

- [Introduction](#)
- [What is a Web Framework?](#)
- [What is Micro-framework?](#)
- [What is Flask?](#)
- [What is WSGI?](#)
- [What is Jinja 2?](#)
- [Why Flask?](#)
- [Templates in Flask](#)

Introduction

Python is the most sort after language for web application development and data science as well. It has risen to this height for its ease of use and variety of supportive libraries. There are legacy frameworks like Java's Enterprise edition and ASP. NET's MVC framework is still popular for enterprise-level development. But Python is the favourite for new POC and small-time development where an audience that of an enterprise is not immediately expected. And, of course, the fact that Python and most of its libraries are open sources and free is an exceptionally helpful and useful factor too.

In Python, for web application development, the leaders are [Django](#) and Flask. Django is very much similar to the MVC framework of ASP. NET. But Flask is different. Let's see how. Read further and learn more about [machine learning deployment](#) using flask and get practical insights.

What is a Web Framework?

To understand Flask, we need to understand what is the traditional web framework. A web application framework is a collection of libraries and modules which helps developers write the business layer without worrying about the protocol, thread management, etc. And Python's Django follows a traditional web framework which is also called an enterprise framework.

What is Micro-framework?

In contrast to a traditional web framework, a micro-framework is a minimalistic framework where developers are provided with a lot of freedom to build the web application layer. As compared to an enterprise framework, a developer would not need to set up many things in a micro-framework to get the web app hosted and running. This is incredibly useful in cases of small web app development where the needs are not the same as an enterprise-level framework which would save lots of development ~ maintenance time and money as a consequence.

What is Flask?

Now, we are well equipped to understand Flask. Flask is Python's micro-framework for web app development. It was developed by **Armin Ronacher**, who led an international team of Python enthusiasts called Poccco. Flask consists of Werkzeug WSGI toolkit and Jinja2 template engine. Both were also developed by Poccco. It was initially released in April 2010. Currently, the latest version as of writing this article is 2.0.2.

What is WSGI?

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

Werkzeug is one of the most advanced WSGI modules that contain various tools and utilities that facilitate web application development. Flask implements Werkzeug.

What is Jinja 2?

Jinja 2 is a template rendering engine. It renders the web pages for the server with any specified custom content given to it by the webserver. Flask renders its HTML based templates using Jinja 2.

Why Flask?

Now, you would be able to understand, when we define Flask as a micro-framework that is built using WSGI and Jinja 2 template engine.

Major advantages of Flask are:

1. Ease of setup and use.
2. Freedom to build the structure of the web application.

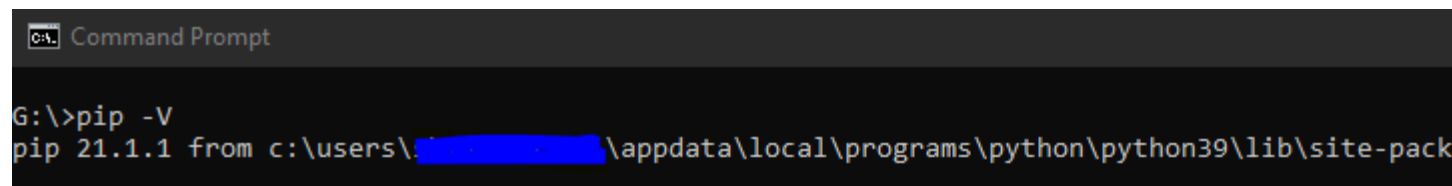
With freedom comes responsibility, similarly, Flask needs the developers to carefully structure it, since Flask doesn't have "flask rules" to follow as compared to frameworks like Django. As the web app increases in complexity, this structuring is what is going to be the foundation.

Setting up Environment

The first thing to set up Flask on your system is to install the Python2.7 version or more. Here, I'll guide you through the process of installing Python on the Windows 10 system.

Using this [Official Python website](#), you can select the correct python version to install as per your specific OS and system configurations. You can select the latest Python version, as of writing this article the latest is **3.9.10**. Please ensure if **pip** has been correctly set up after python installation.

The command to check that is `pip -V`

A screenshot of a Windows Command Prompt window. The title bar says "Command Prompt". The command prompt shows the command `G:\>pip -V` and the output `pip 21.1.1 from c:\users\[redacted]\appdata\local\programs\python\python39\lib\site-pack`.

```
G:\>pip -V
pip 21.1.1 from c:\users\[redacted]\appdata\local\programs\python\python39\lib\site-pack
```

Setting up your virtual environment

Using a virtual environment for a python-based project is considered a best programming practice. In the case of Linux OS, it ensures the OS level python is not

disturbed by installing and using other python packages. And, in the case of other OS, this ensures the settings and files are intact in the virtual environment which is incredibly useful when different projects need different versions of the same packages.

`virtualenv` was the tool of choice for creating virtual environments in Python2. But, `venv` is the tool for Python3 and beyond. `virtualenv` is a third-party package whereas `venv` is a default functionality of Python3.x. We'll be creating a virtual environment using `venv` here.


To work with `venv` you would just need to ensure you have Python 3.x version installed; x being any version upward of 1.

To create a virtual environment, use the following command,

```
python -m venv .
```

If you have python 2.x versions also installed in your system then, considering using `python3` instead of `python`. In my case, Python 3.9.5 is the default version of python installed, so not an issue.

Since I have already navigated inside the folder where I want to create my virtual environment, I have used ``.`` to point to the directory. You can choose to write the full path where you want your virtual environment created as well.



```
C:\> Command Prompt  
G:\dir\Flask>python -m venv .
```

Once completed, the cursor will return, it will not show any output. To verify if the necessary files and folders have been created, you can use the following command:

```

C:\ Command Prompt

G:\dir\Flask>python -m venv .

G:\dir\Flask>dir .
Volume in drive G is New Volume
Volume Serial Number is [REDACTED]

Directory of G:\dir\Flask

06/02/2021  10:22 AM    <DIR>        .
06/02/2021  10:22 AM    <DIR>        ..
06/02/2021  10:22 AM    <DIR>        Include
06/02/2021  10:22 AM    <DIR>        Lib
06/02/2021  10:22 AM             124 pyvenv.cfg
06/02/2021  10:23 AM    <DIR>        Scripts
                1 File(s)            124 bytes
                5 Dir(s)  227,066,822,656 bytes free

```

Now, to activate your virtual environment in Windows OS:

- Navigate to the 'Scripts' folder in the virtual environment directory using, ``cd Scripts``
- And, then use the following command: ``activate``

This is how an activated virtual environment looks like, irrespective of the OS.

```

C:\ Command Prompt

G:\dir\Flask\Scripts>activate
(Flask) G:\dir\Flask\Scripts>

```

(Note: A blue arrow points from the text 'activated virtual environment' to the '(Flask)' prefix in the prompt.)

In the case of Mac OS or Linux OS,

``source env_name/bin/activate``

And, to deactivate, in any OS.

``deactivate``

Now, let's install Flask:

``pip3 install Flask``

The output will be something like

```
Command Prompt

(Flask) G:\dir\Flask\Scripts>pip3 install Flask
Collecting Flask
  Downloading Flask-2.0.1-py3-none-any.whl (94 kB)
    |████████████████████████████████████████| 94 kB 1.1 MB/s
Collecting itsdangerous>=2.0
  Downloading itsdangerous-2.0.1-py3-none-any.whl (18 kB)
Collecting click>=7.1.2
  Downloading click-8.0.1-py3-none-any.whl (97 kB)
    |████████████████████████████████████████| 97 kB 3.3 MB/s
Collecting Werkzeug>=2.0
  Downloading Werkzeug-2.0.1-py3-none-any.whl (288 kB)
    |████████████████████████████████████████| 288 kB 6.4 MB/s
Collecting Jinja2>=3.0
  Downloading Jinja2-3.0.1-py3-none-any.whl (133 kB)
    |████████████████████████████████████████| 133 kB 6.4 MB/s
Collecting colorama
  Downloading colorama-0.4.4-py2.py3-none-any.whl (16 kB)
Collecting MarkupSafe>=2.0
  Downloading MarkupSafe-2.0.1-cp39-cp39-win_amd64.whl (14 kB)
Installing collected packages: MarkupSafe, colorama, Werkzeug, Jinja2, itsdangerous, click
Successfully installed Flask-2.0.1 Jinja2-3.0.1 MarkupSafe-2.0.1 Werkzeug-2.0.1 click-8.0.1
WARNING: You are using pip version 21.1.1; however, version 21.1.2 is available.
You should consider upgrading via the 'g:\dir\flask\scripts\python.exe -m pip install --up
```

Your 1st Flask Application

To start with a boilerplate application, let's use the default as mentioned in the Flask documentation.

I have used **Sublime Text Editor** here; you can choose any text editor of your choice.

Create a ``server.py`` file in the virtual environment and write the following code:

```
G:\dir\Flask\server.py - Sublime Text (UNREGISTERED)
File Edit Selection Find View Goto Tools Project Preferences Help
server.py x
1 from flask import Flask
2 app = Flask(__name__)
3
4 @app.route('/')
5 def hello_world():
6     return 'Hello World!'
7
8 if __name__ == '__main__':
9     app.run()
```

Save this file and run this through the command prompt, just as you would any other python file.

