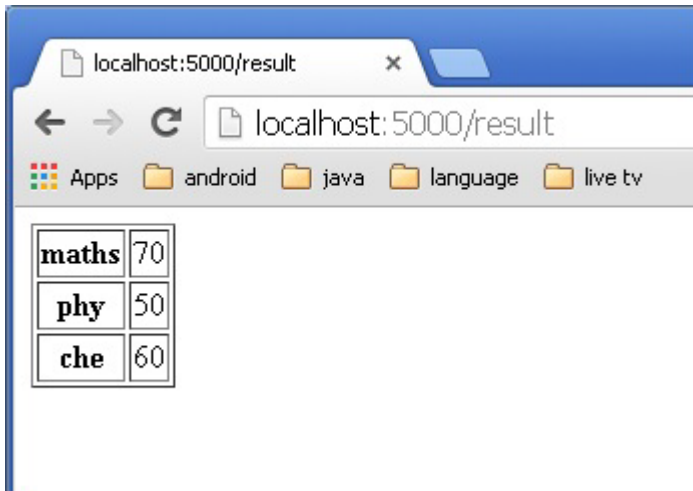
```

<!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>

```

Here, again the Python statements corresponding to the **For** loop are enclosed in `{%..%}` whereas, the expressions **key** and **value** are put inside `{{ }}`.

After the development starts running, open **`http://localhost:5000/result`** in the browser to get the following output.



## 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")
}
```



Flask request object: (Form, args, files, redirect)

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 –

- **Form** – It is a dictionary object containing key and value pairs of form parameters and their values.
- **args** – parsed contents of query string which is part of URL after question mark (?).
- **Cookies** – dictionary object holding Cookie names and values.
- **files** – data pertaining to uploaded file.
- **method** – current request method.

Form request object, render\_template() method, Form data Handling

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**.

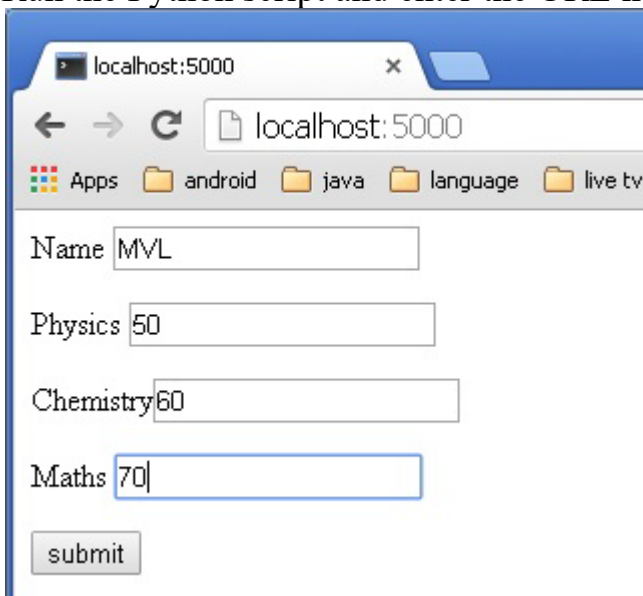
```
<html>
<body>
    <form action = "http://localhost:5000/result" method = "POST">
        <p>Name <input type = "text" name = "Name" /></p>
        <p>Physics <input type = "text" name = "Physics" /></p>
        <p>Chemistry <input type = "text" name = "chemistry" /></p>
```

```
<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>
```

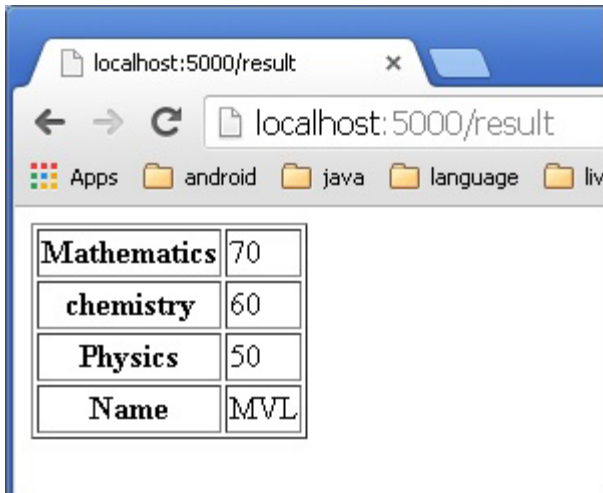
Run the Python script and enter the URL **http://localhost:5000/** in the browser.



The screenshot shows a web browser window with the address bar set to 'localhost:5000'. The browser's tab is labeled 'localhost:5000'. Below the address bar, there are navigation buttons (back, forward, refresh) and a search bar. A folder bar below the search bar contains icons for 'Apps', 'android', 'java', 'language', and 'live tv'. The main content area of the browser displays a form with the following elements:

- A text input field labeled 'Name' containing the text 'MVL'.
- A text input field labeled 'Physics' containing the text '50'.
- A text input field labeled 'Chemistry' containing the text '60'.
- A text input field labeled 'Maths' containing the text '70'.
- A button labeled 'submit' at the bottom left of the form.

When the **Submit** button is clicked, form data is rendered on **result.html** in the form of HTML table.



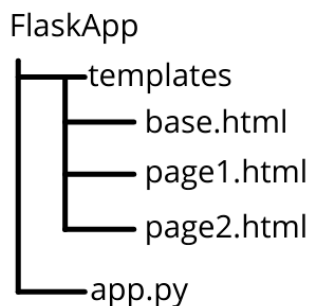
## USING FLASK TEMPLATES FOR DYNAMIC DATA

We have already discussed using templates in flask for storing the non-changing data, but we can also use them dynamically to show data using [Jinja](#). Jinja is used to write python-like syntax in HTML files, which helps in using variables like functionality. In other words, we can make dynamic templates also.

We will use them in a small flask application to get better understanding of how we use them.

## FILE STRUCTURE

The file structure will look like the image given below.



## FLASK APPLICATION

### CREATING APP.PY

Our app.py file consists of only two routes with their templates and a value to pass to that particular template. Down below is code, for the reference.

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

