Flask Rendering Templates

Last Updated : 21 Jun, 2023

Flask is a backend web framework based on the Python programming language. It basically allows the creation of web applications in a Pythonic syntax and concepts. With Flask, we can use Python libraries and tools in our web applications. Using Flask we can set up a web server to load up some basic HTML templates along with Jinja2 templating syntax. In this article, we will see how we can render the HTML templates in Flask.

Rendering a Template in a Flask Application

Setting up Flask is quite easy. We can use a virtual environment to create an isolated environment for our project and then install the Python packages in that environment. After that, we set up the environment variables for running Flask on the local machine. This tutorial assumes that you have a Python environment configured, if not please follow through for setting up Python and pip on your system. Once you are done, you are ready to develop Flask applications.

Setting up the Virtual Environment

To set up a virtual environment, we can make use of the Python Package Manager “pip” to install the “virtualenv” package.

```
pip install virtualenv
```

This will install the package “virtualenv” on your machine. The pip command can be different on the version of your Python installed so please do look at the different syntax of the pip for your version here.

Creating Virtual Environment:

After the package has been installed we need to create a virtual environment in our project folder. So you can locate an empty folder where you want to create the Flask application or create an empty folder in your desired path. To create the environment we simply use the following command.

```
virtualenv venv
```

Here, venv is the name of the environment, after this command has been executed, you will see a folder named "venv" in the current folder. The name "venv" can be anything("env") you like but it is standard to reference a virtual environment at a production level.

Activating Virtual Environment:

Now after the virtual env has been set up and created, we can activate it by using the commands in CMD\Powershell or Terminal:

Note: You need to be in the same folder as the "venv" folder.

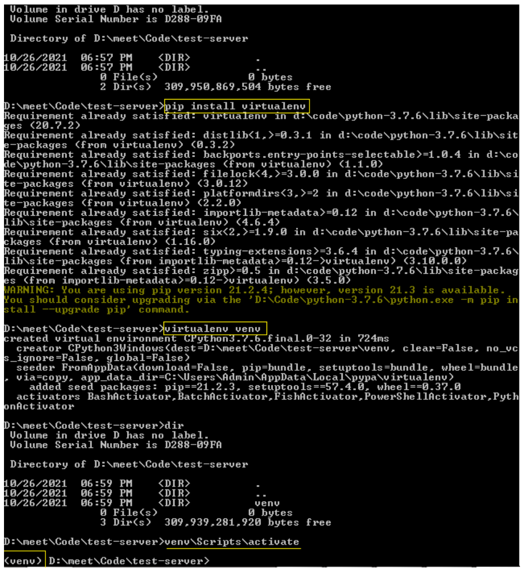
For Windows:

```
venv\Scripts\activate
```

