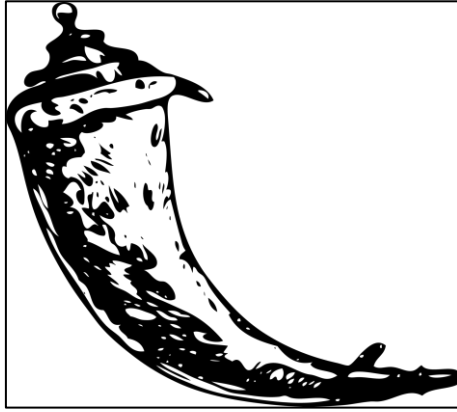


# **Project – 11**

## **Flask**



**Submitted By:**  
**Nishigandha Patil**

# INDEX

<b>Sr. No</b>	<b>Topics</b>	<b>Page. No</b>
1	Introduction to Flask	3
2	Create Application	4
3	Templates	6
4	Flask Form	7
5	Database Integration	9
6	Deployment	10

# INTRODUCTION TO FLASK

Flask is a lightweight and flexible web framework written in Python. It is designed to be easy to use and provides the essentials needed for web development without imposing too many constraints on the developer. 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 a form validation support. Instead, Flask supports the extensions to add such functionality to the application.

## ➤ Why Flask?

1. **Simplicity:** Flask follows a minimalistic design, making it easy to learn and use. Developers can get started quickly with its straightforward syntax and structure.
2. **Flexibility:** Flask does not impose rigid patterns or dependencies, allowing developers to choose components based on their needs. This flexibility is beneficial for building various types of web applications.
3. **Extensibility:** Flask has a modular architecture, and developers can add extensions for specific functionalities as needed. This allows customization without unnecessary bloat.
4. **Microframework:** Flask is often referred to as a microframework, meaning it provides only the essential components for web development. This makes it suitable for small to medium-sized projects where a full-stack framework may be overly complex.
5. **Ease of Integration:** Flask easily integrates with other technologies and libraries. It can be combined with various databases, templating engines, and front-end frameworks, offering developers the freedom to choose their preferred tools.
6. **Scalability:** While Flask is suitable for small projects, it can also scale for larger applications. With the right extensions and design practices, Flask can handle more complex requirements.

➤ **WSGI:** Web Server Gateway Interface (WSGI) has been adopted as a standard for Python web application development. WSGI is a specification for a universal interface between the web server and the web applications.

➤ **Werkzeug:** It is a WSGI toolkit, which implements requests, response objects, and other utility functions. This enables building a web framework on top of it. The Flask framework uses Werkzeug as one of its bases.

## ➤ Prerequisites:

1. **Install Python:** Go to <https://www.python.org/downloads/> download latest version.
2. **PIP:** Use a Python's Package Manager to install any packages/frameworks/modules.
3. **Virtual environment (Optional but Recommended):** It's good practice to work within a virtual environment to isolate your project dependencies.  
**Create venv:** `python -m venv myenv`  
Run command '**Set-ExecutionPolicy unrestricted**' if you got error in running virtual environment.  
**Activate venv:** `myenv\Scripts\activate`
4. **Install Flask:** You can install Django globally as well as in virtual environment using the command: `pip install flask`

# CREATE APPLICATION

## 1. Create python file:

Open VS code → Open Folder → Create Virtual Environment → Install Flask → Create Python file (**Example:** myapp.py)

## 2. Write Flask Minimal Application Code:

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

Codeium: Refactor | Explain | Generate Docstring | X
@app.route('/')
def hello_world():
    return 'Hello World'

