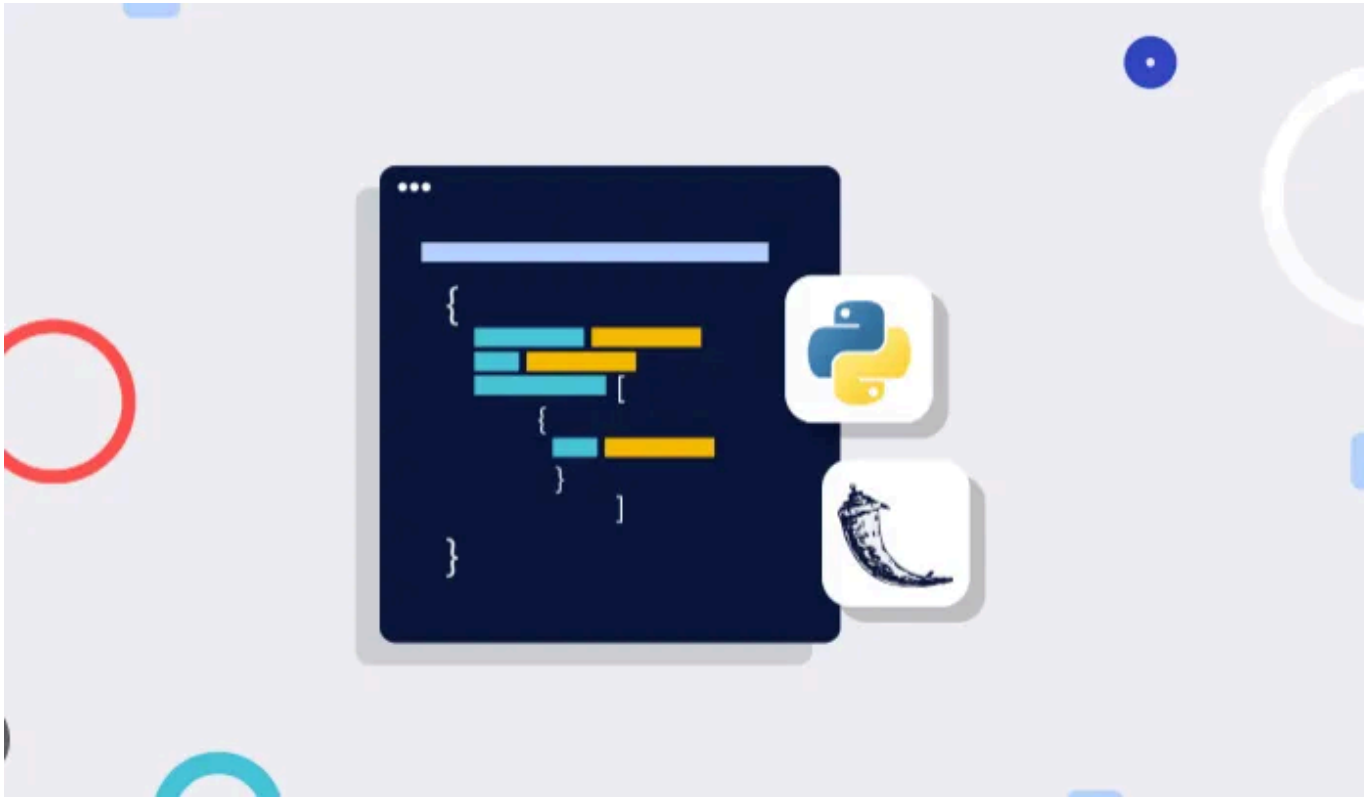
Building a Flask Blog: A Step-by-Step Guide for Beginners

Noran Saber Abdelfattah · Follow

16 min read · Dec 19, 2023

Introduction



In this tutorial, we will systematically guide you through the process of creating a blog post using Flask. I undertook this project as a means of practicing Flask and backend development, and I'll walk you through each step. Let's get started!

