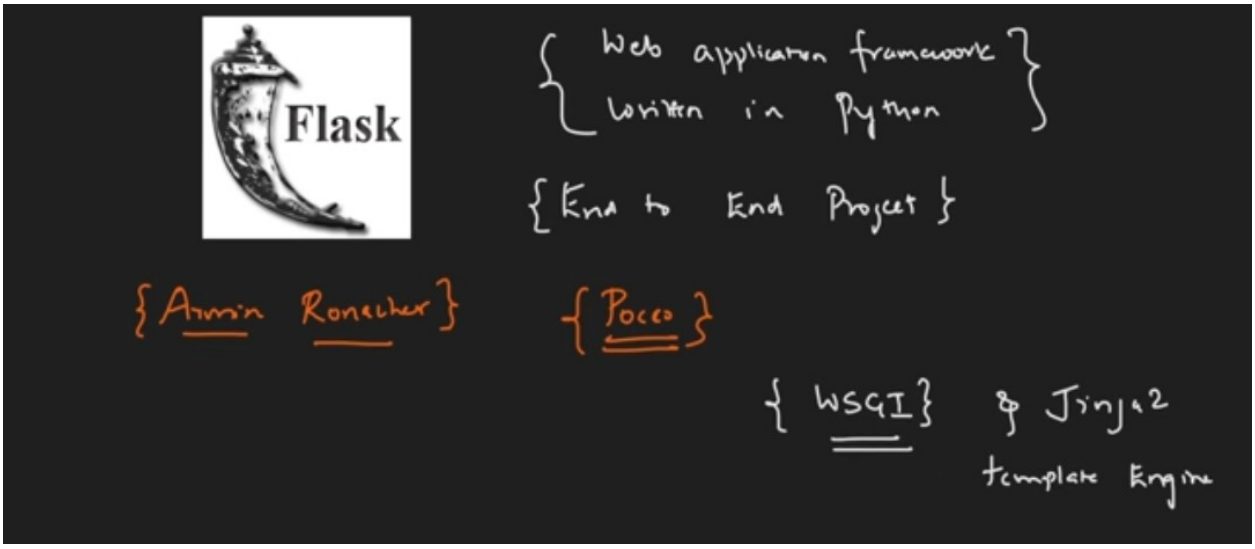
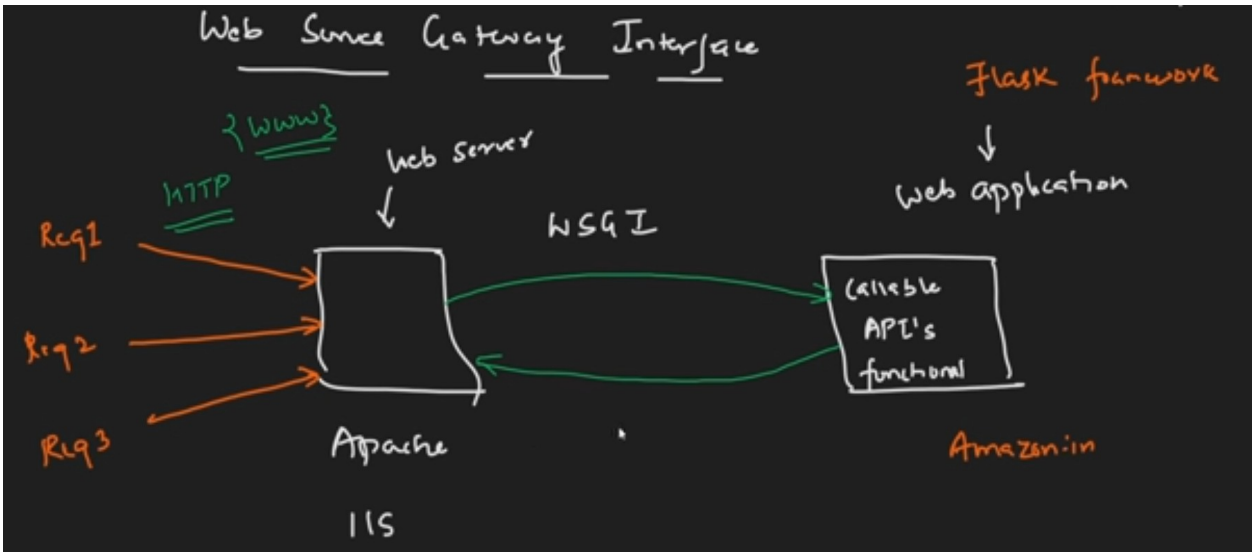
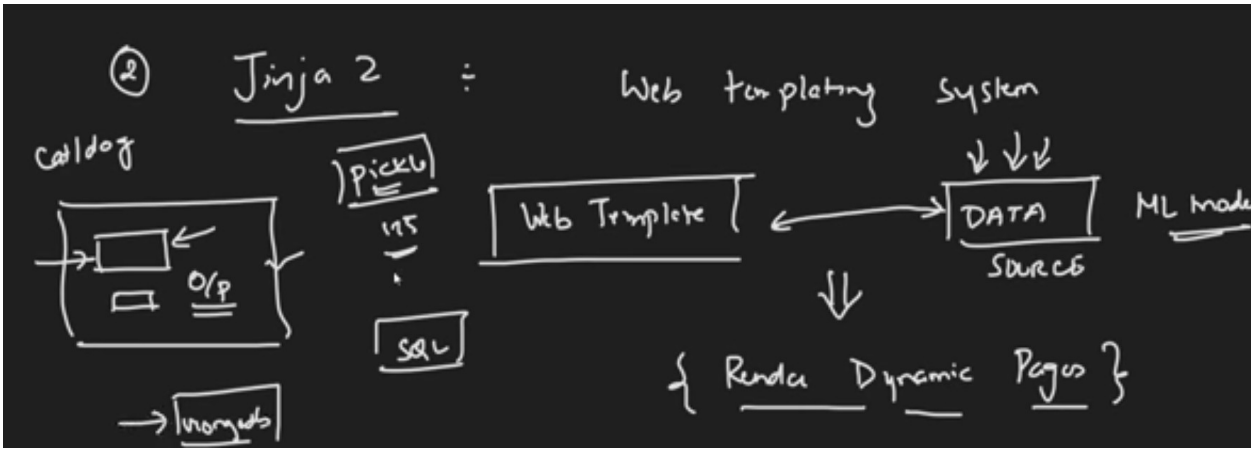
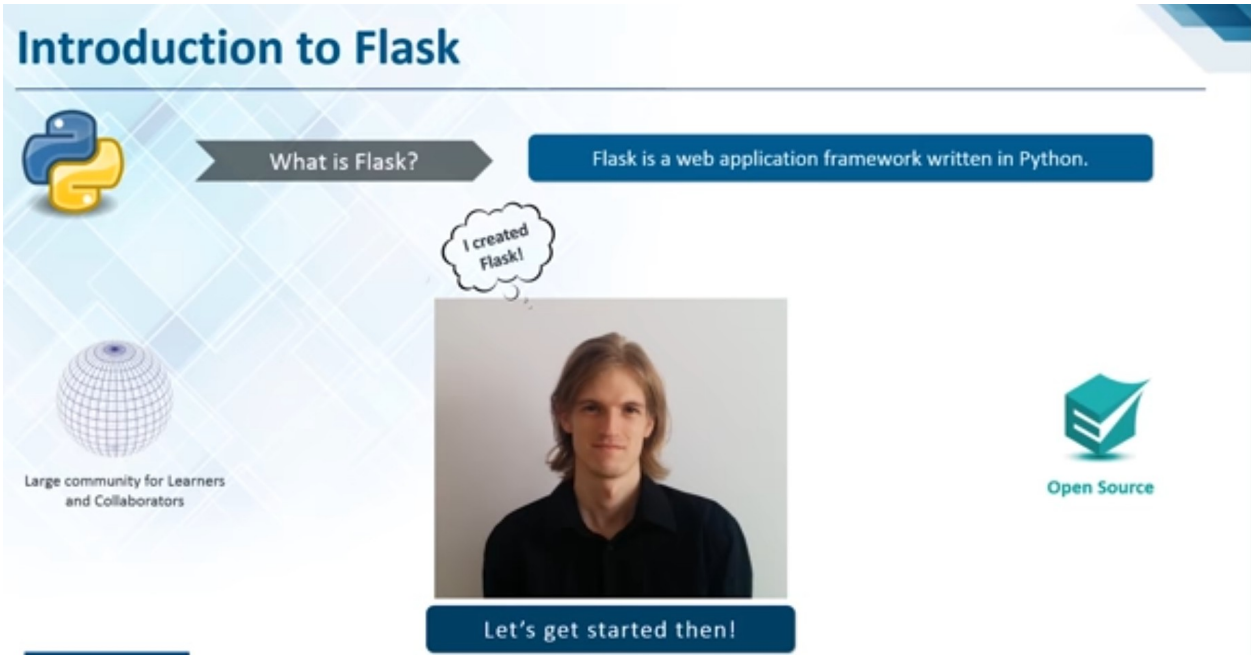

Flask
{ Web application framework
written in Python }
{ End to End Project }
{ Armin Ronacher }
{ Pocco }
{ WSGI } & Jinja2
template Engine


Web Server Gateway Interface
Flask framework
{ www }
Web server
Web application
HTTP
Req1
Req2
Req3
WSGI
callable
API's
functional
Apache
Amazon:in
115


② Jinja 2 : Web templating System
Calldog
Pickle
125
O/p
SQL
mongodb
Web Template
DATA
SOURCE
ML mode
{ Render Dynamic Pages }

Introduction to Python - Flask



Introduction to Flask

Flask!!

python powered

Enthusiasts named Pocco!

Werkzeug WSGI Toolkit

Jinga2 Template Engine

Installation - Prerequisite

Prerequisite

virtualenv → Virtual Python Environment builder

Windows

```
pip install virtualenv
```

ubuntu

```
Sudo apt-get install virtualenv
```

Installation - Flask

Installation

Once installed, new virtual environment is created in a folder

```
mkdir newproj
cd newproj
virtualenv venv
```

To activate corresponding environment, use the following:

Windows

```
venv\scripts\activate
```

```
pip install Flask
```

Flask - Application

Test Installation

Use this simple code, save it as Hello.py

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

@app.route('/')
def hello_world():
   return 'Hello World'

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

Flask - Application

Importing flask module in the project is mandatory!