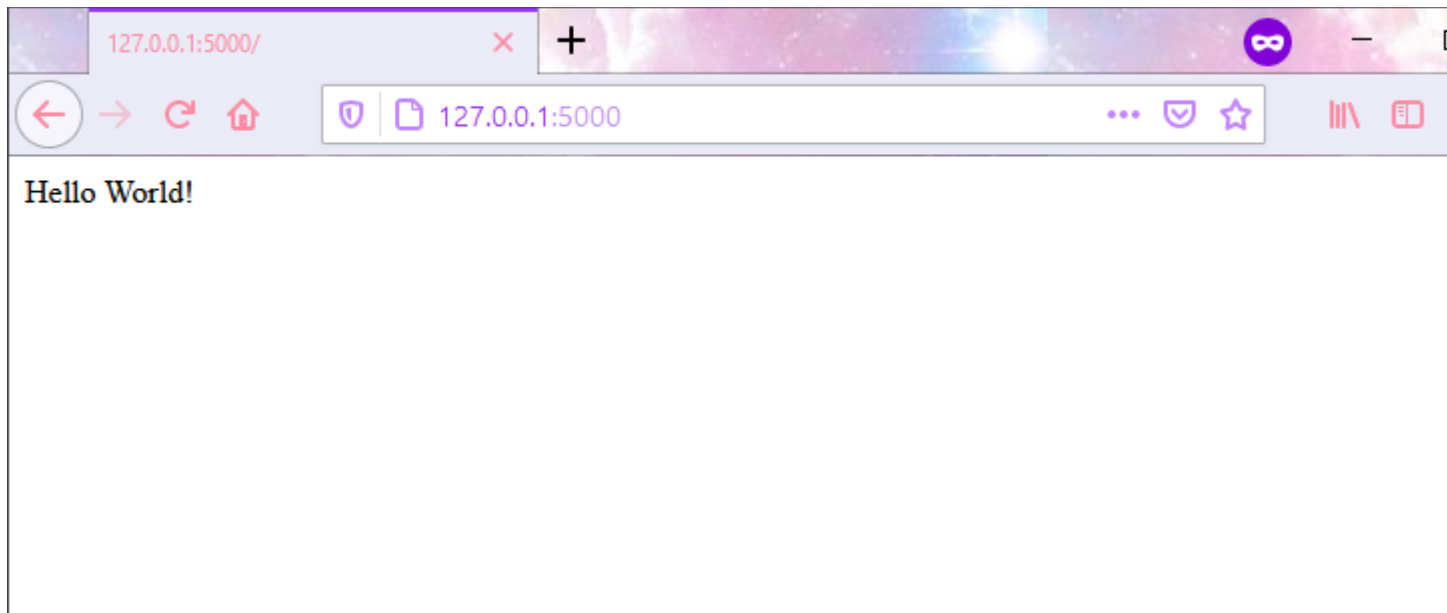
python server.py

Note: Ensure that you never save this as 'Flask' as it would conflict with the Flask itself.

It will give you the following output:

```
Command Prompt - python server.py
(Flask) G:\dir\Flask>python server.py
* Serving Flask app 'server' (lazy loading)
* 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)
127.0.0.1 - - [02/Jun/2021 11:32:13] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [02/Jun/2021 11:32:13] "GET /favicon.ico HTTP/1.1" 404 -
```

Open the localhost URL mentioned in the command prompt to show an output like:



Congratulations, on running your first Flask application! If you had followed through all the steps, there wouldn't be any errors in running the Flask application.

Let's understand the syntax of the program

1. The first line of code `from flask import Flask` is basically importing Flask package in the file.
2. We next create an object of the flask class using `app = flask()`
3. We send the following argument to the default constructor `__name__`, this will help Flask look for templates and static files.
4. Next, we use the route decorator to help define which routes should be navigated to the following function. `@app.route(/)`
5. In this, when we use `/` it lets Flask direct all the default traffic to the following function.
6. We define the function and the action to be performed.
7. The default content type is HTML, so our string will be wrapped in HTML and displayed in the browser.
8. In the main function created, `app.run()` will initiate the server to run on the local development server.

Debugging

When developing a Flask application, it is important to consider using the debugging mode for these two reasons:

1. It will trace errors if and display it.
2. Each time a change is made, you would have to restart the app to be able to view the change, in order to avoid that, the app can be started in debugging mode where the changes will reflect instantaneously.

To debug the code, you need to add the following to your code:

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

Routes

There are few basic functionalities about routes that should be discussed to construct a basic application. The `route()` as noted earlier is a Flask decorator:

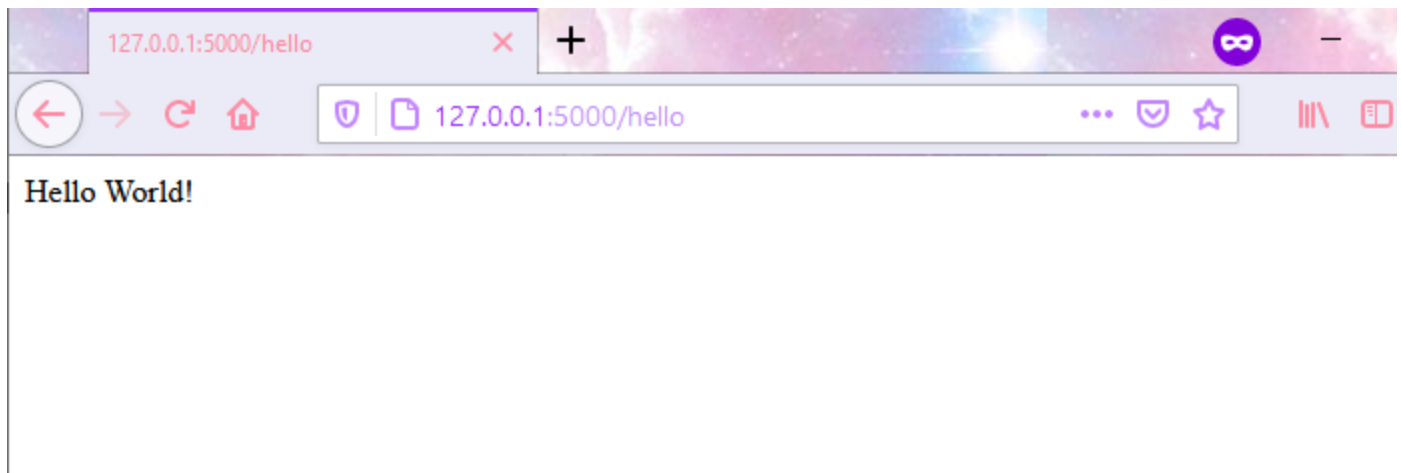
`app.route(rule, options)`

- **Rule** is the URL for which the particular function should be called upon
- **options** are the list of parameters which can be sent to the function as input

You can also use `app.add_url_rule()` to bind a function to the URL.

```
G:\dir\Flask\server.py - Sublime Text (UNREGISTERED)
File Edit Selection Find View Goto Tools Project Preferences Help
server.py x
1 from flask import Flask
2 app = Flask(__name__)
3
4 # @app.route('/')
5 def hello_world():
6     return 'Hello World!'
7
8 app.add_url_rule('/hello', 'hello', hello_world)
9
10 if __name__ == '__main__':
11     app.run(debug = True)
```

The result is the same as routing to `/` which would refer to the homepage of the application.

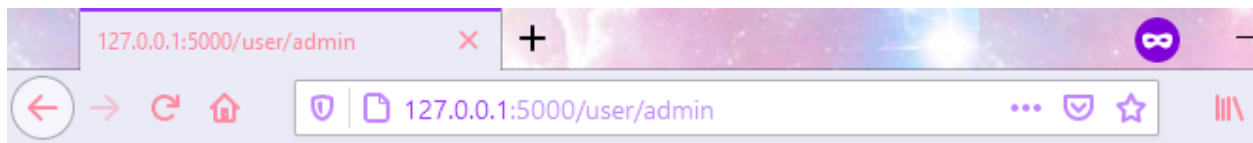


Dynamic Routing

Using `app.route()` we can also send arguments to the python function. Here's the sample code and result.

```
server.py
1  from flask import Flask
2  app = Flask(__name__)
3
4  @app.route('/')
5  def hello_world():
6      return 'Hello World!'
7
8  @app.route('/user/<name>')
9  def greetings(name):
10     return 'Hi ' + name
11
12 app.add_url_rule('/hello', 'hello', hello_world)
13
14 if __name__ == '__main__':
15     app.run(debug = True)
16
```

Result:



Hi admin

The default variable passed in the above case was a **`string`**. There are few other variable types that Werkzeug implementation supports. The same can be found in the [Official Flask Documentation](#) too.

1. **`int`**: accepts positive integers
2. **`float`**: accepts positive float values
3. **`path`**: like **`string`** but also accepts slashes

4. ``uuid``: accepts (universally unique identifier) UUID strings

5. ``string``: (default) accepts any string without a slash

You can create a dynamic Flask URL routing using the ``<>`` & ``:`` symbols. The variable name should be as part of ``<>`` and the type of variable can be mentioned inside the same brackets using a colon (``:``). For example:

```
server.py x
1  from flask import Flask
2  app = Flask(__name__)
3
4  @app.route('/')
5  def hello_world():
6      return 'Hello World!'
7
8  @app.route('/user/<string:name>')
9  def greetings(name):
10     return 'Hi ' + name
11
12  app.add_url_rule('/hello', 'hello', hello_world)
13
14  if __name__ == '__main__':
15     app.run(debug = True)
```

The result remains the same.

