Flask

Flask is a web framework in the sense that it is a Python module that facilitates the development of simple web applications. It has a simple and extensible core: a microframework without an ORM (Object Relational Manager) or similar capabilities.

What is a Web Framework?

A Web Application Framework, or simply a Web Framework, is a set of frameworks and modules that enables web application developers to construct apps without worrying about low-level issues like protocol or thread management.

What is Flask?

Flask is a Python web application framework. Armin Ronacher led a team of international Python enthusiasts named Poocco in developing it. Flask is a web application framework built on the Werkzeug WSGI toolkit and the Jinja2 template engine. Both are Pocco endeavours.
WSGI
The Online Server Gateway Interface (WSGI) has been adopted as a de facto standard for developing Python web applications. WSGI defines a standard interface for web servers and web applications.

Werkzeug

This enables the construction of a web frame on top of it. Werkzeug is the foundation for the Flask framework.

jinja2

jinja2 is a popular Python template engine.
A web template system generates a dynamic web page by combining a template and a given data source.

Microframework

Frequently, Flask is referred to as a microframework. It is intended to keep the application's core scalable and straightforward.

Rather than an abstraction layer for database support, Flask allows extensions that enable the application to incorporate such capabilities.

Why use a flask?

It would help if you now understand when we refer to Flask as a micro-framework built on WSGI and the Jinja 2 template engine.

The perks of Flask include the following:
Ease of installation and use.
Freedom to design the web application's structure.
As with freedom comes responsibility, Flask requires developers to properly build it, as Flask lacks "flask rules" to follow compared to frameworks such as Django. As the web application becomes more complex, this structure will serve as the foundation.

Setting up Environment

The first step in installing Flask on your system is to install Python2.7 or a later version. Please read on for instructions on how to install Python on a Windows 10 PC.

You can use this Official Python website to determine the correct Python version to install based on your operating system and system configuration. You can choose the most recent Python version; as of writing this article, the most recent is 3.9.10. Please verify that pip has been appropriately configured during Python installation.

'pip -V' is the command to verify this.

Setting up your virtual environment

Utilizing a virtual environment for a Python-based project is a recommended best practice. In the case of Linux, it ensures that other python packages are not installed and used at the OS level. Additionally, in the case of other operating systems, this ensures that the virtual environment's settings and files remain intact, which is extremely important when various projects require different versions of the same packages.

In Python2, the tool of choice for building virtual environments was 'virtualenv'. However,' venv' is the Python3 and later tool. 'virtualenv' is a third-party package, whereas 'venv' is a standard Python 3.x feature. Here, we'll use venv to create a virtual environment.

To use 'venv', you need to ensure that you have a Python 3. x version installed; x being any version greater than 1.

To build a virtual environment, type' python -m venv.'

If you also have Python 2. x installed on your system, try using 'python3' rather than Python. This is not a problem because Python 3.9.5 is the default version installed in this circumstance.

Because I've already gone to the folder where I want to establish my virtual environment, I used '.' to point to it. Additionally, you can enter the complete path to the directory in which you wish to establish your virtual environment.

Once done, the cursor will return to its initial state, displaying no output. You can use the following command to verify that the relevant files and folders have been created:

Steps for activating your virtual machine on Windows OS are as follows:

Using the command 'cd Scripts', navigate to the folder 'Scripts' in the virtual environment directory.

After that, execute the command 'activate'.

This is the appearance of an activated virtual environment, regardless of the operating system.