Table of Contents

- Step 1: Install Flask
- Step 2: Creating a Base Application
- Step 3: Utilizing HTML Templates
- Step 4: Setting up the Database
- Step 5: Displaying All Posts
- Step 6: Displaying a Single Post
- Step 7: Editing, Creating, and Deleting Posts

Step 1: Install Flask

The flask can be easily installed using the Python package manager, pip. In the terminal or command prompt, enter the following command and press Enter:

```
pip install flask
```

After finishing the installation process, verify it by executing the following command:

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

This command utilizes the Python command line interface with the `-c` option to execute Python code. It begins by importing the Flask package with `import flask;` and then prints the Flask version, which is accessed through the `flask.__version__` variable.

Step 2: Creating a Base Application

- If you are using the vs code now you can create a hello.py file,
- If you are using nano

```
nano app.py
```

- if you are using Vim

```
vi app.py
```

Create a new file named `app.py` in your project folder and add the following code:

```
from flask import Flask

app = Flask(__name__)
```

```
@app.route('/')
def hello():
    return 'Hello, World!'
```

Explanation:

- We import the `Flask` class from the Flask module.
- We create an instance of the Flask class and name it `app`.
- The `@app.route('/')` decorator associates the function `hello()` with the root URL ('/').
- The `hello()` function returns the string 'Hello, World!' when the root URL is accessed.

Run

```
flask run
```

- To run your application

Step 3: Utilizing HTML Templates

Step 1: Understanding the Need for HTML in Web Applications

Your initial application only delivers a plain message without any structure. To enhance user experience, web applications rely on HTML to organize and present information on the browser. In the upcoming steps, you'll integrate

HTML files into your Flask app. These HTML files, often referred to as templates, will serve as the foundation for creating various pages within your application.

Step 2: Introduction to render_template() in Flask

Flask simplifies HTML management through the `render_template()` helper function, which harnesses the power of the Jinja template engine. By employing this function, you can store your HTML code in separate .html files, promoting a cleaner structure and enabling the use of logic within your HTML. These HTML templates will be pivotal in constructing different sections of your application, such as the main page displaying blog posts, individual blog post pages, and the interface for adding new posts.

In this step, you'll initiate the creation of your primary Flask application by setting up a new file. This lays the groundwork for incorporating HTML templates into your application, facilitating the development of diverse pages with distinct functionalities.

Setting Up Your New Flask Application File

- In your new file, named `app.py`, you're gearing up to create a Flask application instance, just like you did before. Additionally, you're introducing the `render_template()` helper function, a key player in handling HTML templates. These templates, which you're about to organize into a folder named 'templates,' will enhance the visual appeal of your application.

```python
# Import necessary modules
from flask import Flask, render_template

# Create a Flask application instance
app = Flask(__name__)

# Define a view function for the main route '/'
@app.route('/')
def index():
    return render_template('index.html')
```

Understanding the View Function and render_template()

In the provided code, the `index()` view function is your application's entry point for the main route '/'. It is used `render_template()` to pull content from an HTML file, which, in this case, is 'index.html.' However, the 'templates' folder and the 'index.html' file are not yet created. If you were to run the application now, you'd encounter an error. Don't worry; this is a common situation for beginners, and we'll address it shortly.

Preparing for the Next Steps

You’re now equipped with the foundation of your Flask application. Before running it, you’ll encounter an error due to the absence of the ‘templates’ folder and ‘index.html’ file. This intentional error serves as a learning experience. After running the application and witnessing the error, you’ll proceed to create the necessary folder and file in the upcoming steps, resolving the issue and advancing your understanding of the Flask application structure

Addressing the Missing Templates Folder and HTML File

To resolve the earlier error, you need to create a ‘templates’ directory within your ‘flask_blog’ directory. Additionally, you’ll create an ‘index.html’ file inside this newly formed ‘templates’ directory.

```
# Create 'templates' directory
mkdir templates
```

```
# Open 'index.html' for editing
vi templates/index.html
```

Adding Basic HTML Content to ‘index.html’

Inside ‘index.html,’ insert the following HTML code to create a basic structure for your page.

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

Incorporating CSS Styling

Now, you'll set up a CSS file to enhance your application's visual appeal. First, create the necessary folders:

```
# Create 'static' directory
mkdir static

# Create 'css' directory inside 'static'
mkdir static/css

# Open 'style.css' for editing
vi static/css/style.css
```

Adding Styling to 'style.css'

Inside 'style.css,' add the following CSS rules to style the `<h1>` tag:

```
h1 {
    border: 2px #eee solid;
    color: brown;
    text-align: center;
    padding: 10px;
}
```

Linking CSS to 'index.html'

Open 'index.html' again to link the CSS file within the `<head>` section.

```
vi templates/index.html
```

Insert the following line within the `<head>` section:

```
<link rel="stylesheet" href="{{ url_for('static', filename= 'css/style.css') }}"
```

`url_for` is a helpful function in Flask that helps create the correct web address (URL) for a given function or endpoint in your web application.

