Module 15 - FLASK

Welcome to Module 15! This module introduces you to **Flask**, a lightweight and flexible Python web micro-framework. Flask provides the essentials for web development without imposing a rigid structure, giving you freedom to choose your tools.

Chapter 1: Getting Started with Flask

This chapter covers the foundational steps to set up and run a basic Flask application.

1.1 Introduction to Flask

- **What is Flask?**
  - Flask is a **micro-framework** for building web applications in Python.
  - The "micro" in micro-framework means Flask aims to keep the core simple but extensible. It doesn't include features like a database ORM or form validation out of the box (unlike Django), but it's easy to integrate third-party libraries for these functionalities.
  - It provides the essentials: URL routing, request handling, and templating.
- **Why choose Flask?**
  - **Flexibility:** Gives developers more control over component choices (e.g., choose your own ORM, authentication library, etc.).
  - **Lightweight:** Smaller codebase, less boilerplate, making it quick to get started for simpler projects or APIs.
  - **Simple to learn:** Its API is straightforward and intuitive.
  - **Great for APIs:** Often used to build RESTful APIs.
- **Key Concepts:**
  - **Werkzeug:** A WSGI (Web Server Gateway Interface) utility library that Flask uses internally for request and response objects.
  - **Jinja2:** The templating engine Flask uses by default for rendering HTML.

1.2 Installation of Flask

Just like with Django, it's best practice to install Flask within a virtual environment.

- **Prerequisite:** Ensure you have Python installed (Python 3.x recommended).
- **Virtual Environment:** Navigate to your project directory and activate your virtual environment. If you don't have one, create it.

  Bash

  ```
  # Navigate to your project directory (e.g., where you want to keep
  your Flask projects)
  cd D:\Py_Projects\Flask_Projects

  # Create a virtual environment (if you haven't already)
  python -m venv flask_venv
  ```

```
# Activate the virtual environment
# On Windows:
.\flask_venv\Scripts\activate
# On macOS/Linux:
source flask_venv/bin/activate
```

Expected Output: Your terminal prompt will change to include `(flask_venv)`, indicating the virtual environment is active.

- **Install Flask:** With your virtual environment active, use `pip` to install Flask.

  Bash

  ```
  (flask_venv) pip install Flask
  ```

  Expected Output:

  ```
  Collecting Flask
    Downloading Flask-X.Y.Z-py3-none-any.whl (X.X MB)
  Collecting Werkzeug>=X.Y.Z
  Collecting Jinja2>=X.Y.Z
  ... (other dependencies like itsdangerous, click, MarkupSafe)
  Successfully installed Flask-X.Y.Z Jinja2-X.Y.Z MarkupSafe-X.Y.Z
  Werkzeug-X.Y.Z itsdangerous-X.Y.Z
  ```

  Explanation: This confirms Flask and its core dependencies (like Werkzeug and Jinja2) are installed.

1.3 Basic Webpage 1

Let's create your first Flask application with a single webpage.

- **1. Create your Flask application file:**
  - Create a file named `app.py` in your project root directory (e.g., `D:\Py_Projects\Flask_Projects\app.py`).

  File: `app.py` **Add:**

  Python

  ```
  # app.py
  from flask import Flask # Import the Flask class

  app = Flask(__name__) # Create an instance of the Flask class.
                        # __name__ is a special Python variable that
  gets the name of the current module.

  @app.route('/') # This is a decorator that associates a URL path
  ('/') with the hello_world function.
  def hello_world():
      return "Hello, Flask World!" # The function returns the content
  to be displayed in the browser.

  if __name__ == '__main__':
  ```

```
    app.run(debug=True) # Run the development server.
                        # debug=True enables debug mode (auto-reloads
on code changes, provides debugger).
```

Explanation:

- `from flask import Flask`: Imports the necessary `Flask` class.
- `app = Flask(__name__)`: Initializes your Flask application.
- `@app.route('/')`: This is a "decorator." It tells Flask that when a user visits the root URL (`/`), the `hello_world` function should be executed.
- `def hello_world()`:: This is a **view function**. It defines what content Flask should send back to the user when the associated URL is accessed.
- `app.run(debug=True)`: Starts the development server. `debug=True` is very useful during development as it automatically reloads the server on code changes and provides a debugger.

- **2. Run the Flask development server:**
  - Open your terminal, activate your virtual environment, navigate to your project directory (where `app.py` is).