When it comes to Mac OS or Linux OS,

`source env name/bin/activate`

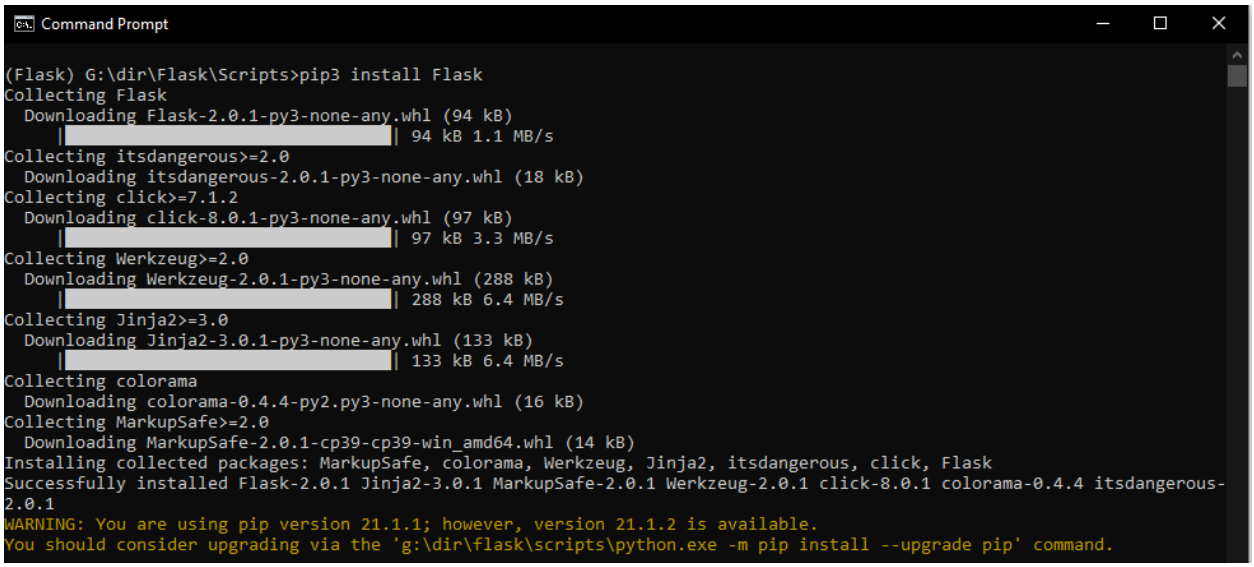
Additionally, to deactivate in any operating system.

`deactivate`

Let's now install Flask:

'pip3 Flask install.'

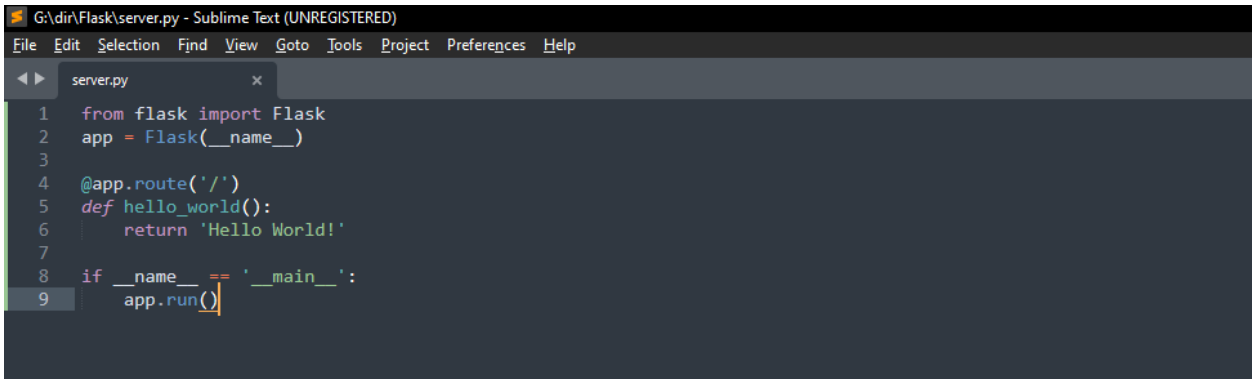
The output will look like this:



Your 1st Flask Application

To begin, let's create a boilerplate application using the basic configuration as described in the Flask instructions.