For Linux/macOS:

```
source venv/bin/activate
```

This should activate the virtualenv with "(venv)" before the command prompt.



```
 Volume in drive D has no label.
 Volume Serial Number is D288-09FA

 Directory of D:\meet\Code\test-server

10/26/2021  06:57 PM    <DIR>          .
10/26/2021  06:57 PM    <DIR>          ..
               0 File(s)              0 bytes
               2 Dir(s)  309,950,869,504 bytes free

D:\meet\Code\test-server>pip install virtualenv
Requirement already satisfied: virtualenv in d:\code\python-3.7.6\lib\site-packa
ges (20.7.2)
Requirement already satisfied: distlib<1,>=0.3.1 in d:\code\python-3.7.6\lib\sit
e-packages (from virtualenv) (0.3.2)
Requirement already satisfied: backports.entry-points-selectable>=1.0.4 in d:\co
de\python-3.7.6\lib\site-packages (from virtualenv) (1.1.0)
Requirement already satisfied: filelock<4,>=3.0.0 in d:\code\python-3.7.6\lib\si
te-packages (from virtualenv) (3.0.12)
Requirement already satisfied: platformdirs<3,>=2 in d:\code\python-3.7.6\lib\si
te-packages (from virtualenv) (2.2.0)
Requirement already satisfied: importlib-metadata>=0.12 in d:\code\python-3.7.6\
lib\site-packages (from virtualenv) (4.6.4)
Requirement already satisfied: six<2,>=1.9.0 in d:\code\python-3.7.6\lib\site-pa
ckages (from virtualenv) (1.16.0)
Requirement already satisfied: typing-extensions>=3.6.4 in d:\code\python-3.7.6\
lib\site-packages (from importlib-metadata>=0.12->virtualenv) (3.10.0.0)
Requirement already satisfied: zipp>=0.5 in d:\code\python-3.7.6\lib\site-packag
es (from importlib-metadata>=0.12->virtualenv) (3.5.0)
WARNING: You are using pip version 21.2.4; however, version 21.3 is available.
You should consider upgrading via the 'D:\Code\python-3.7.6\python.exe -m pip in
stall --upgrade pip' command.

D:\meet\Code\test-server>virtualenv venv
created virtual environment CPython3.7.6.final.0-32 in 724ms
  creator CPython3Windows(dest=D:\meet\Code\test-server\venv, clear=False, no_vc
s_ignore=False, global=False)
  seeder FromAppData(download=False, pip=bundle, setuptools=bundle, wheel=bundle
, via=copy, app_data_dir=C:\Users\Admin\AppData\Local\pypa\virtualenv)
    added seed packages: pip==21.2.3, setuptools==57.4.0, wheel==0.37.0
  activators BashActivator,BatchActivator,FishActivator,PowerShellActivator,Pyth
onActivator

D:\meet\Code\test-server>dir
 Volume in drive D has no label.
 Volume Serial Number is D288-09FA

 Directory of D:\meet\Code\test-server

10/26/2021  06:59 PM    <DIR>          .
10/26/2021  06:59 PM    <DIR>          ..
10/26/2021  06:59 PM    <DIR>          venv
               0 File(s)              0 bytes
               3 Dir(s)  309,939,281,920 bytes free

D:\meet\Code\test-server>venv\Scripts\activate

(venv) D:\meet\Code\test-server>
```

Screenshot of the entire virtualenv setup

As we can see we have successfully created the virtualenv in Windows Operating System, in Linux/macOS the process is quite similar. The (venv) is indicating the current instance of the terminal/CMD is in a virtual environment, anything installed in the current instance of a terminal using pip will be stored in the venv folder without affecting the entire system.

Installing Flask:

After the virtual environment has been set up, we can simply install Flask with the following command:

```
pip install flask
```

This should install the actual Flask Python package in the virtual environment.

Adding Flask to Environment Variables: We need to create an app for Flask to set it as the starting point of our application. We can achieve this by creating a file called "**server.py**" You can call this anything you like, but keep it consistent with other Flask projects you create. Inside the server.py paste the following code:

Python

```python
from flask import Flask

app = Flask(__name__)

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

This is the code for actually running and creating the Flask app. This is so-called the entry point of a Flask web server. As you can see we are importing the Flask module and instantiating with the current file name in "Flask(__name__)". Hence after the check, we are running a function called run().

After this, we need to set the file as the Flask app to the environment variable.

For Windows:

