EXPERIMENT NO. 3

Name of Student	MAHVISH SIDDIQUI
Class Roll No	56
D.O.P.	06/02/2025
D.O.S.	
Sign and Grade	

AIM : To develop a basic Flask application with multiple routes and demonstrate the handling of GET and POST requests.

PROBLEM STATEMENT :

Design a Flask web application with the following features:

1. A homepage (/) that provides a welcome message and a link to a contact form.
    a. Create routes for the homepage (/), contact form (/contact), and thank-you page (/thank_you).
2. A contact page (/contact) where users can fill out a form with their name and email.
3. Handle the form submission using the POST method and display the submitted data on a thank-you page (/thank_you).
    a. On the contact page, create a form to accept user details (name and email).
    b. Use the POST method to handle form submission and pass data to the thank-you page
4. Demonstrate the use of GET requests by showing a dynamic welcome message on the homepage when the user accesses it with a query parameter, e.g., /welcome?name=<user_name>.
    a. On the homepage (/), use a query parameter (name) to display a personalized welcome message.

Theory:

A. List some of the core features of Flask
B. Why do we use Flask(__name__) in Flask?
C. What is Template (Template Inheritance) in Flask?
D. What methods of HTTP are implemented in Flask?
E. What is difference between Flask and Django framework

1. **List some of the core features of Flask**

Core Features of Flask:

Flask is a lightweight and flexible web framework for Python. Some of its core features include:

- **Lightweight**: Flask is a micro-framework, meaning it does not include many built-in features, allowing for easy customization and flexibility.
- **Routing**: Flask uses decorators to handle HTTP requests and map them to Python functions (routes).
- **Template Engine (Jinja2)**: Flask integrates with Jinja2, a powerful template engine, to generate dynamic HTML content.
- **RESTful Request Handling**: Flask easily supports RESTful request handling for APIs.
- **Development Server**: It comes with an inbuilt development server, making it easy to test applications locally.
- **Extensible**: Flask is highly extensible, allowing for the addition of various libraries and tools for functionalities like database integration (e.g., SQLAlchemy).
- **Session Management**: Flask provides session management, allowing for persistent user data across requests.
- **Debug Mode**: Flask has a built-in debug mode that helps developers catch errors in development.

2. **Why do we use Flask(__name__) in Flask?**
    - Flask(__name__) is used to create an instance of the Flask class. It tells Flask where to find the application and its resources.
    - __name__ is passed to indicate where the application is defined (usually the main entry point). It helps Flask determine where to look for static files, templates, and other resources.
        - When __name__ is used, Flask knows whether it's being run directly or N imported as a module.
        - __name__ is used for resource loading and debugging.

3. **What is Template (Template Inheritance) in Flask?**

Templates in Flask are files that contain placeholders for dynamic content. They are typically written in HTML, with embedded Python code for dynamic content (using the Jinja2 template engine).

Template Inheritance: Flask allows you to create a base template that defines the structure of the page, and child templates can extend it and add their own content.

- **Base Template**: Defines common structure, like headers, footers, and navigation.

- **Child Templates**: Extend the base template and inject specific content into defined blocks.

4. **What methods of HTTP are implemented in Flask?**

HTTP Methods in Flask:

Flask, a lightweight web framework for Python, supports various HTTP methods that can be used to handle different types of HTTP requests. The primary HTTP methods are:

1. **GET** – Used to request data from a specified resource. This is the most common HTTP method.
2. **POST** – Used to send data to a server to create a resource.
3. **PUT** – Used to update a current resource with new data.
4. **DELETE** – Used to delete a resource.
5. **PATCH** – Used to apply partial modifications to a resource.
6. **OPTIONS** – Used to describe the communication options for the target resource.
7. **HEAD** – Similar to GET, but it only returns the headers and no body content.

