SET-3

PRACTICAL-1

❖ AIM : A program that creates a simple web server and serves a static HTML page.

Problem Statement:

Develop a Python program that creates a simple web server using Flask and serves a static HTML page. The web server should be able to handle incoming requests and respond by rendering the provided HTML page. The goal is to understand the basics of setting up a web server and serving static content using Flask.

Program Description:

Develop a Python program using Flask to create a simple web server serving a static HTML page. The objective is to use the **flask** command (**flask --app <YourAppName> run**) instead of running the script directly. Ensure proper project organization, write a basic HTML file, and confirm functionality by accessing **http://127.0.0.1:5000/** in a web browser.

Procedure:

Step 1: Install Flask

Make sure Flask is installed. Open a terminal and run the following command:

bash

```bash
pip install Flask
```

Step 2: Create Project Structure

Create a folder for your project and inside it, create two files - **app.py** for your Flask application and a folder named **templates** to store the HTML file.

```
project_folder/

|-- app.py

|-- templates/

|   |-- index.html
```

Step 3: Write the HTML File

Open **templates/index.html** in a text editor and write a simple HTML structure. For example:

index.html

```
<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta http-equiv="X-UA-Compatible" content="IE=edge">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Static HTML Page</title>

</head>

<body>

    <h1>Hello World!</h1>

</body>

</html>
```

Step 4: Write the Flask App

Open **app.py** in a text editor and write the Flask application code:

app.py

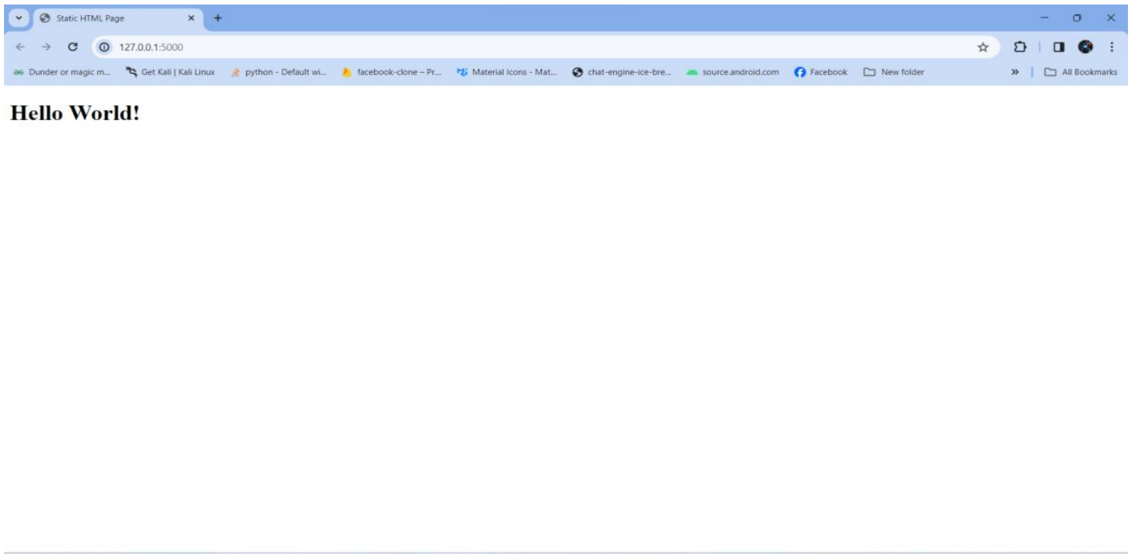
```
from flask import Flask, render_template

app = Flask(__name__)

@app.route("/")

def home():

    return render_template("index.html")

if __name__ == "__main__":

    app.run(debug=True)
```

Step 5: Run the Flask App

In the terminal, navigate to the project folder and run the Flask app using the **flask** command:

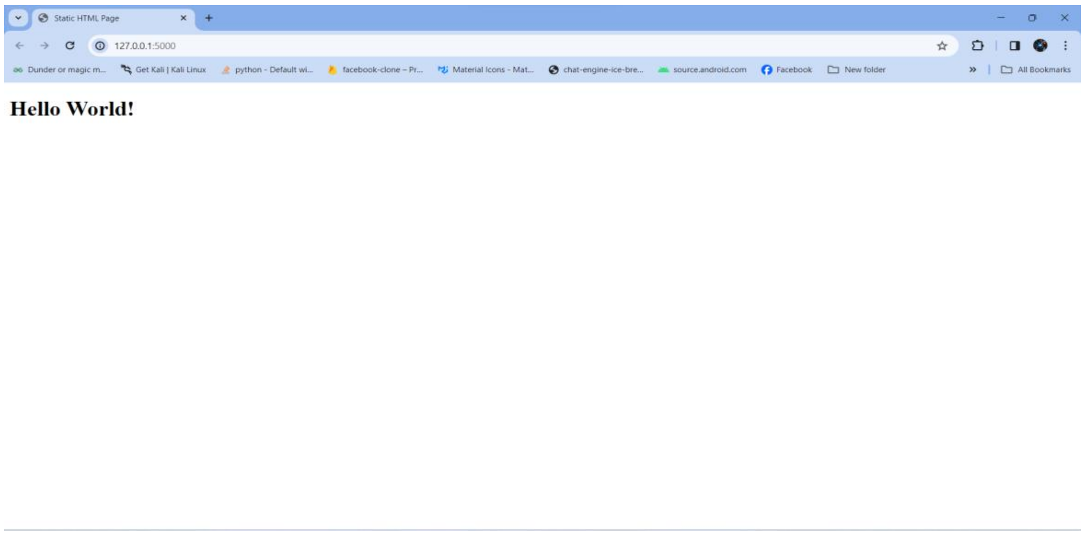
```bash
flask --app app run
```

Expected Output:



Hello World!

Actual Output:



Hello World!