```
set FLASK_APP=server
```

For Linux/macOS:

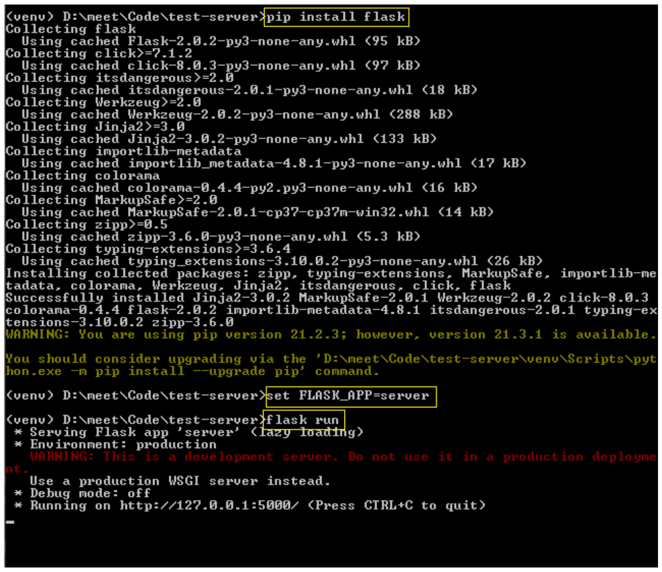
```
export FLASK_APP=server
```

Now, this will set up the Flask starting point to that file we created, so once we start the server the Flask server will find the way to the file “server.py”

To run the server, enter the command :

```
flask run
```

This will run the server and how smartly it detected the server.py file as our actual flask app. If you go to the URL “http://localhost:5000”, you would see nothing than a Not Found message this is because we have not configured our web server to serve anything just yet. You can press **CTRL + C** to stop the server



```
(venv) D:\meet\Code\test-server>pip install flask
Collecting flask
  Using cached Flask-2.0.2-py3-none-any.whl (95 kB)
Collecting click>=7.1.2
  Using cached click-8.0.3-py3-none-any.whl (97 kB)
Collecting itsdangerous>=2.0
  Using cached itsdangerous-2.0.1-py3-none-any.whl (18 kB)
Collecting Werkzeug>=2.0
  Using cached Werkzeug-2.0.2-py3-none-any.whl (288 kB)
Collecting Jinja2>=3.0
  Using cached Jinja2-3.0.2-py3-none-any.whl (133 kB)
Collecting importlib-metadata
  Using cached importlib_metadata-4.8.1-py3-none-any.whl (17 kB)
Collecting colorama
  Using cached colorama-0.4.4-py2.py3-none-any.whl (16 kB)
Collecting MarkupSafe>=2.0
  Using cached MarkupSafe-2.0.1-cp37-cp37m-win32.whl (14 kB)
Collecting zipp>=0.5
  Using cached zipp-3.6.0-py3-none-any.whl (5.3 kB)
Collecting typing-extensions>=3.6.4
  Using cached typing_extensions-3.10.0.2-py3-none-any.whl (26 kB)
Installing collected packages: zipp, typing-extensions, MarkupSafe, importlib-me
tadata, colorama, Werkzeug, Jinja2, itsdangerous, click, flask
Successfully installed Jinja2-3.0.2 MarkupSafe-2.0.1 Werkzeug-2.0.2 click-8.0.3
colorama-0.4.4 flask-2.0.2 importlib-metadata-4.8.1 itsdangerous-2.0.1 typing-ex
tensions-3.10.0.2 zipp-3.6.0
WARNING: You are using pip version 21.2.3; however, version 21.3.1 is available.

You should consider upgrading via the 'D:\meet\Code\test-server\venv\Scripts\pyt
hon.exe -m pip install --upgrade pip' command.

(venv) D:\meet\Code\test-server>set FLASK_APP=server

(venv) D:\meet\Code\test-server>flask run
 * Serving Flask app 'server' (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployme
nt.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
_
```

Flask set up for webserver

Creating Templates in a Flask Application

Now, we can move on to the goal of this article i.e. to render the template. To do that we need to first create the templates, you can use any HTML template but for simplicity, I am going with a basic HTML template. Before that, **create a folder called “templates” in the current folder.** Inside this “**templates**” folder, all of the templates will be residing. Now let us create a basic HTML template: This template must have some Jinja blocks that can be optionally replaced later. We start with a single block called the body.

templates\index.html

HTML

