Building a Website with Flask: Technical Explanation

Introduction

Flask is a lightweight and flexible Python web framework that allows developers to build web applications quickly with minimal overhead. In this essay, I will explain how I used Flask to build my website, focusing on the core concepts such as routing, templating, form handling, and deployment. Each of these concepts plays a crucial role in how the application works and interacts with the user.

Flask: The Micro Framework

Flask is considered a micro-framework because it provides the essential components needed to build a web application without imposing unnecessary complexity. Unlike full-stack frameworks like Django, Flask is unopinionated and doesn't come with predefined tools or structure. This gives developers the flexibility to structure their applications the way they see fit while still providing essential functionality, such as handling requests, templating, and sessions.

The main reason I chose Flask for building my website was its simplicity and ability to scale. Flask allows developers to focus on creating their application logic without the overhead of larger frameworks. The Flask core is minimal, but I could extend its functionality with additional libraries and components as needed.

Routing and Views

In Flask, the central concept is routing—the process of mapping URLs to specific functions (views) in the application. When a user navigates to a particular URL, Flask processes the request, maps it to the corresponding function, and returns the generated response. Each function in Flask corresponds to a route and handles the request and response lifecycle.

For example, when I built the contact form for my website, I defined two main routes: the home page, which displayed the content, and the contact page, which handled form submissions and sent email notifications. Flask uses decorators to bind a function to a route, enabling the app to process the form data and send emails upon submission.

This flexibility allowed me to build the website step by step, creating routes for each page or action, executing the appropriate logic, and returning the result.

Templating with Jinja2

Flask uses Jinja2 as its templating engine. Jinja2 allows developers to dynamically generate HTML by embedding Python-like expressions in HTML files. This means that instead of writing static HTML, you can embed logic directly in the template to render dynamic content based on data passed from Flask.

In my case, Jinja2 was used to render HTML templates for pages like the home page and contact form. The templates are stored in the templates directory, and when Flask renders a template, it substitutes variables and expressions with actual values. For example, when

rendering the contact form, I could pass data such as success or error messages and display them in the template.

One of the advantages of using Jinja2 is its ability to handle conditional logic and loops. For example, I could conditionally display different messages based on whether the user had successfully submitted the form or if there were errors in the submission. Jinja2 also allows for inheritance of templates, meaning I could create a base layout (such as a header, footer, and navigation) and extend it across multiple pages without duplicating code.

Handling Forms and Validating Input

Handling user input through forms is a core part of many web applications. Flask makes it easy to process forms using the request object, which captures data sent via the POST method. When the user submits the form, Flask processes the data and makes it available for further use, such as validation or database insertion.

In my project, the contact form allowed users to submit their name, email, and message. Flask captured the form data using the request.form object. I performed basic validation to ensure that the fields were not empty and that the email address was in the correct format. If the form was valid, the data was processed further (i.e., sending the email to my inbox). If there was an issue, the user would see an error message indicating what went wrong.

In Flask, you can also handle file uploads, sessions, and other types of data through the request object, making it a versatile tool for managing user interactions.

Sending Emails with Flask-Mail

Flask is often used to build dynamic, interactive websites, and one of the key features of my site was the ability to receive and send emails from the contact form. To handle email functionality, I used the Flask-Mail extension, which simplifies sending emails within a Flask application.

Flask-Mail integrates with Flask's configuration system, allowing me to set email parameters such as the mail server, port, and authentication credentials. After receiving form data, Flask-Mail allowed me to create an email message and send it to a specified recipient. The email message could include dynamic content, such as the name, email address, and message of the person submitting the form.

By using Flask-Mail, I could handle email delivery asynchronously, ensuring that the application remained responsive and that users received timely responses without delay.

Environment Configuration with python-dotenv

To avoid exposing sensitive information such as email credentials in the source code, I used python-dotenv to store these details in a .env file. This file is loaded into the Flask app at runtime, keeping the credentials secure and separate from the application code. Flask, by default, doesn't load environment variables, so I used python-dotenv to load the variables stored in the .env file into the Flask environment.

By using environment variables for sensitive configurations, I ensured that the application was both secure and flexible. This approach also made it easy to configure the app for

different environments, such as development, staging, and production, by simply changing the contents of the .env file.

Conclusion

Overall, Flask provided the perfect balance of simplicity and power, making it an ideal choice for building my website from scratch. The concepts I implemented—routing, templating, form handling, and deployment—are fundamental to any web application and showcase the core capabilities of Flask as a web framework.