Result:

Upon executing the provided Python program using Flask and the **flask** command (**flask --app <YourAppName> run**), navigate to **http://127.0.0.1:5000/** in a web browser. You should see a static HTML page with the expected content, confirming the successful setup of a basic web server.

PRACTICAL- 2

❖ AIM: A program that creates a web application that allows users to register and login.

Problem Statement:

Develop a Python program that creates a web application allowing users to register and login. The objective is to implement user authentication functionalities using Flask, ensuring secure and seamless user interaction.

Program Description:

The web application will be built using a combination of front-end and back-end technologies. The front-end will be responsible for creating the user interface, while the back-end will handle user registration, login authentication, and data storage.

Procedure:

1. Setup:

Make sure you have Python installed on your system. Install Flask and Flask-SQLAlchemy:

bash

```
pip install Flask Flask-SQLAlchemy
```

2. Create Project Structure :

Create a folder for your project and inside it, create two files - **app.py** for your Flask application and a folder named **templates** to store the HTML file.

```
    project_folder/

    |-- app.py

    |-- templates/

    |   |-- index.html

    |   |-- login.html

    |   |-- register.html
```

3. Write the html files:

Write the index.html, login.html, register.html and design using CSS(Cascading stylesheet):

index.html

```
<!DOCTYPE html>
<html lang="en">
 <head>
  <meta charset="UTF-8" />
  <meta http-equiv="X-UA-Compatible" content="IE=edge" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  <title>Static HTML Page</title>
 </head>
 <style>
  @import url("https://fonts.googleapis.com/css2?family=Poppins:wght@500&display=swap");
  * {
   margin: 0;
   padding: 0;
   box-sizing: border-box;
  }
  body {
   height: 100vh;
   width: 100%;
   display: flex;
   justify-content: center;
   align-items: center;
   flex-direction: column;
   background: #ff5a5f;
  }
  h1 {
   font-family: "Poppins", sans-serif;
   color: #fff;
   margin: 30px 50px;
   font-size: 3rem;
  }
  input {
   padding: 10px 20px;
   border: 3px solid #fff;
   border-radius: 10px;
   background: rgb(16, 208, 16);
   font-size: 1.5rem;
   color: white;
   font-family: "Poppins", sans-serif;
   font-weight: 300;
   transition: .3s;
   &:hover{
    background: #fff;
    color: #000;
    cursor: pointer;
```

```
      }
    }
  </style>
  <body>
    <h1>Hello, this is a static HTML page served by Flask!</h1>
    <form action="{{ url_for('register') }}">
      <input type="submit" value="Register" />
    </form>
  </body>
</html>
```

login.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>User Login</title>
    <style>
      * {
        margin: 0;
        padding: 0;
        box-sizing: border-box;
      }
      body {
        height: 100vh;
        width: 100%;
        display: flex;
        align-items: center;
        justify-content: center;
        flex-direction: column;
        background: rgb(9, 9, 121);
        background: linear-gradient(
          30deg,
          rgba(9, 9, 121, 1) 0%,
          rgba(2, 0, 36, 1) 29%,
          rgba(0, 212, 255, 1) 100%
        );
      }
      .container {
        display: flex;
        align-items: center;
        justify-content: space-evenly;
        flex-direction: column;
        width: 600px;
        border-radius: 20px;
        height: 500px;
        background: #ffffff5a;
  backdrop-filter: blur(20px);
  & h1 {
    font-family: Arial, Helvetica, sans-serif;
    color: #fff;
    margin: 30px 0;
  }
  & li {
    list-style: none;
  }
  & form {
    & label {
      color: white;
      font-family: Arial, Helvetica, sans-serif;
      font-size: 1.4rem;
      margin: 10px 20px;
    }
    & .log_button {
      color: #fff;
      background: red;
      border: none;
      outline: none;
      padding: 5px 10px;
      border-radius: 10px;
      font-size: 1.2rem;
      transition: 0.3s;
      transform: translateX(130px);
      &:hover {
        background: #fff;
        color: #000;
        cursor: pointer;
      }
  }
  & .password{
      padding: 10px 20px;
      border-radius: 20px;
      outline: none;
      border: none;
  }
  & .username{
      padding: 10px 20px;
      border-radius: 20px;
      outline: none;
      border: none;
  }
  & input {
    margin: 10px 20px;
  }
  }
}
.error {
  color: red;
}
```