Let's look at few more examples using ``uuid`` & ``path`` variable types.

```

server.py x
1  from flask import Flask
2  app = Flask(__name__)
3
4  @app.route('/')
5  def hello_world():
6      return 'Hello World!'
7
8  @app.route('/user/<string:name>')
9  def greetings(name):
10     return 'Hi ' + name
11
12  @app.route('/uniqueid/<uuid:api_key>')
13  def display_key(api_key):
14     return "The API Key " + str(api_key)
15
16  @app.route('/path/<path:sub_path>')
17  def display_path(sub_path):
18     # Accepts any string with slashes
19     return sub_path
20
21  app.add_url_rule('/hello', 'hello', hello_world)
22
23  if __name__ == '__main__':
24     app.run(debug = True)
25

```

The result for `uuid` based results is:



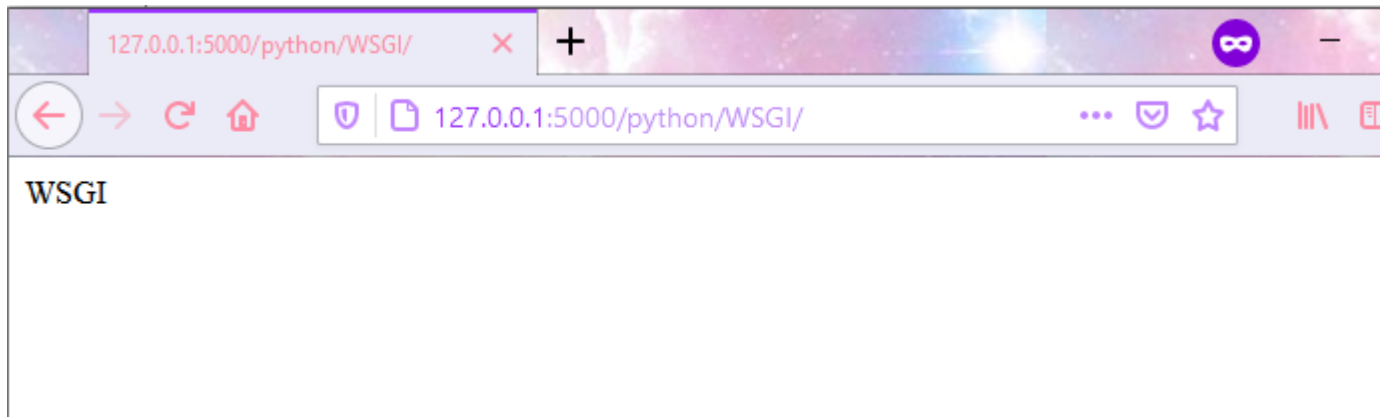
The result for `path` based results is:



Now that we have covered the basics of routing in Flask. There is one more important concept we need to be aware of. The importance of slash (`/`) in the dynamic routing. Let's see with an example.

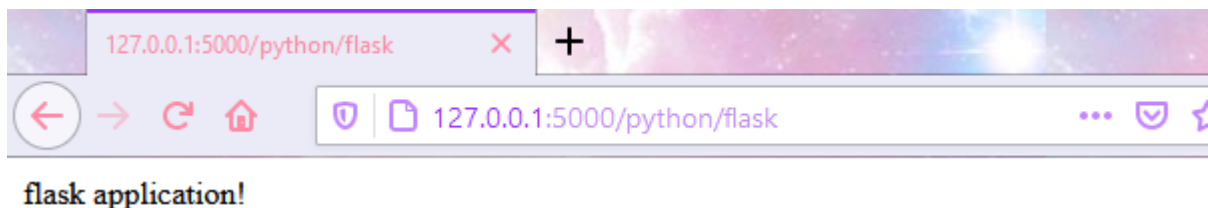
```
server.py
1  from flask import Flask
2  app = Flask(__name__)
3
4  @app.route('/python/flask')
5  def print_python():
6      return 'flask application!'
7
8  @app.route('/python/WSGI/')
9  def print_wsgi():
10     return 'WSGI'
11
12  if __name__ == '__main__':
13     app.run(debug = True)
```

In the above piece of code, we see two routes one not ending with a slash (`/python/flask`) and one ending with a slash (`/python/WSGI/`). The results of the route ending with a slash are:

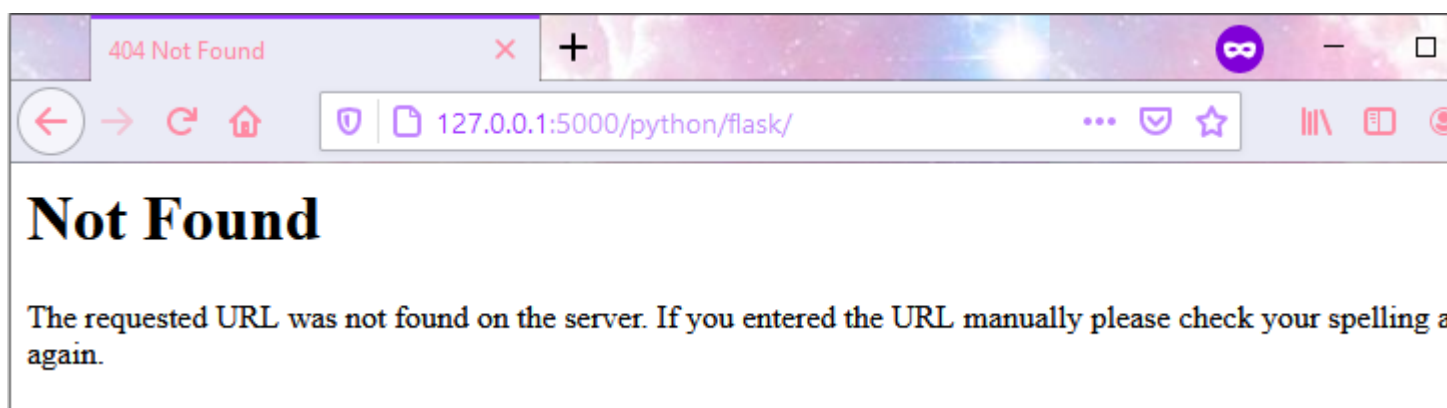


Flask routing creates a URL with the slash at the end if it is explicitly mentioned in the `route()`. Otherwise,

It will work correctly in routing the URL without slash:



But not if the URL entered contains a slash at the end:



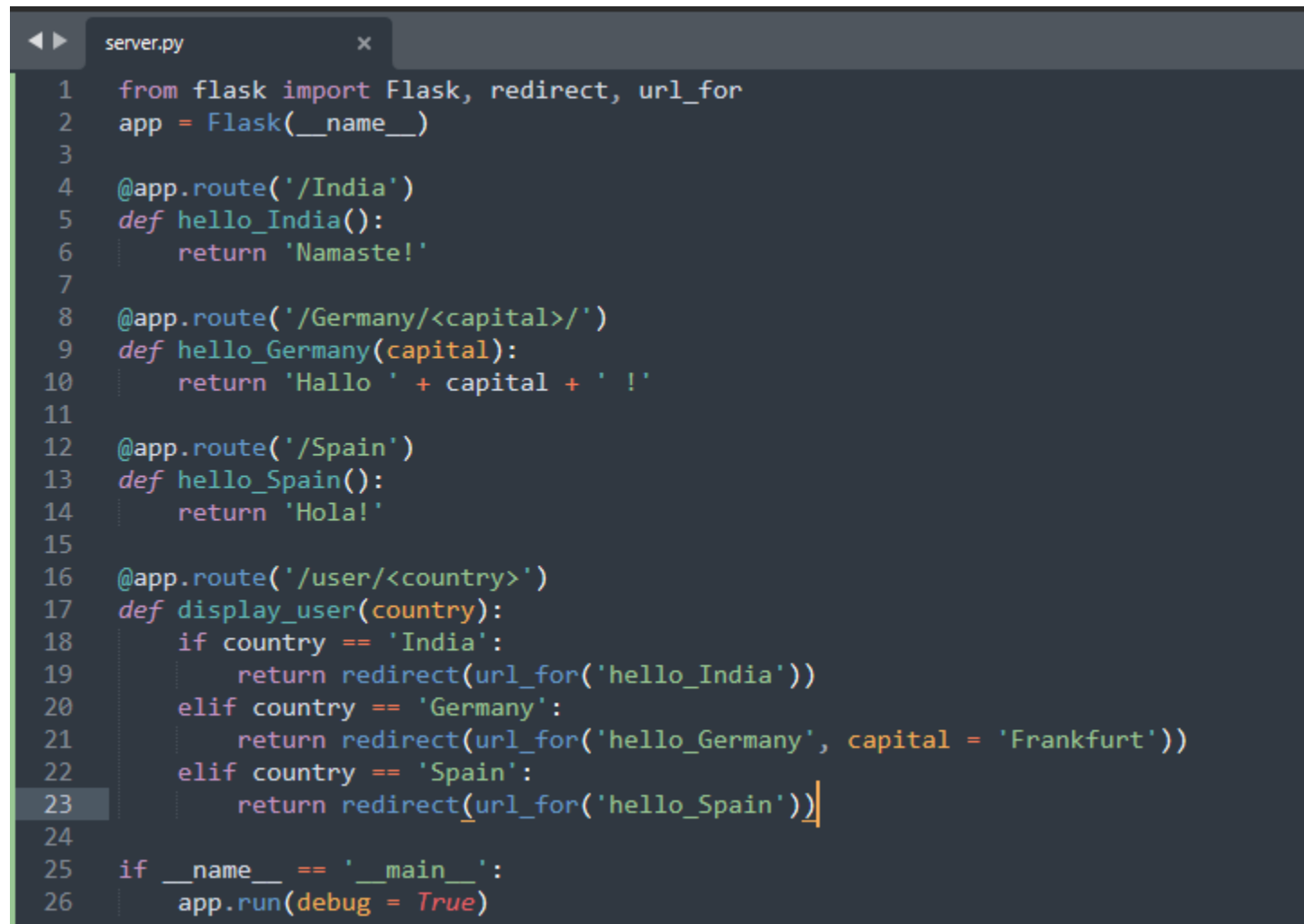
It will result in a **404 Not found error**. This is a useful piece of information to keep in mind.

