Flask Blog Application

Flask Blog Application - Complete Step-by-Step Tutorial

Introduction to the Flask Blog Application

In this comprehensive tutorial, you'll build a fully functional blog application using Flask and SQLite. This is a step up from a simple todo app (which we created earlier) - you'll learn how to create, display, edit, and delete blog posts with a proper user interface using HTML templates and Bootstrap for styling.

What you'll build:

- A homepage displaying all blog posts
- Individual post pages with full content
- A form to create new blog posts
- Edit functionality for existing posts
- Delete functionality with confirmation
- A responsive UI using Bootstrap

What you'll learn:

- Flask routing and view functions
- Jinja2 templating engine
- Template inheritance to avoid code repetition
- Working with SQLite databases
- Form handling and validation
- Flash messages for user feedback
- Static files (CSS) management
- Bootstrap integration

Prerequisites

Before starting, ensure you have:

1. **Python 3.7 or higher** installed
   - Verify: Open Command Prompt and type `python --version`
2. **VS Code** installed
3. **Basic Python knowledge** (variables, functions, loops, dictionaries)
4. **Basic HTML knowledge** (helpful but not required)

Step 1: Project Setup

1.1 Create Project Directory

1. Open **Command Prompt**
2. Navigate to your desired location:

```
cd Desktop
```

3. Create and enter the project folder:

```
mkdir flask_blogcd flask_blog
```

1.2 Open in VS Code

1. Open VS Code
2. Go to **File → Open Folder**
3. Select the `flask_blog` folder

1.3 Create and Activate Virtual Environment

1. Open VS Code terminal: **Terminal → New Terminal**
2. Create virtual environment:

```
python -m venv env
```

3. Activate it:

```
env\Scripts\activate
```

You should see `(env)` at the start of your terminal prompt.

1.4 Install Flask

```
pip install flask
```

Verify installation:

```
python -c "import flask; print(flask.__version__)"
```

Step 2: Create a Simple "Hello World" Application

Let's start with a minimal Flask app to ensure everything works.

2.1 Create hello.py

1. In VS Code, create a new file: `hello.py`
2. Add the following code:

PYTHON

```python
from flask import Flask

# Create Flask application instance
app = Flask(__name__)

# Define route for home page
@app.route('/')
def hello():
    return 'Hello, World! Welcome to FlaskBlog!'

# Run the application
if __name__ == '__main__':
    app.run(debug=True)
```

Code breakdown:

- `Flask(__name__)` - Creates your Flask app instance
- `@app.route('/')` - Decorator that maps the URL `/` to the function below
- `def hello()` - View function that returns the response
- `app.run(debug=True)` - Starts the development server with debug mode
  - Debug mode provides detailed error messages
  - Auto-reloads the server when you change code

2.2 Run Your First App

In the terminal:

```
python hello.py
```

You'll see output like:

```
 * Serving Flask app 'hello'
 * Debug mode: on
 * Running on http://127.0.0.1:5000
 * Restarting with stat
 * Debugger is active!
```

Open your browser and visit: `http://127.0.0.1:5000`

You should see: **Hello, World! Welcome to FlaskBlog!**

🎉 **Your Flask app is running!**

Press `Ctrl + C` to stop the server.

Step 3: Understanding HTML Templates

Instead of returning plain text, web applications return HTML. Flask uses **Jinja2**, a powerful template engine, to generate HTML dynamically.

3.1 Create the Main Application File

1. Create a new file: `app.py`
2. Add this code:

PYTHON

```python
from flask import Flask, render_template

# Create Flask application
app = Flask(__name__)

@app.route('/')
def index():
    """
    Home page route.
    render_template() looks for HTML files in the 'templates' folder.
    """
    return render_template('index.html')

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

What's new:

- `render_template` - Function that renders HTML template files
- `render_template('index.html')` - Looks for `index.html` in a `templates` folder

3.2 Create Templates Folder and First Template

1. Create a folder named `templates` in your `flask_blog` directory
2. Inside `templates`, create `index.html`:

```HTML
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>FlaskBlog</title>
</head>
<body>
   <h1>Welcome to FlaskBlog</h1>
</body>
</html>
```

3.3 Run the Application

```
python app.py
```

Visit `http://127.0.0.1:5000`

You should see a webpage with the heading **Welcome to FlaskBlog**.

Understanding the Flask Project Structure:

```
flask_blog/
│
├── env/                 # Virtual environment
├── templates/           # HTML template files
│   └── index.html
├── hello.py            # Simple test app
└── app.py              # Main application
```

Step 4: Adding CSS Styling

Let's make our blog look better with custom CSS.

4.1 Create Static Files Directory

Flask serves CSS, JavaScript, and images from a `static` folder.

1. Create `static` folder in `flask_blog`
2. Inside `static`, create a `css` folder
3. Inside `css`, create `style.css`

Your structure:

```
flask_blog/
├── static/
│   └── css/
│       └── style.css
└── templates/
    └── index.html
```

4.2 Add CSS Rules

Open `static/css/style.css` and add:

```CSS
/* Basic styling for h1 headings */
h1 {
    border: 2px #eee solid;
    color: brown;
    text-align: center;
    padding: 10px;
}
```

CSS explanation:

- `border` - Adds a 2-pixel solid border with light gray color
- `color` - Sets text color to brown
- `text-align` - Centers the text
- `padding` - Adds 10px space inside the border

4.3 Link CSS to HTML Template

Open `templates/index.html` and update it:

```HTML
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <link rel="stylesheet" href="{{ url_for('static', filename='css/style.css')
}}">
    <title>FlaskBlog</title>
</head>
<body>
   <h1>Welcome to FlaskBlog</h1>
</body>
</html>
```

New concept - `url_for()` function:

- `{{ url_for('static', filename='css/style.css') }}` - Jinja syntax to generate URLs
- `url_for('static', ...)` - Tells Flask to look in the static folder
- `filename='css/style.css'` - Path to the CSS file inside static folder

4.4 Test the Styling

Refresh your browser (or visit `http://127.0.0.1:5000` again).

Your heading should now be brown, centered, and have a border!

Step 5: Using Bootstrap for Professional Styling

Bootstrap is a popular CSS framework that provides pre-made, responsive components. Instead of writing all CSS ourselves, we'll use Bootstrap.

5.1 Create a Base Template

Most web pages share common elements (navigation bar, footer, etc.). Instead of repeating HTML code, we'll create a **base template** that other pages will inherit from.

Create `templates/base.html`:

HTML