Flask constructor takes name of current module (__name__) as **argument**

route() function

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

URL binding with the function

List of parameters to be forwarded to the underlying Rule object

Flask - Application

```
App.run(host, port, options
```

All these parameters are optional

Sl.no	Parameter	Description
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
3	options	To be forwarded to underlying Werkzeug server.

```
Python hello.py
```

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

Flask – Routing

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 – Application

In order to test **Flask** installation, type the following code in the editor as **Hello.py**

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

@app.route('/')
def hello_world():
   return 'Hello World'

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

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	**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 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)
```

Flask App 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.

In our first application, the URL ('/') is associated with the home function that returns a particular string displayed on the web page.

In other words, we can say that if we visit the particular URL mapped to some particular function, the output of that function is rendered on the browser's screen.

Consider the following example.

Example

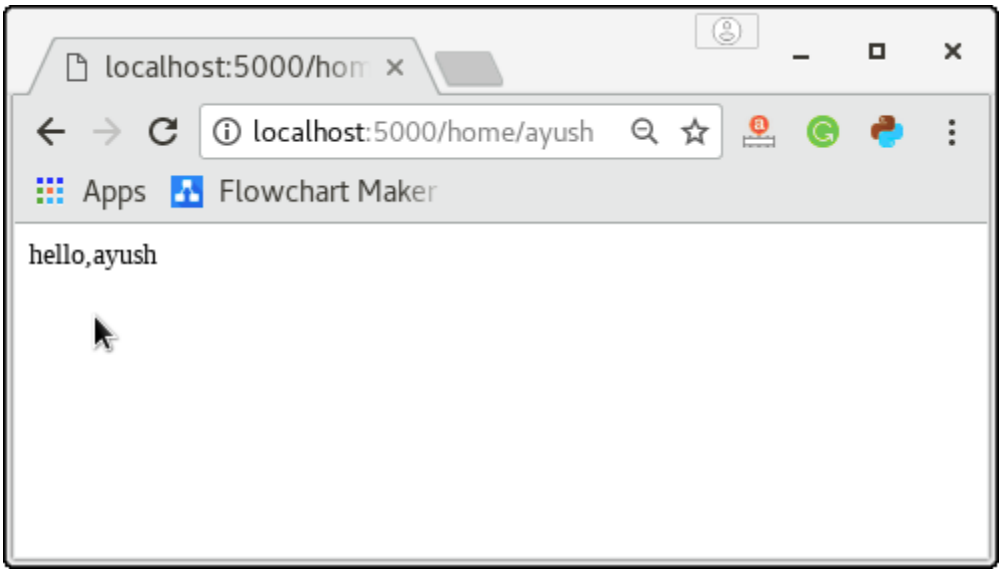
```
1. from flask import Flask
2. app = Flask(__name__)
3.
4. @app.route('/home')
5. def home():
6.     return "hello, welcome to our website";
7.
8. if __name__ =="__main__":
9.     app.run(debug = True)
```

Flask facilitates us to add the variable part to the URL by using the section. We can reuse the variable by adding that as a parameter into the view function. Consider the following example.

Example

```
1. from flask import Flask
2. app = Flask(__name__)
3. 
4. @app.route('/home/<name>')
5. def home(name):
6.     return "hello,"+name;
7. 
8. if __name__ =="__main__":
9.     app.run(debug = True)
```

It will produce the following result on the web browser.

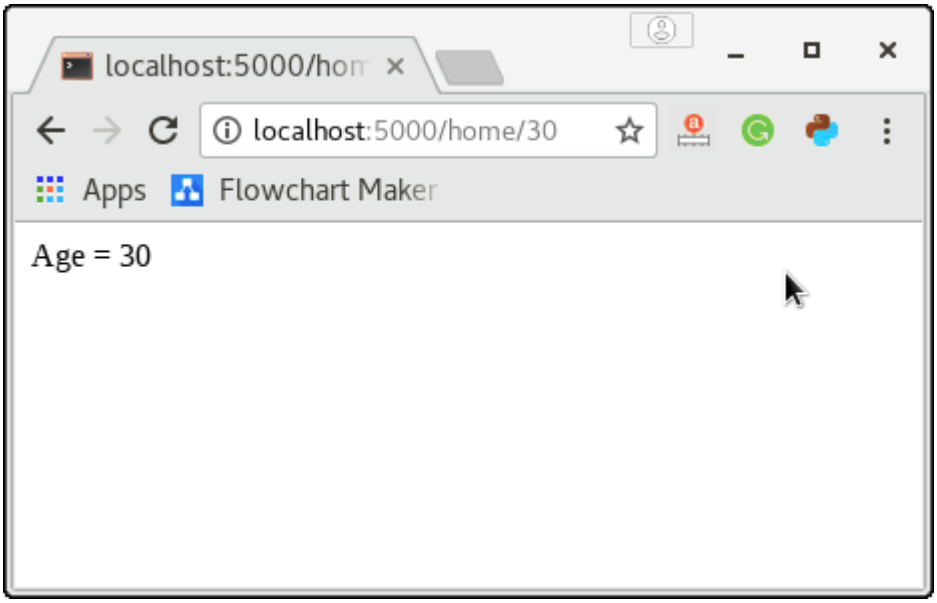


The converter can also be used in the URL to map the specified variable to the particular data type. For example, we can provide the integers or float like age or salary respectively.

Consider the following example.

Example

```
1. from flask import Flask
2. app = Flask(__name__)
3.
4. @app.route('/home/<int:age>')
5. def home(age):
6.     return "Age = %d"%age;
7.
8. if __name__ =="__main__":
9.     app.run(debug = True)
```



The following converters are used to convert the default string type to the associated data type.

1. string: default
2. int: used to convert the string to the integer
3. float: used to convert the string to the float.
4. path: It can accept the slashes given in the URL.

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.

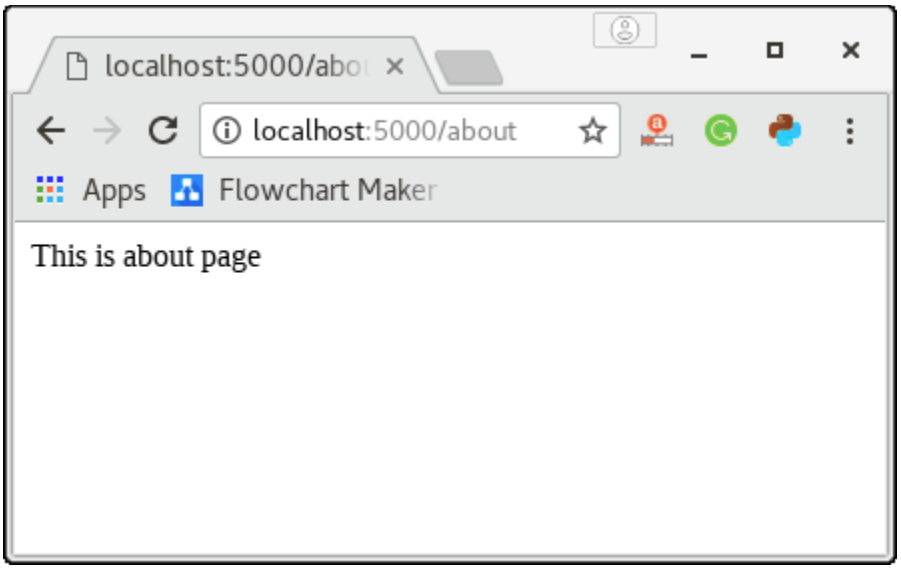
1. add_url_rule(<url rule>, <endpoint>, <view function>)

   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.

   Consider the following example.

Example

```
1. from flask import Flask
2. app = Flask(__name__)
3.
4. def about():
5.     return "This is about page";
6.
7. app.add_url_rule("/about","about",about)
8.
9. if __name__ =="__main__":
10.     app.run(debug = True)
```



Flask URL Building

The url_for() function is used to build a URL to the specific function dynamically. The first argument is the name of the specified function, and then we can pass any number of keyword argument corresponding to the variable part of the URL.

This function is useful in the sense that we can avoid hard-coding the URLs into the templates by dynamically building them using this function.

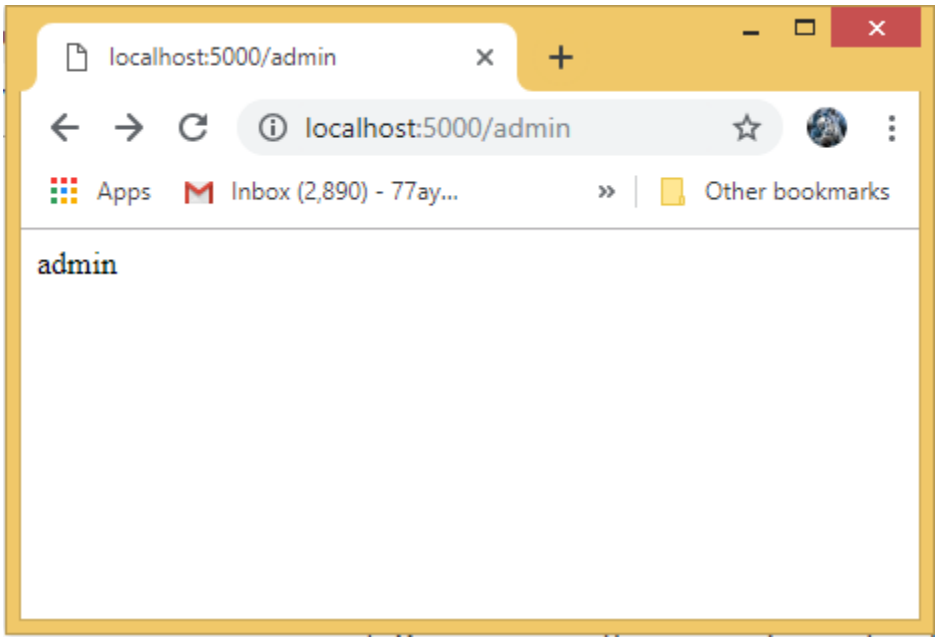
Consider the following python flask script.

Example

```
1. from flask import *
2. app = Flask(__name__)
3. @app.route('/admin')
4. def admin():
5.     return 'admin'
6. @app.route('/librarion')
7. def librarion():
8.     return 'librarion'
9. @app.route('/student')
10. def student():
11.     return 'student'
```