```
      .success {
        color: green;
      }
      .default {
        color: black;
      }
    </style>
  </head>
  <body>
    <div class="container">
      <h1>User Login</h1>
      {% with messages = get_flashed_messages() %} {% if messages %}
      <ul>
        {% for message in messages %}
        <li
          class="{% if 'error' in message %}error{% elif 'success' in message %}success{% else %}default{% endif
%}"
        >
          {{ message }}
        </li>
        {% endfor %}
      </ul>
      {% endif %} {% endwith %}
      <form method="post" action="{{ url_for('login') }}">
        <label for="username" class="username_label">Username:</label>
        <input type="text" name="username" class="username" required />
        <br />
        <label for="password" class="password_label">Password:</label>
        <input type="password" name="password" class="password" required />
        <br />
        <input type="submit" class="log_button" value="Log in" />
      </form>
      <p>
        Don't have an account?
        <a href="{{ url_for('register') }}">Register here</a>.
      </p>
    </div>
  </body>
</html>
```

register.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>User Registration</title>
    <style>
```

```
* {
  margin: 0;
  padding: 0;
  box-sizing: border-box;
}
body {
  height: 100vh;
  width: 100%;
  display: flex;
  align-items: center;
  justify-content: center;
  flex-direction: column;
  background: rgb(9, 9, 121);
  background: linear-gradient(
    30deg,
    rgba(9, 9, 121, 1) 0%,
    rgba(2, 0, 36, 1) 29%,
    rgba(0, 212, 255, 1) 100%
  );
}
.container {
  display: flex;
  align-items: center;
  justify-content: space-evenly;
  flex-direction: column;
  width: 600px;
  border-radius: 20px;
  height: 500px;
  background: #ffffff5a;
  backdrop-filter: blur(20px);
  & h1 {
    font-family: Arial, Helvetica, sans-serif;
    color: #fff;
    margin: 30px 0;
  }
  & li {
    list-style: none;
  }
  & form {
    & label {
      color: white;
      font-family: Arial, Helvetica, sans-serif;
      font-size: 1.4rem;
      margin: 10px 20px;
    }
    & .register_button {
      color: #fff;
      background: red;
      border: none;
      outline: none;
      padding: 5px 10px;
      border-radius: 10px;
```

```
            font-size: 1.2rem;
            transition: 0.3s;
            transform: translateX(130px);
            &:hover {
              background: #fff;
              color: #000;
              cursor: pointer;
            }
          }
          & .password {
            padding: 10px 20px;
            border-radius: 20px;
            outline: none;
            border: none;
          }
          & .username {
            padding: 10px 20px;
            border-radius: 20px;
            outline: none;
            border: none;
          }
          & input {
            margin: 10px 20px;
          }
        }
      }
      .error {
        color: red;
      }
      .success {
        color: green;
      }
      .default {
        color: black;
      }
    </style>
  </head>
  <body>
    <div class="container">
      <h1>User Registration</h1>
      {% with messages = get_flashed_messages() %} {% if messages %}
      <ul>
        {% for message in messages %}
        <li
          class="{% if 'error' in message %}error{% elif 'success' in message %}success{% else %}default{% endif
%}"
        >
          {{ message }}
        </li>
        {% endfor %}
      </ul>
      {% endif %} {% endwith %}
      <form method="post" action="{{ url_for('register') }}">
        <label for="username" class="username_label">Username:</label>
        <input type="text" name="username" class="username" required />
        <br />
        <label for="password" class="password_label">Password:</label>
        <input type="password" name="password" class="password" required />
        <br />
        <input type="submit" class="register_button" value="Register" />
      </form>
      <p>
        Already have an account?
        <a href="{{ url_for('login') }}">Log in here</a>.
      </p>
    </div>
  </body>
</html>
```

4. Run the Application:

Copy the provided Python code into a file named **app.py**.

app.py

```
from flask import Flask, render_template, request, redirect, url_for, session, flash
from flask_sqlalchemy import SQLAlchemy
from werkzeug.security import generate_password_hash, check_password_hash
import secrets

app = Flask(__name__)
app.secret_key = secrets.token_hex(16)
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///users.db'
db = SQLAlchemy(app)

class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(50), unique=True, nullable=False)
    password = db.Column(db.String(256), nullable=False)

with app.app_context():
    db.create_all()
@app.route("/")
def home():
    return render_template("index.html")

@app.route('/register', methods=['GET', 'POST'])
def register():
    if request.method == 'POST':
        username = request.form['username']
        password = request.form['password']

        if User.query.filter_by(username=username).first():
            flash('Username already taken. Please choose another.', 'error')
        else:
            hashed_password = generate_password_hash(password, method='pbkdf2:sha256')
            new_user = User(username=username, password=hashed_password)
            db.session.add(new_user)
            db.session.commit()
            flash('Registration successful. You can now log in.', 'success')
            return redirect(url_for('login'))

    return render_template('register.html')

@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        username = request.form['username']
        password = request.form['password']

        user = User.query.filter_by(username=username).first()

        if user and check_password_hash(user.password, password):
            session['username'] = username
            flash('Login successful!', 'success')
            return redirect(url_for('dashboard'))
        else:
            flash('Invalid username or password. Please try again.', 'error')

    return render_template('login.html')

@app.route('/dashboard')
def dashboard():
    if 'username' in session:
        return f'Welcome to the dashboard, {session["username"]}!'
    else:
        flash('Please log in to access the dashboard.', 'info')
        return redirect(url_for('login'))

@app.route('/logout')
def logout():
    session.pop('username', None)
    flash('You have been logged out.', 'info')
    return redirect(url_for('login'))

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

5. Open a terminal and run the application inside the directory whereapp.py file is stored and run the following command:

bash

