Experiment 5 : Flask Application using `render_template()` function.

Name of Student	**Aryan Patankar**
Class Roll No	**D15A_33**
D.O.P.	**06/03/2025**
D.O.S.	**13/03/2025**
Sign and Grade	

AIM : To create a Flask application that demonstrates template rendering by dynamically generating HTML content using the `render_template()` function.

PROBLEM STATEMENT :

Develop a Flask application that includes:

1. A homepage route (`/`) displaying a welcome message with links to additional pages.
2. A dynamic route (`/user/<username>`) that renders an HTML template with a personalized greeting.
3. Use Jinja2 templating features, such as variables and control structures, to enhance the templates.

Theory:

1. What does the `render_template()` function do in a Flask application?

The `render_template()` function in Flask is used to render HTML templates and return them as responses to client requests. Instead of returning plain text or manually writing HTML inside the Python code, Flask allows the use of separate HTML files stored in the `templates` folder.

Usage Example:

```
from flask import Flask, render_template

app = Flask(_name_)

@app.route('/')
def home():
```

```
    return render_template('index.html')
```

Here, `render_template('index.html')` loads the `index.html` file from the `templates` folder and sends it as a response. This helps in separating logic from presentation, making web applications more organized and maintainable.
Additionally, `render_template()` supports passing dynamic data to templates:

```
@app.route('/user/<name>') def user(name):
    return render_template('user.html', username=name)
```

In `user.html`, we can access `username` using Jinja2 templating:

```
<p>Hello, {{ username }}!</p>
```

---

2. What is the significance of the `templates` folder in a Flask project?

The `templates` folder holds all the HTML files used for rendering web pages in a Flask application. Flask automatically looks for template files inside this directory, making it a convention that helps in maintaining a well-structured project.

Key Significance:

1. **Separation of Concerns** – Keeps the HTML structure separate from Python logic, improving code readability.
2. **Easy Management** – All templates are stored in one location, simplifying maintenance.
3. **Supports Jinja2** – Enables the use of dynamic content within HTML files through Jinja2 templating.
4. **Enables Code Reusability** – Common UI components, such as headers and footers, can be stored in separate template files and reused across multiple pages using template inheritance.

Project Structure Example:

```
/my_flask_app
|── app.py
|── /templates
|    |── index.html
|    |── about.html
|    |── base.html
|── /static
|    |── styles.css
|    |── script.js
```

Here, `index.html` and `about.html` are stored inside the `templates` folder and can be rendered using `render_template()`.

---

3. What is Jinja2, and how does it integrate with Flask?

Jinja2 is a powerful templating engine used in Flask to generate dynamic HTML content. It allows embedding Python-like expressions inside HTML, making web pages more interactive and adaptable based on user input or backend data.

Integration with Flask:

Flask uses Jinja2 by default when rendering templates through `render_template()`. The syntax includes:

- **Variables** – `{{ variable_name }}`
- **Control Structures** – `{% if condition %} ... {% endif %}`
- **Loops** – `{% for item in list %} ... {% endfor %}`

Example Usage:

Python Code (Flask App)

```
@app.route('/greet/<name>')
def greet(name):
    return render_template('greet.html', username=name)
```

Jinja2 Template (`greet.html`)

```
<!DOCTYPE html>
<html>
<head>
    <title>Greeting</title>
</head>
```

```
<body>
    <h1>Hello, {{ username }}!</h1>
</body>
</html>
```

Features of Jinja2 in Flask:

1. **Template Inheritance** – Allows reusing base layouts using `{% extends "base.html" %}` and `{% block content %} ... {% endblock %}`.
2. **Filters** – Modify data output (e.g., `{{ name.upper() }}` converts text to uppercase).
3. **Control Structures** – Supports conditionals and loops for dynamic content.

Jinja2 enhances the flexibility of Flask applications by enabling dynamic content generation within HTML templates.