Instead of writing the URL directly in your HTML code, you use `url_for` it to generate it dynamically. This is handy because if you ever change the structure of your URLs, `url_for` will automatically update them, preventing errors and making your code more flexible and maintainable.

Essentially, it ensures that your links stay accurate and consistent as your application evolves.

Observing Changes in the Browser

Refresh your application in the browser by navigating to http://127.0.0.1:5000/. Now, the text “Welcome to FlaskBlog” should be displayed in brown, centred, and enclosed within a border.

Introducing Template Inheritance with a Base Template

Creating a base template allows you to reuse common HTML code across multiple pages. Start by creating ‘base.html’ within the ‘templates’ directory.

```
vi templates/base.html
```

Defining ‘base.html’ Structure

Inside ‘base.html,’ include the following code:

```
<!doctype html>
<html lang="en">
  <head>
    <!-- Required meta tags -->
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1, shrink-t
    <!-- Bootstrap CSS -->
        <link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstra
        <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>
            <!-- ... (navbar code) ... -->
        </nav>
        <div class="container">
            {% block content %} {% endblock %}
        </div>
        <!-- ... (Bootstrap JavaScript links) ... -->
      </body>
</html>
```

Implementing Template Inheritance in ‘index.html’

Update ‘index.html’ to inherit from ‘base.html’ and customize content.

```
vi templates/index.html
```

Include the following lines at the beginning of ‘index.html’:

```
#now we import the common code from base.html
{% extends 'base.html' %}
{% block content %}
    <h1>{% block title %} Welcome to FlaskBlog {% endblock %}</h1>
{% endblock %}
```

Browser Confirmation

Refresh your browser to observe the changes. Your application should now have a navigation bar and a styled title.

Congratulations! You've successfully utilized HTML templates, introduced static files, implemented Bootstrap for styling, and embraced template inheritance in your Flask application. In the next step, you'll delve into setting up a database to store your application data.

Step 4: Setting up the Database

Introducing a Database for Your Blog Posts

To make your application more dynamic, you'll set up a database to store your blog posts.

For simplicity, you'll use an SQLite database, which is part of the standard Python library. SQLite is convenient for small to medium-sized applications and is a great starting point.

Creating a Schema for Your Database

To start, you'll define the structure of your database using a schema. Open a new file named `schema.sql` in your `flask_blog` directory:

```
vi schema.sql
```

Inside this file, add the following SQL commands:

```
-- flask_blog/schema.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
);
```

Step 1: Introducing a Database for Your Blog Posts

To make your application more dynamic, you'll set up a database to store your blog posts. For simplicity, you'll use an SQLite database, which is part of the standard Python library. SQLite is convenient for small to medium-sized applications and is a great starting point.

Step 2: Creating a Schema for Your Database

To start, you'll define the structure of your database using a schema. Open a new file named `schema.sql` in your `flask_blog` directory:

```
nano schema.sql
```

Inside this file, add the following SQL commands:

```sql
-- flask_blog/schema.sql
CREATE TABLE posts (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    created TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
    title TEXT NOT NULL,
    content TEXT NOT NULL
);
```

Save and close the file.

Explanation:

- `CREATE TABLE posts` : Defines the 'posts' table with columns for post ID, creation timestamp, title, and content.

Initializing the Database

Now, you'll use a Python script (`init_db.py`) to execute the SQL commands and create the database. Open a new file named `init_db.py` in your `flask_blog` directory:

```
vi init_db.py
```

Add the following code to `init_db.py` :

```
# flask_blog/init_db.py
import sqlite3

# open a connection between python script and database.db to create it
connection = sqlite3.connect('database.db')

# open the schema.sql to read what inside it
with open('schema.sql') as f:
    connection.executescript(f.read())

# make the cursor to execute what inside the schema in database
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()
```

Save and close the file.