Bash

```
(flask_venv) flask run
# OR if that doesn't work, ensure FLASK_APP environment variable is
set:
# (flask_venv) set FLASK_APP=app.py (Windows)
# (flask_venv) export FLASK_APP=app.py (macOS/Linux)
# (flask_venv) flask run
# OR simply:
# (flask_venv) python app.py
```

Expected Output:

```
 * 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 stat
 * Debugger is active!
 * Debugger PIN: XXX-XXX-XXX
```

Explanation: This shows your Flask development server is running on `http://127.0.0.1:5000/`.

- **3. Access in browser:**
  - Open your web browser and go to `http://127.0.0.1:5000/`.
  - **Browser Output:** You should see `Hello, Flask World!` displayed on the page.

1.4 Basic Webpage 2

Let's expand our Flask application by adding more routes and understanding dynamic URL segments.

- **1. Add new routes to `app.py`:**

  File: `app.py` **Modify:**

  Python

```
# app.py
from flask import Flask

app = Flask(__name__)

@app.route('/')
def hello_world():
    return "Hello, Flask World!"

@app.route('/about') # New route for the about page
def about():
    return "This is a simple About Us page."

@app.route('/user/<username>') # New route with a dynamic URL segment
def show_user_profile(username):
    # The 'username' captured from the URL is passed as an argument
to the function
    return f'User: {username}'

@app.route('/post/<int:post_id>') # Another dynamic route, specifying
integer type
def show_post(post_id):
    # Flask converts 'post_id' to an integer automatically due to
<int:>
    return f'Post ID: {post_id}'

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

  Explanation:

  - `@app.route('/about')`: Creates a new page accessible at `http://127.0.0.1:5000/about`.
  - `@app.route('/user/<username>')`: This demonstrates dynamic URL segments. The text within `< >` acts as a placeholder. Whatever value is in that part of the URL will be captured and passed as an argument (named `username` in this case) to the `show_user_profile` function.
  - `@app.route('/post/<int:post_id>')`: Similar to `username`, but `<int:post_id>` specifies that the captured segment must be an integer, and Flask will automatically convert it for you. Other converters include `string`, `float`, `path`, and `uuid`.
- **2. Run the server (if not already running) or restart it:**
  - Since `debug=True`, Flask should auto-reload after you save `app.py`. If not, restart it (`Ctrl+C`, then `flask run`).
- **3. Access new URLs in browser:**
  - Go to `http://127.0.0.1:5000/about`

        - **Browser Output:** `This is a simple About Us page.`
    - Go to `http://127.0.0.1:5000/user/Alice`
        - **Browser Output:** `User: Alice`
    - Go to `http://127.0.0.1:5000/user/Bob`
        - **Browser Output:** `User: Bob`
    - Go to `http://127.0.0.1:5000/post/123`
        - **Browser Output:** `Post ID: 123`

1.5 Importing HTML files using Flask

Returning plain strings is not how real websites work. Flask uses **Jinja2** to render HTML templates.

- **1. Create a `templates` folder:**
    - Flask, by default, looks for HTML templates in a folder named `templates` in the same directory as your `app.py` file.
    - Create this folder: `D:\Py_Projects\Flask_Projects\templates\`
- **2. Create an HTML file inside `templates`:**
    - **File:** `templates/index.html`
    - **Add:**

    HTML

    ```
    <!DOCTYPE html>
    <html lang="en">
    <head>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width, initial-
    scale=1.0">
        <title>Home - My Flask App</title>
    </head>
    <body>
        <h1>Welcome to My Flask Application!</h1>
        <p>This content is rendered from an HTML template.</p>
        <p>Current time: {{ current_time }}</p> {# Jinja2 variable
    placeholder #}
        <p>Your name is: {{ user_name }}</p> {# Another variable #}
    </body>
    </html>
    ```

    Explanation: This is a standard HTML file, but it includes `{{ current_time }}` and `{{ user_name }}`. These are **Jinja2 placeholders** (similar to Django Template Language variables) that Flask will replace with actual values passed from the view.

- **3. Modify `app.py` to use `render_template`:**
    - You need to import `render_template` from `flask`.
    - Instead of returning a string, return the result of `render_template()`.
    - Pass data to the template as keyword arguments to `render_template()`.

    File: `app.py` **Modify:**

    Python

```
# app.py
from flask import Flask, render_template # Import render_template
from datetime import datetime # For getting current time

app = Flask(__name__)

@app.route('/')
def hello_world():
    # This route will now redirect to the /home route
    return "Hello, Flask World! (Please go to /home)"

@app.route('/home') # This will be our new primary home page
def home():
    now = datetime.now()
    formatted_time = now.strftime("%Y-%m-%d %H:%M:%S")
    user_name = "Guest" # Example data
    return render_template('index.html', current_time=formatted_time,
user_name=user_name)

@app.route('/about')
def about():
    return "This is a simple About Us page."