```
12. @app.route('/user/<name>')
13. def user(name):
14.     if name == 'admin':
15.         return redirect(url_for('admin'))
16.     if name == 'librarion':
17.         return redirect(url_for('librarion'))
18.     if name == 'student':
19.         return redirect(url_for('student'))
20. if __name__ =='__main__':
21.     app.run(debug = True)
```

The above script simulates the library management system which can be used by the three types of users, i.e., admin, librarion, and student. There is a specific function named user() which recognizes the user the redirect the user to the exact function which contains the implementation for this particular function.



For example, the URL http://localhost:5000/user/admin is redirected to the URL http://localhost:5000/admin, the URL localhost:5000/user/librarion, is redirected to the URL http://localhost:5000/librarion, the URL http://localhost:5000/user/student is redirected to the URL http://localhost/student.

Benefits of the Dynamic URL Building

1. It avoids hard coding of the URLs.

2. We can change the URLs dynamically instead of remembering the manually changed hard-coded URLs.
3. URL building handles the escaping of special characters and Unicode data transparently.
4. The generated paths are always absolute, avoiding unexpected behavior of relative paths in browsers.
5. If your application is placed outside the URL root, for example, in /myapplication instead of /, url_for() properly handles that for you.

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.

login.html

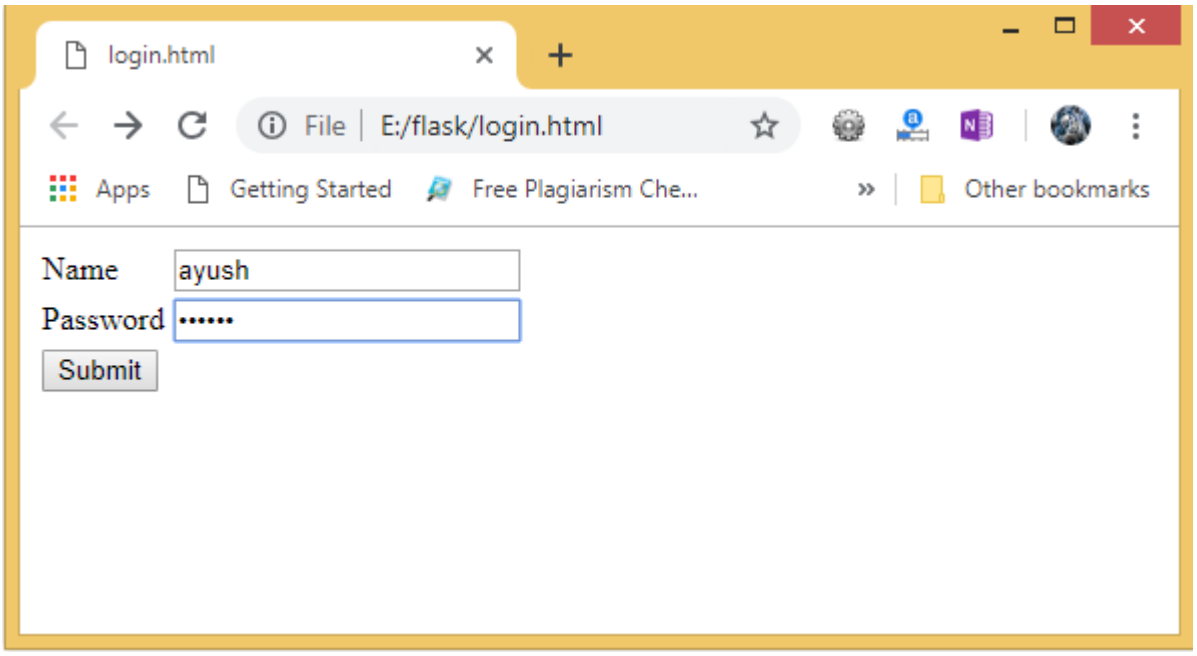
```
<html>
   <body>
      <form action = "http://localhost:5000/login" method = "post">
         <table>
         <tr><td>Name</td>
         <td><input type ="text" name ="uname"></td></tr>
         <tr><td>Password</td>
         <td><input type ="password" name ="pass"></td></tr>
         <tr><td>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

Now, Enter the following code into the script named post_example.py.

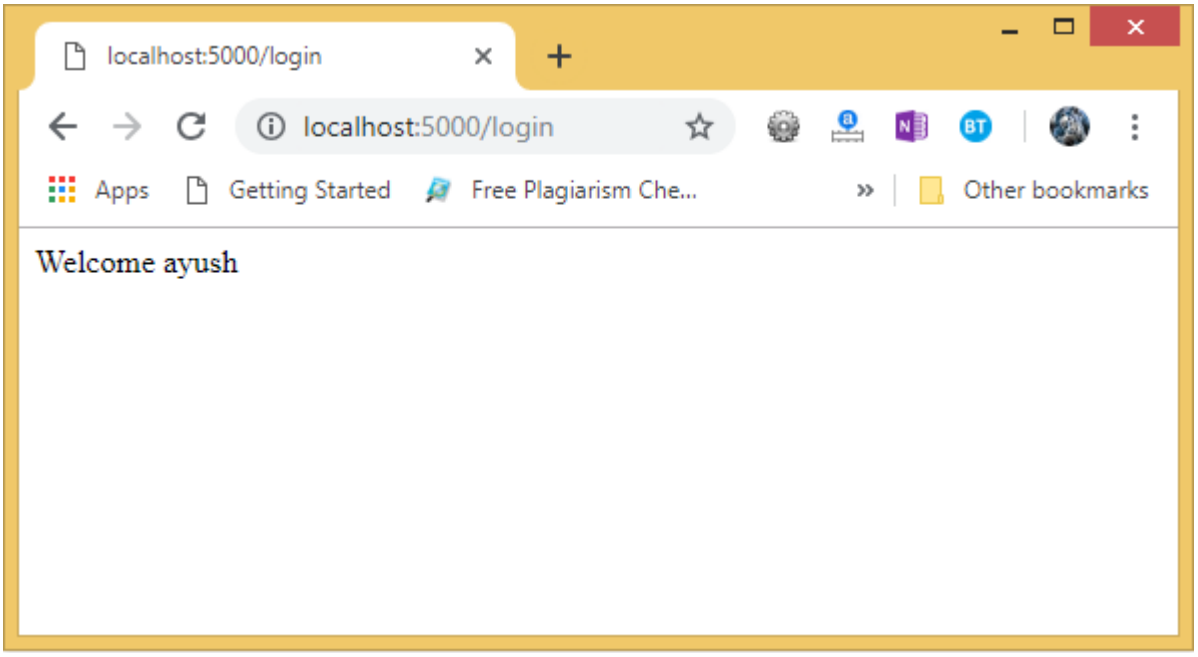
post_example.py

```
from flask import *
app = Flask(__name__)
@app.route('/login',methods = ['POST'])
def login():
      uname=request.form['uname']
      passwrd=request.form['pass']
      if uname=="ayush" and passwrd=="google":
         return "Welcome %s" %uname
if __name__ == '__main__':
   app.run(debug = True)
```

Now, start the development server by running the script using python post_exmple.py and open login.html on the web browser as shown in the following image.



Give the required input and click Submit, we will get the following result.



Hence, the form data is sent to the development server by using the post method.

GET Method

Let's consider the same example for the Get method. However, there are some changes in the data retrieval syntax on the server side. First, create a form as login.html.

login.html

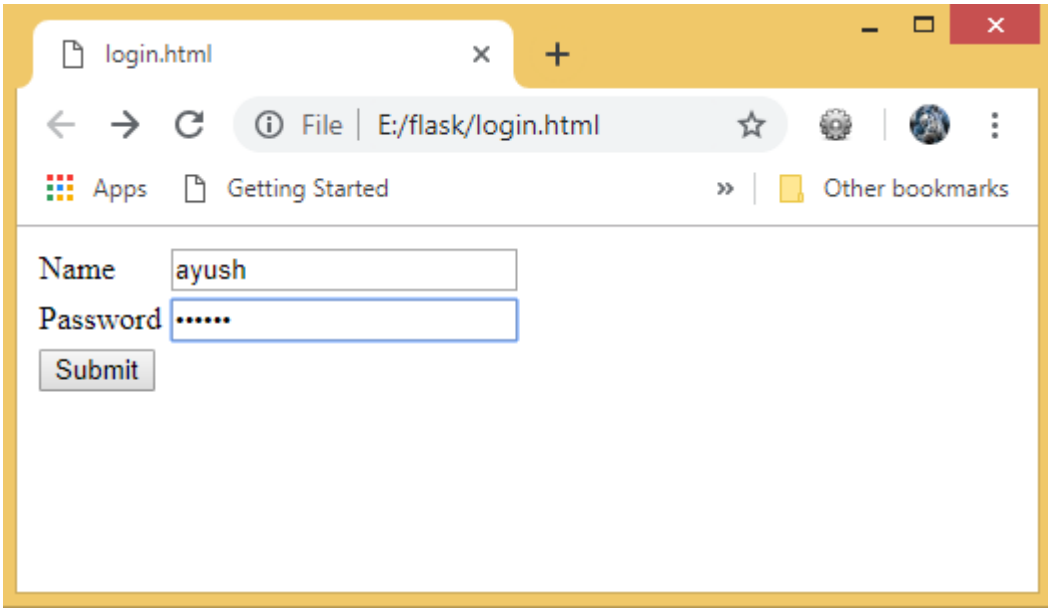
```
<html>
   <body>
      <form action = "http://localhost:5000/login" method = "get">
         <table>
         <tr><td>Name</td>
         <td><input type ="text" name ="uname"></td></tr>
         <tr><td>Password</td>
         <td><input type ="password" name ="pass"></td></tr>
         <tr><td><input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

Now, create the following python script as get_example.py.

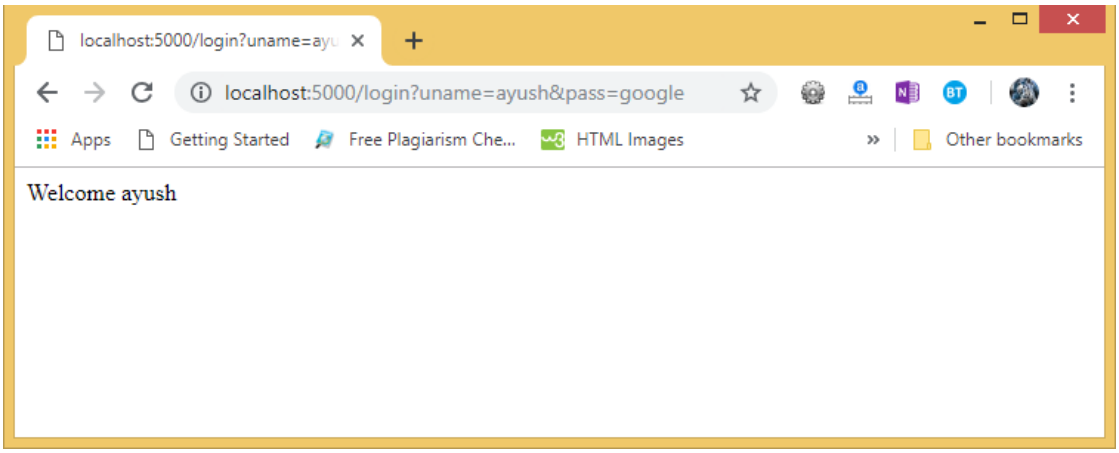
get_example.py

```
from flask import *
app = Flask(__name__)
@app.route('/login',methods = ['GET'])
def login():
      uname=request.args.get('uname')
      passwrd=request.args.get('pass')
      if uname=="ayush" and passwrd=="google":
         return "Welcome %s" %uname

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

Now, open the HTML file, login.html on the web browser and give the required input.