```
<!doctype html>
<html lang="en">
  <head>
    <!-- Required meta tags for responsive design -->
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">

    <!-- Bootstrap CSS from CDN -->
    <link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/css/bootstrap.min.css" integrity="sha384-ggOyR0iXCbMQv3Xipma34MD+dH/1fQ784/j6cY/iJTQUOhcWr7x9JvoRxT2MZw1T" crossorigin="anonymous">

    <title>{% block title %} {% endblock %}</title>
  </head>
  <body>
    <!-- Navigation Bar -->
    <nav class="navbar navbar-expand-md navbar-light bg-light">
        <a class="navbar-brand" href="{{ url_for('index')}}">FlaskBlog</a>
        <button class="navbar-toggler" type="button" data-toggle="collapse" data-target="#navbarNav" aria-controls="navbarNav" aria-expanded="false" aria-label="Toggle navigation">
            <span class="navbar-toggler-icon"></span>
        </button>
        <div class="collapse navbar-collapse" id="navbarNav">
            <ul class="navbar-nav">
            <li class="nav-item active">
                <a class="nav-link" href="#">About</a>
            </li>
            </ul>
        </div>
    </nav>

    <!-- Main Content Container -->
    <div class="container">
        {% block content %} {% endblock %}
    </div>

    <!-- Bootstrap JavaScript (for interactive components) -->
    <script src="https://code.jquery.com/jquery-3.3.1.slim.min.js" integrity="sha384-q8i/X+965DzO0rT7abK41JStQIAqVgRVzpbzo5smXKp4YfRvH+8abtTE1Pi6jizo" crossorigin="anonymous"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.14.7/umd/popper.min.js"
```

```
    integrity="sha384-UO2eT0CpHqdSJQ6hJty5KVphtPhzWj9WO1clHTMGa3JDZwrnQq4sF86dIHNDz0W1"
    crossorigin="anonymous"></script>
        <script
    src="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/js/bootstrap.min.js"
    integrity="sha384-JjSmVgyd0p3pXB1rRibZUAYoIIy6OrQ6VrjIEaFf/nJGzIxFDsf4x0xIM+B07jRM"
    crossorigin="anonymous"></script>
      </body>
    </html>
```

Understanding base.html:

1. **Bootstrap CSS Link** - Loads Bootstrap styles from CDN (Content Delivery Network)
2. `{% block title %} {% endblock %}` - Placeholder for page title
   - Child templates can insert their own titles here
3. **Navigation Bar** - Bootstrap navbar component
   - `{{ url_for('index') }}` - Links to home page
4. `{% block content %} {% endblock %}` - Placeholder for main content
   - Child templates insert their unique content here
5. **JavaScript Scripts** - Required for Bootstrap interactive features (dropdowns, modals, etc.)

5.2 Update index.html to Use Base Template

Open `templates/index.html` and replace everything with:

HTML

```
{% extends 'base.html' %}

{% block content %}
    <h1>{% block title %} Welcome to FlaskBlog {% endblock %}</h1>
{% endblock %}
```

Template Inheritance explained:

- `{% extends 'base.html' %}` - This template inherits from base.html
- `{% block content %}` - Fills in the content block from base.html
- `{% block title %}` - Sets both the page title and heading
  - Appears in browser tab AND as the H1 heading

5.3 Test the New Design

Refresh your browser.

You should now see:

- A professional navigation bar at the top
- The heading with Bootstrap styling
- Responsive design (try resizing the browser window)

Step 6: Setting Up the Database

Now let's create a database to store blog posts.

6.1 Design the Database Schema

Create `schema.sql` in the `flask_blog` folder:

```sql
-- Remove existing posts table if it exists
DROP TABLE IF EXISTS posts;

-- Create posts table with necessary columns
CREATE TABLE posts (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    created TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
    title TEXT NOT NULL,
    content TEXT NOT NULL
);
```

Schema explanation:

- `DROP TABLE IF EXISTS posts;` - Deletes old table (useful during development)
- `CREATE TABLE posts` - Creates a new table named "posts"
- `id INTEGER PRIMARY KEY AUTOINCREMENT` - Unique ID for each post
  - `AUTOINCREMENT` - Database automatically assigns next number
- `created TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP` - Creation timestamp
  - `DEFAULT CURRENT_TIMESTAMP` - Automatically set to current time
- `title TEXT NOT NULL` - Post title (required)
- `content TEXT NOT NULL` - Post content (required)

6.2 Create Database Initialization Script

Create `init_db.py`:

```python
import sqlite3

# Connect to database (creates file if doesn't exist)
connection = sqlite3.connect('database.db')

# Read and execute schema
with open('schema.sql') as f:
    connection.executescript(f.read())

# Create cursor for executing commands
cur = connection.cursor()

# Insert sample blog posts
cur.execute("INSERT INTO posts (title, content) VALUES (?, ?)",
            ('First Post', 'Content for the first post')
            )

cur.execute("INSERT INTO posts (title, content) VALUES (?, ?)",
            ('Second Post', 'Content for the second post')
            )

# Commit changes and close connection
connection.commit()
connection.close()

print("Database initialized successfully!")
```

Code explanation:

- `sqlite3.connect('database.db')` - Creates/opens SQLite database file
- `executescript(f.read())` - Executes all SQL commands from schema.sql
- `cur.execute("INSERT ... ")` - Adds sample posts
  - `?` placeholders prevent SQL injection attacks
- `connection.commit()` - Saves all changes to database

6.3 Initialize the Database

Run the script:

```
python init_db.py
```

Output: `Database initialized successfully!`

You'll see a new file: `database.db` in your project folder.

Current project structure:

```
flask_blog/
│
├── env/
├── static/
│   └── css/
│       └── style.css
├── templates/
│   ├── base.html
│   └── index.html
├── app.py
├── hello.py
├── schema.sql
├── init_db.py
└── database.db          # New file!
```

Step 7: Displaying Blog Posts from Database

Now let's modify our application to read and display posts from the database.

7.1 Update app.py with Database Functions

Open `app.py` and replace its contents with:

PYTHON

```python
import sqlite3
from flask import Flask, render_template

app = Flask(__name__)

def get_db_connection():
    """
    Create a database connection with Row factory.
    Row factory allows us to access columns by name like a dictionary.
    """
    conn = sqlite3.connect('database.db')
    conn.row_factory = sqlite3.Row
    return conn