@app.route('/user/<username>')
def show_user_profile(username):
    return f'User: {username}'

@app.route('/post/<int:post_id>')
def show_post(post_id):
    return f'Post ID: {post_id}'

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

Explanation:

- `from flask import Flask, render_template`: We import `render_template`.
- `return render_template('index.html', current_time=formatted_time, user_name=user_name)`: Flask finds `index.html` in the `templates` folder. `current_time` and `user_name` are passed as variables to the template.

- **4. Run the server (or restart).**
- **5. Access in browser:**
  - Go to `http://127.0.0.1:5000/home`.
  - **Browser Output:** You should see the HTML content, with the current time and "Guest" dynamically inserted.

Chapter 2: Forms and User Interaction

This chapter covers handling user input through forms in Flask.

2.1 Basic Registration Form

We'll create a simple HTML registration form and process its data using Flask.

- **1. Create a registration HTML template:**
  - **File:** `templates/register.html`
  - **Add:**

  HTML

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>Register</title>
</head>
<body>
    <h1>Register for an Account</h1>
    <form method="POST" action="/register"> {# method="POST" is
crucial #}
        <label for="username">Username:</label><br>
        <input type="text" id="username" name="username"
required><br><br>

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

        <label for="password">Password:</label><br>
        <input type="password" id="password" name="password"
required><br><br>

        <button type="submit">Register</button>
    </form>
</body>
</html>
```

  Explanation:

  - `<form method="POST" action="/register">`: This specifies that when the form is submitted, the data will be sent using the HTTP POST method to the `/register` URL.
  - `name="..."`: The `name` attribute on input fields is how Flask (and any web server) identifies the data when the form is submitted (e.g., `request.form['username']`).
- **2. Modify `app.py` to handle GET and POST for registration:**
  - Import `request` from `flask`.
  - Define a route for `/register` that accepts both `GET` (for displaying the form) and `POST` (for processing the submission).

  File: `app.py` **Modify:**

  Python

```
# app.py
from flask import Flask, render_template, request # Import 'request'
```

```
# ... (rest of your existing code) ...

@app.route('/register', methods=['GET', 'POST']) # Allow both GET and
POST methods
def register():
    if request.method == 'POST':
        # This block runs when the form is submitted (POST request)
        username = request.form['username'] # Access data from the
submitted form
        email = request.form['email']
        password = request.form['password']

        # For now, just print the data to the console
        print(f"New user registered: Username={username},
Email={email}, Password={password}")

        # You'd typically save this to a database here in a real app
        return "Registration successful! Check your console." #
Simple confirmation for now
    else:
        # This block runs when the page is first loaded (GET request)
        return render_template('register.html') # Display the empty
registration form

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

Explanation:

- `methods=['GET', 'POST']`: This argument to `@app.route` tells Flask that this view function can handle both GET (initial page load) and POST (form submission) requests.
- `if request.method == 'POST':`: This condition checks if the incoming request is a POST request.
- `request.form['username']`: This is how you access the data submitted through the form fields. `request.form` is a dictionary-like object containing all the form data.

- **3. Run the server (or restart).**
- **4. Access in browser:**
  - Go to `http://127.0.0.1:5000/register`.
  - **Browser Output:** You should see the registration form.
  - **Action:** Fill in some details and click "Register."
  - **Browser Output:** `Registration successful! Check your console.`
  - **Console Output:** You'll see the details printed in your terminal where the Flask server is running.

2.2 Basic Registration Form 2 (Validation and Error Handling)

Now, let's add some basic server-side validation to our registration form.

- **1. Modify `app.py` for validation:**
  - We'll check if fields are empty and pass error messages back to the template.

File: `app.py` **Modify:**

Python

```
# app.py
from flask import Flask, render_template, request

# ... (rest of your existing code) ...

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

        errors = [] # List to store error messages

        if not username:
            errors.append('Username is required.')
        if not email:
            errors.append('Email is required.')
        if not password:
            errors.append('Password is required.')
        # Basic email format check (more robust validation typically
uses libraries like WTForms)
        elif '@' not in email or '.' not in email:
            errors.append('Please enter a valid email address.')

        if errors: # If there are any errors
            # Re-render the form with error messages and previously
entered data
            return render_template('register.html',
                                   errors=errors,
                                   username=username,
                                   email=email) # Pass back existing
data
        else:
            # If no errors, proceed with registration (for now,
print)
            print(f"New user registered: Username={username},
Email={email}, Password={password}")
            return "Registration successful! (Validated)" #
Confirmation message
    else:
        return render_template('register.html')

# ... (rest of your existing code) ...
```

Explanation:

- `errors = []`: An empty list to accumulate error messages.
- `if not username:`: Checks if the string is empty.
- `errors.append(...)`: Adds an error message to the list.
- `if errors:`: If the `errors` list is not empty, it means there were validation failures.