In the virtual environment, create a file named server.py' and write the following code:

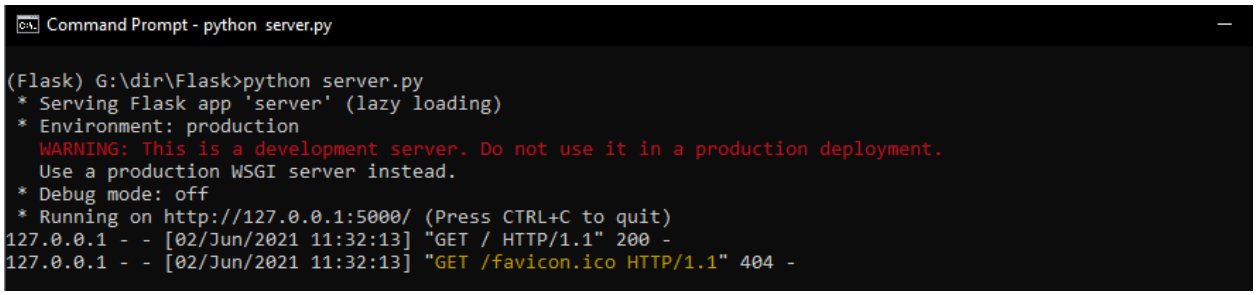


Save this file and execute it via the command prompt, just like any other python file.

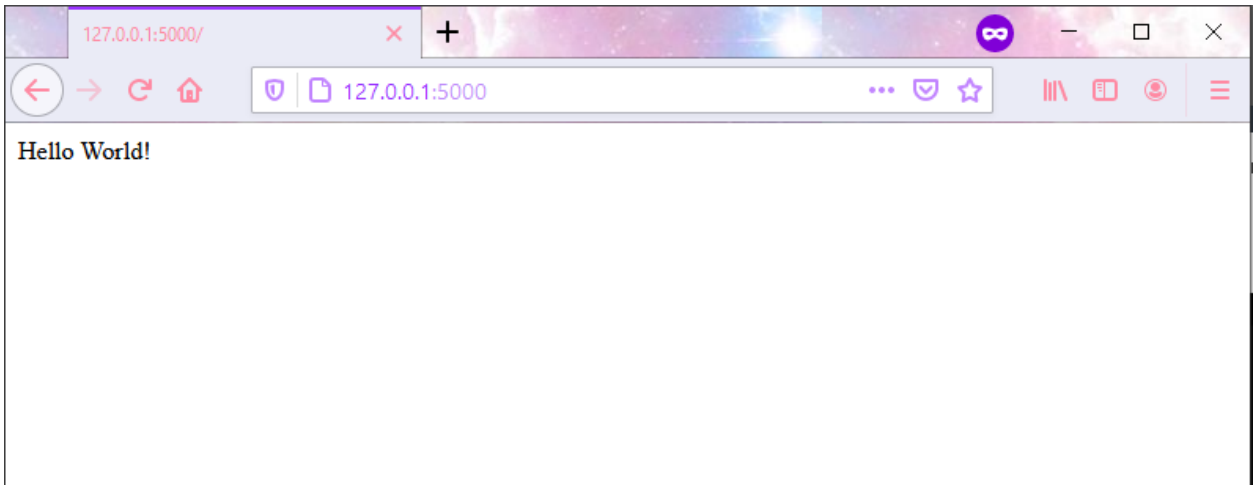
`python server.py`

To avoid confusion, do not save this as 'Flask.' That would cause it to be confused with the Flask itself.

It will output the following:



Open the localhost URL specified in the command prompt to view output similar to this:



You've successfully run your first Flask application! If you followed all of the procedures exactly, there should be no issues running the Flask application.

Let us examine the program's syntax.
The first line of code, 'from flask import Flask', imports the Flask package into the program.
Following that, we use 'app = flask()' to create an instance of the Flask class.
The following input is passed to the default function Object() { [native code] } '__name__'; this instructs Flask to seek for templates and static files.
Following that, we'll utilize the route decorator to specify which routes should be used to reach the following function. `@app.route(/)`
When we use '/' in this, we instruct Flask to deliver all default traffic to the following function.