Now, click the submit button.



As we can check the result. The data sent using the get() method is retrieved on the development server.

The data is obtained by using the following line of code.

1. uname = request.args.get('uname')

Here, the args is a dictionary object which contains the list of pairs of form parameter and its corresponding value.

In the above image, we can also check the URL which also contains the data sent with the request to the server. This is an important difference between the GET requests and

the POST requests as the data sent to the server is not shown in the URL on the browser in the POST requests.

Rendering external HTML files

Flask facilitates us to render the external HTML file instead of hardcoding the HTML in the view function. Here, we can take advantage of the jinja2 template engine on which the flask is based.

Flask provides us the render_template() function which can be used to render the external HTML file to be returned as the response from the view function.

Consider the following example.

Example

To render an HTML file from the view function, let's first create an HTML file named as message.html.

message.html

```
<html>
<head>
<title>Message</title>
</head>
<body>
<h1>hi, welcome to the website </h1>
</body>
</html>
```

script.py

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

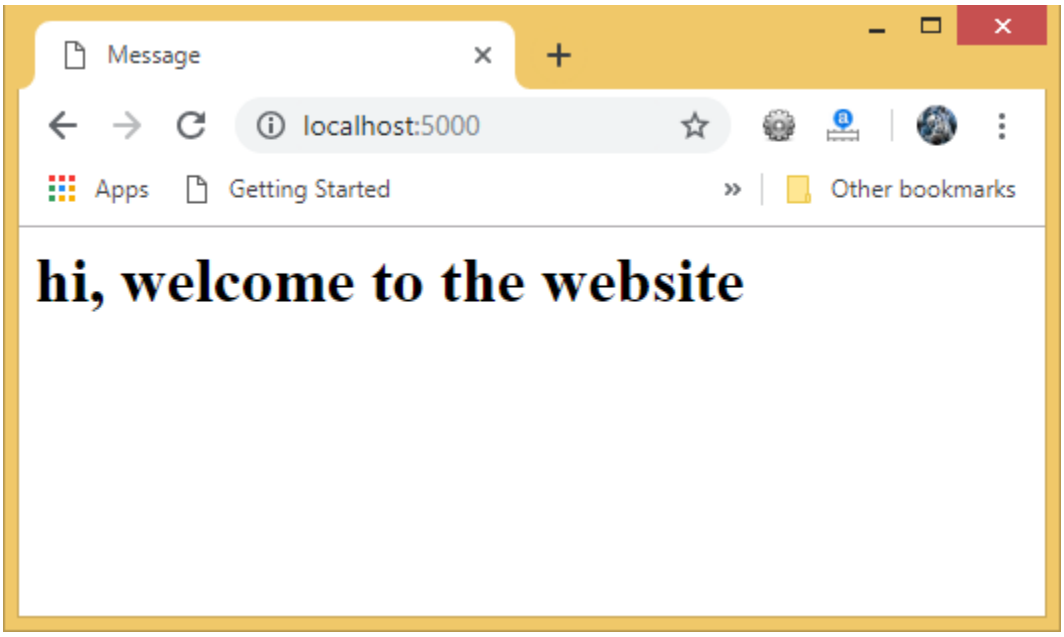
```
@app.route('/')
def message():
    return render_template('message.html')
if __name__ == '__main__':
  app.run(debug = True)
```

Here, we must create the folder **templates** inside the application directory and save the HTML templates referenced in the flask script in that directory.

In our case, the path of our script file **script.py is E:\flask** whereas the path of the HTML template is **E:\flask\templates**.

Application Directory

- script.py
- templates
- message.html



Delimiters

Jinga 2 template engine provides some delimiters which can be used in the HTML to make it capable of dynamic data representation. The template system provides some HTML syntax which are placeholders for variables and expressions that are replaced by their actual values when the template is rendered.

The jinga2 template engine provides the following delimiters to escape from the HTML.

- {% ... %} for statements
- {{ ... }} for expressions to print to the template output
- {# ... #} for the comments that are not included in the template output
- # ... ## for line statements

Example

Consider the following example where the variable part of the URL is shown in the HTML script using the {{ ... }} delimiter.

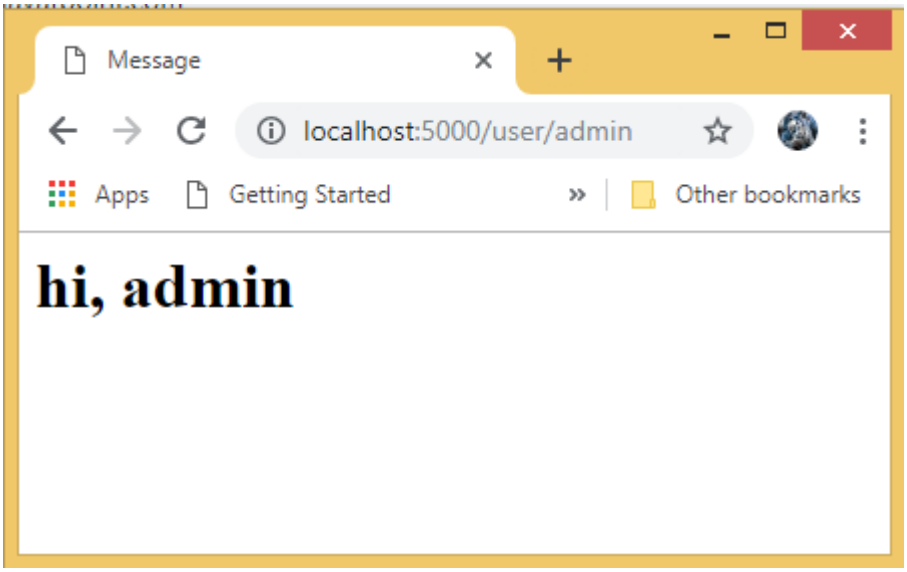
message.html

```
<html>
<head>
<title>Message</title>
</head>
<body>
<h1>hi, {{ name }}</h1>
</body>
</html>
```

script.py

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

@app.route('/user/<uname>')
def message(uname):
```

```
    return render_template('message.html',name=uname)
if __name__ == '__main__':
   app.run(debug = True)
```



The variable part of the URL http://localhost:5000/user/admin is shown in the HTML script using the {{name}} placeholder.

Embedding Python statements in HTML

Due to the fact that HTML is a mark-up language and purely used for the designing purpose, sometimes, in the web applications, we may need to execute the statements for the general-purpose computations. For this purpose, Flask facilitates us the delimiter {%...%} which can be used to embed the simple python statements into the HTML.

Example

In the following example, we will print the table of a number specified in the URL, i.e., the URL http://localhost:5000/table/10 will print the table of 10 on the browser's window.

Here, we must notice that the for-loop statement is enclosed inside {%...%} delimiter, whereas, the loop variable and the number is enclosed inside {{ ... }} placeholders.

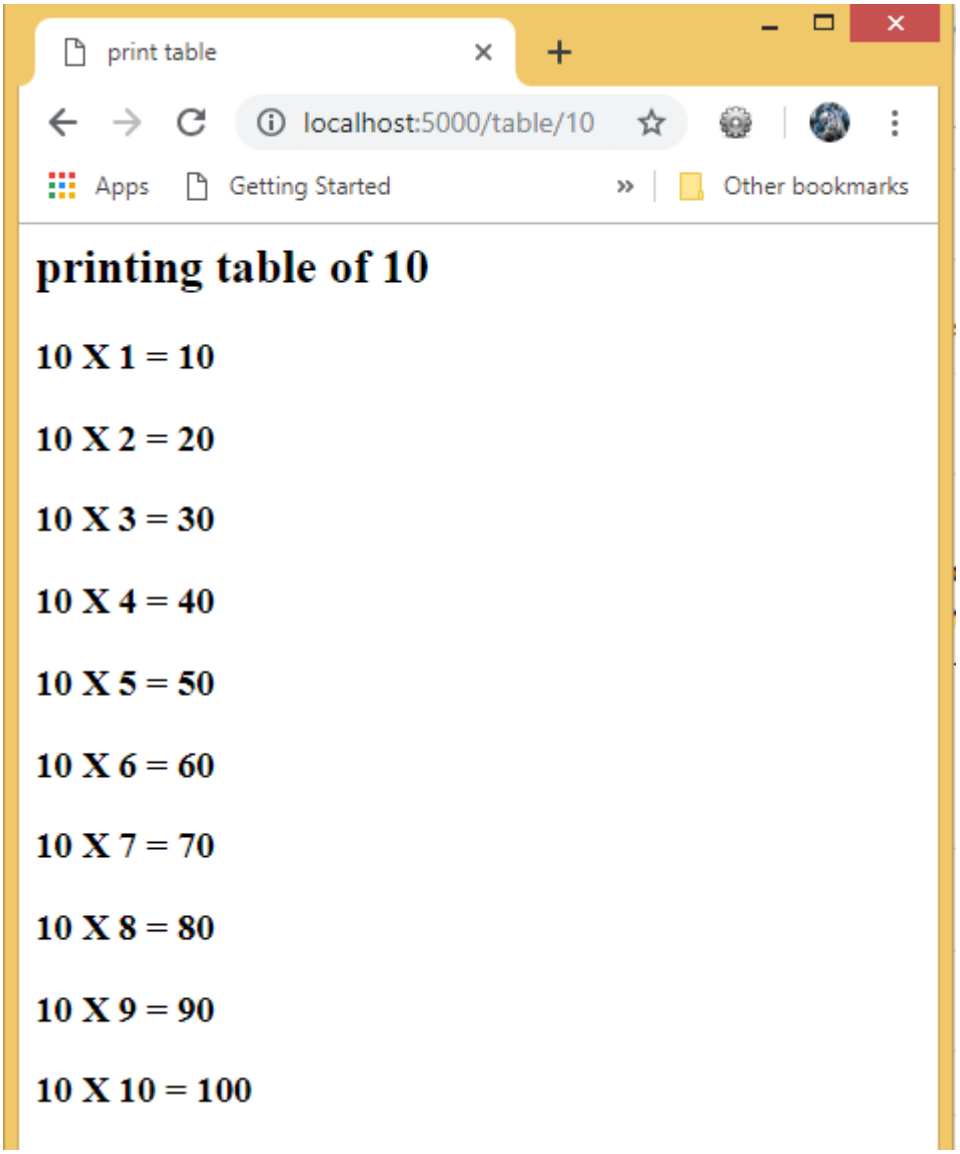
script.py