```
<!DOCTYPE html>
<html>

<head>
    <title>FlaskTest</title>
</head>

<body>
    <h2>Welcome To GFG</h2>
    <h4>Flask: Rendering Templates</h4>
 <!-- this section can be replaced by a child document -->
{% block body %}


<p>This is a Flask application.</p>


{% endblock %}


</body>

</html>
```

Adding Routes and Rendering Templates

A route is a mapping of a URL with a function or any other piece of code to be rendered on the webserver. In the Flask, we use the function decorate @app.route to indicate that the function is bound with the URL provided in the parameter of the route function.

Creating the basic route: In this case, we are binding the **URL "/"** which is the base URL for the server with the **function "index"**, you can call it whatever you like but it makes more sense to call it index here. The function simply returns something here it calls the function render_template. The render_template finds the app by default in the templates folder. So, we just need to provide the name of the template instead of the entire path to the template. The index function renders a template index.html and hence we see the result in the browser.

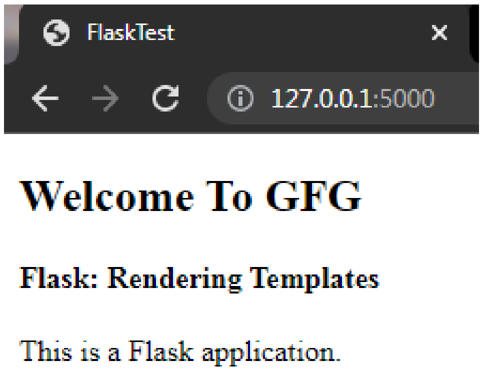
Now, we need a way to actually link the template with a specific route or URL. This means whenever the user goes to a specific URL then a specific template should be rendered or generated. Now, we need to change the “server.py” with the following:

Python

```
from flask import Flask, render_template

app = Flask(__name__)


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

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

Output:

We have imported the render_template function from the Flask module and added a route.



Welcome To GFG

Flask: Rendering Templates

This is a Flask application.

render the basic template

Templating With Jinja2 in Flask

Now, we'll create a new route for demonstrating the usage of the Jinja template. We need to add the route, so just add one more chunk of the code to the "server.py file"

Python

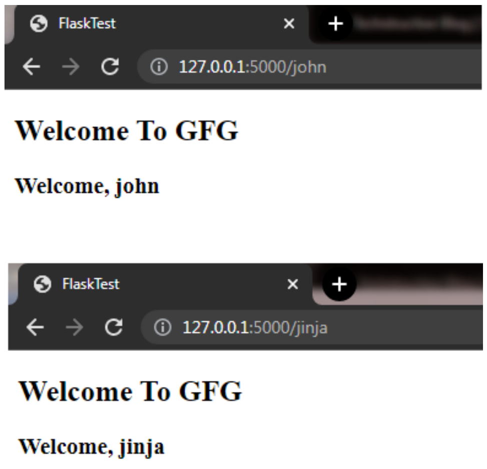
```python
@app.route("/<name>")
def welcome(name):
    return render_template("welcome.html", name=name)
```

Now, this might look pretty easy to understand, we are simply creating a route "/<name>" which will be bound to the welcome function. The "<name>" is standing for anything after the "/". So we take that as the parameter to our function and pass it to the render_template function as name. So, after passing the variable name in the render_template function, it would be accessible in the template for us to render that variable. You can even perform an operation on the variable and then parse it.

No, we need to create another template called "welcome.html" inside the template folder. This file should contain the following markup

HTML

```html
<!DOCTYPE html>
<html>

<head>
    <title>FlaskTest</title>
</head>

<body>
    <h2>Welcome To GFG</h2>
    <h3>Welcome, {{name}}</h3>
</body>

</html>
```



Welcome To GFG

Welcome, john

Welcome To GFG

Welcome, jinja

Using Jinja template

Flask – Jinja Template Inheritance Example

Now, we need a way to actually inherit some templates instead of reusing them, we can do that by creating the blocks in Jinja. They allow us to create a template block and we can use them in other templates with the name given to the block.