We define the function to be performed and the action to be taken.
Since HTML is the default content type, our string will be wrapped in HTML and shown in the browser.
'app. run()' in the primary function generated will start the server on the local development server.

Debugging

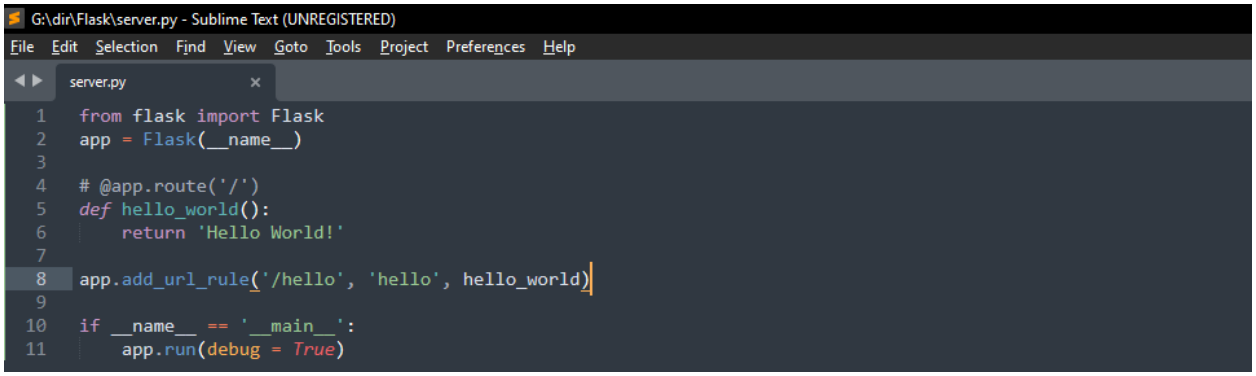
It is critical to consider using the debugging mode when designing a Flask application for two reasons:
It will track down and display errors if they occur.
Each time a change is made, the app must be restarted to examine it; to avoid this, the app can be started in debugging mode, where changes are immediately reflected.

To debug the code, you must include the following:

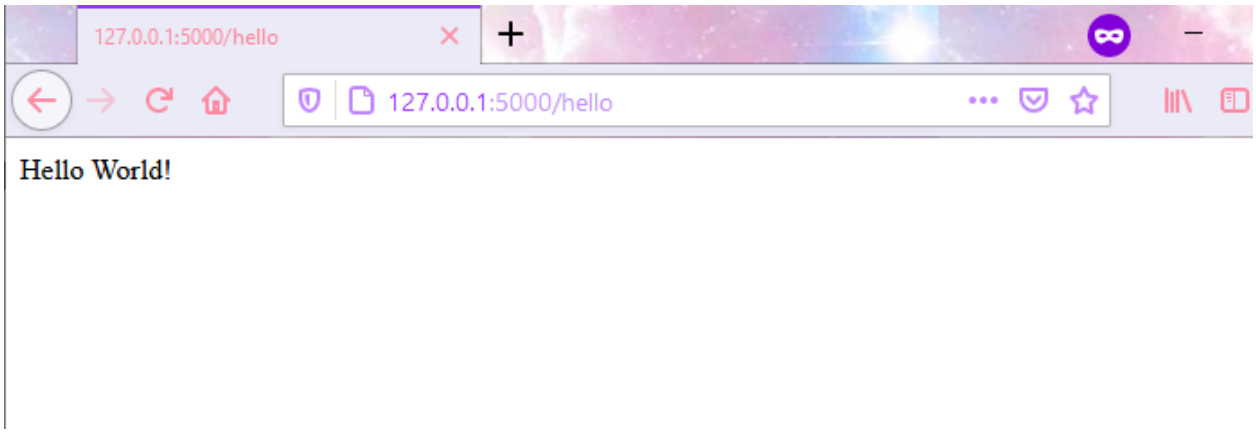
```
if __name__ == '__main__':
    app.run(debug = True)
```

Routes

Routes As previously stated, 'route()' is a Flask decorator:
`app.route(rule, options)`
Using the rule, you can give a URL for the function to be called, as well as a list of possible parameters.
Additionally, you can use 'app. add_URL_rule()' to associate a function with the URL.