```
1. from flask import *
```

```
2.  app = Flask(__name__)
3.
4.  @app.route('/table/<int:num>')
5.  def table(num):
6.      return render_template('print-table.html',n=num)
7.  if __name__ == '__main__':
8.    app.run(debug = True)
```

print-table.html

```
<html>
<head>
<title>print table</title>
</head>
<body>
<h2> printing table of {{n}}</h2>
{% for i  in range(1,11): %}
   <h3>{{n}} X {{i}} = {{n * i}} </h3>
{% endfor %}
</body>
</html>
```



Flask Request Object

In the client-server architecture, the request object contains all the data that is sent from the client to the server. As we have already discussed in the tutorial, we can retrieve the data at the server side using the HTTP methods.

In this section of the tutorial, we will discuss the Request object and its important attributes that are given in the following table.

SN	Attribute	Description

1	Form	It is the dictionary object which contains the key-value pair of form parameters and their values.
2	args	It is parsed from the URL. It is the part of the URL which is specified in the URL after question mark (?).
3	Cookies	It is the dictionary object containing cookie names and the values. It is saved at the client-side to track the user session.
4	files	It contains the data related to the uploaded file.
5	method	It is the current request method (get or post).

Form data retrieval on the template

In the following example, the / URL renders a web page customer.html that contains a form which is used to take the customer details as the input from the customer.

The data filled in this form is posted to the /success URL which triggers the print_data() function. The print_data() function collects all the data from the request object and renders the result_data.html file which shows all the data on the web page.

The application contains three files, i.e., script.py, customer.html, and result_data.html.

script.py

```
1. from flask import *
2. app = Flask(__name__)
3.
4. @app.route('/')
5. def customer():
6.     return render_template('customer.html')
7.
8. @app.route('/success',methods = ['POST', 'GET'])
9. def print_data():
10.    if request.method == 'POST':
11.        result = request.form
```