@app.route('/')
def index():
    """
    Home page - displays all blog posts.
    """
    # Connect to database
    conn = get_db_connection()

    # Get all posts
    posts = conn.execute('SELECT * FROM posts').fetchall()

    # Close connection
    conn.close()

    # Render template with posts
    return render_template('index.html', posts=posts)

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

What's new:

1. `get_db_connection()` **function:**
   - Reusable function for database connections
   - `conn.row_factory = sqlite3.Row` - Makes rows behave like dictionaries
   - Can access columns like `row['title']` instead of `row[1]`
2. **Updated** `index()` **function:**
   - Connects to database
   - Executes SQL query: `SELECT * FROM posts`

- `.fetchall()` - Gets all results as a list
- Closes connection (important to prevent memory leaks!)
- Passes `posts` to template

7.2 Update index.html to Display Posts

Open `templates/index.html` and update it:

```HTML
{% extends 'base.html' %}

{% block content %}
    <h1>{% block title %} Welcome to FlaskBlog {% endblock %}</h1>

    {% for post in posts %}
        <a href="#">
            <h2>{{ post['title'] }}</h2>
        </a>
        <span class="badge badge-primary">{{ post['created'] }}</span>
        <hr>
    {% endfor %}
{% endblock %}
```

Jinja2 template syntax:

- `{% for post in posts %}` - Loop through all posts
- `{{ post['title'] }}` - Display post title
  - `{{ ... }}` - Variable/expression delimiter in Jinja2
- `{{ post['created'] }}` - Display creation timestamp
- `<span class="badge badge-primary">` - Bootstrap badge component
- `{% endfor %}` - End the loop
- `<hr>` - Horizontal line separator

7.3 Test the Blog Posts Display

Run the application:

```
python app.py
```

Visit `http://127.0.0.1:5000`

You should see:

- Welcome to FlaskBlog heading
- Two blog posts with titles and timestamps
- Each post is a clickable link (though it doesn't go anywhere yet)

Step 8: Displaying Individual Blog Posts

Now let's create a page to view a single post in detail.

8.1 Create Helper Function to Get Single Post

Open `app.py` and add these imports at the top:

PYTHON

```python
import sqlite3
from flask import Flask, render_template
from werkzeug.exceptions import abort
```

Then add this function after `get_db_connection()`:

PYTHON

```python
def get_post(post_id):
    """
    Get a single post by ID.
    Returns 404 error if post doesn't exist.
    """
    conn = get_db_connection()
    post = conn.execute('SELECT * FROM posts WHERE id = ?',
                        (post_id,)).fetchone()
    conn.close()

    if post is None:
        abort(404)  # Return 404 Not Found error

    return post
```

Function explanation:

- Takes `post_id` as parameter
- Executes SQL query with `WHERE id = ?` to get specific post
- `.fetchone()` - Returns single result (or None if not found)
- `abort(404)` - Returns HTTP 404 error if post doesn't exist
- Returns the post if found

8.2 Create Route for Individual Posts

Add this route at the end of `app.py` (before `if __name__ == '__main__':`):

```python
@app.route('/<int:post_id>')
def post(post_id):
    """
    Display a single post.
    URL format: /1, /2, /3, etc.
    """
    post = get_post(post_id)
    return render_template('post.html', post=post)
```

Route explanation:

- `/<int:post_id>` - Variable route with type converter
  - `<int:post_id>` - Accepts only integers, passes value to function
  - Examples: `/1`, `/2`, `/42`
- Calls `get_post()` to retrieve the post
- Renders `post.html` template with post data

8.3 Create Post Template

Create `templates/post.html`:

```html
{% extends 'base.html' %}

{% block content %}
    <h2>{% block title %} {{ post['title'] }} {% endblock %}</h2>
    <span class="badge badge-primary">{{ post['created'] }}</span>
    <p>{{ post['content'] }}</p>
{% endblock %}
```

Template explanation:

- Extends base.html for consistent layout
- Displays post title in `<h2>` tag
- Shows creation date with Bootstrap badge
- Displays full post content in `<p>` tag

8.4 Update index.html to Link to Posts

Open `templates/index.html` and update the link:

```HTML
{% extends 'base.html' %}

{% block content %}
    <h1>{% block title %} Welcome to FlaskBlog {% endblock %}</h1>

    {% for post in posts %}
        <a href="{{ url_for('post', post_id=post['id']) }}">
            <h2>{{ post['title'] }}</h2>
        </a>
        <span class="badge badge-primary">{{ post['created'] }}</span>
        <hr>
    {% endfor %}
{% endblock %}
```

What changed:

- `href="{{ url_for('post', post_id=post['id']) }}"` - Dynamic URL generation
  - `url_for('post', ...)` - Links to the `post()` view function
  - `post_id=post['id']` - Passes the post ID as a parameter
  - Flask generates URLs like `/1`, `/2`, etc.

8.5 Test Individual Post Pages

Visit these URLs:

- `http://127.0.0.1:5000/1` - First post
- `http://127.0.0.1:5000/2` - Second post
- `http://127.0.0.1:5000/999` - Should show 404 error

Click on post titles from the home page - they should now work!

Step 9: Creating New Blog Posts

Let's add functionality to create new posts.

9.1 Import Additional Flask Functions

Update the imports in `app.py`:

```PYTHON
import sqlite3
from flask import Flask, render_template, request, url_for, flash, redirect
from werkzeug.exceptions import abort
```

New imports:

- `request` - Access form data submitted by users
- `url_for` - Generate URLs for redirects
- `flash` - Show one-time messages to users
- `redirect` - Redirect users to different pages

9.2 Add Secret Key for Flash Messages

Flask uses sessions to store flash messages. Sessions need a secret key for security.