Explanation:

- `sqlite3.connect('database.db')` : Establishes a connection to a new SQLite database file named 'database.db.'
- `with open('schema.sql')` : Opens the schema file and executes its contents to create the 'posts' table.
- `cur.execute()` : Insert two example blog posts into the 'posts' table.

- `connection.commit()` : Commits the changes to the database.
- `connection.close()` : Closes the connection.

Executing the Initialization Script

Run the initialization script in the terminal:

```
python init_db.py
```

Upon completion, a new file `database.db` will be created in your `flask_blog` directory, signifying a successful database setup.

In the next step, you'll retrieve and display the inserted blog posts on your application's homepage.

Step 5: Displaying All Posts

Displaying Posts on the Homepage

Now that your database is set up, let's modify the `index()` view function to show all the blog posts on your homepage.

Modification in app.py:

- Open and Import sqlite3
- Open the `app.py` file:

```
vi app.py
```

Add the import statement for `sqlite3` at the top of the file:

```
import sqlite3
from flask import Flask, render_template
# ... (existing code) ...
```

Modification: Creating a Database Connection Function

Below the imports, add a function to create a database connection:

```
def get_db_connection():
    conn = sqlite3.connect('database.db')
    conn.row_factory = sqlite3.Row
    return conn
```

Explanation:

This function establishes a connection to the SQLite database file, sets up the row factory for name-based column access, and returns the connection object.

In a database, information is stored in tables, and each row in a table represents a record. Now, think of a row factory as a set of instructions for how the database should give you those records.

So, when we say `conn.row_factory = sqlite3.Row`, it's like telling the database, "Hey, when you give me a row (a record), make it like a dictionary. Let me access the columns by their names."

Here's why we need it:

By default, without `row_factory`, when you get a row from a SQLite database, you access the columns by their index numbers (like 0, 1, 2, ...). But that can be confusing and error-prone.

With `row_factory`, you can use names instead of numbers. For example, if you have a column named 'title' in your database, instead of doing `row[1]` to get its value, you can do `row['title']`, which is much clearer and less prone to mistakes.

Modification: Updating the index() Function

Modify the `index()` function to retrieve and display all blog posts:

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

Explanation:

The `index()` the function now uses the `get_db_connection()` function to open a database connection.

It then executes an SQL query to fetch all entries from the 'posts' table and passes them to the `render_template` function along with the 'index.html' template.

Save and close the `app.py` file.

Displaying Posts in the HTML Template

Now that the posts are passed to the template, update `index.html` to display them:

```
vi templates/index.html
```

Modify the file content as follows:

```
{% 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 %}
```

Explanation:

The template now uses a Jinja for loop (`{% for post in posts %}`) to iterate over each post in the list.

Inside the loop, it displays the post title in a `<h2>` heading within a `<a>` tag. Additionally, it shows the post-creation date using a Bootstrap badge.

Save and close the `index.html` file.

View the Result in the Browser

Navigate to your application's index page in your browser. You should now see all the blog posts displayed on your homepage.

Step 6: Displaying a Single Post

Displaying Individual Blog Posts

Now, let's create a new route and HTML template to display individual blog posts by their ID. This means that URLs like http://127.0.0.1:5000/1 will show the first post.

Open `app.py` for Editing:

```
vi app.py
```

Import `abort` Function:

At the top of the file, import the `abort` function, so if you didn't find the page it displays 404 error:

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

Create `get_post()` Function:

Below the `get_db_connection()` function, add a new function `get_post()`:

```
def get_post(post_id):
    # open the connection to db
    conn = get_db_connection()
    # select the post base on it's id
    post = conn.execute('SELECT * FROM posts WHERE id = ?', (post_id,)).fetchone
    # clos the connection
    conn.close()
    # checking if we already have the post or not
    if post is None:
        abort(404)
    return post
```

Explanation:

This function retrieves a blog post by its ID. If the post doesn't exist, it responds with a 404 Not Found error.

id = ? is acting like a placeholder for the variable

Create New Route and View Function:

Add a new route and view function to display individual blog posts:

```
@app.route('/<int:post_id>')
def post(post_id):
    # we got the post the user clicked on through the function we wrote before,
    we save the value of the post in post variable
    post = get_post(post_id)
    # we render the post page, pass the post variable as an argument,
    #why? to be able to use it in the html page
    return render_template('post.html', post=post)
```

Explanation:

This new route uses a variable rule `<int:post_id>` to capture a positive integer from the URL and pass it to the `post()` view function.