So, let us re-use our “index.html” and create a block in there. T do that we use “{% block <name> %} (where name = ‘body’) to start the block, this will take everything above it and store it in a virtual block of template, to end the block you simply use “{% endblock %}” this will copy everything below it.

templates/index.html

HTML

```
<!DOCTYPE html>
<html>
<head>
<title>FlaskTest</title>
</head>
<body>
<h2>Welcome To GFG</h2>
<h4>Flask: Rendering Templates</h4>
<a href="{{ url_for('home') }}">Home</a>
<a href="{{ url_for('index') }}">Index</a>
{% block body %}


<p>This is a Flask application.</p>


{% endblock %}
</body>
</html>
```

So, here we are not including the <p> tags as everything below the {% endblock %} and everything above the {% block body %} tag is copied. We are also using absolute URLs. The URLs are dynamic and quite easy to understand. We enclose them in “{{ }}” as part of the Jinja2 syntax. The url_for function reverses the entire URL for us, we just have to pass the name of the function as a string as a parameter to the function.

Now, we’ll create another template to reuse this created block “body”, let’s create the template “home.html” with the following contents:

templates/home.html

HTML

```
{% extends 'index.html' %}

{% block body %}

<p> This is a home page</p>

{% endblock %}
```

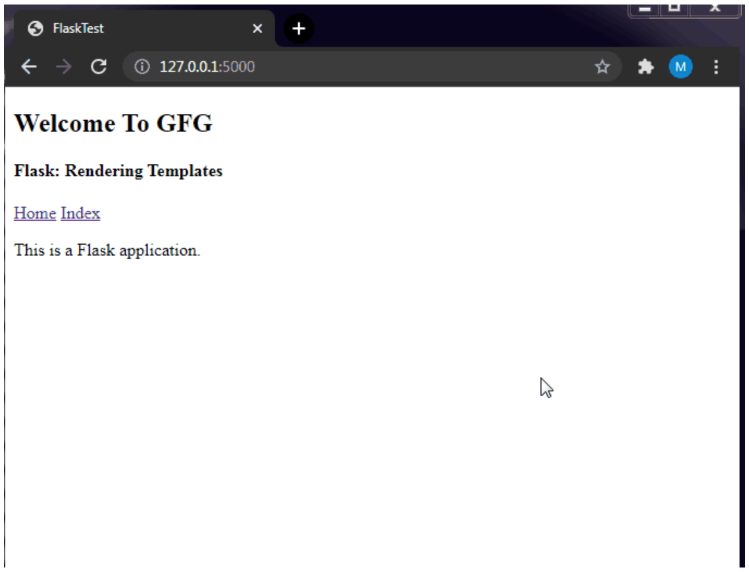
This looks like a two-liner but will also **extend** *(not include*) the index.html. This is by using the **{% extends <file.html> %}** tags, they parse the block into the mentioned template. After this, we can add the things we want. If you use the **include** tag it will not put the replacement paragraph in the correct place on the index.html page. It will create an invalid HTML file, but since the browser is very forgiving you will not notice unless you look at the source generated. The body text must be properly nested.

Finally, the piece left here is the route to home.html, so let's create that as well. Let's add another route to the "server.py file"

Python

```python
@app.route("/home")
def home():
    return render_template("home.html")
```

So, this is a route bound to the "/home" URL with the home function that renders the template "home.html" that we created just right now.



Welcome To GFG

Flask: Rendering Templates

Home Index

This is a Flask application.