```

@app.route("/")
def index():
    data = "This is the body of homepage"
    return render_template("page1.html",content = data)

@app.route("/page2")
def page2():
    data = ['A','B','C']
    return render_template("page2.html",content = data)

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

```

## CREATING TEMPLATES

Now creating a folder **templates** and inside it creating three HTML files, BASE.HTML, PAGE1.HTML and PAGE2.HTML. Firstly adding some code to BASE.HTML, as the name suggests it's base file, that means it will be included in the another HTML file using `{% extends "base.html" %}`.

One thing to note that is we use that syntax in the file which we have to include something. Following code is for BASE.HTML.

```

<html>
<head><title> This is created by base.html template </title></head>
<body>
    <h2>Commom Part of both pages</h2>
    <br>
    <h3>Changed Part :</h3>
    <div>
        =><a href = '{{ url_for("index") }}'>Index Page</a><br>
        =><a href = '{{ url_for("page2") }}'>Page 2</a>
        {% block page_content %}
        {% endblock %}
    </div>
</body>
</html>

```

Now adding the code for other files, PAGE1.HTML and PAGE2.HTML.

```

<!-- page1.html -->
{% extends "base.html" %}

{% block page_content %}
<p> This is from "index.html"</p>
    {{ content }}
{% endblock %}
<!-- page2.html -->
{% extends "base.html" %}

{% block page_content %}

```

```

<p> This is from "page2.html"</p>
{% for i in content %}
    data - {{ i }} <br>
{% endfor %}

{% endblock %}

```

Let's discuss what we have done in another two files, firstly PAGE1.HTML, at starting we used this `{% extends "base.html" %}` to add base.html in page1.html. Then, we created a block named **page\_content**. Inside it put the value **content** which we pass when we render the template, then we close the block we created. In this case **content** is a string. Now, you will notice we have done the same with **page2.html**. But all that's changed is instead of passing an string in `RENDER_TEMPLATE()`. This time we passed a list, so to access them individually we use JINJA'S **for** loop.



## FLASK SESSION, CREATING SESSION, SESSION VARIABLE, SESSION.POP()

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'))

```

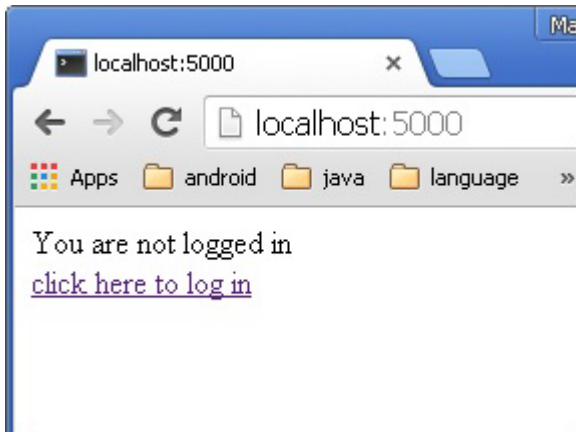
Run the application and visit the homepage. (Ensure to set **secret\_key** of the application)

```

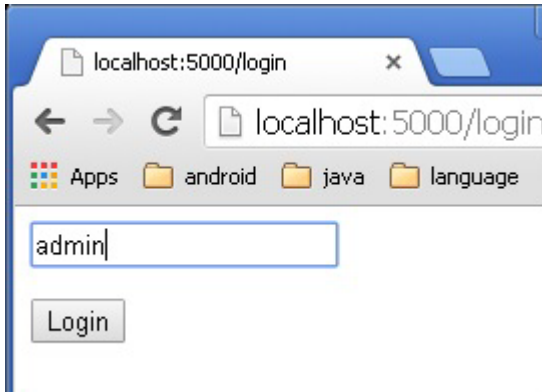
from flask import Flask, session, redirect, url_for, escape, request
app = Flask(__name__)
app.secret_key = 'any random string'

