**FLASK**

What is Web Framework?

Web Application Framework or simply Web Framework represents a collection of libraries and modules that enables a web application developer to write applications easily

What is Flask?

Flask is a web application framework written in Python.

Jinja2

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

Flask is often referred to 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 form validation support. Instead, Flask supports the extensions to add such functionality to the application.

**Flask – Environment**

**Setting Up the Environment**

1. **Install Flask**: You need to have Flask installed. You can do this using pip:

bash

pip install flask

1. **Environment Variables**: Flask uses environment variables to manage configurations. Commonly used environment variables include:
   * FLASK\_APP: Specifies the entry point of your application (e.g., FLASK\_APP=app.py).
   * FLASK\_ENV: Set this to development to enable the debug mode (e.g., FLASK\_ENV=development).
2. **Run the Application**: To run your Flask app with the environment variables set, you can use the following commands:

Bash:Terminal

export FLASK\_APP=app.py

export FLASK\_ENV=development

flask run

On Windows, you should use set instead of export:

cmd

set FLASK\_APP=app.py

set FLASK\_ENV=development

flask run

**Configuration Settings**

Flask applications can be configured using a configuration file or directly in the code. Here are some common settings:

Config.py

class Config:

DEBUG = True

SECRET\_KEY = 'your\_secret\_key'

DATABASE\_URI = 'sqlite:///example.db'

app.config.from\_object(Config)

**Debug Mode**

Running Flask in debug mode provides helpful error messages and automatically reloads the application when code changes are detected:

Bash:Terminal

export FLASK\_ENV=development

**Virtual Environment**

Using a virtual environment helps to manage dependencies and avoid conflicts:

Bash:Terminal

python -m venv venv

source venv/bin/activate # On Windows use `venv\Scripts\activate`

**Example Structure**

A typical Flask project structure might look like this:

my\_flask\_app/

│

├── app.py

├── config.py

├── templates/

│ └── base.html

├── static/

│ └── styles.css

└── venv/

**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.

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. In Debug Mode, The flask run command can do more than just start the development server. By enabling debug mode to True, the server will automatically reload if code changes, and will show an interactive debugger in the browser if an error occurs during a request. |
| 4 | **options**  To be forwarded to the server. |

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 during passing the debug parameter to the **run()** method.

app.debug = True

app.run()

OR

app.run(debug = True)

**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 in place of  **route()**

Now, decorator is not needed in this style.

def hello\_world():

return ‘hello world’

app.add\_url\_rule(‘/’, ‘hello’, hello\_world)

**Flask – Variable Rules**

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/Prathiba\_R** is entered as a **URL** in the browser, **‘Prathiba\_R’** will be supplied to **hello()** function as a keyword 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)

open the browser and enter URL **localhost:5000/hello/Prathiba\_R .**

The following output will be displayed in the browser.

Hello Prathiba\_R!

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

from flask import Flask

app = Flask(\_\_name\_\_)

@app.route('/add/<int:num1>/<int:num2>')

def add\_numbers(num1, num2):

    result = num1 + num2

    return f'The sum of {num1} and {num2} is {result}'

if \_\_name\_\_ == '\_\_main\_\_':

    app.run(debug=True)

on browser: **localhost:5000/add/3/5**

|  |  |
| --- | --- |
| **Sr.No.** | **Converters & Description** |
| 1 | **int**  accepts integer |
| 2 | **float**  For floating point value |
| 3 | **path**  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()

Visit the URL **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

Enter this URL in the browser − **localhost:5000/rev/1.1**

The **revision()** function takes up the floating point number as argument. The following result appears in the browser window −

Revision Number 1.100000

**Flask – URL Building**

The **url\_for()** function is very useful for dynamically building a URL for a specific function. The function accepts the name of a function as first argument, and one or more keyword arguments, each corresponding to the variable part of URL.

The following script demonstrates use of **url\_for()** function.

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.

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

The application response in browser is −

Hello Admin

Enter the following URL in the browser − **localhost:5000/user/Prathiba**

The application response now changes to −

Hello Prathiba as Guest