```
flask --app app run
```

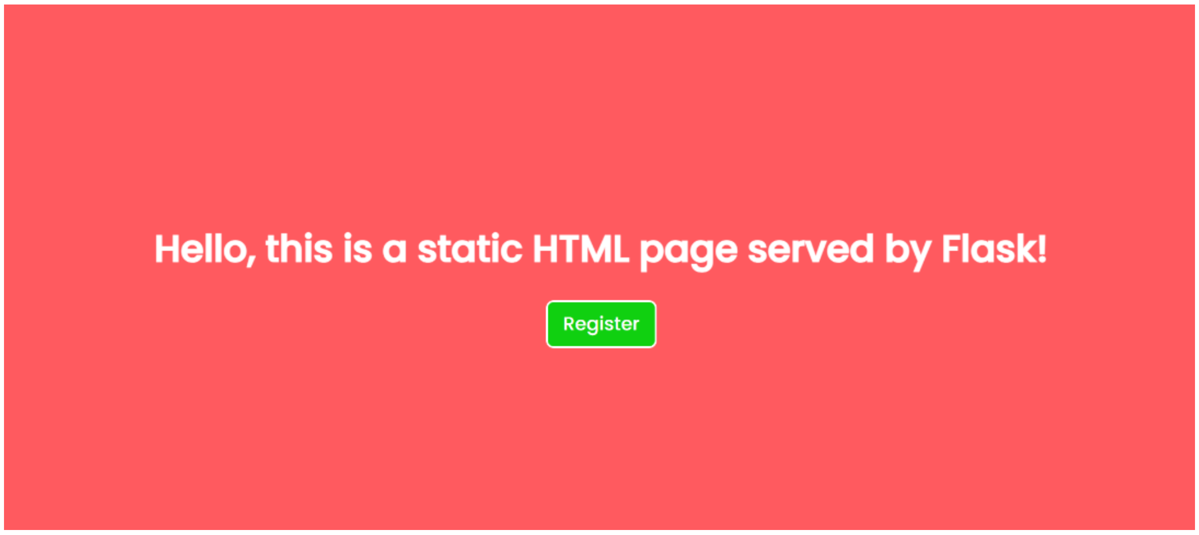
6. The application will be accessible at **http://127.0.0.1:5000/** in your web browser. **Access Home Page:**
   - Navigate to **http://127.0.0.1:5000/**.
   - The home page should be rendered, displaying a link to the login page.

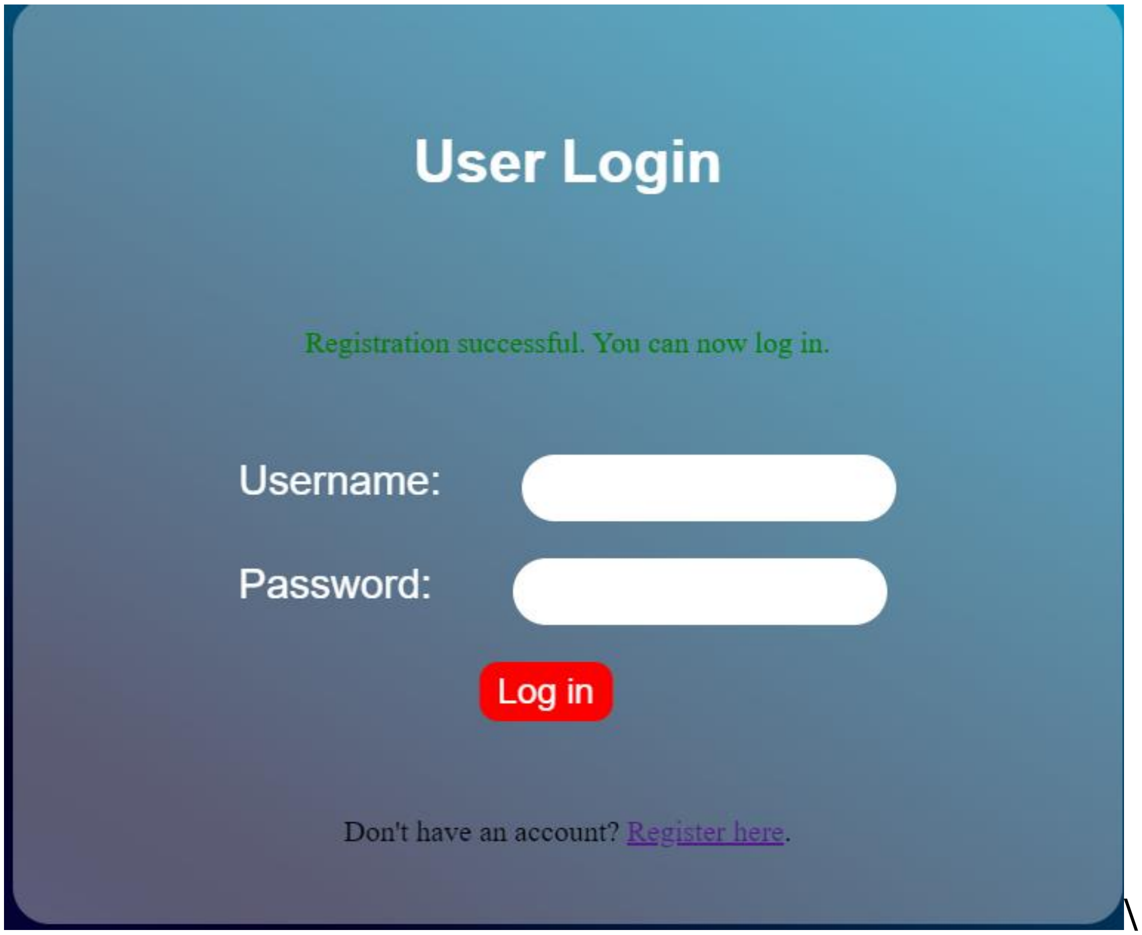
7. **User Registration:**
8. Click on the "Register here" link.

- Enter a unique username and password in the registration form.
- Submit the form to register and receive a success message.

9. **User Login:**
    - Go back to the login page.
    - Enter the registered credentials and log in.
    - Receive a login success message.
10. **Dashboard Access:**
    - Navigate to **http://127.0.0.1:5000/dashboard**.
    - Access the protected dashboard route, displaying a welcome message with the username.
11. **Logout:**
    - Click on the "Logout" link.
    - Log out and receive a logout success message.
12. **Database Exploration:**
    - Check the SQLite database file (**users.db**) to see the stored user information.
13. **Customization:**
    - Modify HTML templates (**login.html** and **register.html**) for appearance changes.
    - Explore Flask features for additional functionalities and security enhancements.
14. **Testing:**
    - Test different scenarios, such as attempting to register with an existing username, entering incorrect login credentials, and accessing the dashboard without logging in.
15. **Further Enhancements:**
    - Add more features, such as password reset functionality, email verification, or profile management.
    - Implement stronger security measures, like using Flask-WTF for form handling and Flask-Login for user session management.

Expected Output:

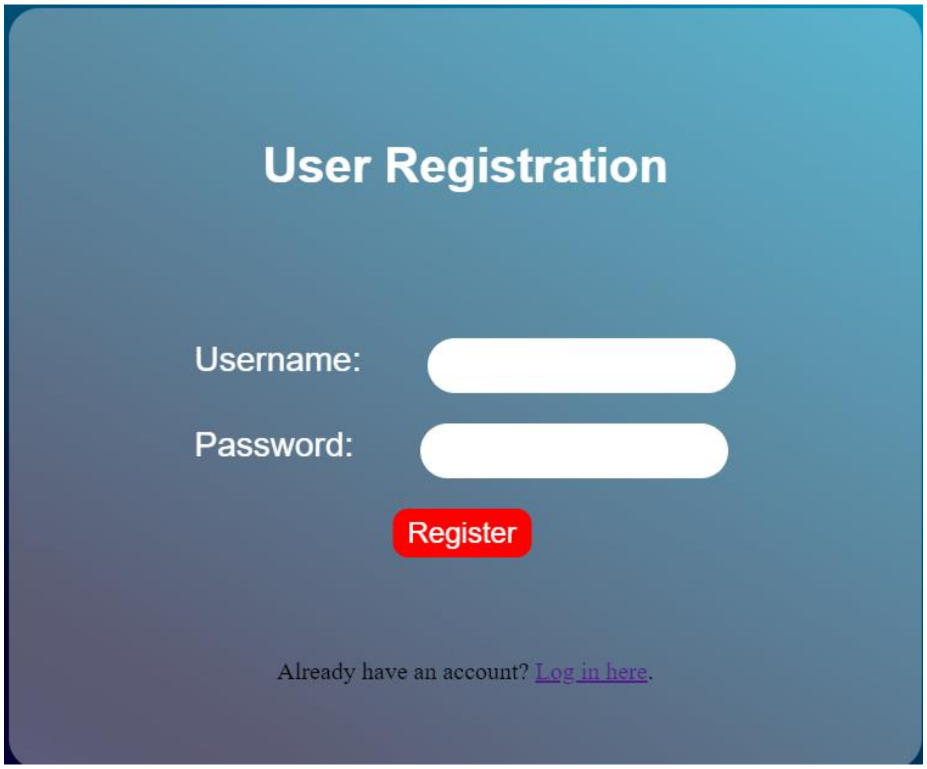

Hello, this is a static HTML page served by Flask!



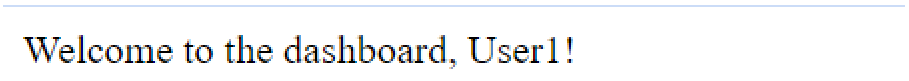
Registration successful. You can now log in.

Don't have an account? Register here.

\



Already have an account? Log in here.


Welcome to the dashboard, User1!

Result:

Upon successful completion of the registration and login processes, the web application will display a confirmation message or redirect the user to a dashboard page. Additionally, the user's information will be securely stored in the database for future logins. The application will provide a secure and user-friendly experience for managing user accounts on the web platform.

PRACTICAL- 3

❖ AIM: A program that creates a web application that allows users to upload and download files.

Problem Statement:

Create a Flask web application for file management, allowing users to upload and download files. The application should provide a user-friendly interface with a file upload form, display a list of uploaded files, and enable users to download files from the list.

Program Description:

Develop a Flask-based web application featuring:

1. **File Upload:**
    - Users can upload files through a web form.
    - Prevent form submission without selecting a file.
    - Save uploaded files to a server directory (e.g., **uploads**).
2. **File Display:**
    - Display a dynamic list of uploaded files on the main page.
    - Each file entry includes a "Download" button.
3. **File Download:**
    - Enable users to download files by clicking the corresponding "Download" button.
4. **User Interface:**
    - Design a clean and user-friendly interface using HTML templates.
    - Separate application code (**app.py**) and HTML templates.

Procedure:

1. Setup:

- Make sure you have Python installed on your system.
- Install Flask:

bash