- `return render_template('register.html', errors=errors, username=username, email=email)`: If errors occur, the template is re-rendered, passing the `errors` list and the data the user previously entered.

- **2. Modify `templates/register.html` to display errors and pre-fill fields:**

  File: `templates/register.html` **Modify:**

  HTML

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>Register</title>
</head>
<body>
    <h1>Register for an Account</h1>

    {# Display error messages if any #}
    {% if errors %}
        <ul style="color: red;">
            {% for error in errors %}
                <li>{{ error }}</li>
            {% endfor %}
        </ul>
    {% endif %}

    <form method="POST" action="/register">
        <label for="username">Username:</label><br>
        <input type="text" id="username" name="username" value="{{
username if username is not none else '' }}" required><br><br>
        {# `value="{{ username if username is not none else '' }}"`
pre-fills the field #}

        <label for="email">Email:</label><br>
        <input type="email" id="email" name="email" value="{{ email
if email is not none else '' }}" required><br><br>

        <label for="password">Password:</label><br>
        <input type="password" id="password" name="password"
required><br><br>

        <button type="submit">Register</button>
    </form>
</body>
</html>
```

  Explanation:

  - `{% if errors %}`: Jinja2 conditional to check if the `errors` list (passed from the view) exists and is not empty.
  - `{% for error in errors %}`: Loops through the `errors` list and displays each error.
  - `value="{{ username if username is not none else '' }}"`: This is Jinja2 syntax to pre-fill the input field with the `username` value if it was

passed from the view; otherwise, it sets it to an empty string. This is crucial for user experience.

- **3. Run the server (or restart).**
- **4. Access in browser:**
  - Go to `http://127.0.0.1:5000/register`.
  - **Action:** Try submitting the form with empty fields or an invalid email.
  - **Browser Output:** Error messages will appear above the form, and the valid data you entered will remain in the fields.
  - **Action:** Submit with valid data.
  - **Browser Output:** `Registration successful! (Validated)`

2.3 Basic Registration Form 3 (Confirmation Display)

Instead of a plain text message, let's display a proper confirmation message on a new page. This leads into the concept of redirecting.

- **1. Modify `app.py` to redirect after successful registration:**
  - Import `redirect` and `url_for` from Flask.
  - After successful registration, redirect to a new route, `registration_success`.

File: `app.py` **Modify:**

Python

```
# app.py
from flask import Flask, render_template, request, redirect, url_for
# Import redirect and url_for

# ... (rest of your existing code) ...

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

        errors = []

        if not username:
            errors.append('Username is required.')
        if not email:
            errors.append('Email is required.')
        if not password:
            errors.append('Password is required.')
        elif '@' not in email or '.' not in email:
            errors.append('Please enter a valid email address.')

        if errors:
            return render_template('register.html',
                                   errors=errors,
                                   username=username,
                                   email=email)
        else:
```

```
            print(f"New user registered: Username={username},
Email={email}, Password={password}")
            # Redirect to a success page after successful
registration
            return redirect(url_for('registration_success',
username=username))
    else:
        return render_template('register.html')

@app.route('/registration_success/<username>') # New route for
success page
def registration_success(username):
    return render_template('success.html', username=username)

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

Explanation:

- `return redirect(url_for('registration_success', username=username))`: This tells the browser to make a new GET request to the URL generated by `url_for('registration_success', username=username)`. `url_for` is safer than hardcoding URLs.
- `url_for('registration_success', username=username)`: It takes the name of the *view function* (`registration_success`) and any arguments (`username`) that the route (`/registration_success/<username>`) expects. This generates the correct URL, e.g., `/registration_success/Alice`.