Add this after `app = Flask(__name__)`:

```python
app = Flask(__name__)
app.config['SECRET_KEY'] = 'your_secret_key_here_change_this_in_production'
```

Important notes:

- Secret key should be a long, random string
- In production, use environment variables (not hardcoded)
- Never share your secret key publicly

9.3 Create the Create Route

Add this route to `app.py`:

```python
@app.route('/create', methods=('GET', 'POST'))
def create():
    """
    Create a new blog post.
    GET: Display the form
    POST: Process the form and save to database
    """
    if request.method == 'POST':
        # Get form data
        title = request.form['title']
        content = request.form['content']

        # Validate title
        if not title:
            flash('Title is required!')
        else:
            # Insert into database
            conn = get_db_connection()
            conn.execute('INSERT INTO posts (title, content) VALUES (?, ?)',
                         (title, content))
            conn.commit()
            conn.close()

            # Redirect to home page
            return redirect(url_for('index'))

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

Function breakdown:

1. `methods=('GET', 'POST')` - Accept both request types
   - GET: Display the form
   - POST: Process form submission
2. `if request.method == 'POST':` - Check if form was submitted
3. `request.form['title']` - Access form data by field name
4. **Validation:**
   - `if not title:` - Check if title is empty
   - `flash('Title is required!')` - Show error message
5. **Database insertion:**
   - SQL INSERT statement
   - `commit()` - Save changes

- `close()` - Close connection

6. `redirect(url_for('index'))` - Redirect to home page after success

9.4 Create the Form Template

Create `templates/create.html`:

```HTML
{% extends 'base.html' %}

{% block content %}
<h1>{% block title %} Create a New Post {% endblock %}</h1>

<form method="post">
    <div class="form-group">
        <label for="title">Title</label>
        <input type="text" name="title"
               placeholder="Post title" class="form-control"
               value="{{ request.form['title'] }}">
    </div>

    <div class="form-group">
        <label for="content">Content</label>
        <textarea name="content" placeholder="Post content"
                  class="form-control">{{ request.form['content'] }}</textarea>
    </div>

    <div class="form-group">
        <button type="submit" class="btn btn-primary">Submit</button>
    </div>
</form>
{% endblock %}
```

Form explanation:

1. `method="post"` - Form submits POST request
2. `name="title"` - Field name (matches `request.form['title']`)
3. `value="{{ request.form['title'] }}"` - Preserve data if validation fails
    - If user forgets title, they don't lose their content
4. **Bootstrap classes:**
    - `form-group` - Groups label and input
    - `form-control` - Styles input fields
    - `btn btn-primary` - Styles button

9.5 Update base.html to Show Flash Messages and Add Link

Open `templates/base.html` and update the navbar and content area:

```HTML
<nav class="navbar navbar-expand-md navbar-light bg-light">
    <a class="navbar-brand" href="{{ url_for('index')}}">FlaskBlog</a>
    <button class="navbar-toggler" type="button" data-toggle="collapse" data-target="#navbarNav" aria-controls="navbarNav" aria-expanded="false" aria-label="Toggle navigation">
        <span class="navbar-toggler-icon"></span>
    </button>
    <div class="collapse navbar-collapse" id="navbarNav">
        <ul class="navbar-nav">
        <li class="nav-item">
            <a class="nav-link" href="#">About</a>
        </li>
        <li class="nav-item">
            <a class="nav-link" href="{{url_for('create')}}">New Post</a>
        </li>
        </ul>
    </div>
</nav>

<div class="container">
    {% for message in get_flashed_messages() %}
        <div class="alert alert-danger">{{ message }}</div>
    {% endfor %}
    {% block content %} {% endblock %}
</div>
```

What changed:

1. **New nav link:**
   - `href="{{url_for('create')}}"` - Links to create page
   - Now visible in navigation bar
2. **Flash messages display:**
   - `{% for message in get_flashed_messages() %}`
   - Loops through all flashed messages
   - `alert alert-danger` - Bootstrap alert styling (red box)

9.6 Test Creating Posts

1. Visit `http://127.0.0.1:5000/create`
2. Try submitting without a title - you should see an error message
3. Fill in both title and content - click Submit

4. You should be redirected to home page with your new post visible

---

Step 10: Editing Existing Posts

Let's add the ability to edit posts.

10.1 Create Edit Route

Add this route to `app.py`:

PYTHON

```python
@app.route('/<int:id>/edit', methods=('GET', 'POST'))
def edit(id):
    """
    Edit an existing post.
    GET: Display form with current post data
    POST: Update the post in database
    """
    post = get_post(id)

    if request.method == 'POST':
        title = request.form['title']
        content = request.form['content']

        if not title:
            flash('Title is required!')
        else:
            conn = get_db_connection()
            conn.execute('UPDATE posts SET title = ?, content = ?'
                         ' WHERE id = ?',
                         (title, content, id))
            conn.commit()
            conn.close()
            return redirect(url_for('index'))

    return render_template('edit.html', post=post)
```

Function explanation:

- Similar to `create()` but with differences:
- Gets existing post using `get_post(id)`
- Uses SQL UPDATE instead of INSERT
- `WHERE id = ?` - Updates only the specific post
- Passes `post` to template to show current values

10.2 Create Edit Template

Create `templates/edit.html`:

```HTML
{% extends 'base.html' %}

{% block content %}
<h1>{% block title %} Edit "{{ post['title'] }}" {% endblock %}</h1>

<form method="post">
    <div class="form-group">
        <label for="title">Title</label>
        <input type="text" name="title" placeholder="Post title"
               class="form-control"
               value="{{ request.form['title'] or post['title'] }}">
    </div>

    <div class="form-group">
        <label for="content">Content</label>
        <textarea name="content" placeholder="Post content"
                  class="form-control">{{ request.form['content'] or
post['content'] }}</textarea>
    </div>

    <div class="form-group">
        <button type="submit" class="btn btn-primary">Submit</button>
    </div>
</form>
<hr>
{% endblock %}
```

Key differences from create.html:

- `{{ request.form['title'] or post['title'] }}`
  - Shows form data if validation failed
  - Otherwise shows current database value
- Title shows which post you're editing

10.3 Add Edit Link to Index Page

Open `templates/index.html` and update:

```HTML
{% extends 'base.html' %}

{% block content %}
    <h1>{% block title %} Welcome to FlaskBlog {% endblock %}</h1>
    {% for post in posts %}
        <a href="{{ url_for('post', post_id=post['id']) }}">
            <h2>{{ post['title'] }}</h2>
        </a>
        <span class="badge badge-primary">{{ post['created'] }}</span>
        <a href="{{ url_for('edit', id=post['id']) }}">
            <span class="badge badge-warning">Edit</span>
        </a>
        <hr>
    {% endfor %}
{% endblock %}
```

What's new:

- `href="{{ url_for('edit', id=post['id']) }}"` - Link to edit page
- `badge badge-warning` - Yellow badge for Edit button

10.4 Test Editing

1. Go to home page
2. Click "Edit" on any post
3. Modify the title or content
4. Submit and verify changes appear on home page

Step 11: Deleting Posts

Finally, let's add delete functionality.

11.1 Create Delete Route

Add to `app.py`:

```python
@app.route('/<int:id>/delete', methods=('POST',))
def delete(id):
    """
    Delete a post.
    Only accepts POST requests for security.
    """
    post = get_post(id)
    conn = get_db_connection()
    conn.execute('DELETE FROM posts WHERE id = ?', (id,))
    conn.commit()
    conn.close()
    flash('"{}" was successfully deleted!'.format(post['title']))
    return redirect(url_for('index'))
```

Function explanation:

- `methods=('POST',)` - Only accepts POST requests
  - Can't delete by just visiting a URL (security measure)
  - Must use a form with POST method
- Gets post first (to show title in flash message)
- Executes SQL DELETE statement
- Flashes success message
- Redirects to home page

Why POST only?

- GET requests should never modify data
- Prevents accidental deletions from web crawlers
- Prevents deletions from browser prefetching

11.2 Add Delete Button to Edit Page

Open `templates/edit.html` and add this form before `{% endblock %}`:

```html
<hr>

<form action="{{ url_for('delete', id=post['id']) }}" method="POST">
    <input type="submit" value="Delete Post"
            class="btn btn-danger btn-sm"
            onclick="return confirm('Are you sure you want to delete this post?')">
</form>

{% endblock %}
```

Form explanation:

- `action="{{ url_for('delete', id=post['id']) }}"` - Form submits to delete route
- `method="POST"` - Uses POST method (required by our route)
- `class="btn btn-danger"` - Red button (Bootstrap danger style)
- `onclick="return confirm( ... )"` - JavaScript confirmation dialog
  - Shows "Are you sure?" popup
  - Only submits if user clicks "OK"

11.3 Update Flash Message Styling

Flash messages for deletions should be green (success) not red (danger).

Open `templates/base.html` and update the flash message section:

```HTML
<div class="container">
    {% for message in get_flashed_messages() %}
        <div class="alert alert-success">{{ message }}</div>
    {% endfor %}
    {% block content %} {% endblock %}
</div>
```

Changed `alert-danger` to `alert-success` for green success messages.

11.4 Test Deleting

1. Go to home page
2. Click "Edit" on a post
3. Scroll down and click "Delete Post"
4. Confirm the deletion
5. You should see a success message and the post should be gone

---

Step 12: Complete Application Code

Here's your complete `app.py` file:

PYTHON

```python
import sqlite3
from flask import Flask, render_template, request, url_for, flash, redirect
from werkzeug.exceptions import abort

# Create Flask application
app = Flask(__name__)
app.config['SECRET_KEY'] = 'your_secret_key_here_change_this_in_production'

def get_db_connection():
    """
    Create a database connection with Row factory.
    Row factory allows us to access columns by name like a dictionary.
    """
    conn = sqlite3.connect('database.db')
    conn.row_factory = sqlite3.Row
    return conn

def get_post(post_id):
    """
    Get a single post by ID.
    Returns 404 error if post doesn't exist.
    """
    conn = get_db_connection()
    post = conn.execute('SELECT * FROM posts WHERE id = ?',
                        (post_id,)).fetchone()
    conn.close()

    if post is None:
        abort(404)  # Return 404 Not Found error

    return post

@app.route('/')
def index():
    """
    Home page - displays all blog posts.
    """
    conn = get_db_connection()
    posts = conn.execute('SELECT * FROM posts').fetchall()
    conn.close()
    return render_template('index.html', posts=posts)

@app.route('/<int:post_id>')
def post(post_id):
    """
```

```
    Display a single post.
    URL format: /1, /2, /3, etc.
    """
    post = get_post(post_id)
    return render_template('post.html', post=post)

@app.route('/create', methods=('GET', 'POST'))
def create():
    """
    Create a new blog post.
    GET: Display the form
    POST: Process the form and save to database
    """
    if request.method == 'POST':
        title = request.form['title']
        content = request.form['content']

        if not title:
            flash('Title is required!')
        else:
            conn = get_db_connection()
            conn.execute('INSERT INTO posts (title, content) VALUES (?, ?)',
                         (title, content))
            conn.commit()
            conn.close()
            return redirect(url_for('index'))

    return render_template('create.html')

@app.route('/<int:id>/edit', methods=('GET', 'POST'))
def edit(id):
    """
    Edit an existing post.
    GET: Display form with current post data
    POST: Update the post in database
    """
    post = get_post(id)

    if request.method == 'POST':
        title = request.form['title']
        content = request.form['content']

        if not title:
            flash('Title is required!')
        else:
            conn = get_db_connection()
```

```
            conn.execute('UPDATE posts SET title = ?, content = ?'
                         ' WHERE id = ?',
                         (title, content, id))
            conn.commit()
            conn.close()
            return redirect(url_for('index'))

    return render_template('edit.html', post=post)

@app.route('/<int:id>/delete', methods=('POST',))
def delete(id):
    """
    Delete a post.
    Only accepts POST requests for security.
    """
    post = get_post(id)
    conn = get_db_connection()
    conn.execute('DELETE FROM posts WHERE id = ?', (id,))
    conn.commit()
    conn.close()
    flash('"{}" was successfully deleted!'.format(post['title']))
    return redirect(url_for('index'))

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

Step 13: Final Project Structure

Your complete project should look like:

```
flask_blog/
│
├── env/                          # Virtual environment (don't commit to Git)
│
├── static/                       # Static files (CSS, JS, images)
│   └── css/
│       └── style.css             # Custom CSS (optional with Bootstrap)
│
├── templates/                    # HTML templates
│   ├── base.html                 # Base template (parent)
│   ├── index.html                # Home page
│   ├── post.html                 # Individual post page
│   ├── create.html               # Create post form
│   └── edit.html                 # Edit post form
│
├── app.py                        # Main application
├── hello.py                      # Initial test app (can delete)
├── schema.sql                    # Database schema
├── init_db.py                    # Database initialization
└── database.db                   # SQLite database file
```

Step 14: Understanding Key Concepts

14.1 Flask Routing

Routes map URLs to Python functions:

PYTHON

```python
@app.route('/')                    # Home page
@app.route('/<int:id>')            # Variable route with integer
@app.route('/create', methods=['GET', 'POST'])  # Multiple methods
```

14.2 Jinja2 Template Syntax

Variables and expressions:

HTML

```
{{ variable }}                    <!-- Display variable -->
{{ post['title'] }}               <!-- Dictionary access -->
{{ url_for('index') }}            <!-- Function call -->
```