```bash
pip install Flask
```

2. Project structure:

```
project_folder/

|-- app.py

|-- templates/
```

```
|   |-- index.html
```

3. Write html files:

Write index.html file and style it:

index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>File Upload and Download</title>
</head>
<body>
  <h1>File Upload and Download</h1>
  <form action="/upload" method="post" enctype="multipart/form-data">
    <label for="file">Choose a file:</label>
    <input type="file" name="file" id="file" required>
    <br>
    <input type="submit" value="Upload">
  </form>

  <h2>Uploaded Files</h2>
  {% for filename in filenames %}
    <div>
      <span>{{ filename }}</span>
      <a href="{{ url_for('download_file', filename=filename) }}" download>
        <button>Download</button>
      </a>
    </div>
  {% endfor %}
</body>
</html>
```

4. Run the Application:

- Copy the provided Python code into a file named **app.py**.

app.py

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

import os

app = Flask(__name__)

UPLOAD_FOLDER = 'uploads'

app.config['UPLOAD_FOLDER'] = UPLOAD_FOLDER
```

```
os.makedirs(UPLOAD_FOLDER, exist_ok=True)
@app.route('/')
def index():
    filenames = os.listdir(app.config['UPLOAD_FOLDER'])
    return render_template('index.html', filenames=filenames)
@app.route('/upload', methods=['POST'])
def upload_file():
    if 'file' not in request.files:
        return "No file part"
    file = request.files['file']
    if file.filename == '':
        return "No selected file"
    file.save(os.path.join(app.config['UPLOAD_FOLDER'], file.filename))
    return redirect(url_for('index'))
@app.route('/download/<filename>')
def download_file(filename):
    return send_from_directory(app.config['UPLOAD_FOLDER'], filename)
if __name__ == '__main__':
    app.run(debug=True)
```

- Create a folder named **uploads** in the same directory as **app.py** to store uploaded files.
- Open a terminal and open the directory where the app.py file is saved and run the command:

bash