Demonstrating block and URLs

As we can see the URL generated is dynamic, otherwise, we would have to hardcode both the template page paths. And also the block is working and inheriting the template as provided in the base templates. Open the page source in the browser to check it is properly formed HTML.

```
<!DOCTYPE html>
<html>
<head>
<title>FlaskTest</title>
</head>
<body>
   <h2>Welcome To GFG</h2>
   <h4>Flask: Rendering Templates</h4>
   <a href="/home">Home</a>
   <a href="/">Index</a>
   <a href="/about">About</a>
   <a href="/documentation">Documentation</a>

<p> This is a home page</p>
<p>must use extends not include</p>
</body>
</html>
```

Inducing Logic in Templates: We can use for loops if conditions in templates. this is such a great feature to leverage on. We can create some great dynamic templates without much of a hassle. Let us create a list in Python and try to render that on an HTML template.

Using for loops in templates: For that, we will create another route, this time at "/about", this route will bind to the function about that renders the template "about.html" but we will add some more things before returning from the function. We will create a list of some dummy strings and then parse them to the render_template function.

Python

```
@app.route("/about")
def about():
    sites = ['twitter', 'facebook', 'instagram', 'whatsapp']
    return render_template("about.html", sites=sites)
```

So, we have created the route at "/about" bound to the about function. Inside that function, we are first creating the list "Sites" with some dummy strings and finally while returning, we parse them to the render_template function as sites, you can call anything you like but remember to use that name in the templates. Now, to create the templates, we'll create the template "about.html" with the following contents:

templates/about.html

HTML

```
{% extends 'index.html' %}
{% block body %}
<ul>
    {% for social in sites %}
    <li>{{ social }}</li>
    {% endfor %}
</ul>
{% endblock %}
```

We can use for loops in templates enclosed in "{% %}" we can call them in a regular Pythonic way. The sites are the variable(list) that we parsed in the route function. We can again use the iterator as a variable enclosed in "{{ }}".

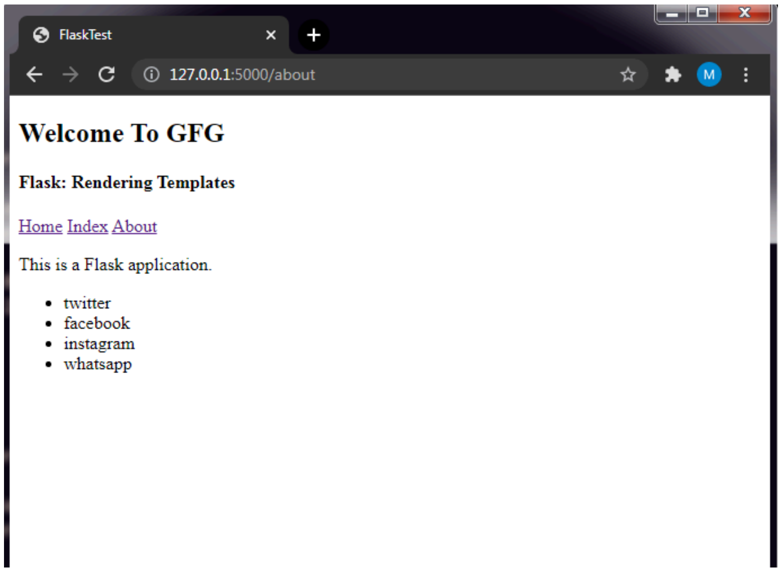
This is like joining the puzzle pieces, the values of variables are accessed with "{{ }}", and any other structures or blocks are enclosed in "{% %}.

Now to make it more accessible you can add its URL to the index.html like so:

HTML

```html
<!DOCTYPE html>
<html>

<head>
    <title>FlaskTest</title>
</head>

<body>
    <h2>Welcome To GFG</h2>
    <h4>Flask: Rendering Templates</h4>
    <a href="{{ url_for('home') }}">Home</a>
    <a href="{{ url_for('index') }}">Index</a>
    <a href="{{ url_for('about') }}">About</a>
    {% block body %}


<p>This is a Flask application.</p>


    {% endblock %}
</body>

</html>
```

This is not mandatory but it creates an accessible link for ease.