Control structures:

```HTML
{% for item in items %}           <!-- Loop -->
    {{ item }}
{% endfor %}

{% if condition %}                <!-- Conditional -->
    <p>True</p>
{% else %}
    <p>False</p>
{% endif %}

{% extends 'base.html' %}         <!-- Template inheritance -->
{% block content %}               <!-- Define/override block -->
    <!-- Content here -->
{% endblock %}
```

14.3 Database Operations Pattern

Always follow this pattern:

1. **Connect**

```PYTHON
conn = get_db_connection()
```

2. **Execute SQL**

```PYTHON
result = conn.execute('SELECT * FROM posts').fetchall()
```

3. **Commit (for INSERT/UPDATE/DELETE)**

```PYTHON
conn.commit()
```

4. **Close**

```PYTHON
conn.close()
```

14.4 Form Handling Pattern

Two-step process:

1. **GET request** - Display form
2. **POST request** - Process form data

```python
@app.route('/create', methods=('GET', 'POST'))
def create():
    if request.method == 'POST':
        # Process form
        data = request.form['field_name']
        # Validate, save, redirect
    # Display form
    return render_template('form.html')
```

Step 15: Testing Your Complete Application

15.1 Manual Testing Checklist

Test each feature:

- [ ] **Home page** (`/`)
  - Shows all posts
  - Timestamps are displayed
  - Edit links work
- [ ] **Individual post** (`/1`, `/2`, etc.)
  - Shows full post content
  - Title and date displayed
- [ ] **Create post** (`/create`)
  - Form displays correctly
  - Submitting without title shows error
  - Successful creation redirects to home
- [ ] **Edit post** (`/1/edit`)
  - Form pre-filled with current data
  - Changes save correctly
  - Delete button works
- [ ] **Delete post**
  - Confirmation dialog appears
  - Post is removed
  - Success message displayed

15.2 Create Test Script

Create `test_blog.py` for automated testing:

PYTHON

```
"""
Manual test script for Flask Blog
Run while the Flask app is running
"""

print("=" * 60)
print("FLASK BLOG - MANUAL TEST GUIDE")
print("=" * 60)

print("\n1. HOME PAGE TEST")
print("   URL: http://127.0.0.1:5000/")
print("   Expected: See all blog posts with Edit links")
print("   Action: Visit URL and verify posts are displayed")

print("\n2. VIEW SINGLE POST TEST")
print("   URL: http://127.0.0.1:5000/1")
print("   Expected: See full content of first post")
print("   Action: Visit URL and verify full post displays")

print("\n3. CREATE POST TEST")
print("   URL: http://127.0.0.1:5000/create")
print("   Expected: See form with Title and Content fields")
print("   Actions:")
print("   a) Submit empty form - should see 'Title is required' error")
print("   b) Fill form and submit - should redirect to home with new post")

print("\n4. EDIT POST TEST")
print("   URL: Click 'Edit' on any post from home page")
print("   Expected: Form pre-filled with current post data")
print("   Action: Modify and submit - verify changes on home page")

print("\n5. DELETE POST TEST")
print("   URL: Go to edit page and click 'Delete Post'")
print("   Expected: Confirmation dialog, then post removed")
print("   Action: Confirm deletion - verify post is gone")

print("\n6. 404 ERROR TEST")
print("   URL: http://127.0.0.1:5000/999")
print("   Expected: 404 Not Found error page")
print("   Action: Visit URL with non-existent post ID")

print("\n" + "=" * 60)
print("Start Flask app with: python app.py")
```

```
print("Then follow each test step above")
print("=" * 60)
```

Run it to see the test guide:

```
python test_blog.py
```

Step 16: Common Issues and Solutions

Issue 1: Template Not Found Error

Error message:

```
jinja2.exceptions.TemplateNotFound: index.html
```

Solutions:

1. Ensure `templates` folder exists in project root
2. Check file is named exactly `index.html` (case-sensitive)
3. Verify you're running `app.py` from the correct directory

Issue 2: Database Locked

Error message:

```
sqlite3.OperationalError: database is locked
```

Solutions:

1. Close all database connections in code
2. Make sure no other program has the database open
3. Restart the Flask application

Issue 3: Secret Key Warning

Warning message:

```
WARNING: The config value 'SECRET_KEY' is not set.
```

Solution: Add to `app.py`:

PYTHON

```
app.config['SECRET_KEY'] = 'your_secret_key_here'
```

Issue 4: Form Data Not Showing

Problem: `request.form['title']` raises KeyError

Solutions:

1. Ensure form has `method="post"`
2. Verify input has `name="title"` attribute
3. Check route accepts POST: `methods=('GET', 'POST')`

Issue 5: Bootstrap Not Loading

Problem: Page looks unstyled

Solutions:

1. Check internet connection (Bootstrap loads from CDN)
2. Verify CDN links in `base.html` are correct
3. Check browser console for errors (F12)

Issue 6: Port Already in Use

Error message:

```
OSError: [Errno 98] Address already in use
```

Solutions:

1. Stop other Flask apps running on port 5000
2. Change port: `app.run(debug=True, port=5001)`
3. Kill process using port (in Command Prompt):

```
netstat -ano | findstr :5000taskkill /PID <PID_NUMBER> /F
```

Step 17: Enhancing Your Blog

17.1 Add Character Limits

Limit title length in the form:

```HTML
<input type="text" name="title"
       placeholder="Post title"
       class="form-control"
       maxlength="100"
       value="{{ request.form['title'] or post['title'] }}">
```

17.2 Add Content Validation

In `app.py` create and edit routes:

```PYTHON
if request.method == 'POST':
    title = request.form['title']
    content = request.form['content']

    # Validate both fields
    if not title:
        flash('Title is required!')
    elif not content:
        flash('Content is required!')
    elif len(title) > 100:
        flash('Title is too long (max 100 characters)')
    else:
        # Save to database
        # ...
```

17.3 Improve Post Display

Show post excerpts on home page:

Update `templates/index.html`:

```HTML
{% extends 'base.html' %}

{% block content %}
    <h1>{% block title %} Welcome to FlaskBlog {% endblock %}</h1>
    {% for post in posts %}
        <a href="{{ url_for('post', post_id=post['id']) }}">
            <h2>{{ post['title'] }}</h2>
        </a>
        <span class="badge badge-primary">{{ post['created'] }}</span>
        <a href="{{ url_for('edit', id=post['id']) }}">
            <span class="badge badge-warning">Edit</span>
        </a>

        <!-- Show first 200 characters of content -->
        <p>{{ post['content'][:200] }}{% if post['content']|length > 200 %} ... {%
endif %}</p>

        <hr>
    {% endfor %}
{% endblock %}
```

17.4 Format Timestamps

Make dates more readable using Jinja filters.