OUTPUT:-

app.py

```
from flask import Flask,

render_template app = Flask(_name_)

# 1. Homepage Route
(/) @app.route('/')
def home():
   user_list = ["Alice", "Bob", "Charlie", "David"]
   return render_template('home.html', users=user_list)

# 2. Dynamic User Route (/user/<username>)
@app.route('/user/<username>')
def user_profile(username):
   user_data = {
      "Alice": {"age": 30, "city": "New York"},
      "Bob": {"age": 25, "city": "Los Angeles"},
      "Charlie": {"age": 35, "city": "Chicago"},
      "David": {"age": 28, "city": "Houston"},
   }
   if username in user_data:
      return render_template('user.html', username=username,
   data=user_data[username]) else:
      return render_template('user.html', username=username, data=None)

if _name___== '_main_':
   app.run(debug=True)
```

Templates

1. home.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Homepage</title>
    <link rel="stylesheet" href="{{ url_for('static', filename='style.css') }}">
</head>
<body>
    <div class="container">
        <h1>Welcome to Our Site!</h1>
        <p>Here are some users you can visit:</p>
        <ul>
            {% for user in users %}
            <li><a href="{{ url_for('user_profile', username=user) }}">{{ user }}</a></li>
            {% endfor %}
        </ul>
    </div>
</body>
</html>
```

2. user.html

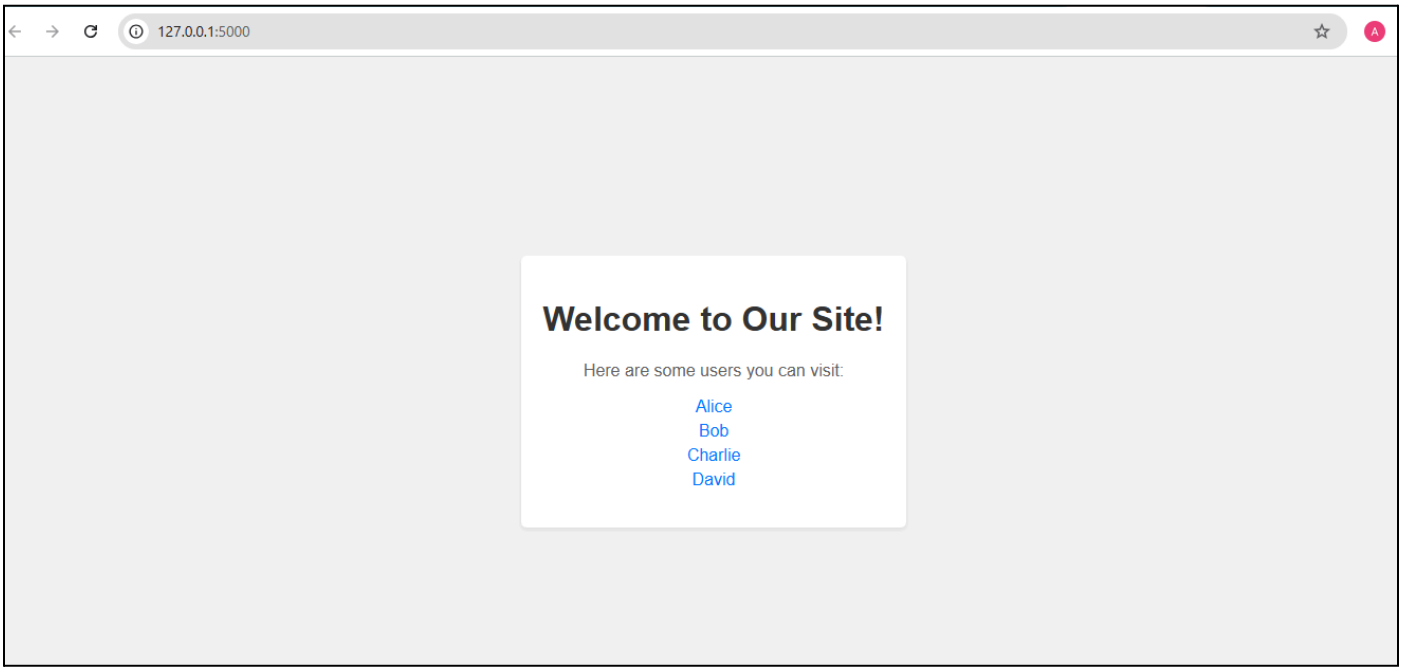
```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>User Profile</title>
    <link rel="stylesheet" href="{{ url_for('static', filename='style.css') }}">
</head>
<body>
    <div class="container">
        <h1>User Profile</h1>
        {% if data %}
        <p>Hello, {{ username }}!</p>
        <p>Age: {{ data.age }}</p>
        <p>City: {{ data.city }}</p>
        {% else %}
        <p>User {{ username }} not found.</p>
        {% endif %}
    </div>
</body>
</html>
```

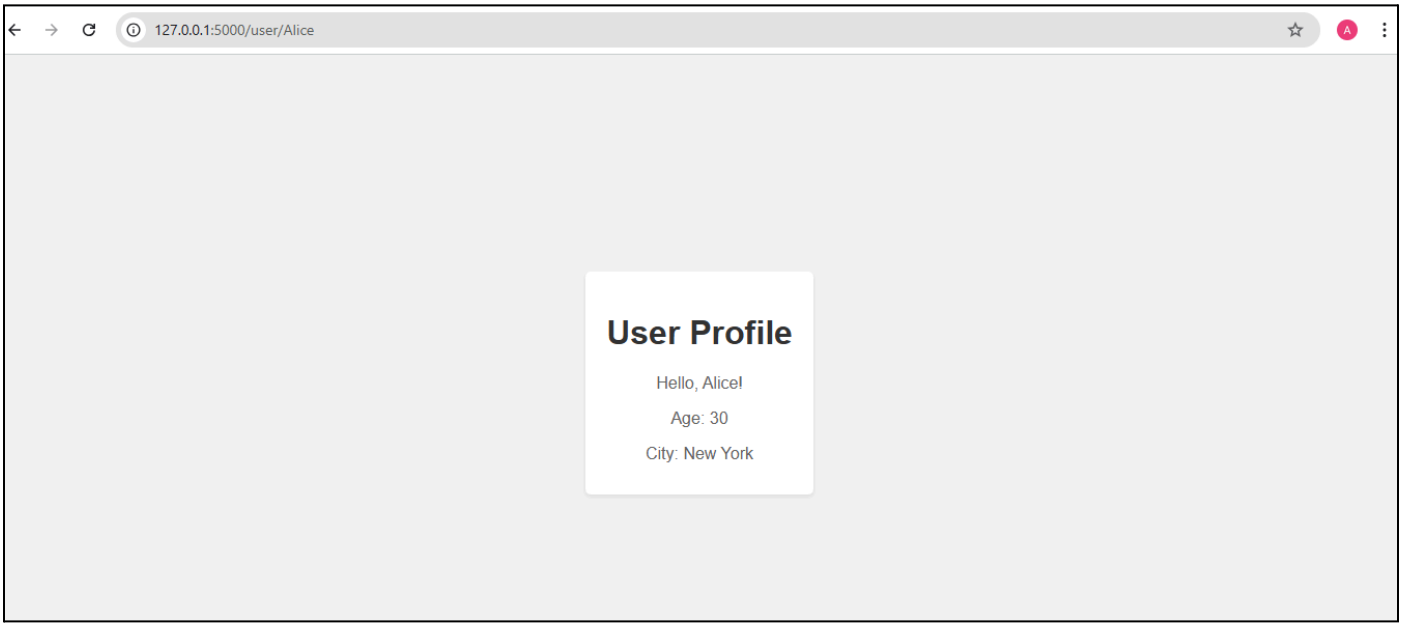
style.css:

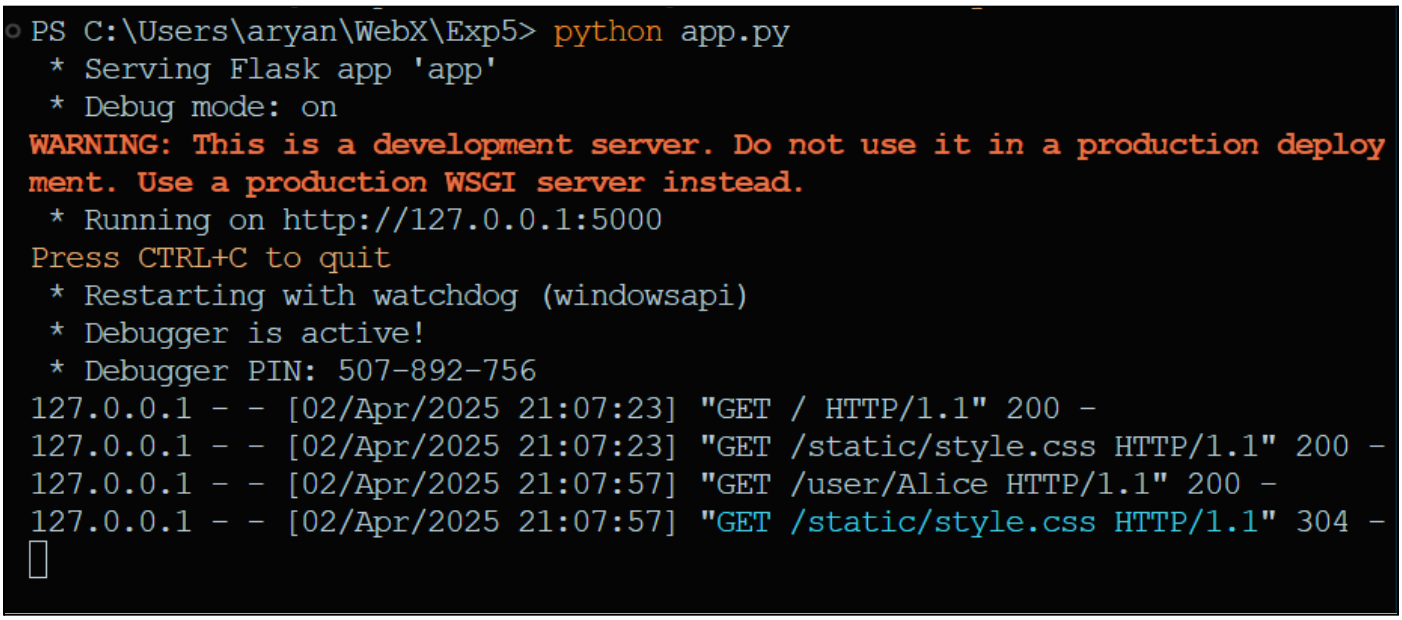
```
/* style.css */
body {
    font-family: Arial, sans-serif;
    background-color: #f4f4f4;
    margin: 0;
    padding: 0;
    display: flex;
    justify-content: center;
    align-items: center;
    min-height: 100vh;
}