URL Building

There's another approach to building URLs dynamically in Flask applications. Let's look at `url_for()` method. This is basically the building routing approach in reverse order. The easy way of hard-coding routing is what we have seen using `app.route()` and `app.add_url_rule()` methods.

How Does that work?

Let's see with an example. Shall we?

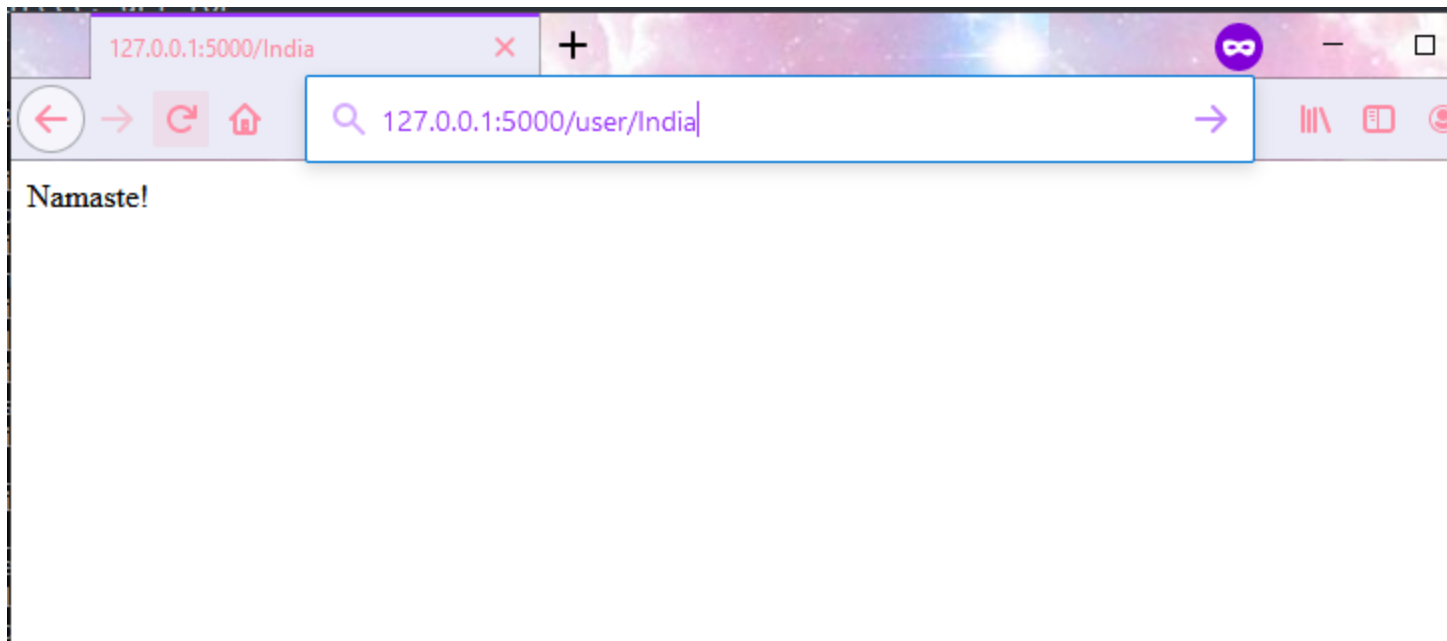


```
1  from flask import Flask, redirect, url_for
2  app = Flask(__name__)
3
4  @app.route('/India')
5  def hello_India():
6      return 'Namaste!'
7
8  @app.route('/Germany/<capital>/')
9  def hello_Germany(capital):
10     return 'Hallo ' + capital + ' !'
11
12 @app.route('/Spain')
13 def hello_Spain():
14     return 'Hola!'
15
16 @app.route('/user/<country>')
17 def display_user(country):
18     if country == 'India':
19         return redirect(url_for('hello_India'))
20     elif country == 'Germany':
21         return redirect(url_for('hello_Germany', capital = 'Frankfurt'))
22     elif country == 'Spain':
23         return redirect(url_for('hello_Spain'))
24
25 if __name__ == '__main__':
26     app.run(debug = True)
```

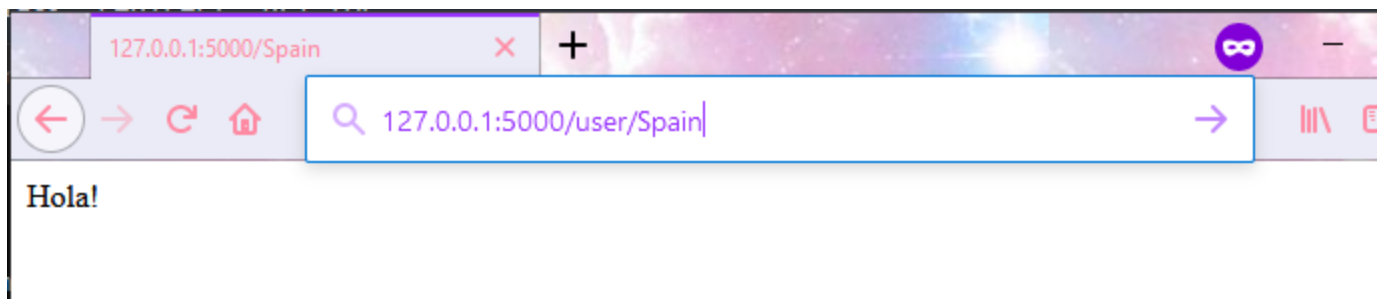
The first thing to note is the additional imports. `redirect` & `url_for()` need to be imported separately, to begin with.

In the above example, we've 3 simple routes and their functions hard-coded in the traditional manner. And, then we have a special function `display_user()`. This

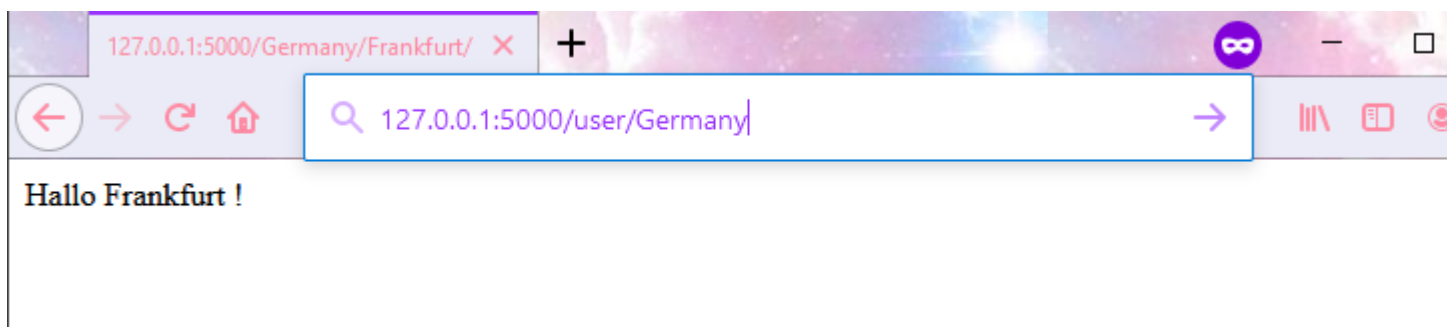
method takes a user-defined input through their URL. Based on the input of the ``country`` variable, ``url_for()`` method redirects it to the appropriate function call.



Similarly, for Spain.



Now, in case of Germany, we're additionally passing ``capital`` variable also to the function.



Why should you consider Dynamic URL Building?

1. Reversing of routing is far more descriptive than hard-coding (hard-coding is bad practice) of routes.
2. This way escape characters & Unicode characters can be handled efficiently.
3. The generated pathways are absolute, removing the uncertainty of unexpected pathways.
4. Changes can be made easily now that the paths are built at the same place.
5. If the app is placed outside of the root, dynamic URL building will handle that as well.

HTTP Methods

HTTP methods are at the core of communication between different parties of the world wide web, one of them being the Flask app you will be designing. These methods are standard across any frameworks, so let's see how Flask handles them.

1. GET

It is the most basic and unencrypted form of sending data to the website by appending the content to the URL. It can be used in cases where data of invaluable nature (which will not be an issue if disclosed) is being shared with the server. It is most commonly used in cases of fetching data or loading a new HTML page.

2. POST

The POST method is the second most used method after GET. The data is encrypted and sent to the server. It is not appended to the URL, it is sent in the body of HTML, most commonly as part of HTML form data. It is also not cached on the system. This is one of the most trusted methods to send sensitive data to the server like login credentials, etc.