The view function then uses the `get_post()` function to retrieve the blog post and render the `post.html` template.

Save and Close `app.py`.

Create `post.html` Template

Now, create a new HTML template to display individual blog posts:

```
vi templates/post.html
```

Add the Following Code:

```
{% 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 %}
```

Explanation:

This template extends the base template and displays the title, creation date, and content of an individual blog post.

Save and Close `post.html`.

Step 3: Update Links on `index.html`

Now, let's make each post title on the index page link to its respective individual post page. Open `index.html` for editing:

```
vi templates/index.html
```

Update the link inside the for loop:

```
{% 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 %}
```

Explanation:

We use the `url_for()` function to generate the proper URL for each post based on its ID.

Save and Close `index.html`.

Now, your links on the index page should function as expected. You can visit URLs like http://127.0.0.1:5000/1 to see individual blog posts.

Step 7: Editing, Creating, and Deleting Posts

Step-by-Step Guide to Creating a New Blog Post

When creating a post, the fundamental concept involves providing a user with a form. This form allows the user to input both the title and content of the post they wish to create. Once the user completes the form, they click on the submit button. At this point, you gather the information provided by the user and add it to the database. This process enables the system to save the post, making it available for rendering and display.

Step 1: Import Necessary Modules and Set Secret Key

Open `app.py` for editing:

```
vi app.py
```

Add the following imports at the top of the file:

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

Explanation:

These imports are necessary for handling form submissions, generating URLs, flashing messages, and redirecting.

Set a secret key for session security:

```
app = Flask(__name__)
app.config['SECRET_KEY'] = 'your secret key'
```

Explanation:

The `SECRET_KEY` is crucial for securing sessions, allowing Flask to remember information between requests.

Step 2: Create a New Route for the Creation

Add a new route for creating a new post:

```
@app.route('/create', methods=('GET', 'POST'))
def create():
    return render_template('create.html')
```

Explanation:

This route handles both GET and POST requests. GET requests display the form, and POST requests handle form submissions (When the user clicks on submit).

Step 3: Create `create.html` Template

Create a new HTML template for the post-creation form:

```
vi templates/create.html
```

Add the following code:

```
{% 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 %}
```

Explanation:

This template displays a form with input fields for the post title and content.

The values are pre-filled with any submitted data to prevent loss during validation.

Step 4: Handle POST Requests in `create()` Function

Modify the `create()` view function in `app.py`:

```
@app.route('/create', methods=('GET', 'POST'))
def create():
    # if the user clicked on Submit, it sends post request
    if request.method == 'POST':
        # Get the title and save it in a variable
        title = request.form['title']
        # Get the content the user wrote and save it in a variable
        content = request.form['content']
      if not title:
                  flash('Title is required!')
              else:
                  # Open a connection to databse
                  conn = get_db_connection()
```

```
                # Insert the new values in the db
                conn.execute('INSERT INTO posts (title, content) VALUES (?, ?)
                             (title, content))
                conn.commit()
                conn.close()
                # Redirect the user to index page
                return redirect(url_for('index'))
        return render_template('create.html')
```

Explanation:

This modification checks if the request is a POST request. If so, it extracts the title and content from the form data.

If the title is missing, a flash message is shown. Otherwise, the new post is inserted into the database, and the user is redirected to the index page.

Step 5: Update the Navigation Bar in `base.html`

To create a post, you need to click on a link that redirects you to a page containing a form. On this page, you'll fill out the form with the necessary information. In this step, we'll add the link to facilitate the process.

Open `base.html` for editing:

```
vi templates/base.html
```

Add a new navigation item for creating a new post and display flashed messages:

```
...
<li class="nav-item">
    <a class="nav-link" href="{{url_for('create')}}">New Post</a>
</li>
...
{% for message in get_flashed_messages() %}
    <div class="alert alert-danger">{{ message }}</div>
{% endfor %}
...
```

Explanation:

The navigation bar now includes a link for creating a new post. Flashed messages are displayed below the navigation bar.