.container {
    background-color: #fff;
    padding: 20px;
    border-radius: 5px;
    box-shadow: 0 2px 5px rgba(0, 0, 0, 0.1);
    text-align: center;
}

h1 {
    color: #333;
}

p {
    color: #666;
    margin-bottom: 10px;
```

```
}

ul {
    list-style: none;
    padding: 0;
}

li {
                                                    margin-bottom: 5px;

}


a {
    color: #007bff;
    text-decoration: none;
}

a:hover {
    text-decoration: underline;
}
```

Results:


127.0.0.1:5000
Welcome to Our Site!
Here are some users you can visit:
Alice
Bob
Charlie
David


127.0.0.1:5000/user/Alice
User Profile
Hello, Alice!
Age: 30
City: New York


PS C:\Users\aryan\WebX\Exp5> python app.py
* Serving Flask app 'app'
* Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
* Running on http://127.0.0.1:5000
Press CTRL+C to quit
* Restarting with watchdog (windowsapi)
* Debugger is active!
* Debugger PIN: 507-892-756
127.0.0.1 - - [02/Apr/2025 21:07:23] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [02/Apr/2025 21:07:23] "GET /static/style.css HTTP/1.1" 200 -
127.0.0.1 - - [02/Apr/2025 21:07:57] "GET /user/Alice HTTP/1.1" 200 -
127.0.0.1 - - [02/Apr/2025 21:07:57] "GET /static/style.css HTTP/1.1" 304 -