3. HEAD

The HEAD method is very similar to the Get method. This is can be cached on the end-user's system, is unencrypted but it is not supposed to have a response body.

For example, a URL request can have a large download attached to it, but, using HEAD method, the URL can request to check the `content-length` from the header alone to get the file size without downloading the entire file.

4. PUT

The PUT method is similar to the POST method, except for the fact, calling a POST method multiple times will send that many requests to the server as opposed to the PUT method, wherein it will just replace the current result or response with the same response.

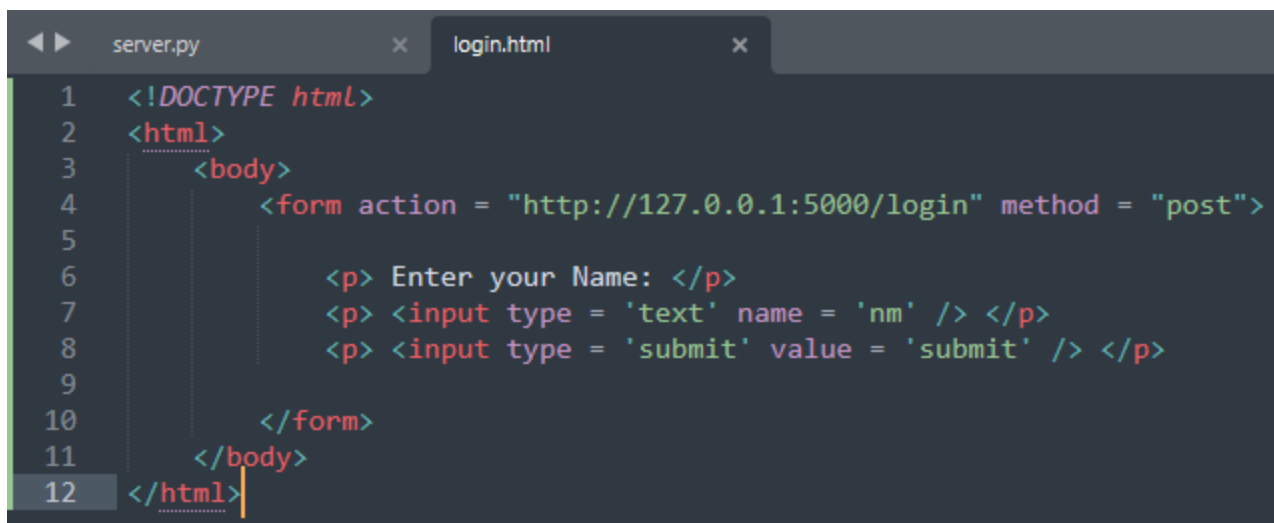
In a nutshell, PUT is used to create a resource most often, so that subsequent requests do not create redundant resources (and end-user receives a `201 created error`) whereas, POST should be used when updating resources, there can be multiple update requests.

5. DELETE

The DELETE sends a request to erase the particular resource to the server.

You can add any of the methods to the route by adding an argument to the `app.route()` method. For example:

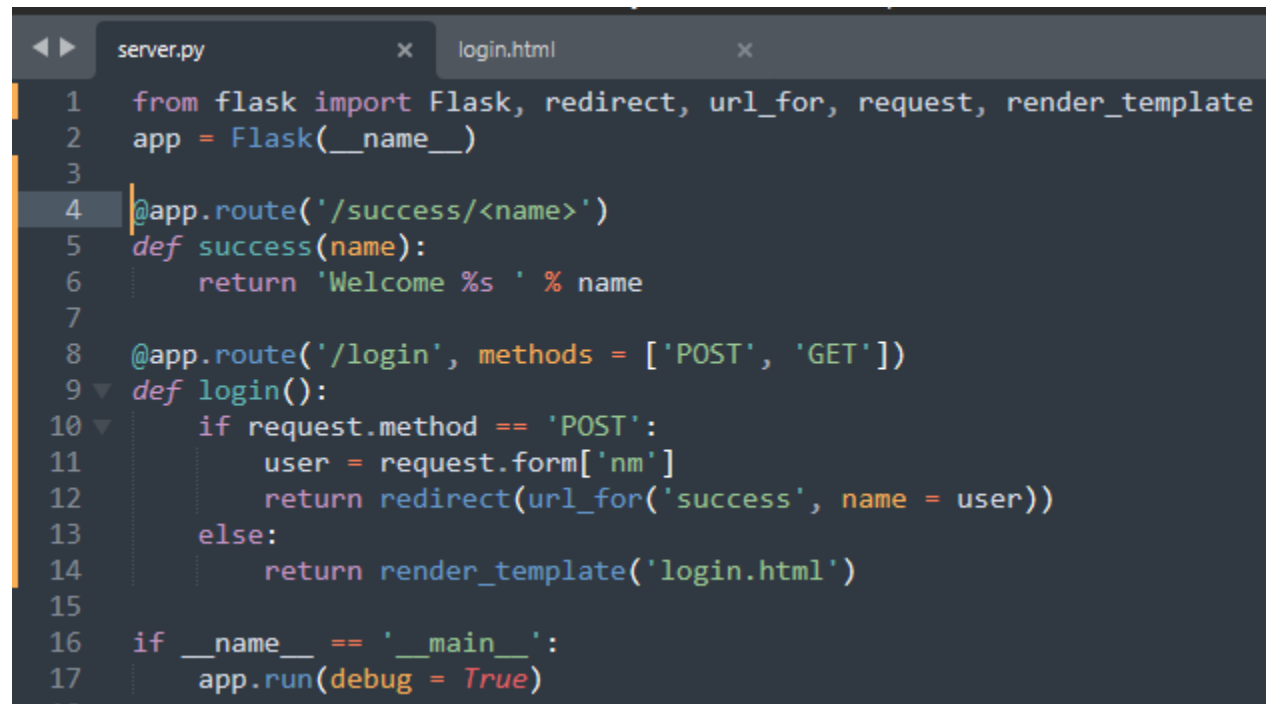
Note: Create a folder in your project folder by the name of `Templates`. Create a `login.html` file inside this folder structure.



```
1  <!DOCTYPE html>
2  <html>
3      <body>
4          <form action = "http://127.0.0.1:5000/login" method = "post">
5              <p> Enter your Name: </p>
6              <p> <input type = 'text' name = 'nm' /> </p>
7              <p> <input type = 'submit' value = 'submit' /> </p>
8          </form>
9      </body>
10 </html>
```

In the `login.html` file, we've created a basic HTML form of **POST** HTTP method type and with one textbox to take user input and one Submit button.

Then, in your `server.py` file make the following changes:



```
1 from flask import Flask, redirect, url_for, request, render_template
2 app = Flask(__name__)
3
4 @app.route('/success/<name>')
5 def success(name):
6     return 'Welcome %s ' % name
7
8 @app.route('/login', methods = ['POST', 'GET'])
9 def login():
10     if request.method == 'POST':
11         user = request.form['nm']
12         return redirect(url_for('success', name = user))
13     else:
14         return render_template('login.html')
15
16 if __name__ == '__main__':
17     app.run(debug = True)
```

In the `server.py` file, we notice that we need an additional Flask functionality to be imported, namely, `render_template`. We'll venture in deep regarding that functionality later. Then, we create a routing URL for `login`. In the `@app.route('/login', methods = ['POST', 'GET'])` we've added the `methods` argument which contains the list of HTTP methods which will be allowed.

In that function, if the HTTP request type is GET, we use `render_template()` functionality to call upon the static HTML page of `login.html`. And, if the request type is POST, we extract the `nm` parameter from the HTML Form and `redirect` the data to `success` URL.

This is one simple example using **GET** and **POST** HTTP method requests. Similarly, any of the HTTP methods can be incorporated based on the need using `method` argument in `@app.route()` decorator.

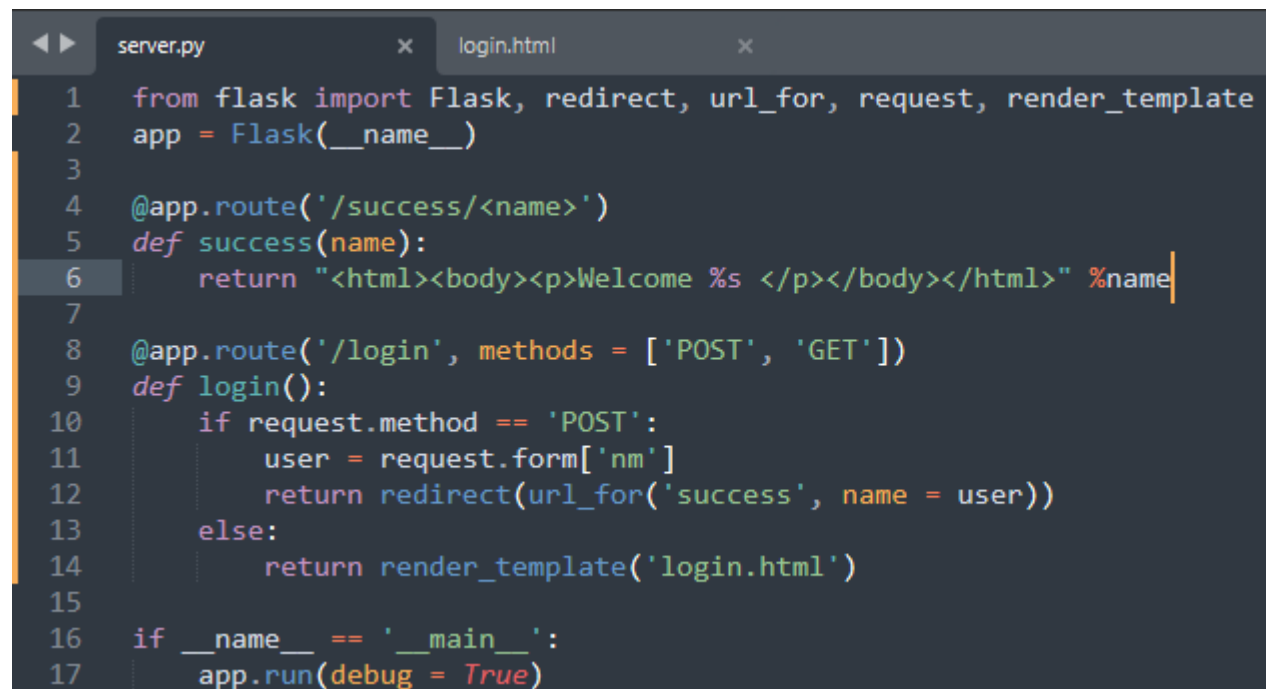
Templates in Flask

Now that we've seen a working example of loading static HTML pages in a flask application, let's explore Templates more.