Save and Close Files.

Now, navigate to http://127.0.0.1:5000/create in your browser, and you can create and submit new posts with the provided form. The flashed messages will inform you of any issues, and successful submissions will redirect you to the index page.

To enable post-editing

Open `app.py` For editing, Open the `app.py` file using your preferred text editor.

```
vi app.py
```

Add the `edit()` View Function: Add the following `edit()` view function at the end of the file.

```
flask_blog/app.py
. . .

@app.route('/<int:id>/edit', methods=('GET', 'POST'))
def edit(id):
    # Get the post to be edited by it's id
    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()
            # Update the table
            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)
```

This function handles both GET and POST requests. For a GET request, it renders the `edit.html` template, and for a POST request, it updates the post in the database.

Create `edit.html` Template: Create a new file named `edit.html` inside the `templates` folder.

```
vi templates/edit.html
```

Add the following code inside the file:

```
flask_blog/templates/edit.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'] }}">
        </input>
    </div>

    <div class="form-group">
        <label for="content">Content</label>
        <textarea name="content" placeholder="Post content"
                  class="form-control">{{ request.form['content'] or post['conte
    </div>
    <div class="form-group">
        <button type="submit" class="btn btn-primary">Submit</button>
    </div>
</form>
<hr>
{% endblock %}
```

This template provides a form for editing the post title and content. The existing data is pre-filled in the form.

Navigate to the Edit Page: Visit the edit page for the first post using the following URL: http://127.0.0.1:5000/1/edit, You will see an “Edit ‘First Post’”

page.

Add Edit Links to Index Page: Open the `index.html` template file.

```
vi templates/index.html
```

Update the file to include edit links for each post:

```
# flask_blog/templates/index.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 %}
```

Now, each post on the index page will have an “Edit” link, allowing you to easily access the edit page for each post.

To add the functionality of deleting a post to your Flask application, follow these steps:

Open `app.py` for Editing: Open the `app.py` file using your preferred text editor.

```
vi app.py
```

Add the `delete()` View Function: Add the following `delete()` view function at the end of the file.

```
flask_blog/app.py
# ....

@app.route('/<int:id>/delete', methods=('POST',))
def delete(id):
    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'))
```

This function handles only POST requests. It receives the post ID from the URL, deletes the post from the database, flashes a success message, and redirects the user to the index page.

Update `edit.html` Template: Open the `edit.html` template file.

- We are enhancing the `edit.html` file by adding a compact link for deleting the post.

```
vi templates/edit.html
```

Add the following form tag after the `<hr>` tag and directly before the `{% endblock %}` line:

```
# flask_blog/templates/edit.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 %}
```

This form allows users to delete the current post. It triggers the `delete()` view function and displays a confirmation message using JavaScript `confirm()` before submitting the request.

Test Deletion: Navigate to the edit page of a blog post and try deleting it: http://127.0.0.1:5000/1/edit. Confirm the deletion when prompted.

Conclusion

In conclusion, this tutorial has provided a comprehensive guide for beginners on creating a blog using Flask, a Python web framework. Starting from the installation of Flask to implementing key features like displaying posts, creating a database, and incorporating HTML templates, the tutorial

systematically walks through each step. The structure of the tutorial is well-organized, making it suitable for beginners to follow along.

Your feedback is invaluable! ❤️

If you're eager to delve into the job market, check out the website to review requirements, qualifications, and salary information. Take a glimpse into your potential future.

Remote Rocketship

Feel free to connect 🌹me on:

Twitter

Linkedin

Github

Sincerely, **Noran**🌹

Flask | Python | Programming | Programming Languages | Database

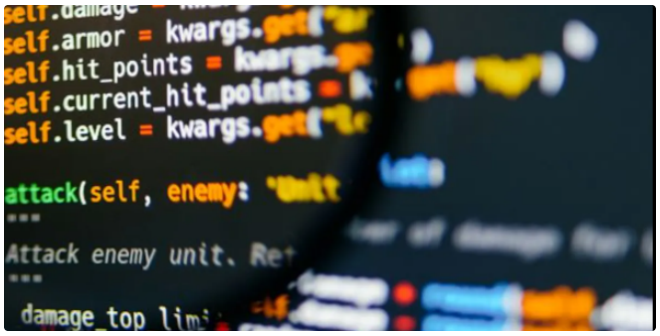
Written by Noran Saber Abdelfattah

Follow

342 Followers

I write about programming topics, i hope you love my words, Watch me on Youtube.
Youtube: https://www.youtube.com/channel/UCPNshzGPa2ZQUeCuJekEsqQ