```
12.     return render_template("result_data.html",result = result)
13.
14. if __name__ == '__main__':
15.   app.run(debug = True)
```

customer.html

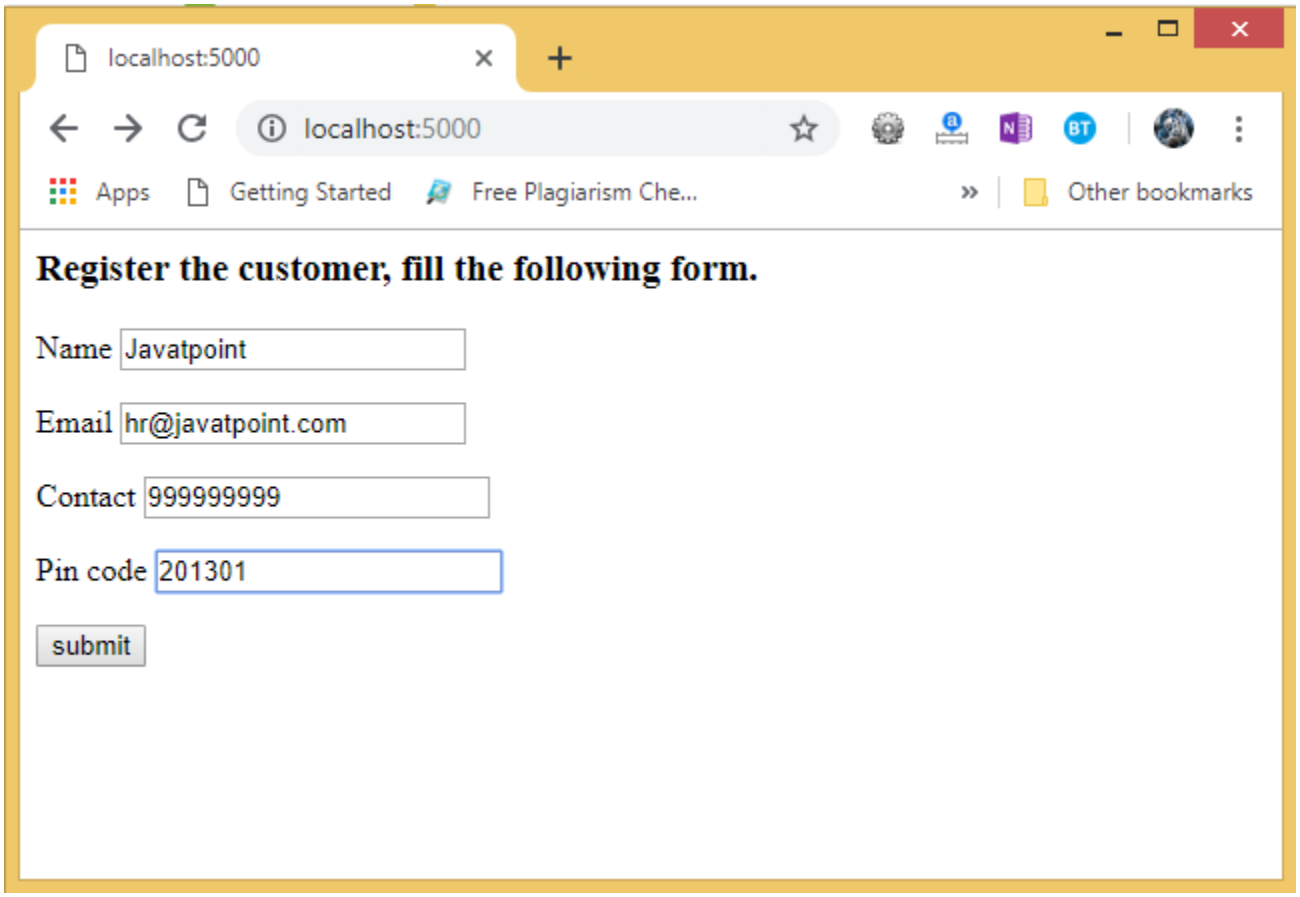
```
1.  <html>
2.    <body>
3.       <h3>Register the customer, fill the following form.</h3>
4.       <form action = "http://localhost:5000/success" method = "POST">
5.          <p>Name <input type = "text" name = "name" /></p>
6.          <p>Email <input type = "email" name = "email" /></p>
7.          <p>Contact <input type = "text" name = "contact" /></p>
8.          <p>Pin code <input type ="text" name = "pin" /></p>
9.          <p><input type = "submit" value = "submit" /></p>
10.      </form>
11.   </body>
12. </html>
```

result_data.html

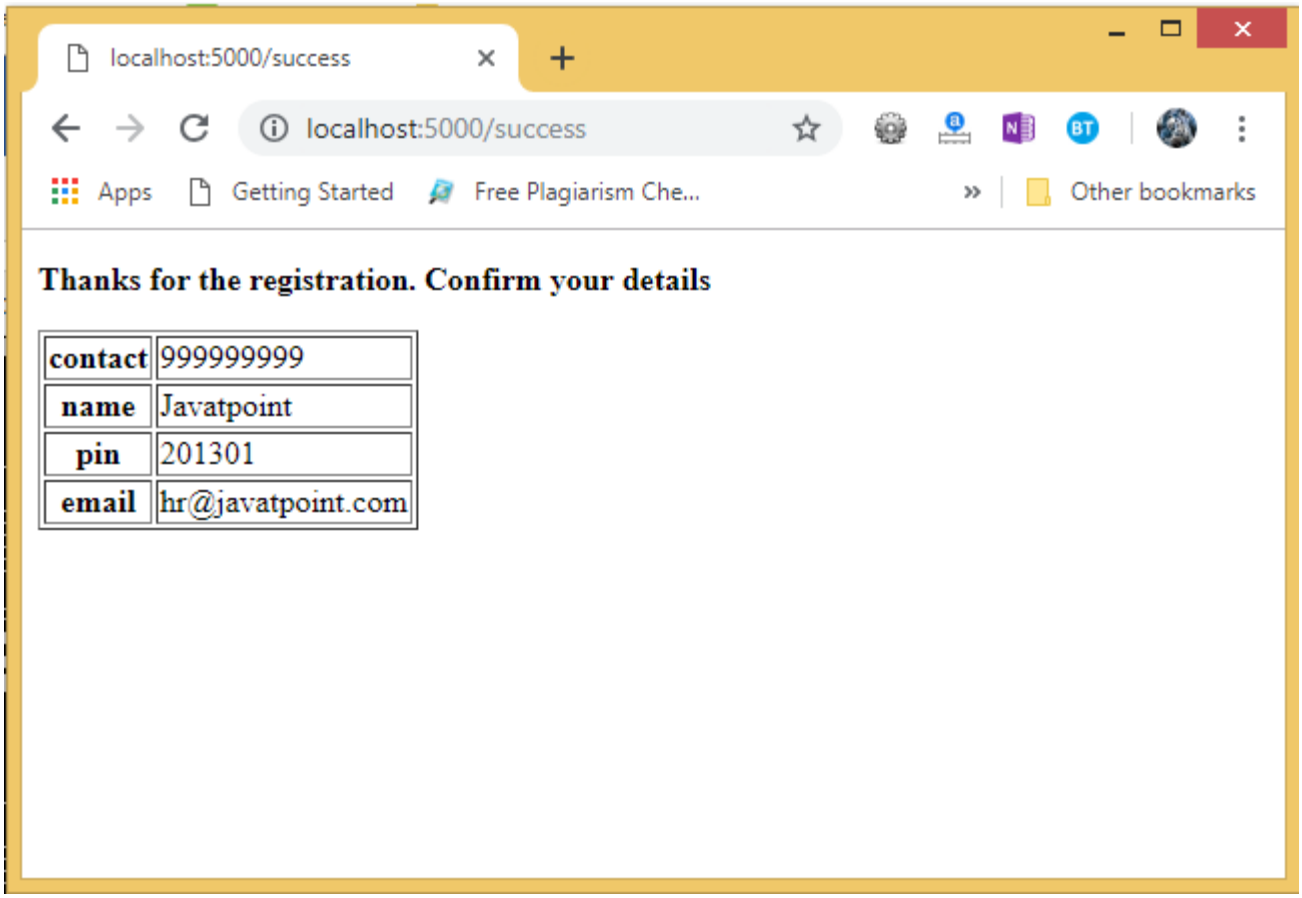
```
1.  <!doctype html>
2.  <html>
3.    <body>
4.       <p><strong>Thanks for the registration. Confirm your details</strong></p>
5.       <table border = 1>
6.          {% for key, value in result.items() %}
7.             <tr>
8.                <th> {{ key }} </th>
9.                <td> {{ value }} </td>
10.            </tr>
11.         {% endfor %}
12.      </table>
13.   </body>
```

14. </html>

To run this application, we must first run the script.py file using the command python script.py. This will start the development server on localhost:5000 which can be accessed on the browser as given below.



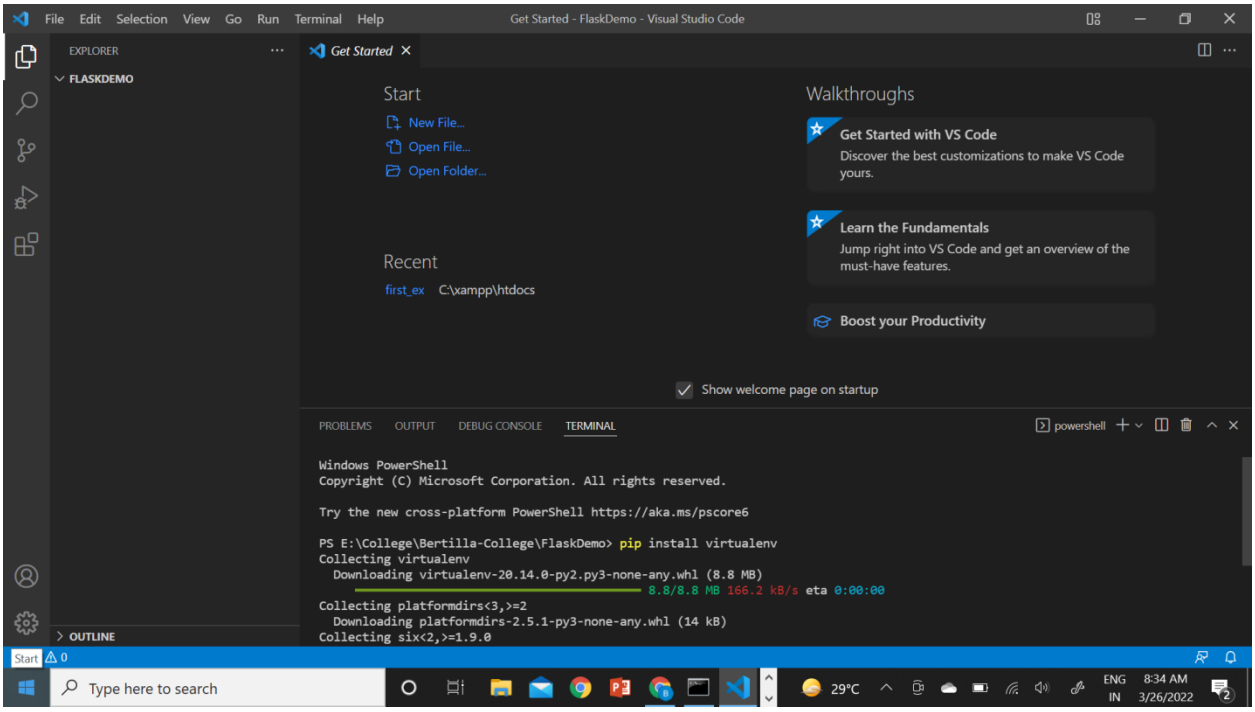
Now, hit the submit button. It will transfer to the /success URL and shows the data entered at the client.


localhost:5000/success
localhost:5000/success
Apps
Getting Started
Free Plagiarism Che...
Other bookmarks
Thanks for the registration. Confirm your details
contact 999999999
name Javatpoint
pin 201301
email hr@javatpoint.com

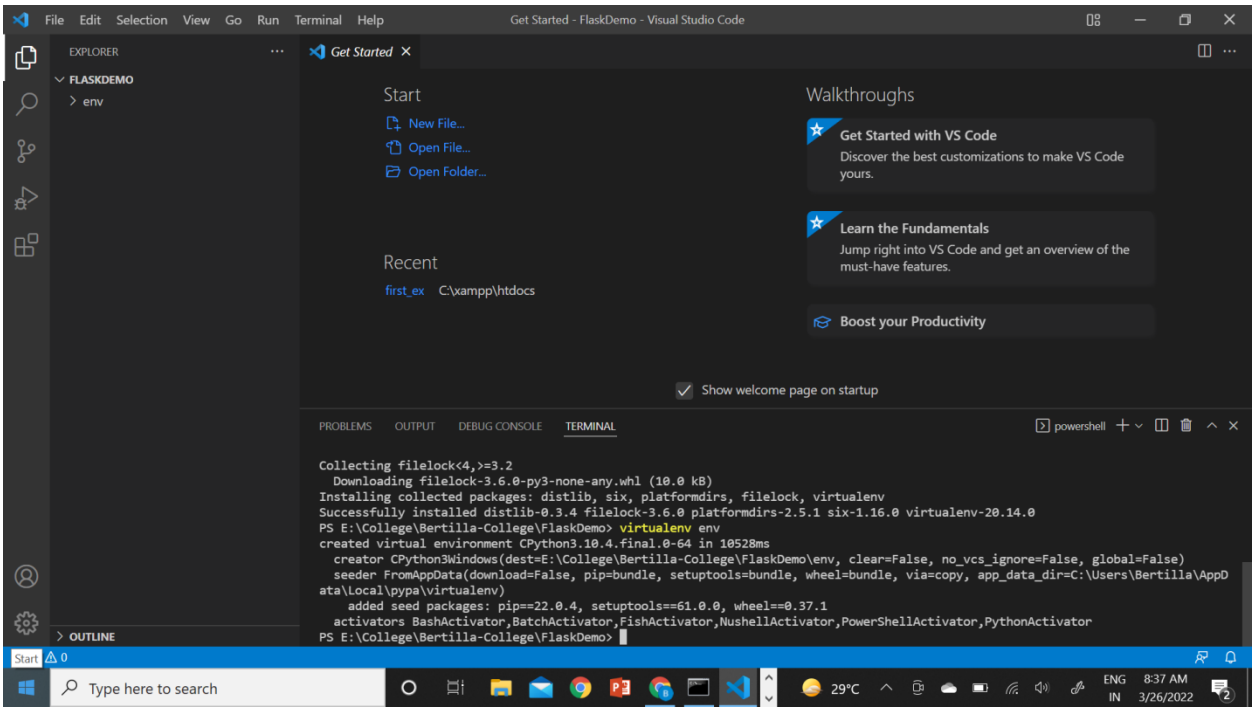
Flask installation:

1.Create an application Folder

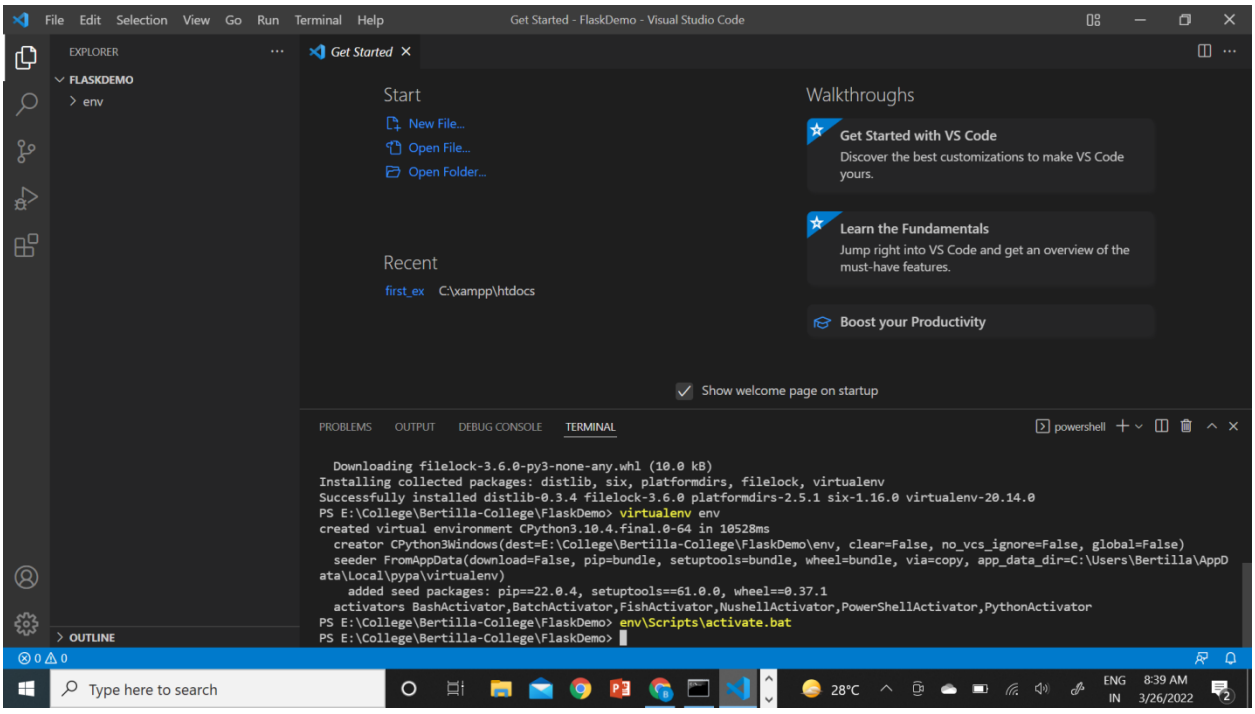
2.Install the virtual environment



3.Create a virtual environment

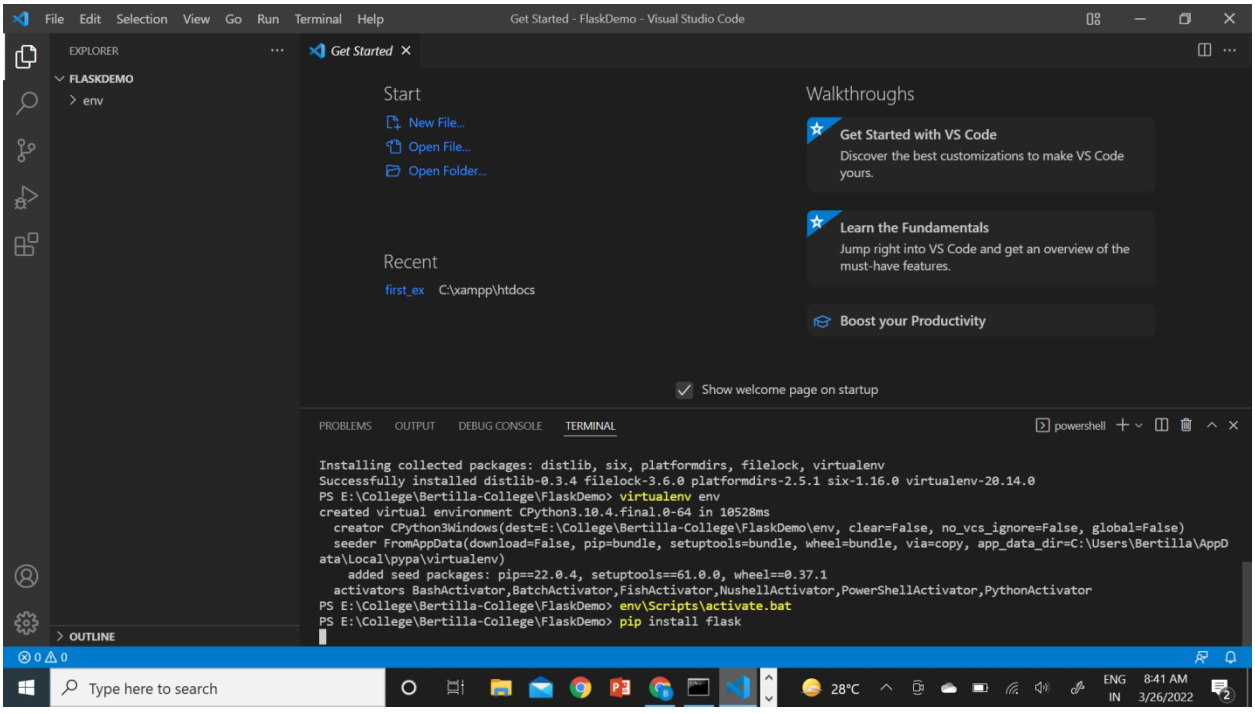