The effect is the same as routing to '/', which refers to the application's homepage.



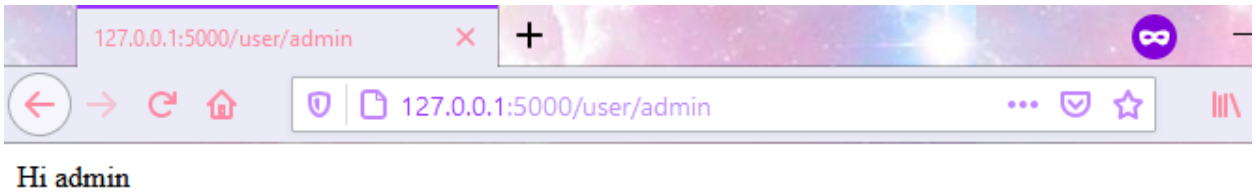
Dynamic Routing

Additionally, we may pass arguments to the python method via 'app. route()'. Here is an example of the code and the result.

server.py

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

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

@app.route('/user/<name>')
def greetings(name):
    return 'Hi ' + name

app.add_url_rule('/hello', 'hello', hello_world)

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

As a result, in the above situation, the default variable given was a 'string'. The Werkzeug implementation supports other variable types. The same information is available in the Flask Official Documentation.



'float' accepts positive floating-point values
'path': similar to string but accepts slashes.
'uuid': accepts the (universally unique identifier) UUID identifiers
'string': (default) accepts any string that does not contain a slash.

Using the '< >' and ':' symbols, you will establish a dynamic Flask URL routing. The variable name should be part of '< >', and the variable type can be specified using a colon (':') within the same brackets. Consider the following:

server.py

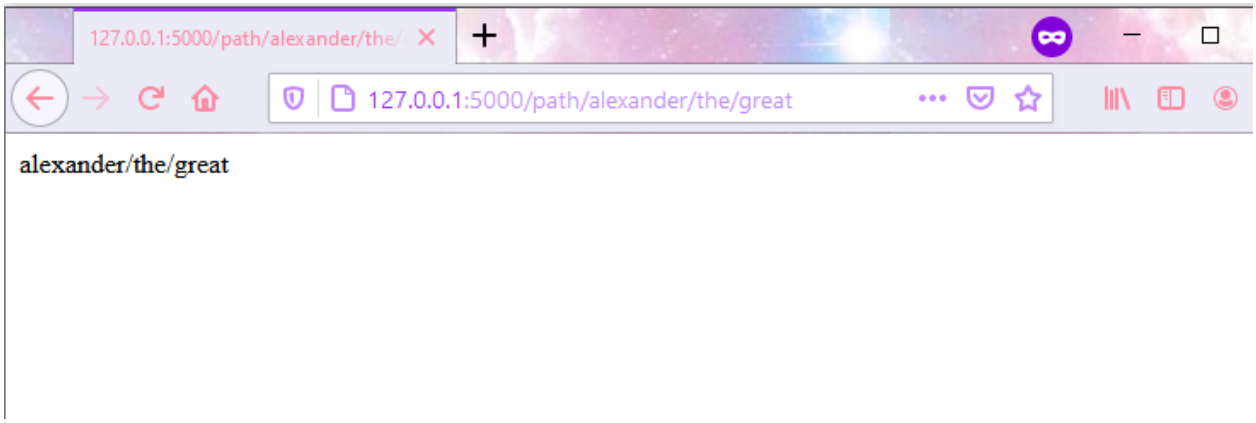
```
from flask import Flask
app = Flask(__name__)

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

@app.route('/user/<string:name>')
def greetings(name):
    return 'Hi ' + name

app.add_url_rule('/hello', 'hello', hello_world)

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

The result remains unaltered.
Consider a few further instances utilizing the 'uuid' and 'path' variable types.

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

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

@app.route('/user/<string:name>')
def greetings(name):
    return 'Hi ' + name

@app.route('/uniqueid/<uuid:api_key>')
def display_key(api_key):
    return "The API Key " + str(api_key)

@app.route('/path/<path:sub_path>')
def display_path(sub_path):
    # Accepts any string with slashes
    return sub_path

app.add_url_rule('/hello', 'hello', hello_world)

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

The following is the result for 'uuid'-based results:

The following is the result for 'path'-based results:

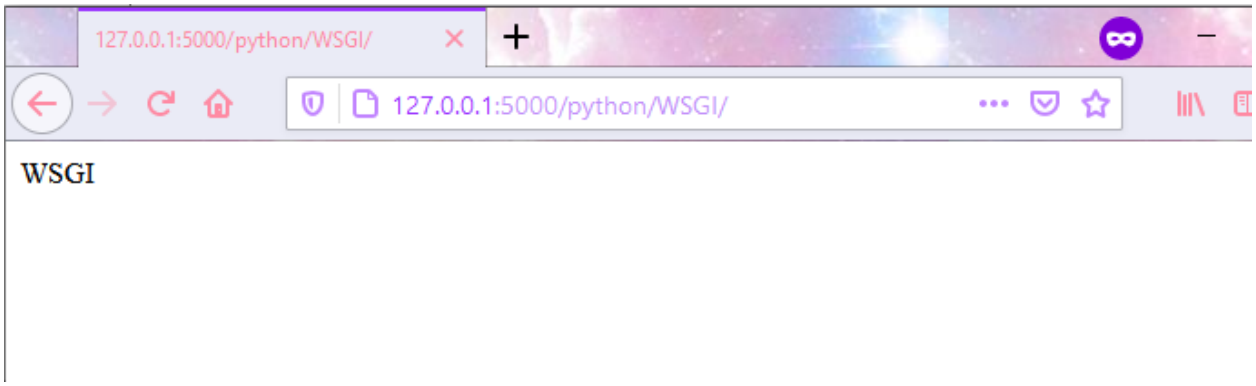


Now that we've covered the fundamentals of Flask routing. There is one other critical concept that we must understand. The significance of the forward-slash ('/') in dynamic routing. Consider the following example.

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)
```

We observe two routes in the following piece of code: one that does not end in a slash('/python/flask') and one that does end in a slash('/python/WSGI/'). The following are the results of the route ending in a slash:



Flask routing creates a URL with a trailing slash if specified explicitly in the 'route()' method. Otherwise, it will correctly route the URL without '/ ' :

However, this is not the case if the URL entered contains a trailing slash at the end:

As a result, you will receive a 404 Not Found error. This is critical knowledge to retain.

URL Building

There is another way to generate URLs in Flask applications dynamically. Consider the 'url for()' technique. This is essentially the approach to creating routing in reverse order. The simplest approach of hard-coding routing is demonstrated by using the 'app. route()' and 'app.add URL rule()' methods.

How Does that work?