In Flask, you can define route handlers for each of these methods using the `@app.route` decorator with the `methods` parameter:

```
@app.route('/example', methods=['GET', 'POST'])
def example():
    if request.method == 'GET':
        return 'This is a GET request'
    elif request.method == 'POST':
        return 'This is a POST request'
```

5. What is difference between Flask and Django framework

Difference Between Flask and Django:

Both **Flask** and **Django** are popular web frameworks for Python, but they differ in their design philosophies, features, and usage. Here are some of the key differences:

Aspect	Flask	Django
Type	Microframework (minimalistic)	Full-stack framework (batteries-included)
Philosophy	Simplicity and flexibility, minimal setup	Convention over configuration, ready-to-use tools
Learning Curve	Steeper (requires more manual configuration)	Gentler (includes built-in features, less configuration)
Architecture	Very modular and unopinionated. Developers build their structure.	Monolithic with a prescribed structure (MVC pattern).
Flexibility	Highly flexible and allows developers to choose their tools	Less flexible, but has built-in tools for everything (e.g., ORM, admin, authentication)
Built-in Features	Minimal built-in tools (you add extensions for ORM, form handling, etc.)	A lot of built-in tools (ORM, authentication, admin panel, etc.)
Template Engine	Jinja2 (similar to Django templates)	Django templates (own templating engine)
Routing	Explicit routing and URL handling	URL routing via regular expressions

ORM	Flask does not include an ORM, but you can use extensions like SQLAlchemy.	Django comes with its own ORM system out of the box.
Admin Interface	No built-in admin interface, but extensions like Flask-Admin exist.	Django provides a fully-featured admin interface by default.
Community & Ecosystem	Smaller, but vibrant and growing community	Larger community and many available plugins