```

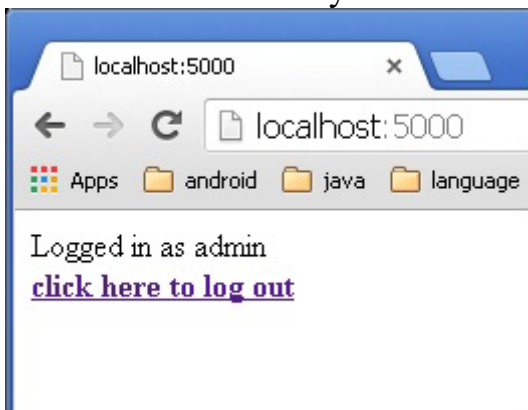
The output will be displayed as shown below. Click the link “**click here to log in**”.



The link will be directed to another screen. Type 'admin'.



The screen will show you the message, '**Logged in as admin**'.



FILE UPLOADING: REQUEST.FILES[] OBJECT, SAVE() METHOD, SAVING FILE TO SPECIFIC FOLDER.

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.

It is possible to define the path of default upload folder and maximum size of uploaded file in configuration settings of Flask object.

app.config['UPLOAD_FOLDER']	Defines path for upload folder
app.config['MAX_CONTENT_PATH']	Specifies maximum size of file yo be

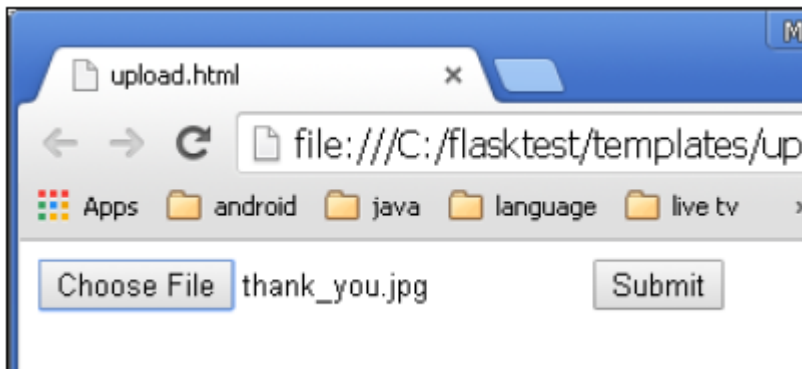


uploaded – in bytes

The following code has **‘/upload’** URL rule that displays **‘upload.html’** from the templates folder, and **‘/upload-file’** URL rule that calls **uploader()** function handling upload process. **‘upload.html’** has a file chooser button and a submit button.

```
<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>
```

You will see the screen as shown below.



Click **Submit** after choosing file. Form's post method invokes **‘/upload\_file’** URL. The underlying function **uploader()** does the save operation.

Following is the Python code of Flask application.

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

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

@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'

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

REDIRECTING : REDIRECT() METHOD, LOCATION, STATUS CODE AND RESPONSE.

Flask class has a **redirect()** function. When called, it returns a response object and redirects the user to another target location with specified 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.
- **statuscode** sent to browser's header, defaults to 302.
- **response** parameter is used to instantiate response.

The following status codes are standardized –

- HTTP\_300\_MULTIPLE\_CHOICES
- HTTP\_301\_MOVED\_PERMANENTLY
- HTTP\_302\_FOUND
- HTTP\_303\_SEE\_OTHER
- HTTP\_304\_NOT\_MODIFIED
- HTTP\_305\_USE\_PROXY
- HTTP\_306\_RESERVED
- HTTP\_307\_TEMPORARY\_REDIRECT

The **default status** code is **302**, which is for '**found**'.

In the following example, the **redirect()** function is used to display the login page again when a login attempt fails.

```
from flask import Flask, redirect, url_for, render_template, request
# Initialize the Flask application
app = Flask(__name__)

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

@app.route('/login', methods = ['POST', 'GET'])
def login():
    if request.method == 'POST' and request.form['username'] == 'admin' :
        return redirect(url_for('success'))
    else:
        return redirect(url_for('index'))

@app.route('/success')
def success():
    return 'logged in successfully'

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

FLASK CLASS HAS **ABORT()** FUNCTION WITH AN ERROR CODE.

Flask.abort(code)

The **Code** parameter takes one of following values –

- **400** – for Bad Request

- **401** – for Unauthenticated
- **403** – for Forbidden
- **404** – for Not Found
- **406** – for Not Acceptable
- **415** – for Unsupported Media Type
- **429** – Too Many Requests

Let us make a slight change in the **login()** function in the above code. Instead of re-displaying the login page, if ‘**Unauthourized**’ page is to be displayed, replace it with call to **abort(401)**.

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

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

@app.route('/login', methods = ['POST', 'GET'])
def login():
    if request.method == 'POST':
        if request.form['username'] == 'admin' :
            return redirect(url_for('success'))
        else:
            abort(401)
    else:
        return redirect(url_for('index'))

@app.route('/success')
def success():
    return 'logged in successfully'

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