Welcome To GFG

Flask: Rendering Templates

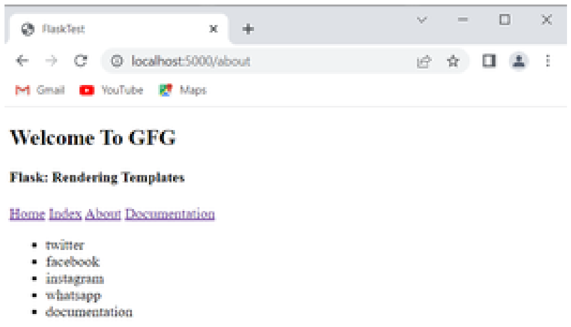
Home Index About

This is a Flask application.

- twitter
- facebook
- instagram
- whatsapp

Demonstrating for loop-in templates

As we can see it has dynamically created all the lists in the template. This can be used for fetching the data from the database if the app is production ready. Also, it can be used to create certain repetitive tasks or data which is very hard to do them manually.



Welcome To GFG

Flask: Rendering Templates

Home Index About Documentation

- twitter
- facebook
- instagram
- whatsapp
- documentation

Corrected extends file

This correctly defined extends file removed the placeholder paragraph and replaces it in the body of the HTML.

If statement in HTML Template in Python Flask

We can even use if-else conditions in flask templates. Similar to the syntax for the for loops we can leverage that to create dynamic templates. Let's see an example of a role for a website.

Let's build the route for the section contact. This URL is "contact/<role>", which is bound to the function contact which renders a template called "contacts.html". this takes in the argument as role. Now we can see some changes here, this is just semantic changes nothing new, we can use the variable person as a different name in the template which was assigned as the values of the role.

Python

```python
@app.route("/contact/<role>")
def contact(role):
    return render_template("contact.html", person=role)
```

So, this creates the route as desired and parses the variable role as a person to the template. Now let us create the template.

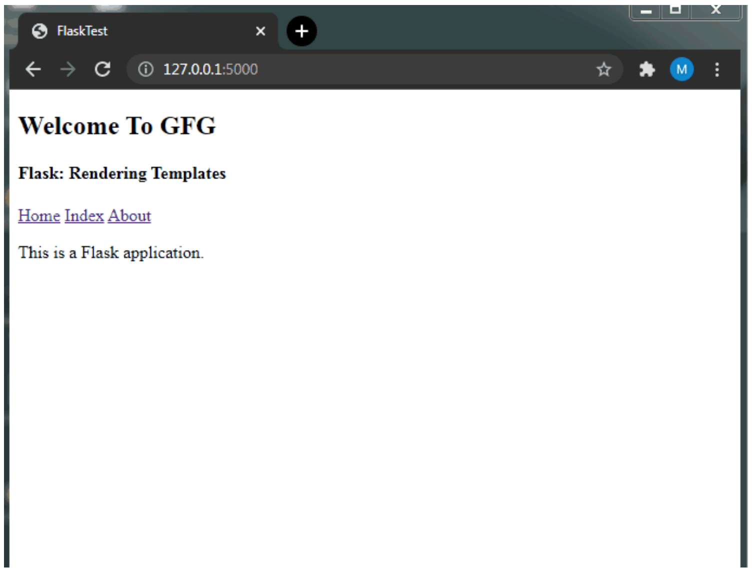
template/contact.html

HTML

```html
{% extends 'index.html' %}

{% block body %}
  {% if person == "admin" %}


<p> Admin Section </p>


  {% elif person == "maintainer" %}


<p> App Source Page for Maintainer</p>


  {% elif person == "member" %}


<p> Hope you are enjoying our services</p>


  {% else %}


<p> Hello, {{ person }}</p>
```

```
  {% endif %}
{% endblock %}
```

So, in the template, we are checking the value of the variable person which is obtained from the URL and parsed from the render_template function. The if-else syntax is similar to Python with just "{% %}" enclosed. The code is quite self-explanatory as we create if-elif and else ladder, checking for a value and creating the HTML elements as per the requirement.