'**Web Templating system**' refers to displaying an HTML page to the end-user. This revolves around 3 parts:

1. A Core template engine: In this case it is Jinja2
2. Data source: It can be any content hard-coded in HTML or passed to the HTML file
3. Template processor: In this, python expressions will help in processing the templates.

In a simple Flask application, we can hard-code our HTML page from the main python script file. For example:

A screenshot of a code editor with two tabs: 'server.py' and 'login.html'. The 'server.py' tab is active, showing Python code for a Flask application. The code includes imports for Flask, redirect, url_for, request, and render_template. It defines a Flask app, a route for '/success/<name>' with a 'success' function that returns an HTML string, and a route for '/login' with a 'login' function that handles POST and GET requests. The 'login' function either redirects to the success page or renders the 'login.html' template. Finally, it runs the app in debug mode.

```
1 from flask import Flask, redirect, url_for, request, render_template
2 app = Flask(__name__)
3
4 @app.route('/success/<name>')
5 def success(name):
6     return "<html><body><p>Welcome %s </p></body></html>" %name
7
8 @app.route('/login', methods = ['POST', 'GET'])
9 def login():
10     if request.method == 'POST':
11         user = request.form['nm']
12         return redirect(url_for('success', name = user))
13     else:
14         return render_template('login.html')
15
16 if __name__ == '__main__':
17     app.run(debug = True)
```

In the above example, let's replace the content of `success()` function. This gives us the same output as previously.

While this one way of approaching HTML content, most often, this wouldn't be the optimal way for any project other than a Hello World program. Real-life projects will have more complexity.

Project Folder Structure

Before continuing further, let's look at the Project Folder structure. It is important we know this before continuing ahead.

| Name | Date modified | Type | Size |
|------------|-------------------|---------------------|------|
| Include | 6/2/2021 10:22 AM | File folder | |
| Lib | 6/2/2021 10:22 AM | File folder | |
| Scripts | 6/2/2021 11:26 AM | File folder | |
| Templates | 6/8/2021 9:57 AM | File folder | |
| login | 6/8/2021 9:59 AM | Firefox HTML Doc... | 1 KB |
| pyvenv.cfg | 6/2/2021 10:22 AM | CFG File | 1 KB |
| server | 6/8/2021 12:26 PM | Python File | 2 KB |

This is the Flask application project folder. You'll have to create a folder by the name of **Templates** in which you can save all your HTML files. This path structuring is necessary to enable Jinja2 template engine to identify and process the HTML file.

Static Files

Similar to the HTML files, all the necessary static files such as images, videos, CSS files, JS files, etc need to be placed under the folder name of **static**. This needs to be created in the root folder same as **Templates**.

Jinja 2

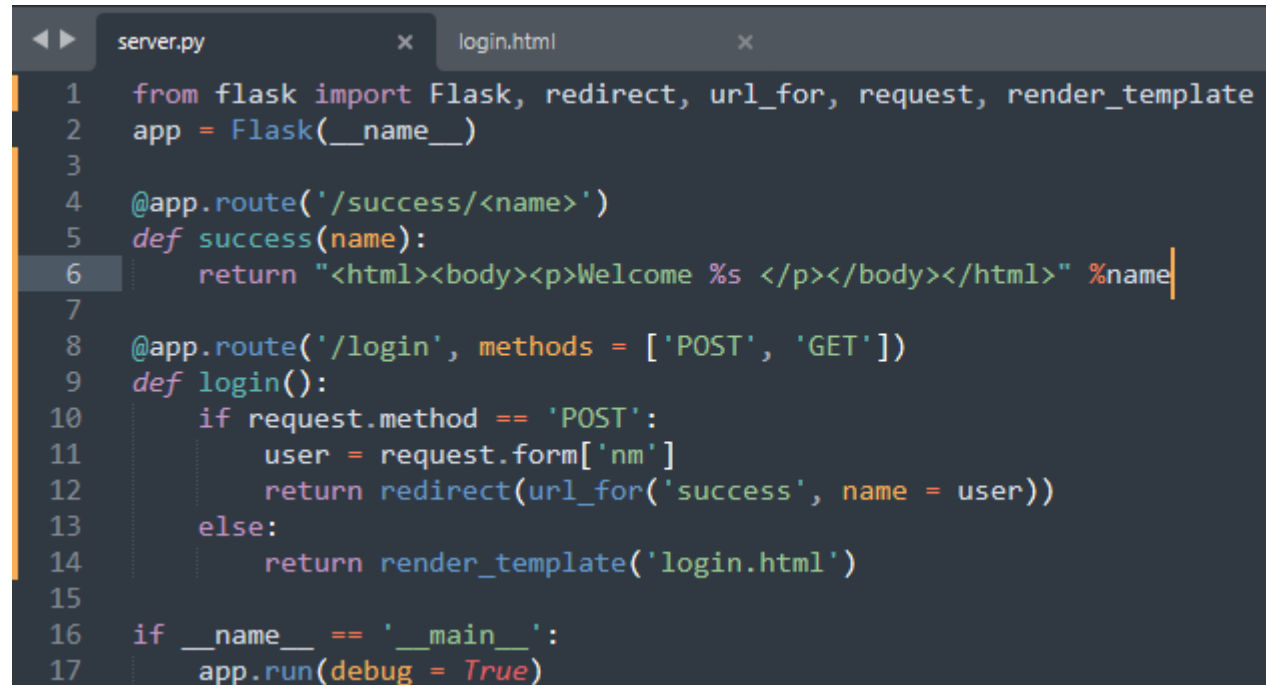
Jinja2 being the web templating system has few templates which would allow us to create Python expressions in the HTML file.

1. `{{ }}` : This will process the Python code written and display accordingly. Example seen earlier.
2. `{ %.....% }` : For statements.
3. `{# ... #}` : For comments not to be included in the template output.

For more detailed read about the same, you can check out [Jinja2 Official Documentation](#).

Rendering Templates

`render_template()` is the Flask function which helps render the HTML file provided to it.

A screenshot of a code editor with two tabs: 'server.py' and 'login.html'. The 'server.py' tab is active, showing Python code for a Flask application. The code includes imports for Flask, redirect, url_for, request, and render_template. It defines a Flask app, a success route, a login route, and a main block to run the app with debug mode enabled. The login route uses render_template to serve 'login.html' when the method is not POST.

```
1 from flask import Flask, redirect, url_for, request, render_template
2 app = Flask(__name__)
3
4 @app.route('/success/<name>')
5 def success(name):
6     return "<html><body><p>Welcome %s </p></body></html>" %name
7
8 @app.route('/login', methods = ['POST', 'GET'])
9 def login():
10     if request.method == 'POST':
11         user = request.form['nm']
12         return redirect(url_for('success', name = user))
13     else:
14         return render_template('login.html')
15
16 if __name__ == '__main__':
17     app.run(debug = True)
```

Referring to the above example, we see the most basic implementation of the function. Only the HTML page's name is passed as argument.

In the templates, you can also access the `request`, `sessions`, `get_flashed_message` and `g` objects also. Each of these can be further looked up in Flask official Documentation. And, templates especially useful when **Template Inheritance** is used.