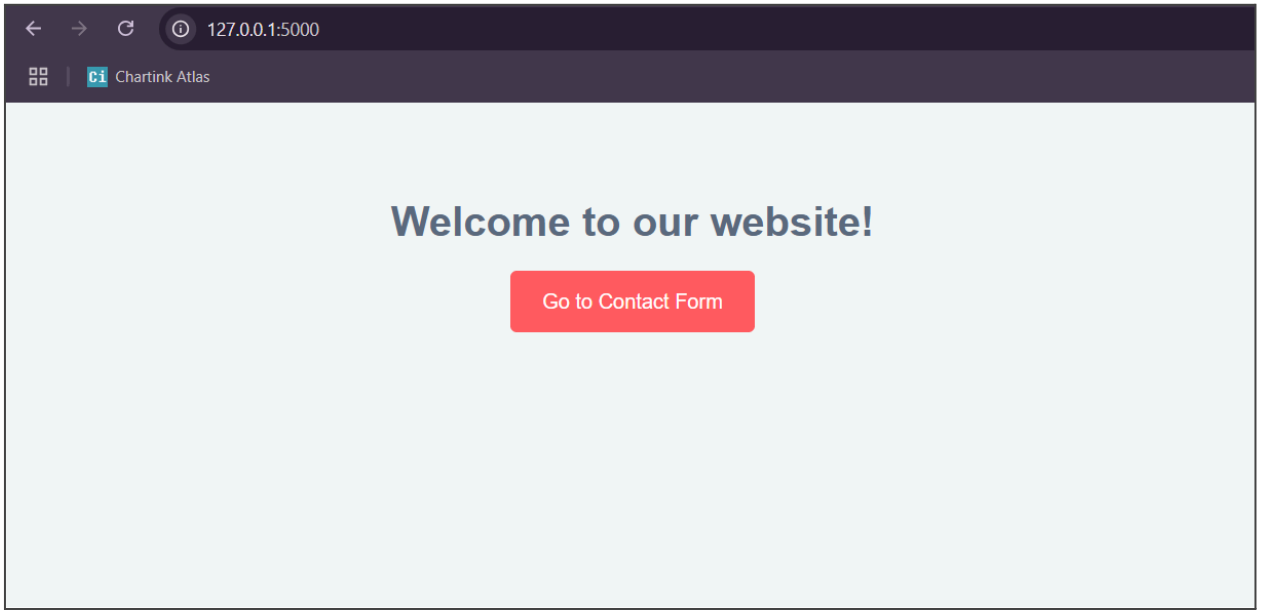
Routing

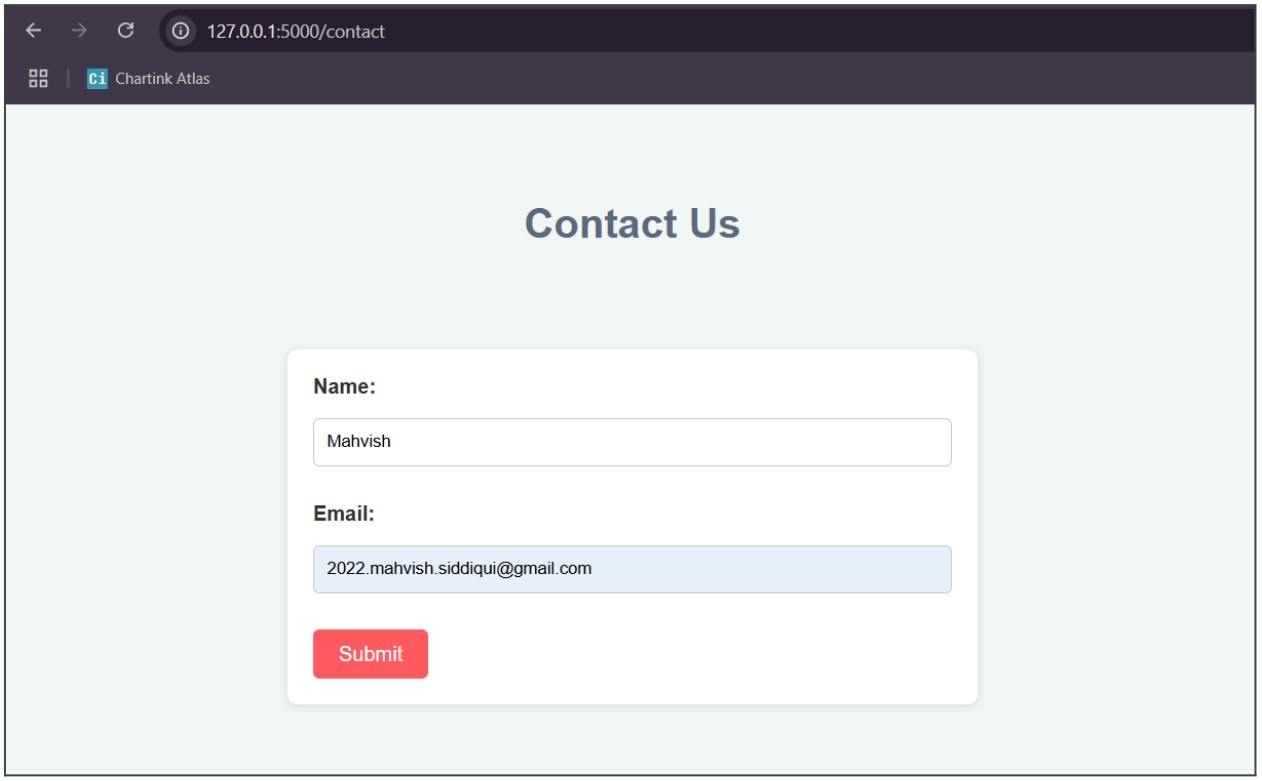
URL building

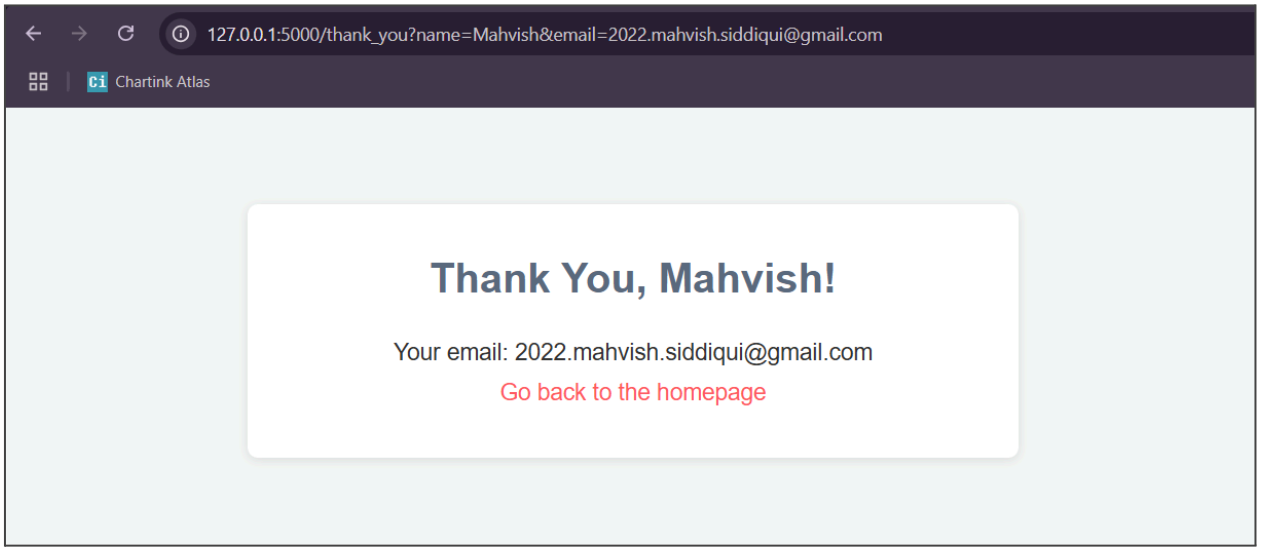
GET REQUEST

POST REQUEST

OUTPUT







CODE: app.py

```
from flask import Flask, render_template, request, redirect, url_for

app = Flask(__name__)

@app.route('/')
def home():
    name = request.args.get('name', '')
    if name:
        message = f"Welcome, {name}!"
    else:
        message = "Welcome to our website!"
    return render_template('home.html', message=message)
```

```
@app.route('/contact', methods=['GET', 'POST'])
def contact():
    if request.method == 'POST':

        name = request.form['name']
        email = request.form['email']
        return redirect(url_for('thank_you', name=name, email=email))

    return render_template('contact.html')

@app.route('/thank_you')
def thank_you():
    name = request.args.get('name')
    email = request.args.get('email')
    return render_template('thank_you.html', name=name, email=email)

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

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='styles.css') }}">
</head>
<body>
    <div class="container">
        <h1>{{ message }}</h1>
        <p><a href="{{ url_for('contact') }}" class="button">Go to Contact Form</a></p>
    </div>
</body>
</html>
```

contact.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Contact Form</title>
    <link rel="stylesheet" href="{{ url_for('static', filename='styles.css') }}">
</head>
<body>
    <div class="container">
        <h1>Contact Us</h1>
```

```
        <form action="{{ url_for('contact') }}" method="POST">
            <label for="name">Name:</label>
            <input type="text" id="name" name="name" required><br><br>

            <label for="email">Email:</label>
            <input type="email" id="email" name="email" required><br><br>

            <input type="submit" value="Submit">
        </form>
    </div>
</body>
</html>
```

Contact.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Thank You</title>
    <link rel="stylesheet" href="{{ url_for('static', filename='styles.css') }}">
</head>
<body>
    <div class="container thank-you">
        <h1>Thank You, {{ name }}!</h1>
        <p>Your email: {{ email }}</p>
        <p><a href="{{ url_for('home') }}">Go back to the homepage</a></p>
    </div>
</body>
</html>
```

Conclusion:

In this experiment, we developed a basic Flask web application with multiple routes and demonstrated handling of both GET and POST requests. The application included a homepage, a contact form, and a thank-you page. We implemented a contact form to accept user details and used the POST method to process and display the data. Additionally, we used a GET request with a query parameter to personalize the welcome message on the homepage. This task provided hands-on experience with route creation, form handling, and request methods in Flask.