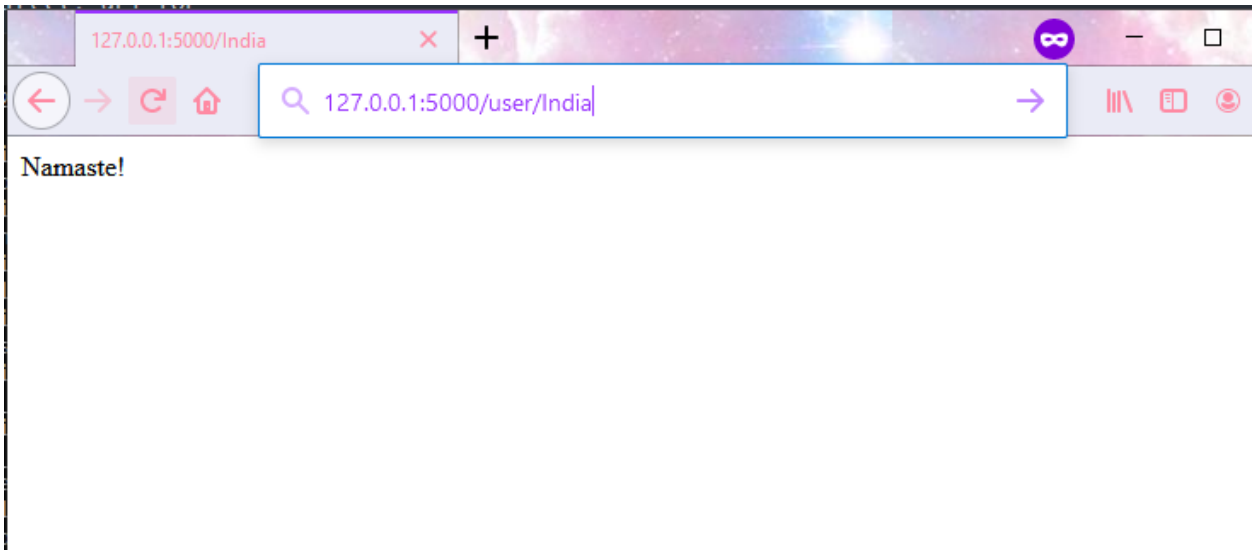
Consider the following example.

server.py

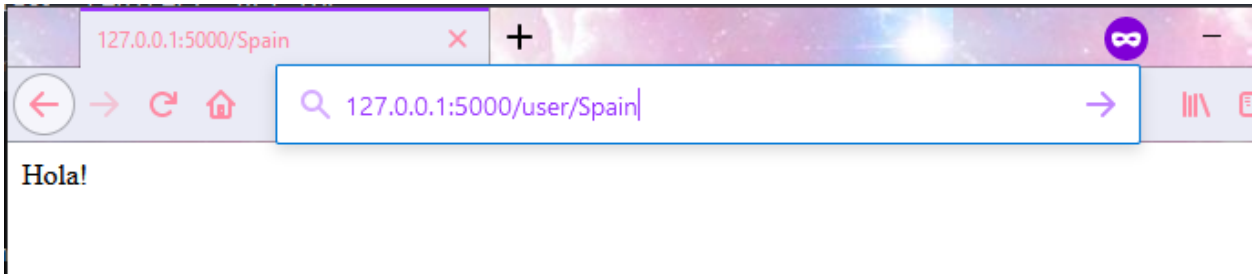
```python
from flask import Flask, redirect, url_for
app = Flask(__name__)

@app.route('/India')
def hello_India():
    return 'Namaste!'

@app.route('/Germany/<capital>/')
def hello_Germany(capital):
    return 'Hallo ' + capital + ' !'

@app.route('/Spain')
def hello_Spain():
    return 'Hola!'

@app.route('/user/<country>')
def display_user(country):
    if country == 'India':
        return redirect(url_for('hello_India'))
    elif country == 'Germany':
        return redirect(url_for('hello_Germany', capital = 'Frankfurt'))
    elif country == 'Spain':
        return redirect(url_for('hello_Spain'))

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

The first point to make is the increased imports. `redirect` & `URL for()` need to be imported separately, to begin with.

In the preceding example, we've hard-coded three simple routes and their associated functions. Then there is a specific function called 'display_user()'. This function accepts user-defined parameters via their URL. The 'url_for()' method redirects the 'country' variable to the proper function call based on its value.



Likewise, for Spain.



Now, for Germany, we're also passing the 'capital' variable to the function.