Create a custom filter in `app.py`:

```PYTHON
from datetime import datetime

@app.template_filter('datetimeformat')
def datetimeformat(value, format='%B %d, %Y at %I:%M %p'):
    """
    Format a timestamp for display.
    Example: January 15, 2024 at 02:30 PM
    """
    return datetime.strptime(value, '%Y-%m-%d %H:%M:%S').strftime(format)
```

Use in templates:

```HTML
<span class="badge badge-primary">{{ post['created']|datetimeformat }}</span>
```

17.5 Add Post Count

Show number of posts on home page.

Update `index()` in `app.py`:

```python
@app.route('/')
def index():
    conn = get_db_connection()
    posts = conn.execute('SELECT * FROM posts').fetchall()
    post_count = len(posts)
    conn.close()
    return render_template('index.html', posts=posts, post_count=post_count)
```

Display in `index.html`:

```html
<h1>{% block title %} Welcome to FlaskBlog {% endblock %}</h1>
<p class="text-muted">{{ post_count }} post{% if post_count ≠ 1 %}s{% endif %}
published</p>
```

17.6 Add Search Functionality

Add a simple search route in `app.py`:

```python
@app.route('/search')
def search():
    """
    Search posts by title or content.
    """
    query = request.args.get('q', '')

    if query:
        conn = get_db_connection()
        posts = conn.execute(
            'SELECT * FROM posts WHERE title LIKE ? OR content LIKE ?',
            (f'%{query}%', f'%{query}%')
        ).fetchall()
        conn.close()
    else:
        posts = []

    return render_template('search.html', posts=posts, query=query)
```

Create `templates/search.html`:

```HTML
{% extends 'base.html' %}

{% block content %}
    <h1>{% block title %} Search Results {% endblock %}</h1>

    <form method="get" action="{{ url_for('search') }}">
        <div class="form-group">
            <input type="text" name="q" class="form-control"
                   placeholder="Search posts..."
                   value="{{ query }}">
        </div>
        <button type="submit" class="btn btn-primary">Search</button>
    </form>

    <hr>

    {% if query %}
        {% if posts %}
            <p>Found {{ posts|length }} result(s) for "{{ query }}"</p>
            {% for post in posts %}
                <a href="{{ url_for('post', post_id=post['id']) }}">
                    <h3>{{ post['title'] }}</h3>
                </a>
                <p>{{ post['content'][:150] }}...</p>
                <hr>
            {% endfor %}
        {% else %}
            <p>No posts found for "{{ query }}"</p>
        {% endif %}
    {% endif %}
{% endblock %}
```

Add search link to navbar in `base.html`:

```HTML
<li class="nav-item">
    <a class="nav-link" href="{{ url_for('search') }}">Search</a>
</li>
```

Step 18: Best Practices Summary

18.1 Code Organization

Good practices:

- One route per function
- Helper functions for reusable code
- Clear, descriptive function names
- Comments explaining complex logic

18.2 Database Safety

Always:

- Use parameterized queries (`?` placeholders)
- Close connections after use
- Validate user input before database operations

Never:

- Use string formatting for SQL queries
- Trust user input directly

18.3 Security Considerations

For production:

1. Use environment variables for secret keys
2. Implement user authentication
3. Add CSRF protection
4. Validate and sanitize all input
5. Use HTTPS
6. Set secure session cookies

18.4 Error Handling

Add try-except blocks for production:

PYTHON

```python
@app.route('/')
def index():
    try:
        conn = get_db_connection()
        posts = conn.execute('SELECT * FROM posts').fetchall()
        conn.close()
        return render_template('index.html', posts=posts)
    except Exception as e:
        flash(f'An error occurred: {str(e)}')
        return redirect(url_for('index'))
```

Step 19: Deployment Preparation

19.1 Create requirements.txt

List all dependencies:

```
pip freeze > requirements.txt
```

This creates a file with all installed packages and versions.

19.2 Create .gitignore

If using Git, create `.gitignore`:

```
# Virtual environment
env/
venv/

# Database
*.db

# Python cache
__pycache__/
*.pyc

# IDE
.vscode/
.idea/

# Environment variables
.env
```

19.3 Environment Variables

For production, use environment variables for sensitive data.

Create `.env` file (don't commit to Git):

```
SECRET_KEY=your_very_long_random_secret_key_here
DATABASE_URL=database.db
```

Update `app.py`:

```python
import os
from dotenv import load_dotenv

load_dotenv()

app.config['SECRET_KEY'] = os.getenv('SECRET_KEY', 'dev-key')
```

Install python-dotenv:

```
pip install python-dotenv
```

Step 20: Next Steps and Further Learning

20.1 Feature Ideas to Add

Beginner:

- Add "About" page content
- Sort posts by date (newest first)
- Add post categories
- Count views per post

Intermediate:

- User authentication (login/logout)
- Comments system
- Like/favorite posts
- Image uploads

Advanced:

- User profiles
- Rich text editor (WYSIWYG)
- Tags and search filters
- RESTful API
- Deploy to cloud (Heroku, PythonAnywhere)

20.2 Flask Extensions to Explore

- **Flask-Login** - User session management
- **Flask-WTF** - Form handling and validation
- **Flask-SQLAlchemy** - ORM for database operations
- **Flask-Migrate** - Database migrations
- **Flask-Mail** - Send emails

- **Flask-Admin** - Admin interface

20.3 Learning Resources

Official Documentation:

- Flask: https://flask.palletsprojects.com/
- Jinja2: https://jinja.palletsprojects.com/
- SQLite: https://www.sqlite.org/docs.html

Tutorials:

- Flask Mega-Tutorial by Miguel Grinberg
- Real Python Flask tutorials
- Corey Schafer's Flask series (YouTube)

Books:

- "Flask Web Development" by Miguel Grinberg
- "Python Web Development with Flask" by Gareth Dwyer

Complete Command Reference

Project Setup

```shell
# Create project folder
mkdir flask_blog
cd flask_blog

# Create virtual environment
python -m venv env

# Activate virtual environment
env\Scripts\activate

# Install Flask
pip install flask

# Create requirements file
pip freeze > requirements.txt
```