if __name__ == '__main__': #Run the Application in Debug Mode
    app.run(debug=True)
```

- Flask constructor takes the name of **current module** (`__name__`) as argument.
- The **route ()** function of the Flask tells the application (website) which URL should call the associated function. Routes are defined using the **@app.route** decorator.
- Finally, the **run ()** method of Flask class runs the application on the local development server. Set **debug=True** to enable Debugger Mode. You can set the PORT number: `app.run (debug=True, port= 8000)`

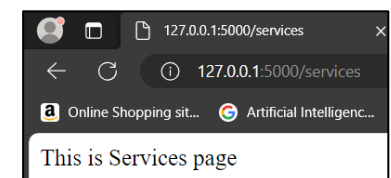
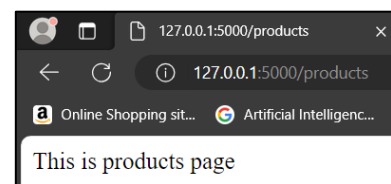
### ➤ Run Application:

Open Terminal → Use Command: **python app\_name.py** (**Example:** python myapp.py)

### ➤ 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.

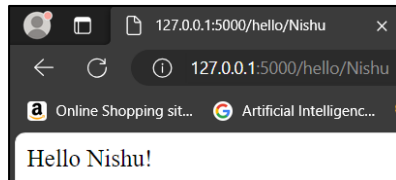
```
myapp.py > ...
1 from flask import Flask
2 app = Flask(__name__) #Flask Constructor
3
4 Codeium: Refactor | Explain | Generate Docstring | X
5 @app.route('/')
6 def hello_world():
7     return 'Hello World'
8
9 Codeium: Refactor | Explain | Generate Docstring | X
10 @app.route('/products') #http://127.0.0.1:5000/products
11 def products():
12     return 'This is products page'
13
14 Codeium: Refactor | Explain | Generate Docstring | X
15 @app.route('/services') #http://127.0.0.1:5000/services
16 def services():
17     return 'This is Services page'
18
19 if __name__ == '__main__': #Run the Application in Debug Mode
20     app.run(debug=True)
```



➤ **Variable Rules:**

```
@app.route('/hello/<name>')
def hello_name(name):
    return 'Hello %s!' % name
```

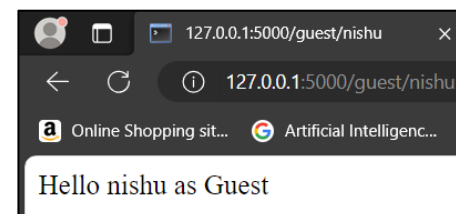
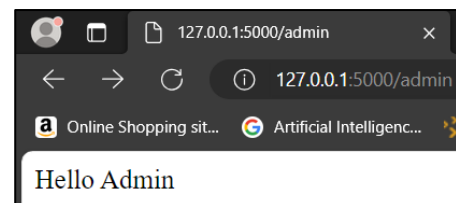
Open the browser and enter URL <http://localhost:5000/hello/Nishu>.



➤ **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.

```
myapp.py > ...
1 from flask import Flask, redirect, url_for
2 app = Flask(__name__) #Flask Constructor
3
4 @app.route('/admin')
5 def hello_admin():
6     return 'Hello Admin'
7
8 @app.route('/guest/<guest>')
9 def hello_guest(guest):
10     return 'Hello %s as Guest' % guest
11
12 @app.route('/user/<name>')
13 def hello_user(name):
14     if name == 'admin':
15         return redirect(url_for('hello_admin'))
16     else:
17         return redirect(url_for('hello_guest', guest = name))
18
19 if __name__ == '__main__':
20     app.run(debug=True)
```



➤ **Middleware:**

Flask supports middleware for handling tasks like authentication, logging, etc.

➤ **Extensions:**

Flask has a rich ecosystem of extensions that add additional functionality. Some popular ones include Flask-SQLAlchemy, Flask-Login, Flask-WTF, etc.

➤ **RESTful APIs:**

Flask can be used to build RESTful APIs by returning JSON responses and handling HTTP methods.

# TEMPLATES

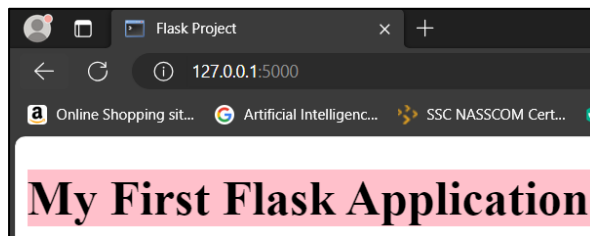
Flask uses **Jinja2** templates for rendering dynamic content. Templates allow you to embed variables and control structures in HTML. To work with HTML Templates, you need to import **render\_template** function.

**VS Code Extension:** Jinja Snippets flask

1. Create **templates** Folder in your Folder Directory.
2. Create HTML File inside it. (index.html)

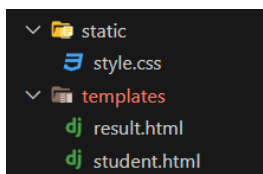
```
templates > index.html > ...
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="UTF-8">
5   <meta name="viewport" content="width=device-width, initial-scale=1.0">
6   <link rel="stylesheet" href="/static/style.css" type="text/css">
7   <title>Flask Project</title>
8 </head>
9 <body>
10  <h1> My First Flask Application</h1>
11 </body>
12 </html>
```

```
myapp.py > ...
1 from flask import Flask,render_template
2 app = Flask(__name__)
3 @app.route('/')
4 def index():
5     return render_template('index.html')
6
7
8 if __name__ == '__main__':
9     app.run(debug=True)
```



## ➤ Static:

A web application often requires a static file such as a JavaScript file or a CSS file supporting the display of a web page. Create **static** Folder in your Flask App Directory to store static files for webpages.



## ➤ Request and Response Handling:

Flask provides request and response objects for handling incoming HTTP requests and generating responses.

# FLASK FORM

The Form data received by the triggered function can collect it in the form of a dictionary object and forward it to a template to render it on a corresponding web page. Flask-WTF or Flask-Forms can be used to handle web forms.