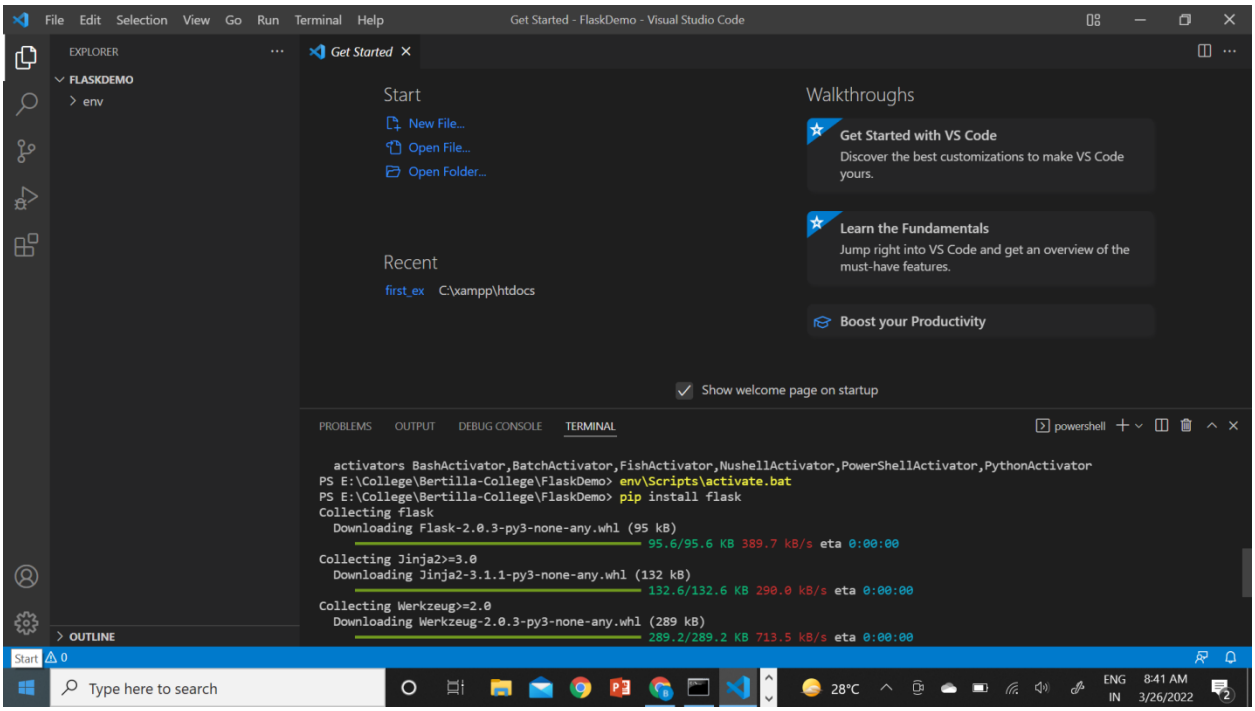
4.Activate the environment:



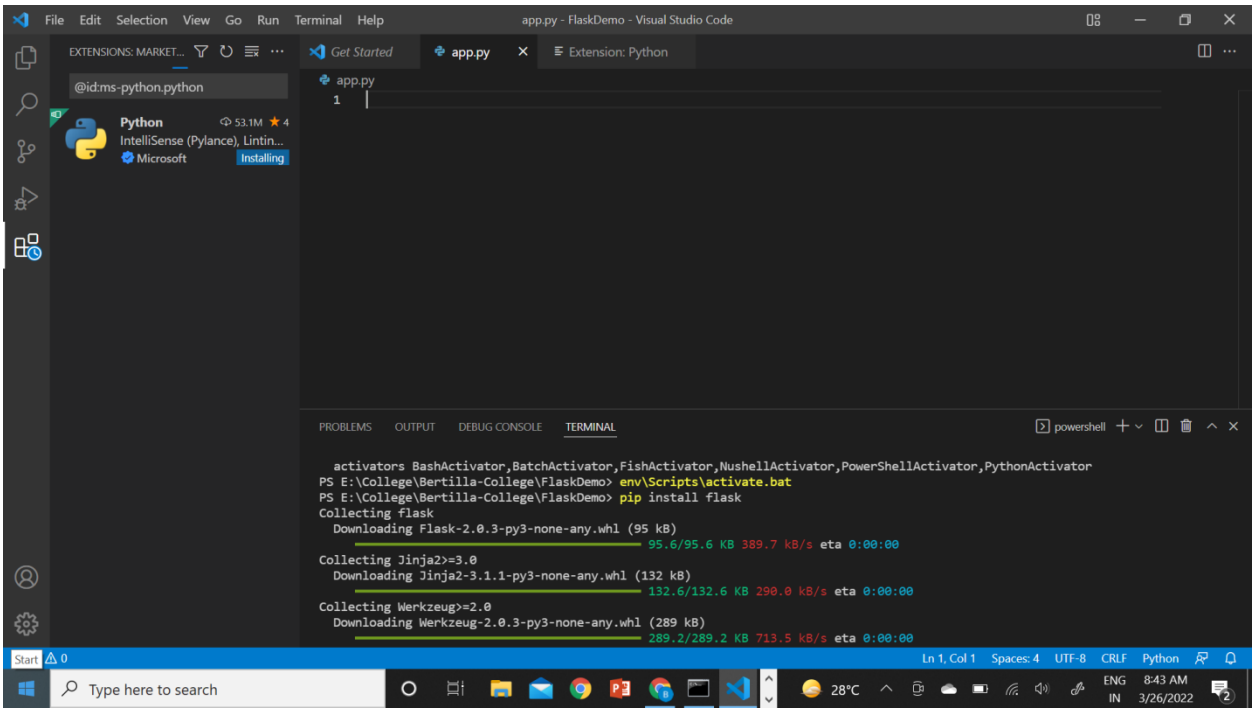
On MAC OS:

5.Install flask:





6.Create the app.py file



7.Execute the flask run command:

PS E:\College\Bertilla-College\FlaskDemo2> flask run

* Environment: production

WARNING: This is a development server. Do not use it in a production deployment.

Use a production WSGI server instead.

* Debug mode: off

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

Errors:

flask run command:

Windows PowerShell

set FLASK_APP=hello.py

****Flask Jinga Filters(Refer text book)**

Flask Cookies

The cookies are stored in the form of text files on the client's machine. Cookies are used to track the user's activities on the web and reflect some suggestions according to the user's choices to enhance the user's experience.

Cookies are set by the server on the client's machine which will be associated with the client's request to that particular server in all future transactions until the lifetime of the cookie expires or it is deleted by the specific web page on the server.

In flask, the cookies are associated with the Request object as the dictionary object of all the cookie variables and their values transmitted by the client. Flask facilitates us to specify the expiry time, path, and the domain name of the website.

1. response.set_cookie(<title>, <content>, <expiry time>)

   In Flask, the cookies are set on the response object by using the set_cookie() method on the response object. The response object can be formed by using the make_response() method in the view function.

In addition, we can read the cookies stored on the client's machine using the get() method of the cookies attribute associated with the Request object.

1. request.cookies.get(<title>)

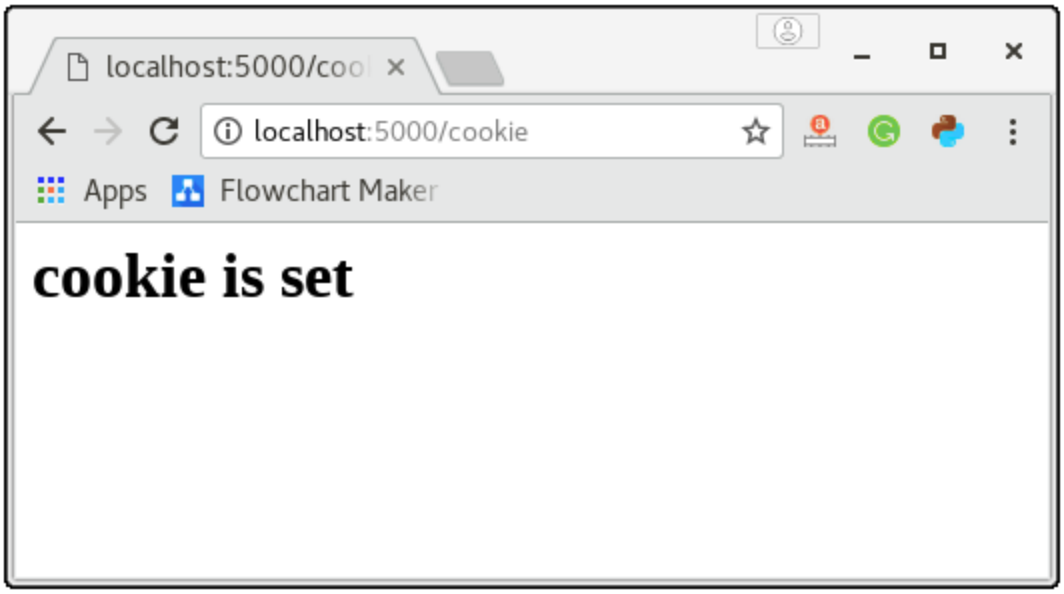
A simple example to set a cookie with the title 'foo' and the content 'bar' is given below.

Example

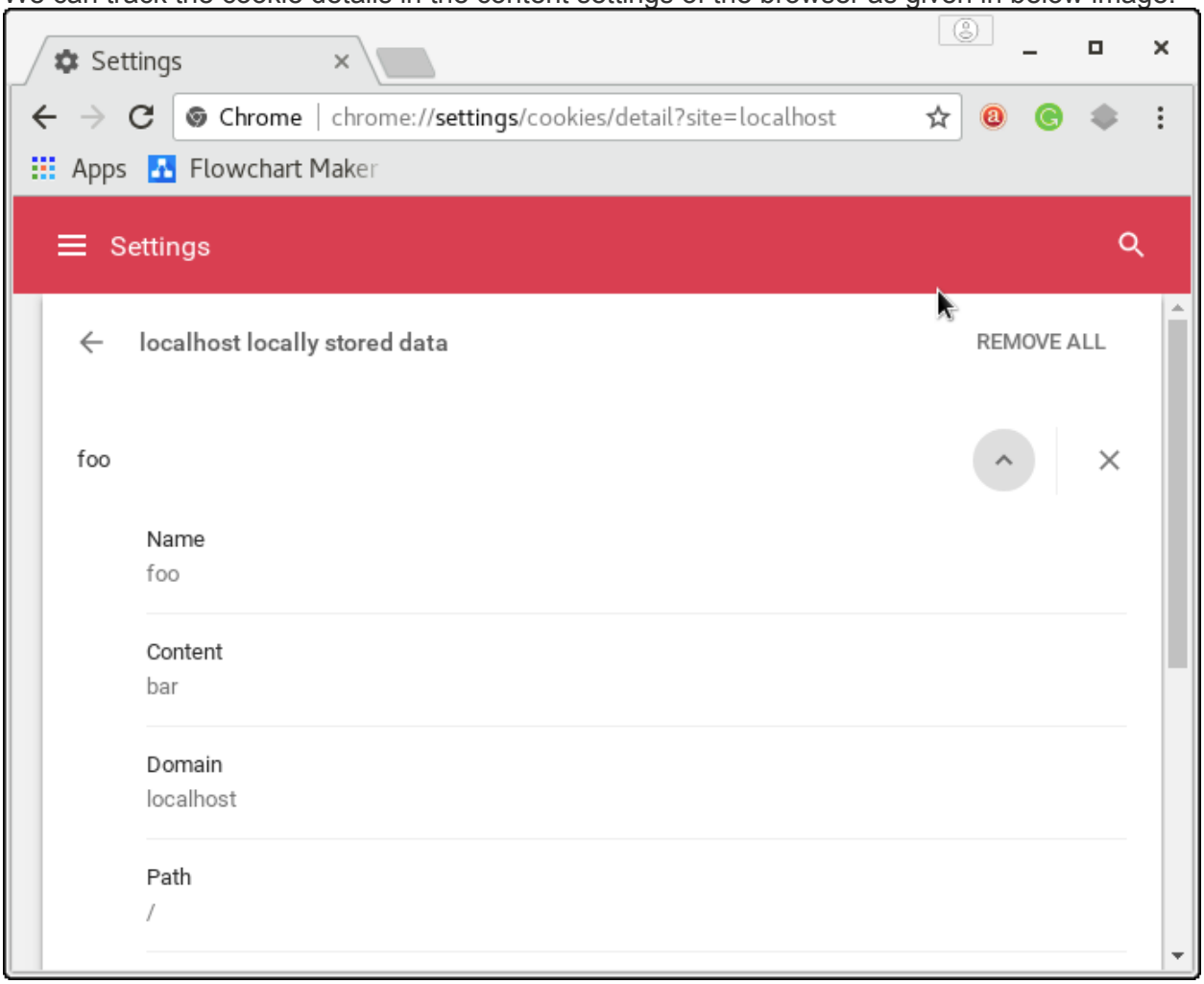
```
from flask import *