Request object

The communication from end-user to server running Flask happens through the usage of Flask's global **Request** object. This has to be specifically imported from Flask module. **Request** objects important attributes are:

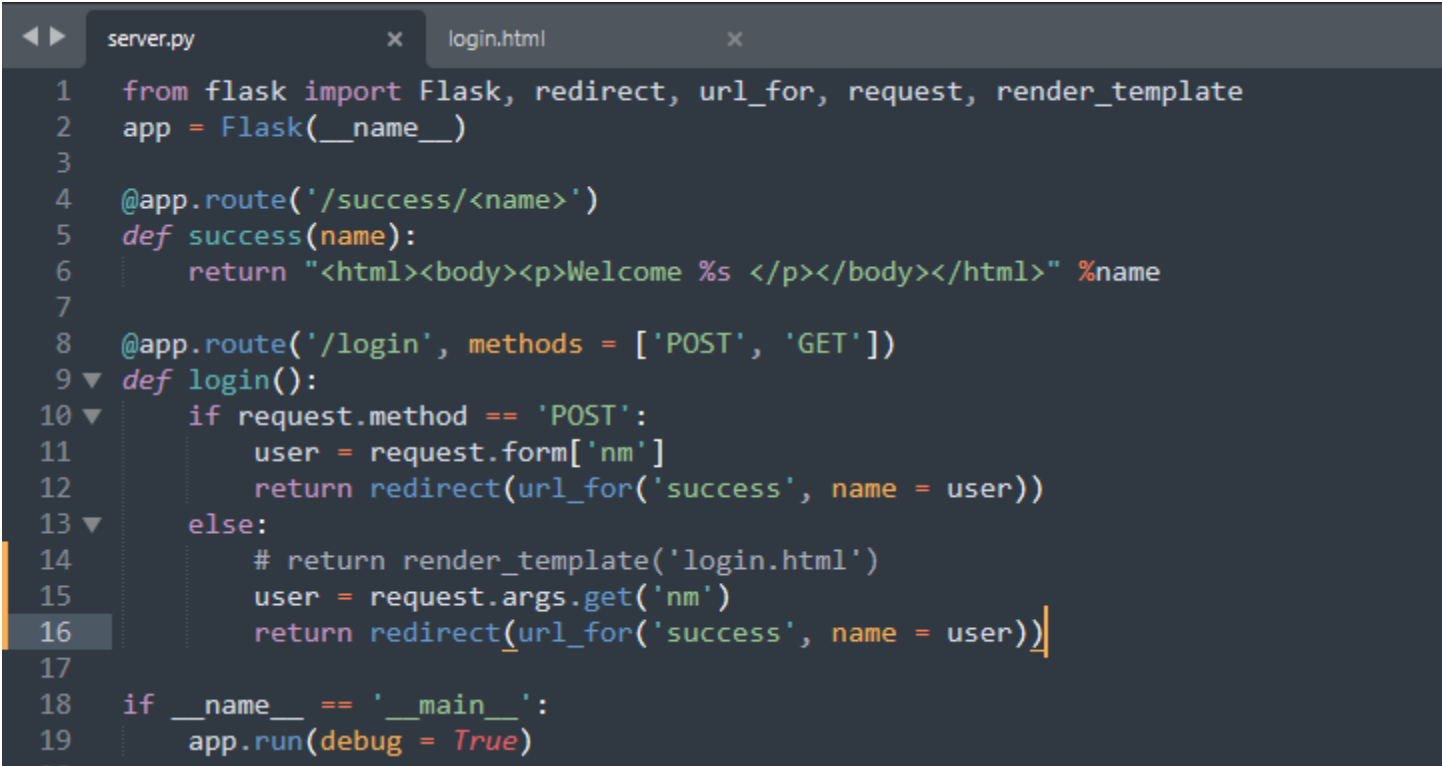
1. **method**:- It defines the HTTP method request object type.

2. **Form**:- It contains a dictionary object of all the Form attributes and its values.
3. **args**:- contents sent as part of query string in the URL after a (?) mark.
4. **files**:- data in which was uploaded to the application.

Of these, we've already seen the **Form** attribute.

```
8 @app.route('/login', methods = ['POST', 'GET'])
9 def login():
10     if request.method == 'POST':
11         user = request.form['nm']
12         return redirect(url_for('success', name = user))
```

In this earlier example, when we used `request.form['nm']` – it requested for a particular value for the key 'nm' from the Form dictionary. There are multiple other ways to get data from the Request object. Let's see the example of **args** attribute.

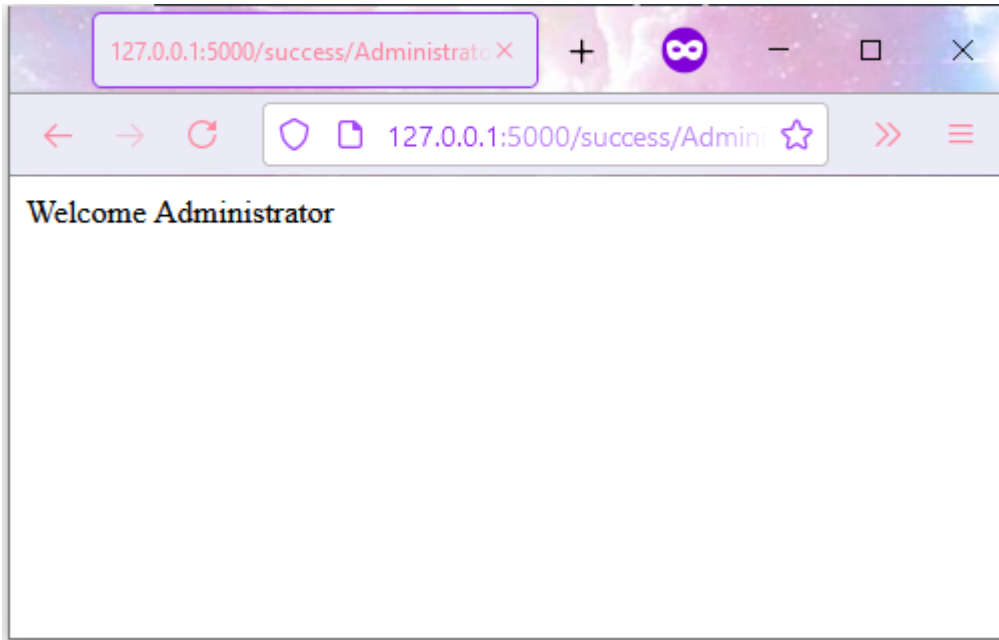


```
server.py x login.html x
1 from flask import Flask, redirect, url_for, request, render_template
2 app = Flask(__name__)
3
4 @app.route('/success/<name>')
5 def success(name):
6     return "<html><body><p>Welcome %s </p></body></html>" %name
7
8 @app.route('/login', methods = ['POST', 'GET'])
9 def login():
10     if request.method == 'POST':
11         user = request.form['nm']
12         return redirect(url_for('success', name = user))
13     else:
14         # return render_template('login.html')
15         user = request.args.get('nm')
16         return redirect(url_for('success', name = user))
17
18 if __name__ == '__main__':
19     app.run(debug = True)
```

This is the same sample re-worked to change the execution when **GET** HTTP method is called. Earlier, we were directly calling the `login.html` using `render_template()`. Now, we're taking an input of username from the **args** attribute of **Request** object.

When running the application, call the URL :

`http://127.0.0.1:5000/login?nm=Administrator`. You'll notice, it redirects you to show the following page:



By appending the value for 'nm' in the query string after a (?), the Flask application wraps it as part of **args** attribute of **Request** object and sends it to the python function linked to it.

Other than **Form** and **args** attribute, we've also seen the **method** attribute, of how the HTTP method can be chosen and to received using **method** argument in `app.route()` decorator.

The only attribute new to us, is **files**. It is used in case when end-user needs to upload a file to the server via application.

Let's explore that with an example.

```

server.py x login.html x upload.html x
1 from flask import Flask, redirect, url_for, request, render_template
2 from werkzeug.utils import secure_filename
3 app = Flask(__name__)
4
5 @app.route('/upload')
6 def upload_view():
7     return render_template('upload.html')
8
9 @app.route('/uploader', methods = ['GET', 'POST'])
10 def uploading():
11     if request.method == 'POST':
12         file = request.files['file']
13         file.save(secure_filename(file.filename))
14         return 'File Uploaded Successfully'
15
16 if __name__ == '__main__':
17     app.run(debug = True)

```

In the above code, first we note `Werkzeug.utils` as an additional package we use to import `secure_filename`. This file is necessary in case you would like to save the file using `secure_filename` function, which is the best practice to retain the file name of the uploaded document as provided by the end-user. You can also do it without using that function, but as the function name suggests, it is not secure otherwise.