```
myapp.py > ...
1  from flask import Flask,render_template,request
2  app = Flask(__name__)
3
4
5  @app.route('/')
6  def student():
7      return render_template('student.html')
8
9  @app.route('/result',methods = ['POST', 'GET'])
10 def result():
11     if request.method == 'POST':
12         result = request.form
13         return render_template("result.html",result = result)
14
15 if __name__ == '__main__':
16     app.run(debug=True)
```

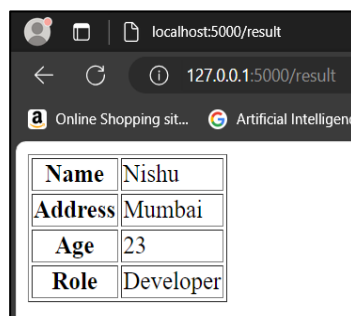
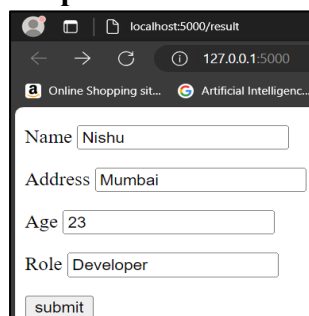
**student.html: To get the information.**

```
<body>
<form action = "/result" method = "POST">
    <p>Name <input type = "text" name = "Name" /></p>
    <p>Address <input type = "text" name = "Address" /></p>
    <p>Age <input type = "text" name = "Age" /></p>
    <p>Role <input type = "text" name = "Role" /></p>
    <p><input type = "submit" value = "submit" /></p>
</form>
</body>
```

**result.html: To store the information in table.**

```
templates > dj result.html
1 <table border = 1>
2     {% for key, value in result.items() %}
3     <tr>
4         <th> {{ key }} </th>
5         <td> {{ value }} </td>
6     </tr>
7     {% endfor %}
8 </table>
```

**Output:**



Name	Nishu
Address	Mumbai
Age	23
Role	Developer

## ➤ WTForms:

Flask-WTF extension provides a simple interface with this WTForms library.

Using Flask-WTF, we can define the form fields in our Python script and render them using an HTML template. It is also possible to apply validation to the WTF field.

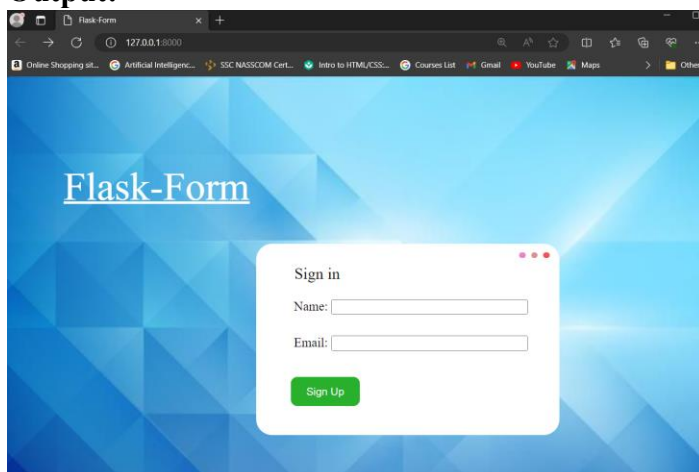
**First, Flask-WTF extension needs to be installed:** flask-WTF extension.