- **2. Create a success HTML template:**

  File: `templates/success.html` **Add:**

  HTML

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>Registration Success</title>
</head>
<body>
    <h1>Registration Successful!</h1>
    <p>Welcome, {{ username }}! Your account has been created.</p>
    <p><a href="/">Go to Home</a></p> {# Simple link back to home #}
</body>
</html>
```

- **3. Run the server (or restart).**
- **4. Access in browser:**
  - Go to `http://127.0.0.1:5000/register`.
  - **Action:** Fill in valid details and click "Register."
  - **Browser Output:** You will be redirected to `http://127.0.0.1:5000/registration_success/YourUsername` and see the success message.

2.4 Redirecting using Hyperlinks

This section builds directly on 2.3. We've already introduced `redirect()` and `url_for()` with the registration success example.

- **Purpose of `redirect()`:** To tell the browser to navigate to a different URL. This is essential for:
  - **Post-Redirect-Get (PRG) pattern:** After a form submission (POST), you typically redirect to a GET request to prevent duplicate submissions if the user refreshes the page.
  - Navigating between different parts of your application.
  - Handling successful operations.
- **Purpose of `url_for()`:** To dynamically build URLs for your application.
  - Instead of hardcoding `/register` or `/home`, you use `url_for('register')` or `url_for('home')`.
  - **Advantages:**
    - **Maintainability:** If you change a URL route in `app.py`, `url_for` automatically updates the URL everywhere it's used.
    - **Readability:** More descriptive than hardcoded paths.
    - **Reverse URL lookup:** It looks up the URL based on the *view function name*.
- **Examples of `url_for` in action (from previous code):**
  - `return redirect(url_for('registration_success', username=username))`
  - `<p><a href="/">Go to Home</a></p>` (This could be `url_for('home')` for better practice)
- **Let's update the home page to include navigation links using `url_for`:**

  File: `templates/index.html` **Modify:**

  HTML

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>Home - My Flask App</title>
</head>
<body>
    <h1>Welcome to My Flask Application!</h1>
    <p>This content is rendered from an HTML template.</p>
    <p>Current time: {{ current_time }}</p>
    <p>Your name is: {{ user_name }}</p>

    <h2>Navigation</h2>
    <ul>
        <li><a href="{{ url_for('home') }}">Home</a></li> {# Uses
url_for #}
        <li><a href="{{ url_for('about') }}">About Us</a></li> {#
Uses url_for #}
        <li><a href="{{ url_for('register') }}">Register</a></li> {#
Uses url_for #}
```

```
        <li><a href="{{ url_for('show_user_profile',
username='GuestUser') }}">View a Sample User Profile</a></li> {#
Passing argument #}
    </ul>
</body>
</html>
```

Explanation: Now, all internal links are generated dynamically using `url_for`, making your application more robust.

2.5 Redirecting using Hyperlinks 2 (Flash Messages)

When you redirect after a form submission, you often want to display a one-time message (e.g., "Successfully registered!"). Since the context is lost on redirect, Flask provides **flash messages**.

- **How it works:**
    1. In the view, you `flash()` a message.
    2. Flask stores this message in the user's session.
    3. After the redirect, the new request can retrieve (`get_flashed_messages()`) and display these messages.
    4. Once retrieved, they are removed from the session.
- **Prerequisite:** Flask needs a `secret_key` to manage sessions securely.
- **1. Modify `app.py` to use `flash` messages:**
    - Add `app.secret_key`.
    - Import `flash` and `get_flashed_messages`.
    - Call `flash()` after successful registration.

File: `app.py` **Modify:**

Python

```
# app.py
from flask import Flask, render_template, request, redirect, url_for,
flash, get_flashed_messages # Import flash, get_flashed_messages

app = Flask(__name__)
app.secret_key = 'your_super_secret_key_here' # IMPORTANT: Change
this to a strong, unique key in production!

# ... (rest of your existing code) ...

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

        errors = []

        if not username:
            errors.append('Username is required.')
        if not email:
            errors.append('Email is required.')
```

```
        if not password:
            errors.append('Password is required.')
        elif '@' not in email or '.' not in email:
            errors.append('Please enter a valid email address.')

        if errors:
            # Flash error messages directly
            for error in errors:
                flash(error, 'error') # 'error' is a category, can be
used for styling
            return render_template('register.html',
                                   username=username,
                                   email=email) # Don't pass 'errors'
directly anymore
        else:
            print(f"New user registered: Username={username},
Email={email}, Password={password}")
            flash('Registration successful!', 'success') # Flash a
success message
            return redirect(url_for('registration_success',
username=username))
    else:
        return render_template('register.html')

@app.route('/registration_success/<username>')
def registration_success(username):
    return render_template('success.html', username=username)

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

Explanation:

- `app.secret_key = '...'`: A secret key is used by Flask to securely sign session cookies and other data. **Never use a hardcoded, easily guessable key in production.**
- `flash(message, category)`: Stores the message in the session. `category` is optional but useful for styling (e.g., 'success', 'error', 'info').

- **2. Modify templates to display flash messages:**
  - You need to use `get_flashed_messages(with_categories=True)` in your template.

File: `templates/register.html` **Modify:**

HTML

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>Register</title>
</head>
<body>
    <h1>Register for an Account</h1>
```

```
    {# Display flash messages here #}
    {% with messages = get_flashed_messages(with_categories=true) %}
      {% if messages %}
        <ul class="flashes">
        {% for category, message in messages %}
          <li class="{{ category }}">{{ message }}</li> {# Use
category for styling #}
        {% endfor %}
        </ul>
      {% endif %}
    {% endwith %}

    <form method="POST" action="/register">
        <label for="username">Username:</label><br>
        <input type="text" id="username" name="username" value="{{
username if username is not none else '' }}" required><br><br>

        <label for="email">Email:</label><br>
        <input type="email" id="email" name="email" value="{{ email
if email is not none else '' }}" required><br><br>

        <label for="password">Password:</label><br>
        <input type="password" id="password" name="password"
required><br><br>

        <button type="submit">Register</button>
    </form>
</body>
</html>
```