```
flask --app app run
```

- The application will be accessible at **http://127.0.0.1:5000/** in your web browser.

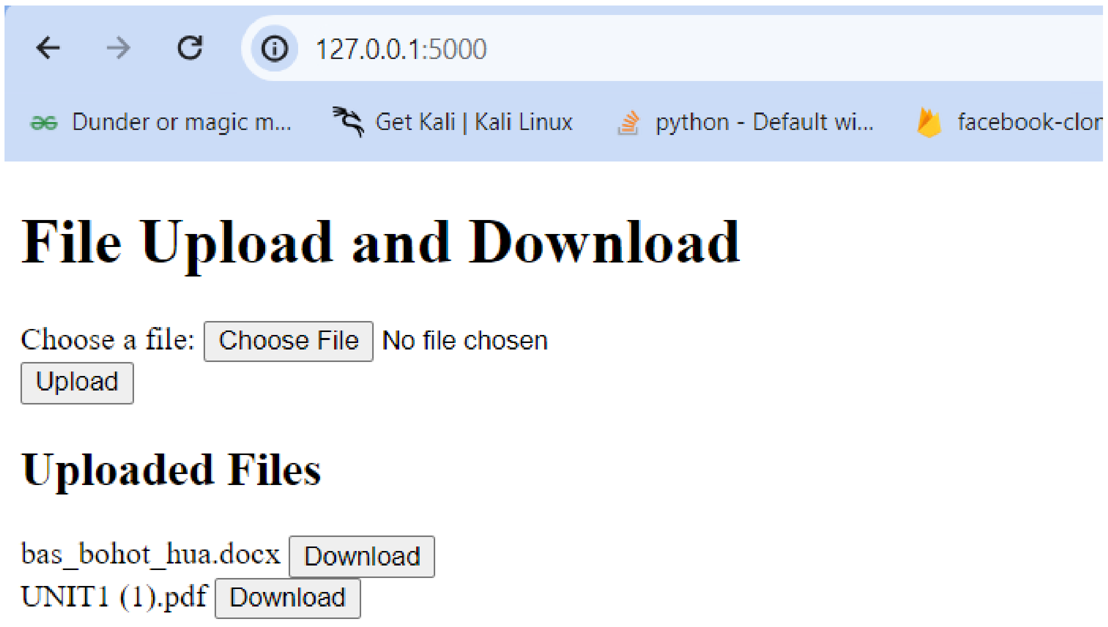
1. **File Upload:**
    - Navigate to **http://127.0.0.1:5000/**.
    - Use the file input form to choose a file and click "Upload."
    - The uploaded file will be saved in the **uploads** folder.
2. **File Download:**
    - View the list of uploaded files on the same page.

- Each file has a "Download" button.
- Click on the "Download" button next to a file to download it.

3. **Customization:**
   - Modify HTML template (**index.html**) for appearance changes.
   - Explore Flask features for additional functionalities, such as handling different file types or implementing user authentication.
4. **Testing:**
   - Test by uploading various file types and sizes.
   - Download the uploaded files to verify the functionality.

- This comprehensive procedure includes the Python code, HTML template, and steps to run, use, and customize the Flask application for file upload and download.

- **Expected Output:**



File Upload and Download

Choose a file: Choose File No file chosen
Upload

Uploaded Files

bas_bohot_hua.docx Download
UNIT1 (1).pdf Download

Result:

Upon completion, users can:

- Upload files using the provided form.
- View a list of uploaded files with download options.

- Download files from the dsisplayed list.
- Experience an organized and visually appealing file management interface.

PRACTICAL- 4

❖ AIM: A program that creates a web application that displays data from a database in a tabular format.

Problem Statement:

Develop a Flask web application that retrieves and displays data from a database in a tabular format. The program should use SQLAlchemy to interact with the database and Pandas to format the data into an HTML table.

Program Description:

The program utilizes Flask, SQLAlchemy, and Pandas to create a web application. It defines a simple SQLAlchemy model (**Person**) to represent data with attributes like name and age. Sample data is inserted into an SQLite database. The main route (/) queries the database, converts the data to a Pandas DataFrame, and renders it as an HTML table using a Flask template (**index.html**).

Procedure:

1. **Setup:**

- Ensure you have Python installed on your system.
- Install Flask and SQLAlchemy:

bash

```
pip install Flask Flask-SQLAlchemy pandas
```

2. **Project structure:**

```
project_folder/

|-- app.py

|-- templates/

|   |-- index.html
```

3. **Write the html files:**

Write index.html file:

index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
```

```
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Data Display</title>
    <link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/css/bootstrap.min.css">
</head>
<body>
    <div class="container mt-5">
        <h1>Data Display</h1>
        <!-- Render the HTML table -->
        {{ table_html | safe }}
    </div>
</body>
</html>
```

4. **Run the Application:**

- Copy the provided Python code into a file named **app.py**.