Welcome To GFG

Flask: Rendering Templates

Home Index About

This is a Flask application.

So, we can see that the template is rendering the contents as per the role variable passed in the URL. Don't try to create a URL link for this as it would not work since we need to enter the role variable manually. There needs to be some workaround done to use it.

So that was about using and rendering the templates in Flask. We have leveraged the Jinja templating syntax with Python to create some dynamic templates.

Looking to dive into the world of programming or sharpen your Python skills? Our **Master Python: Complete Beginner to Advanced Course** is your ultimate guide to becoming proficient in Python. This course covers everything you need to build a solid foundation from fundamental programming concepts to advanced techniques. With **hands-on projects**, real-world examples, and expert guidance, you'll gain the confidence to tackle complex **coding challenges**. Whether you're starting from scratch or aiming to enhance your skills, this course is the perfect fit. Enroll now and master Python, the language of the future!

M meet... 12

Previous Article

Changing Host IP Address in Flask

Next Article

CSRF Protection in Flask

Similar Reads

Server Side Rendering vs Client Side Rendering vs Server Side Generation

In the world of web development, there are several approaches to rendering web pages: server-side rendering, client-side rendering, and server-side generation....

4 min read

Explain lifecycle of component re-rendering due to re-rendering of parent...

React is a javascript library that renders components written in JSX. You can build and render any component as per your usage. States can be updated accordingl...

2 min read

How does SSR(Server-Side Rendering) differ from CSR(client-side renderin...

Server-Side Rendering (SSR) and Client-Side Rendering (CSR) are two different approaches used in web development to render web pages to users. Each...

4 min read

Free Website Templates - HTML and CSS Templates with Source Code

HTML (HyperText Markup Language) and CSS (Cascading Style Sheets) are the fundamental building blocks for designing and developing any web page or...

5 min read

Flask - Templates

In this article, we are going to learn about the flask templates in Python. Python is a high-level, open-source, object-oriented language, consisting of libraries,...

8 min read

Unit Testing Jinja Templates with Flask

Unit testing is an essential practice in software development to ensure the accuracy and functionality of code. When working with Jinja templates in Pytho...

4 min read

Documenting Flask Endpoint using Flask-Autodoc

Documentation of endpoints is an essential task in web development and being able to apply it in different frameworks is always a utility. This article discusses...

4 min read

How to use Flask-Session in Python Flask ?

Flask Session - Flask-Session is an extension for Flask that supports Server-side Session to your application.The Session is the time between the client logs in to...

4 min read

How to Integrate Flask-Admin and Flask-Login

In order to merge the admin and login pages, we can utilize a short form or any other login method that only requires the username and password. This is know...

8 min read

Minify HTML in Flask using Flask-Minify

Flask offers HTML rendering as output, it is usually desired that the output HTML should be concise and it serves the purpose as well. In this article, we would...

12 min read

Article Tags : Python Web Technologies python Python Flask +2 More

Practice Tags : python python

Web Design

Computer Science

Operating Systems
Computer Network
Database Management System
Software Engineering
Digital Logic Design
Engineering Maths
Software Development
Software Testing

DevOps

Git
Linux
AWS
Docker
Kubernetes
Azure
GCP
DevOps Roadmap

System Design

High Level Design
Low Level Design
UML Diagrams
Interview Guide
Design Patterns
OOAD
System Design Bootcamp
Interview Questions

Inteview Preparation

Competitive Programming
Top DS or Algo for CP
Company-Wise Recruitment Process
Company-Wise Preparation
Aptitude Preparation
Puzzles

School Subjects

Mathematics
Physics
Chemistry
Biology
Social Science
English Grammar
Commerce
World GK

GeeksforGeeks Videos

DSA
Python
Java
C++
Web Development
Data Science
CS Subjects