File: `templates/success.html` **Add the flash message display block as well (optional, but good for PRG confirmation):**

HTML

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>Registration Success</title>
</head>
<body>
    {# Display flash messages here #}
    {% with messages = get_flashed_messages(with_categories=true) %}
      {% if messages %}
        <ul class="flashes">
        {% for category, message in messages %}
          <li class="{{ category }}">{{ message }}</li>
        {% endfor %}
        </ul>
      {% endif %}
    {% endwith %}

    <h1>Registration Successful!</h1>
    <p>Welcome, {{ username }}! Your account has been created.</p>
    <p><a href="{{ url_for('home') }}">Go to Home</a></p> {# Updated
with url_for #}
</body>
```

```
</html>
```

Explanation:

- `{% with messages = get_flashed_messages(with_categories=true) %}`: Retrieves messages from the session. `with_categories=true` means each message comes as a tuple `(category, message)`.
- The loop then displays each message, applying the `category` as a CSS class for styling.

- **3. Run the server (or restart).**
- **4. Access in browser:**
  - Go to `http://127.0.0.1:5000/register`.
  - **Action:** Submit with invalid data.
  - **Browser Output:** Error messages appear on the *same page* (no redirect).
  - **Action:** Submit with valid data.
  - **Browser Output:** Redirects to `success.html`, and a green "Registration successful!" message appears at the top.

Chapter 3: Assets and Layout

This chapter covers handling static files and improving template structure.

3.1 Adding Images to Flask

Flask serves static files (images, CSS, JavaScript) from a default `static` folder.

- **1. Create a `static` folder and an `images` subfolder:**
  - Create these folders in your project root: `D:\Py_Projects\Flask_Projects\static\images\`
- **2. Place an image file inside `static/images`:**
  - Find any small image (e.g., a logo) and save it as `flask_logo.png` (or any other name) inside `D:\Py_Projects\Flask_Projects\static\images\`.
- **3. Modify `templates/index.html` to display the image:**
  - Use `url_for('static', filename='...')` to generate the correct URL for the image.

File: `templates/index.html` **Modify:**

HTML

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>Home - My Flask App</title>
</head>
<body>
```

```
    <h1>Welcome to My Flask Application!</h1>
    <p>This content is rendered from an HTML template.</p>
    <p>Current time: {{ current_time }}</p>
    <p>Your name is: {{ user_name }}</p>

    <img src="{{ url_for('static', filename='images/flask_logo.png')
}}" alt="Flask Logo" width="100">

    <h2>Navigation</h2>
    <ul>
        <li><a href="{{ url_for('home') }}">Home</a></li>
        <li><a href="{{ url_for('about') }}">About Us</a></li>
        <li><a href="{{ url_for('register') }}">Register</a></li>
        <li><a href="{{ url_for('show_user_profile',
username='GuestUser') }}">View a Sample User Profile</a></li>
    </ul>
</body>
</html>
```

Explanation: `url_for('static', filename='images/flask_logo.png')` tells Flask to look in the `static` folder for `images/flask_logo.png`. This function generates the correct URL path (e.g., `/static/images/flask_logo.png`).

- **4. Run the server (or restart).**
- **5. Access in browser:**
  - Go to `http://127.0.0.1:5000/home`.
  - **Browser Output:** You should now see the image displayed on the page.

3.2 Template Inheritance

Template inheritance allows you to create a base template that contains common elements (like headers, footers, navigation) and then extend it in other templates, avoiding repetition.

- **1. Create a `base.html` template:**
  - This will contain the common structure.