Database Operations

```shell
# Initialize database
python init_db.py

# Reset database (delete and recreate)
del database.db
python init_db.py
```

Running the Application

```shell
# Activate environment (if not already active)
env\Scripts\activate

# Run application
python app.py

# Run on different port
python app.py --port 5001
```

Troubleshooting

```shell
# Check Flask version
python -c "import flask; print(flask.__version__)"

# List installed packages
pip list

# Check which Python
where python

# Deactivate virtual environment
deactivate
```

Final Checklist

Before considering your blog complete, verify:

- [ ] All routes work correctly (/, /create, /edit, /delete)
- [ ] Forms validate input properly
- [ ] Flash messages display correctly
- [ ] Navigation links work

- [ ] Database operations succeed
- [ ] Bootstrap styling applied
- [ ] No console errors (F12 in browser)
- [ ] Code is commented and organized
- [ ] Secret key is set
- [ ] requirements.txt is created

Congratulations! 🎉

You've built a complete, functional blog application with Flask! You now understand:

✅ **Flask fundamentals** - Routing, view functions, request/response ✅ **Jinja2 templating** - Template inheritance, loops, conditionals ✅ **Database operations** - CRUD with SQLite ✅ **Form handling** - GET/POST requests, validation ✅ **User feedback** - Flash messages ✅ **Static files** - CSS, Bootstrap integration ✅ **Project structure** - Organizing a Flask application

What You've Accomplished:

- **Homepage** displaying all posts
- **Individual post pages** with full content
- **Create functionality** with validation
- **Edit functionality** with pre-filled forms
- **Delete functionality** with confirmation
- **Professional UI** with Bootstrap
- **Flash messages** for user feedback
- **Responsive design** that works on mobile

Troubleshooting Guide

Quick Diagnostic Commands

```shell
# Check if Flask is installed
pip show flask

# Verify you're in the right directory
dir
# Should show: app.py, templates/, static/, database.db

# Check if virtual environment is active
where python
# Should point to env\Scripts\python.exe

# Test database exists and has data
python -c "import sqlite3; conn = sqlite3.connect('database.db'); print('Posts:', conn.execute('SELECT COUNT(*) FROM posts').fetchone()[0]); conn.close()"
```

Common Error Messages and Fixes

1. ModuleNotFoundError: No module named 'flask'

```shell
# Solution:
env\Scripts\activate
pip install flask
```

2. AssertionError: View function mapping is overwriting

- You have duplicate route decorators
- Check for functions with the same name

3. ValueError: View function did not return a response

- Ensure all routes return something
- Add `return render_template( ... )` or `return redirect( ... )`

4. jinja2.exceptions.UndefinedError

- Variable doesn't exist in template context
- Check you're passing the variable from the route

Appendix: Complete File Listings

A. schema.sql

```sql
DROP TABLE IF EXISTS posts;

CREATE TABLE posts (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    created TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
    title TEXT NOT NULL,
    content TEXT NOT NULL
);
```

B. init_db.py

```python
import sqlite3

connection = sqlite3.connect('database.db')

with open('schema.sql') as f:
    connection.executescript(f.read())

cur = connection.cursor()

cur.execute("INSERT INTO posts (title, content) VALUES (?, ?)",
            ('First Post', 'Content for the first post')
            )

cur.execute("INSERT INTO posts (title, content) VALUES (?, ?)",
            ('Second Post', 'Content for the second post')
            )

connection.commit()
connection.close()

print("Database initialized successfully!")
```

C. Complete templates/base.html

HTML

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">
    <link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/css/bootstrap.min.css" integrity="sha384-ggOyR0iXCbMQv3Xipma34MD+dH/1fQ784/j6cY/iJTQUOhcWr7x9JvoRxT2MZw1T" crossorigin="anonymous">
    <title>{% block title %} {% endblock %}</title>
  </head>
  <body>
    <nav class="navbar navbar-expand-md navbar-light bg-light">
        <a class="navbar-brand" href="{{ url_for('index')}}">FlaskBlog</a>
        <button class="navbar-toggler" type="button" data-toggle="collapse" data-target="#navbarNav" aria-controls="navbarNav" aria-expanded="false" aria-label="Toggle navigation">
            <span class="navbar-toggler-icon"></span>
        </button>
        <div class="collapse navbar-collapse" id="navbarNav">
            <ul class="navbar-nav">
            <li class="nav-item">
                <a class="nav-link" href="#">About</a>
            </li>
            <li class="nav-item">
                <a class="nav-link" href="{{url_for('create')}}">New Post</a>
            </li>
            </ul>
        </div>
    </nav>
    <div class="container">
        {% for message in get_flashed_messages() %}
            <div class="alert alert-success">{{ message }}</div>
        {% endfor %}
        {% block content %} {% endblock %}
    </div>
    <script src="https://code.jquery.com/jquery-3.3.1.slim.min.js" integrity="sha384-q8i/X+965DzO0rT7abK41JStQIAqVgRVzpbzo5smXKp4YfRvH+8abtTE1Pi6jizo" crossorigin="anonymous"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.14.7/umd/popper.min.js" integrity="sha384-UO2eT0CpHqdSJQ6hJty5KVphtPhzWj9WO1clHTMGa3JDZwrnQq4sF86dIHNDz0W1"
```

```
    crossorigin="anonymous"></script>
        <script
    src="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/js/bootstrap.min.js"
    integrity="sha384-JjSmVgyd0p3pXB1rRibZUAYoIIy6OrQ6VrjIEaFf/nJGzIxFDsf4x0xIM+B07jRM"
    crossorigin="anonymous"></script>
      </body>
    </html>
```

Summary of Key Learning Points

Flask Architecture

- **App instance** - `Flask(__name__)`
- **Routes** - `@app.route('/')`
- **View functions** - Python functions that return responses
- **Templates** - HTML files with Jinja2 syntax

Jinja2 Features

- **Variables** - `{{ variable }}`
- **Loops** - `{% for item in items %}`
- **Conditionals** - `{% if condition %}`
- **Inheritance** - `{% extends 'base.html' %}`
- **Blocks** - `{% block content %}`

Database Pattern

1. Connect to database
2. Execute SQL query
3. Fetch results
4. Commit (for modifications)
5. Close connection

Request-Response Cycle

1. User requests URL
2. Flask matches route
3. View function executes
4. Database operations (if needed)
5. Template rendered
6. HTML sent to browser

Happy Coding! You're now a Flask developer! 🚀