app.py

```
from flask import Flask, render_template
from flask_sqlalchemy import SQLAlchemy
import pandas as pd

app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///example.db'
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False

# Create a SQLAlchemy instance
db = SQLAlchemy(app)

# Define a model for the data
class Person(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(50), nullable=False)
    age = db.Column(db.Integer, nullable=False)

# Sample data for demonstration
sample_data = [{'name': 'John', 'age': 25},
               {'name': 'Alice', 'age': 30},
               {'name': 'Bob', 'age': 22}]

# Populate the database with sample data
with app.app_context():
    db.create_all()
    for entry in sample_data:
        person = Person(name=entry['name'], age=entry['age'])
        db.session.add(person)
    db.session.commit()

# Define a route to display data in tabular format
```

```
@app.route('/')
def display_data():
   # Query data from the database
   data = Person.query.all()

   # Convert the data to a Pandas DataFrame
   df = pd.DataFrame([(person.name, person.age) for person in data], columns=['name', 'age'])

   # Convert the DataFrame to HTML for rendering in the template
   table_html = df.to_html(classes='table table-striped', index=False)

   return render_template('index.html', table_html=table_html)

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

- Open a terminal and go to the directory where the app.py file is stored and run the application:

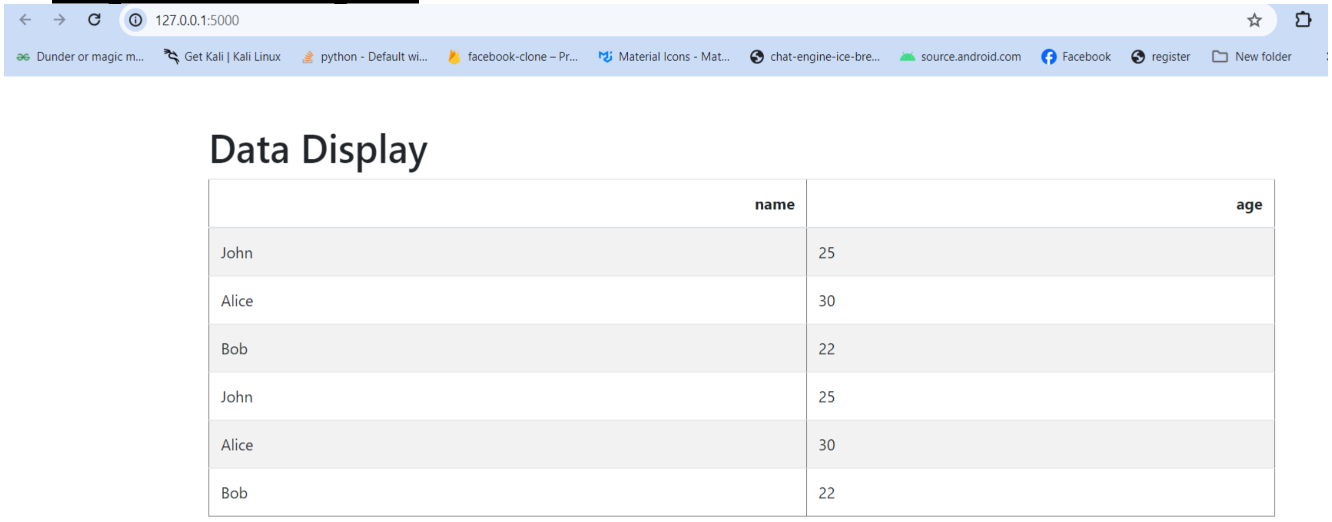
bash

```
flask --app app run
```

  - The application will be accessible at **http://127.0.0.1:5000/** in your web browser.

5. Access Tabular Data:
   - Navigate to http://127.0.0.1:5000/.
   - Observe the tabular display of data retrieved from the database.
6. Database Interaction:
   - Explore the SQLite database (example.db) created by the application.
   - Observe how the sample data is inserted into the database during application startup.
7. Customization:
   - Modify HTML template (index.html) for appearance changes.
   - Adjust the SQLAlchemy model (Person) or database configurations for different data structures.
8. Testing:
     - Test with various data sets to observe how the tabular display adjusts dynamically.

- **Expected Output:**



Data Display

name	age
John	25
Alice	30
Bob	22
John	25
Alice	30
Bob	22

Result:

Upon completion of the steps, you will have a running Flask web application that retrieves data from a database and displays it in a tabular format on the main page. The application showcases the integration of SQLAlchemy and Pandas to handle database interaction and data presentation.

PRACTICAL- 5

❖ AIM: A program that creates a web application that accepts user input and sends it to a server-side script for processing.

Problem Statement:

Develop a Flask web application that allows users to input data via a form on the main page. The program should process the user input on the server side and display the result back to the user.

Program Description:

You are tasked with creating a web application using Flask that enables users to input data on the main page through a form. The entered data should be sent to the server, processed by a server-side script, and the result displayed on the same page. The provided code includes a basic structure for achieving this, where user input is obtained from a form, and a simple processing logic is applied.

Procedure:

1. Setup:

- Ensure you have Python installed on your system.
- Install Flask:

bash

```
pip install Flask
```

2. Project Structure:

```
project_folder/

|-- app.py

|-- templates/

|   |-- index.html
```

3. Write html file:

Write index.html file:

index.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>User Input</title>
  </head>
  <style>
    * {
      margin: 0;
      padding: 0;
      box-sizing: border-box;
    }
    body {
      height: 100vh;
      width: 100%;
      background: #a2d2ff;
      display: flex;
      align-items: center;
      justify-content: center;
      flex-direction: column;
    }
    .container {
      display: flex;
      align-items: center;
      justify-content: space-evenly;
      flex-direction: column;
      width: 500px;
      height: 600px;
      border-radius: 20px;
      background: #ffffff5a;
      backdrop-filter: blur(20px);
      & h1{
        font-family: Arial, Helvetica, sans-serif;
        color: #3a86ff;
        font-size: 2rem;
      }
      & label{
        color: #3a86ff;
        font-family: Arial, Helvetica, sans-serif;
        font-size: 1.2rem;
        padding: 10px;
        margin: 10px 20px;
      }
      & .enter{
        padding: 10px 20px;
        border: none;
        outline: none;
        border-radius: 20px;
      }
      & .submit{
        padding: 10px 20px;
        color: #fff;
        background: #2a9d8f;
        outline: none;
        border: none;
        border-radius: 10px;
        transition: .3s;
        transform: translateX(150px);
        margin: 30px;
        &:hover{
          color: #000;
          cursor: pointer;
          background: #fff;
        }
      }
      & h2{
        font-family: Arial, Helvetica, sans-serif;
        color: #3a86ff;
        font-size: 2rem;
      }
    }
  </style>
  <body>
    <div class="container">
      <h1>User Input Form</h1>
      <form method="post" action="/">
        <label for="user_input">Enter something:</label>
        <input type="text" class="enter" name="user_input" id="user_input" required />
        <br />
        <input class="submit" type="submit" value="Submit" />
      </form>

      {% if result %}
      <div>
        <h2>Result:</h2>
        <p>{{ result }}</p>
      </div>
      {% endif %}
    </div>
  </body>
</html>
```

4. Run the Application:

- Copy the provided Python code into a file named app.py.