app = Flask(__name__)

@app.route('/cookie')
def cookie():
   res = make_response("<h1>cookie is set</h1>")
   res.set_cookie('foo','bar')
   return res

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

The above python script can be used to set the cookie with the name 'foo' and the content 'bar' on the browser for the website **localhost:5000**.

Run this python script using the command python script.py and check the result on the browser.



We can track the cookie details in the content settings of the browser as given in below image.



Flask cookies Example:

Create cookie

In Flask, set the cookie on the response object.Use the `make_response()` function to get the response object from the return value of the view function.After that, the cookie is stored using the `set_cookie()` function of the response object.

It is easy to read back cookies.The `get()` method of the request.cookies property is used to read the cookie.

In the following Flask application, when you access the ‘ /‘ URL, a simple form opens.

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

This HTML page contains a 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>
```

Set cookie

The form is published to the ‘/setcookie’ URL.The associated view function sets the cookie name userID and renders another page.

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

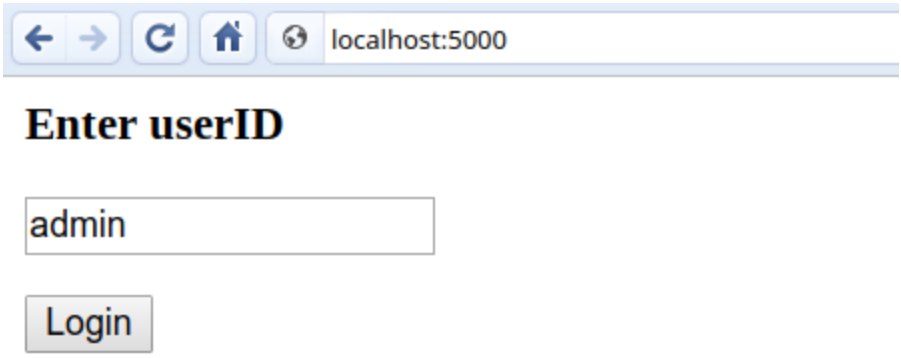
```
    resp = make_response(render_template('readcookie.html'))
    resp.set_cookie('userID', user)

    return resp
```

Get cookie

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

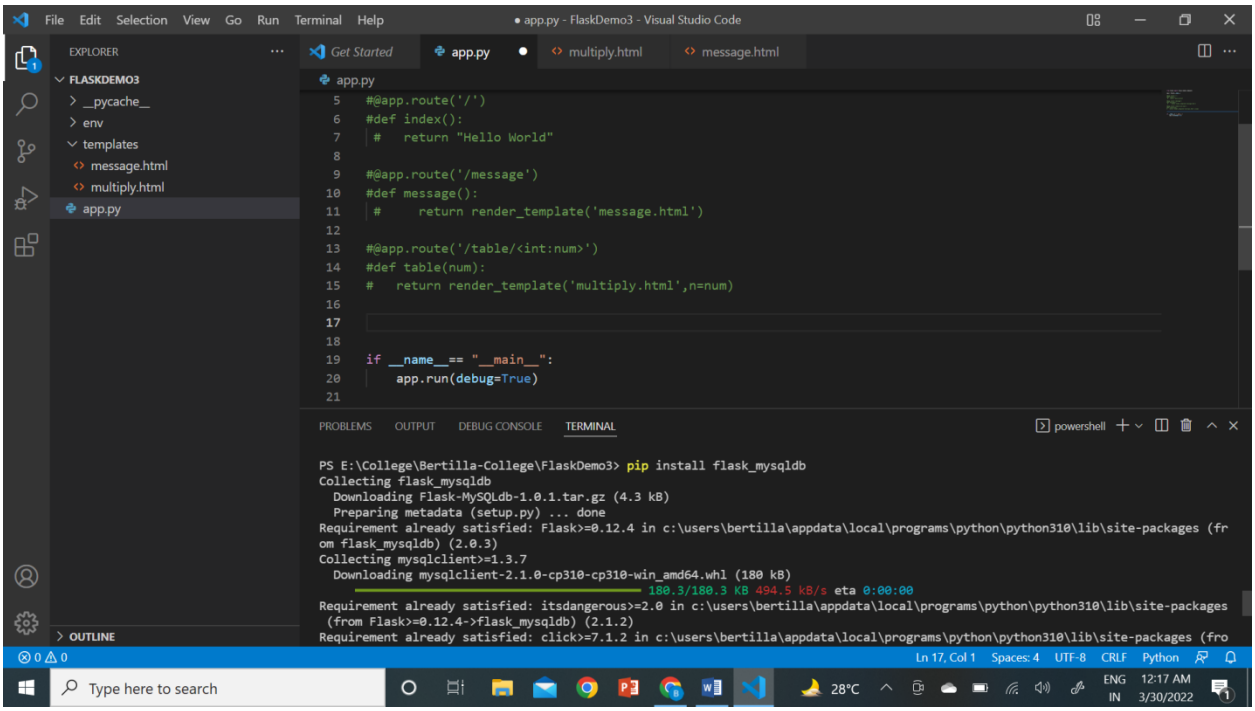
```
@app.route('/getcookie')
def getcookie():
    name = request.cookies.get('userID')
    return '<h1>welcome ' + name + '</h1>'
```

Run the app and access *localhost:5000/*



After you click login, the cookie is set and you can read the cookie.

Flask with mysql database:




```
DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3307
DB_DATABASE=blog
DB_USERNAME=root
DB_PASSWORD=
```

Example:

```
from flask import Flask,render_template,request

from flask_mysqldb import MySQL

app = Flask(__name__)

app.config['MYSQL_HOST']="127.0.0.1"
app.config['MYSQL_USER']="root"
app.config['MYSQL_PASSWORD']=""
app.config['MYSQL_PORT']="3307"
app.config['MYSQL_DB']="blog"

mysql = MySQL(app)
@app.route('/',methods=['GET','POST'])
def index():
```

```
    if request.method=='POST':
        username=request.form['username']
        email=request.form['email']

        cur= mysql.connection.cursor()
        cur.execute("insert into users (name,email) values
(%s,%s)",(username,email))
        #cur.execute("insert into users (name,email) values ('a','a@gmail.com')")
        mysql.connection.commit()

        cur.close()

        return "success"

    return render_template('index.html')

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