File: `templates/base.html` **Add:**

HTML

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>{% block title %}My Flask App{% endblock %}</title> {#
Default title #}
    {# Stylesheet link will go here in next section #}
</head>
<body>
    <nav>
        <a href="{{ url_for('home') }}">Home</a> |
        <a href="{{ url_for('about') }}">About</a> |
        <a href="{{ url_for('register') }}">Register</a>
        {# Add more navigation links as needed #}
```

```
    </nav>
    <div class="container">
        {% with messages = get_flashed_messages(with_categories=true)
%}
          {% if messages %}
            <ul class="flashes">
            {% for category, message in messages %}
              <li class="{{ category }}">{{ message }}</li>
            {% endfor %}
            </ul>
          {% endif %}
        {% endwith %}

        {% block content %}{% endblock %} {# This is where child
templates will insert their unique content #}
    </div>
    <footer>
        <p>&copy; {{ '2025' }} My Flask App</p>
    </footer>
</body>
</html>
```

Explanation:

- `{% block title %}{% endblock %}`: Defines a block named `title`. Child templates can override this. If not overridden, "My Flask App" will be the default.
- `{% block content %}{% endblock %}`: Defines a block named `content`. This is the main area where unique content from child templates will be placed.
- Common navigation (`<nav>`) and footer (`<footer>`) are included directly.
- Flash message display block is moved here, so all pages inheriting from `base.html` can display messages.

- **2. Modify existing templates to extend `base.html`:**
  - Remove the boilerplate HTML (head, body, etc.).
  - Use `{% extends 'base.html' %}` and `{% block ... %}`.

File: `templates/index.html` **Modify:**

HTML

```
{% extends "base.html" %} {# Inherit from base.html #}
{% block title %}Home{% endblock %} {# Override the title block #}
{% block content %} {# Fill the content block #}
    <h1>Welcome to My Flask Application!</h1>
    <p>This content is rendered from an HTML template.</p>
    <p>Current time: {{ current_time }}</p>
    <p>Your name is: {{ user_name }}</p>

    <img src="{{ url_for('static', filename='images/flask_logo.png')
}}" alt="Flask Logo" width="100">
    {# Navigation removed, as it's in base.html #}
{% endblock %}
```

File: `templates/register.html` **Modify:**

HTML

```
{% extends "base.html" %}
{% block title %}Register{% endblock %}
{% block content %}
    <h1>Register for an Account</h1>

    {# Flash messages block removed, as it's in base.html #}

    <form method="POST" action="{{ url_for('register') }}"> {#
Updated action to use url_for #}
        <label for="username">Username:</label><br>
        <input type="text" id="username" name="username" value="{{
username if username is not none else '' }}" required><br><br>

        <label for="email">Email:</label><br>
        <input type="email" id="email" name="email" value="{{ email
if email is not none else '' }}" required><br><br>

        <label for="password">Password:</label><br>
        <input type="password" id="password" name="password"
required><br><br>

        <button type="submit">Register</button>
    </form>
{% endblock %}
```

File: `templates/success.html` **Modify:**

HTML

```
{% extends "base.html" %}
{% block title %}Registration Success{% endblock %}
{% block content %}
    {# Flash messages block removed, as it's in base.html #}

    <h1>Registration Successful!</h1>
    <p>Welcome, {{ username }}! Your account has been created.</p>
    <p><a href="{{ url_for('home') }}">Go to Home</a></p>
{% endblock %}
```

New file: `templates/about.html` (since we need a proper template for /about) Add:

HTML

```
{% extends "base.html" %}
{% block title %}About Us{% endblock %}
{% block content %}
    <h1>About Our Flask Application</h1>
    <p>This is a simple demonstration of Flask features, including
template inheritance.</p>
    <p>We aim to make web development easy and fun!</p>
{% endblock %}
```

Explanation:

    - Each child template starts with `{% extends "base.html" %}`.
    - It then defines the `block`s it wants to override or fill. Content outside a `block` in a child template will be ignored.
- **3. Update `app.py` to render `about.html`:**

  File: `app.py` **Modify:**

  Python

```
# app.py
# ... (imports) ...