app.py

```
from flask import Flask, render_template, request
app = Flask(__name__)
```

```
# Define a route for the main page
@app.route('/', methods=['GET', 'POST'])
def index():
    result = None
    if request.method == 'POST':
        # Get user input from the form
        user_input = request.form.get('user_input')
        result = f"You entered: {user_input}"
    return render_template('index.html', result=result)

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

- Open a terminal and run the application:

bash

```bash
flask --app app run
```

- The application will be accessible at **http://127.0.0.1:5000/** in your web browser.

5. **Access the Web Application:**

    - Navigate to http://127.0.0.1:5000/.
    - Observe the main page with a form that has an input field.

6. **User Input:**

    - Enter text into the input field and submit the form.

7. **Server-side Processing:**

    - Observe how the entered data is processed on the server side.
    - The provided code processes the input by displaying a message. You can replace this with your own processing logic.

8. **Result Display:**

    - View the result displayed on the same page, indicating the processed user input.
    - The result is dynamically updated based on the user's input and server-side processing.

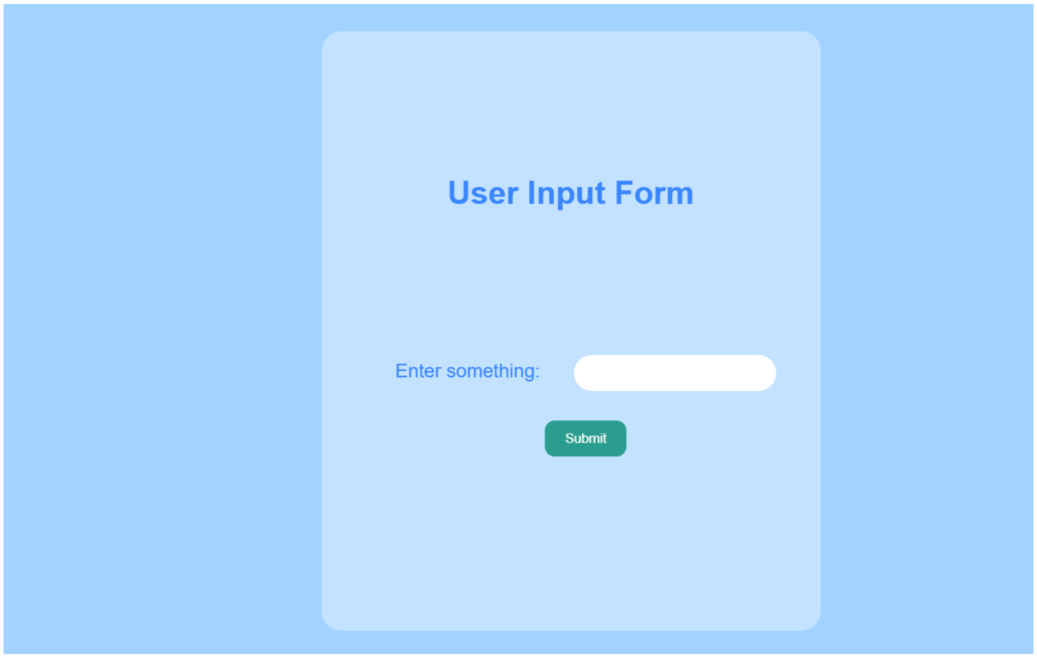
9. **Customization:**

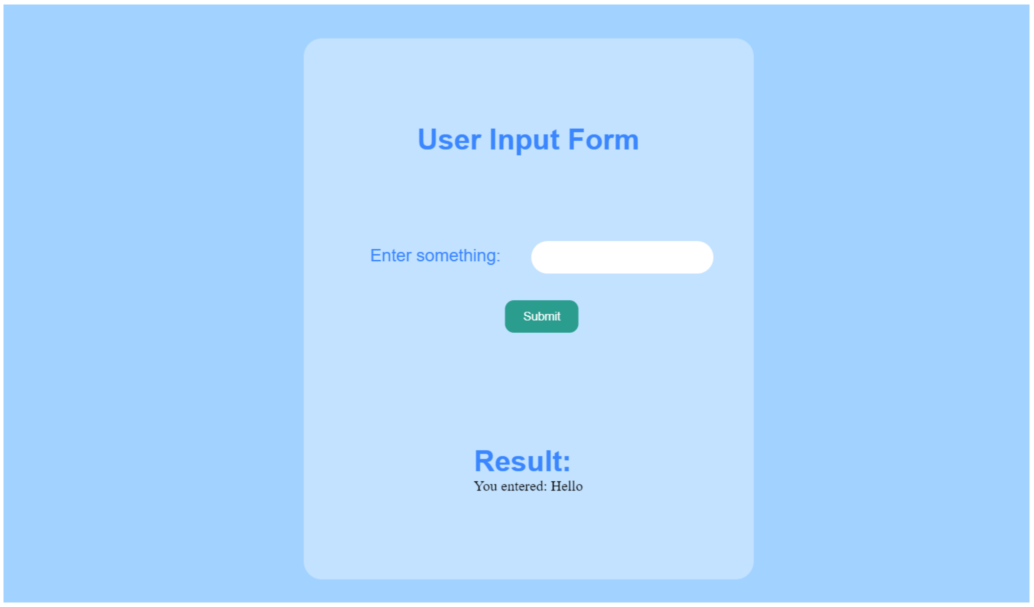
    - Modify HTML template (index.html) for appearance changes.
    - Adjust the server-side processing logic in the Python code based on the specific requirements.

10. **Testing:**

- Test with different input values to observe how the server processes and displays results dynamically.

Expected Outcomes:





You entered: Hello

Result:

Upon completion of the steps, you will have a running Flask web application that accepts user input, sends it to the server for processing, and displays the result on the same page. The application provides a foundation for implementing server-side interactions and dynamic content updates based on user input.