```
form.py > ...
1  from flask import Flask, render_template, request
2  from flask_wtf import FlaskForm
3  from wtforms import StringField, SubmitField
4
5  app = Flask(__name__)
6  app.config['SECRET_KEY'] = 'nishu'
7
8  Codeium: Explain | Codiumate: Options | Test this class
9  class MyForm(FlaskForm):
10     name = StringField('Name: ')
11     email = StringField('Email: ')
12     submit = SubmitField('Sign Up')
13
14  Codeium: Refactor | Explain | Generate Docstring | X
15  @app.route("/", methods=['GET', 'POST'])
16  def login():
17     form = MyForm()
18     if request.method == 'POST':
19         return "Form Submitted Successfully!"
20     else:
21         return render_template('index.html', form=form)
22
23  if __name__ == "__main__":
24     app.run(debug=True, port=8000)
```

## index.html:

```
<body>
<form action="" method="POST">
  <p class="name">Flask-Form</p>
  <center>
    <div class="container">
      <p class="signin">Sign in</p><br>
      <p class="input">
        {{form.name.label()}}
        {{form.name(size=32)}}
      </p><br>
      <p class="input">
        {{form.email.label()}}
        {{form.email(size=32)}}
      </p><br>
      <p>
        {{form.submit(class="signupbtn")}}
      </p>
    </div>
    <div class="box">
      <div class="red"></div>
      <div class="blue"></div>
      <div class="yellow"></div>
    </div>
  </div>
</form>
</body>
```

## Output:





# DATABASE INTEGRATION

Flask can be easily integrated with databases, and one common choice is to use **SQLAlchemy** as an Object-Relational Mapping (ORM) tool.

```
exampledb.py > ...
from flask import Flask, render_template, request, redirect, url_for
from flask_sqlalchemy import SQLAlchemy
app = Flask(__name__)
# Configure the SQLite database
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///example.db'
db = SQLAlchemy(app)

# Define a simple model for demonstration purposes
Codeium: Explain | Codiumate: Options | Test this class
class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(80), unique=True, nullable=False)

# Create the database tables
with app.app_context():
    db.create_all()

# Route to display all users
Codeium: Refactor | Explain | Generate Docstring | X
@app.route('/')
Codeiumate: Options | Test this function
def index():
    users = User.query.all()
    return render_template('index.html', users=users)

# Route to add a new user
Codeium: Refactor | Explain | Generate Docstring | X
@app.route('/add_user', methods=['POST'])
Codeiumate: Options | Test this function
def add_user():
    if request.method == 'POST':
        username = request.form['username']
        new_user = User(username=username)

        # Add the new user to the database
        db.session.add(new_user)
        db.session.commit()

        return redirect(url_for('index'))

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

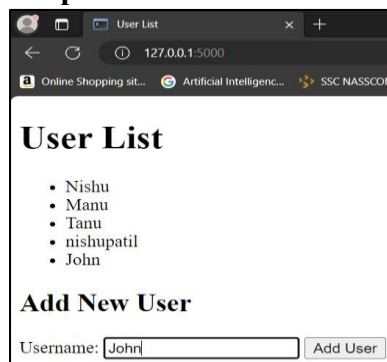
## index.html:

```
<body>
  <h1>User List</h1>

  <ul>
    {% for user in users %}
      <li>{{ user.username }}</li>
    {% endfor %}
  </ul>

  <h2>Add New User</h2>
  <form action="/add_user" method="post">
    <label for="username">Username:</label>
    <input type="text" id="username" name="username" required>
    <button type="submit">Add User</button>
  </form>
</body>
```

## Output:



# DEPLOYMENT

Deploying a Flask application involves making it accessible over the internet so that users can interact with it. There are various ways to deploy a Flask app, and the choice depends on your project requirements, infrastructure, and personal preferences.

## ➤ Choose a Deployment Platform:

- **Platform as a Service (PaaS):** Platforms like Heroku, Google App Engine, or Microsoft Azure App Service provide easy deployment options. You just need to push your code, and the platform takes care of the deployment process.
- **Infrastructure as a Service (IaaS):** Services like AWS EC2, DigitalOcean, or Linode allow you to set up virtual machines and deploy your Flask app manually.
- **Containerization:** You can use Docker to containerize your Flask app and deploy it on container orchestration platforms like Kubernetes.

## ➤ Prepare Your Application for Deployment:

- Ensure your Flask app is configured properly for production. Set **debug=False** and use a **secure secret key**.
- Update your 'requirements.txt' with production dependencies.
- Consider using a production-ready web server like Gunicorn (Green Unicorn) instead of the built-in development server.

## ➤ Deploy on Heroku (Example):

- Install the Heroku CLI: <https://devcenter.heroku.com/articles/heroku-cli>
- Create a Profile in your project's root directory with the following content:

```
web: gunicorn your_app_module:app
```

- Update your requirements file with Gunicorn if not already included:

```
gunicorn==20.1.0
```

- Commit your changes and push your code to a version control system (e.g., Git).
- Create a Heroku app:

```
heroku create your-app-name
```

- Deploy your app:

```
git push heroku master
```

- Open your app in the browser:

```
heroku open
```