# ... (app = Flask(__name__), app.secret_key) ...

# ... (hello_world, home routes) ...

@app.route('/about') # Updated to render a template
def about():
    return render_template('about.html') # Render the about.html
template

# ... (user, post routes, register, registration_success) ...
```

- **4. Run the server (or restart).**
- **5. Access in browser:**
    - Navigate between `http://127.0.0.1:5000/home`, `http://127.0.0.1:5000/about`, and `http://127.0.0.1:5000/register`.
    - **Browser Output:** Notice that the navigation bar and footer remain consistent across all pages, while the main content changes.

3.3 Styling the Website

Finally, let's add some CSS to make our Flask application look better.

- **1. Create a CSS file:**
    - Create a CSS file inside your `static` folder.
    - **File:** `static/css/style.css` (Create the `css` subfolder first: `D:\Py_Projects\Flask_Projects\static\css\`)
    - **Add:**

  CSS

```
/* static/css/style.css */
body {
    font-family: 'Segoe UI', Tahoma, Geneva, Verdana, sans-serif;
    background-color: #eef2f7;
    color: #333;
    margin: 0;
    line-height: 1.6;
}

nav {
    background-color: #2c3e50;
    padding: 15px 0;
```

```
    text-align: center;
    box-shadow: 0 2px 5px rgba(0,0,0,0.2);
}

nav a {
    color: #ecf0f1;
    text-decoration: none;
    padding: 10px 20px;
    display: inline-block;
    font-weight: bold;
    transition: background-color 0.3s ease;
}

nav a:hover {
    background-color: #34495e;
    border-radius: 5px;
}

.container {
    padding: 30px;
    max-width: 800px;
    margin: 30px auto;
    background-color: #ffffff;
    border-radius: 10px;
    box-shadow: 0 4px 10px rgba(0,0,0,0.15);
}

h1 {
    color: #3498db;
    text-align: center;
    margin-bottom: 25px;
}

p {
    margin-bottom: 15px;
}

footer {
    text-align: center;
    margin-top: 40px;
    padding: 20px;
    color: #7f8c8d;
    font-size: 0.9em;
    border-top: 1px solid #ddd;
}

/* Styles for flash messages */
.flashes {
    list-style-type: none;
    padding: 15px 20px;
    margin-bottom: 20px;
    border-radius: 8px;
    font-weight: bold;
    text-align: center;
}

.flashes li.error {
    background-color: #f8d7da;
    color: #721c24;
    border: 1px solid #f5c6cb;
}
```

```
.flashes li.success {
    background-color: #d4edda;
    color: #155724;
    border: 1px solid #c3e6cb;
}

/* Form specific styling */
form {
    margin-top: 20px;
    padding: 20px;
    border: 1px solid #eee;
    border-radius: 8px;
    background-color: #fcfcfc;
}
form label {
    display: block;
    margin-bottom: 8px;
    font-weight: bold;
    color: #555;
}
form input[type="text"],
form input[type="email"],
form input[type="password"] {
    width: calc(100% - 20px);
    padding: 10px;
    margin-bottom: 15px;
    border: 1px solid #ccc;
    border-radius: 5px;
    box-sizing: border-box;
}
form button {
    background-color: #28a745;
    color: white;
    padding: 12px 20px;
    border: none;
    border-radius: 5px;
    cursor: pointer;
    font-size: 1em;
    transition: background-color 0.3s ease;
}
form button:hover {
    background-color: #218838;
}
```

- **2. Link the CSS file in `base.html`:**
  - Use `url_for('static', filename='...')` to link your stylesheet in the `<head>` section.

File: `templates/base.html` **Modify:**

HTML

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
```

```
    <title>{% block title %}My Flask App{% endblock %}</title>
    <link rel="stylesheet" href="{{ url_for('static',
filename='css/style.css') }}"> {# Link to your CSS file #}
</head>
<body>
    <nav>
        <a href="{{ url_for('home') }}">Home</a> |
        <a href="{{ url_for('about') }}">About</a> |
        <a href="{{ url_for('register') }}">Register</a>
    </nav>
    <div class="container">
        {% with messages = get_flashed_messages(with_categories=true)
%}
          {% if messages %}
            <ul class="flashes">
            {% for category, message in messages %}
              <li class="{{ category }}">{{ message }}</li>
            {% endfor %}
            </ul>
          {% endif %}
        {% endwith %}

        {% block content %}{% endblock %}
    </div>
    <footer>
        <p>&copy; {{ '2025' }} My Flask App</p>
    </footer>
</body>
</html>
```

Explanation: The `url_for('static', filename='css/style.css')` generates the correct path to your stylesheet (e.g., `/static/css/style.css`), which the browser then loads and applies to your HTML.

- **3. Run the server (or restart).**
- **4. Access in browser:**
    - Navigate through your application: `http://127.0.0.1:5000/home`, `http://127.0.0.1:5000/about`, `http://127.0.0.1:5000/register`.
    - **Browser Output:** Your application should now appear much more styled, with a professional look for navigation, containers, forms, and messages.