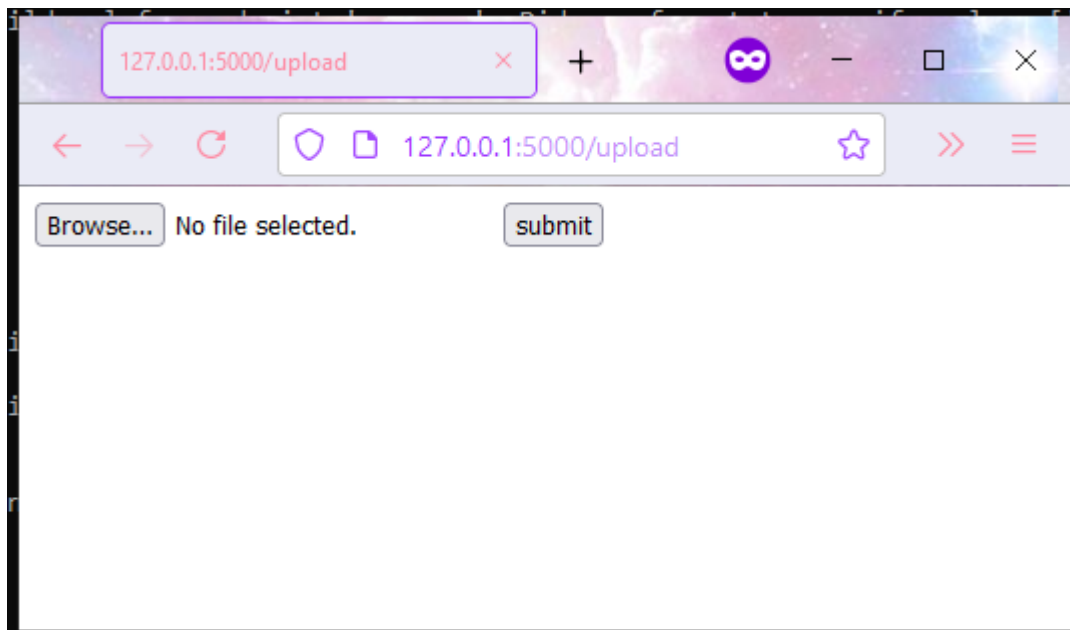
Next, we note that we've created two URLs one, to render the **upload.html** and other to retrieve the file using **Request** object and saving it in the server.

```

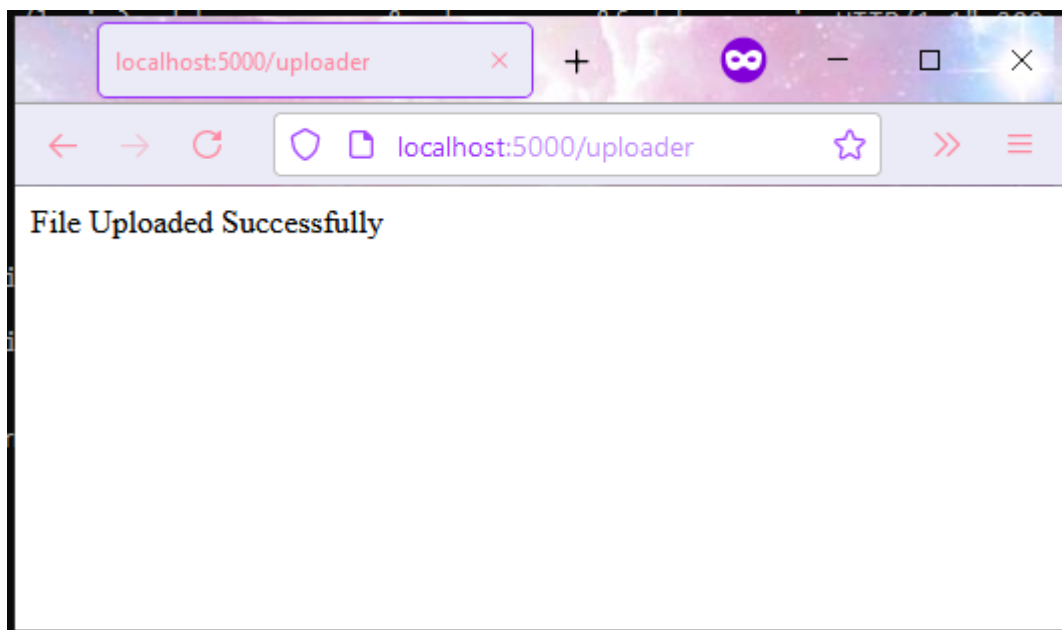
server.py x login.html x upload.html x
1 <html>
2     <body>
3         <form action = "http://localhost:5000/uploader" method = 'POST' enctype
4             <input type = 'file' name = 'file' />
5             <input type = 'submit' value = 'submit' />
6         </form>
7     </body>
8 </html>

```

The above shown is the **upload.html** HTML file. The attribute `enctype = multipart/form-data` in form tag is extremely important, failing which the file will not get uploaded to the application server.



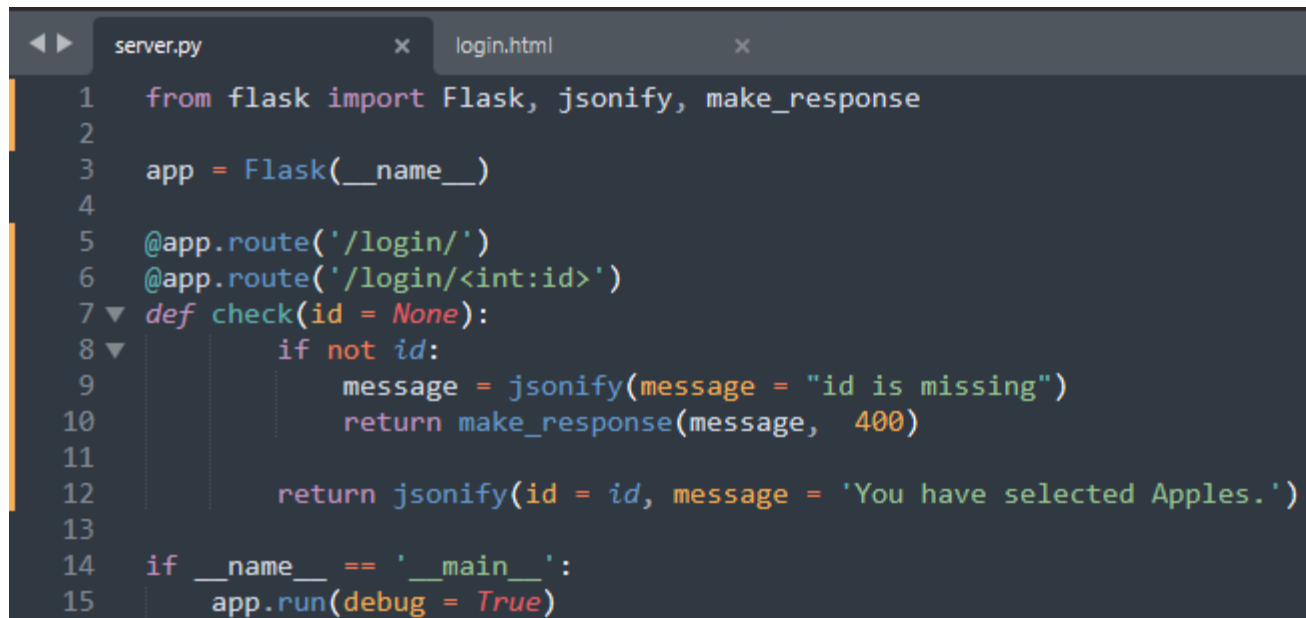
The above is the output of the first URL **upload**. And below is the output after selecting the file and clicking on the Submit button.



Response Object

Using **Request** object, we saw how to interpret the data sent from end-user to the application server. And, **Response** object does the opposite of that, it helps

package and send data from application to the end-user. Flask is the most preferred language for creating API's. And, this is the most important part of it.

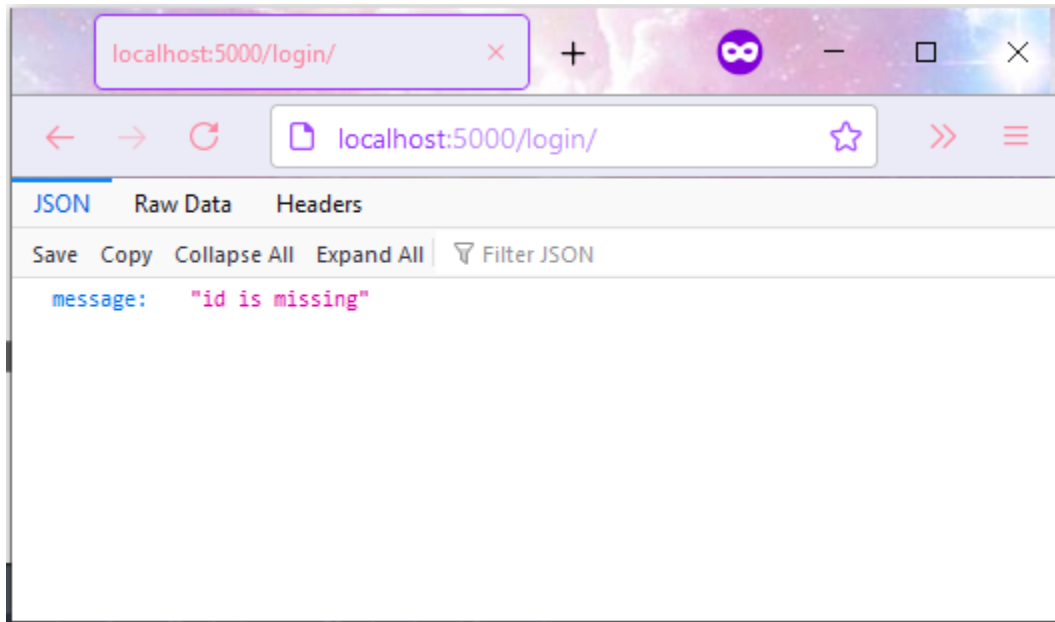


```
server.py x login.html x
1 from flask import Flask, jsonify, make_response
2
3 app = Flask(__name__)
4
5 @app.route('/login/')
6 @app.route('/login/<int:id>')
7 def check(id = None):
8     if not id:
9         message = jsonify(message = "id is missing")
10        return make_response(message, 400)
11
12    return jsonify(id = id, message = 'You have selected Apples.')
13
14 if __name__ == '__main__':
15    app.run(debug = True)
```

As seen above, we have first imported `jsonify` & `make_response` global objects from Flask. We, the create a URL function `check()` with `id` argument which can have a default of `None` type. Inside this function, we check for `id` argument:

1. If `id` has not been provided, we use `jsonify()` method to create a dictionary with a message which we send back with **statuscode** 400 wrapped in a `make_response()` object.
2. If `id` is provided, we just send back a dictionary with two sets of key-value pair.

The below is output for (i).



The below is output for (ii).