More from Noran Saber Abdelfattah



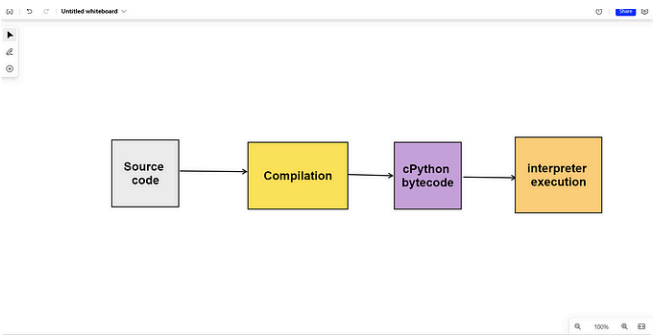
Noran Saber Abdelfattah

Simple Guide to Creating a Command-Line Interface (CLI) in...

Introduction

6 min read · Sep 16, 2023

134



Noran Saber Abdelfattah

Demystifying Python Bytecode: A Guide to Understanding and...

Introduction

11 min read · Jun 8, 2023

72

Noran Saber Abdelfattah

Understanding Classes in Python: From Templates to Objects

Introduction

20 min read · Jun 28, 2023

417 7

Noran Saber Abdelfattah

Understanding the Stack Data Structure in C: Introduction,...

introduction

16 min read · Jun 17, 2023

7 1

See all from Noran Saber Abdelfattah

Recommended from Medium

Dylan Cooper in Stackademic

Mojo, 90,000 Times Faster Than Python, Finally Open Sourced!

On March 29, 2024, Modular Inc. announced the open sourcing of the core components o...

10 min read · Apr 8, 2024

2.7K 21

Andrii Hrimov

Flask project structure template

Flask is a popular micro web framework that is used very often, even with the advent of...

10 min read · Nov 26, 2023

87 1

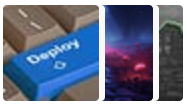
Lists

Coding & Development

11 stories · 566 saves

General Coding Knowledge

20 stories · 1119 saves



Predictive Modeling w/ Python

20 stories · 1106 saves

Practical Guides to Machine Learning

10 stories · 1320 saves

Max Mason

Building a Web Application with Flask, SQLAlchemy, and React: A...

In this tutorial, we'll walk through the step-by-step process of configuring a web applicatio...

5 min read · Nov 9, 2023

3

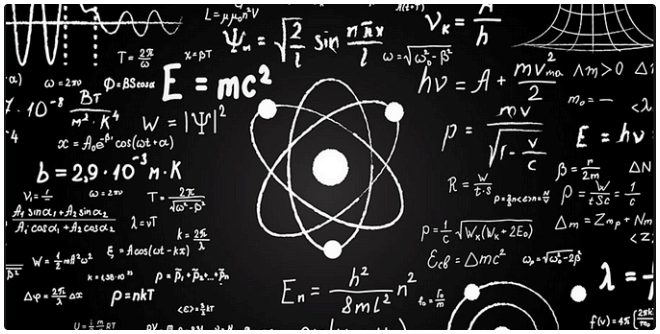
Benedict Neo in bitgrit Data Science Publication

Forget `pip install`, Use This Instead

Install Python packages up to 100x ⚡ faster than before.

4 min read · Mar 27, 2024

1.1K 12



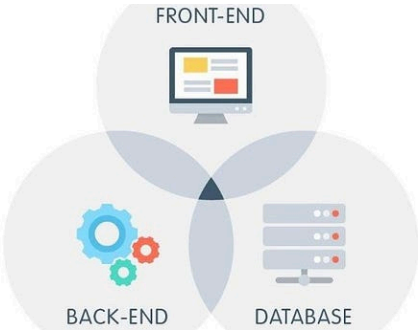
Zeid Zandi

How to Manage Constants in Python: Best Practices and...

Introduction:

6 min read · Oct 30, 2023

501 1



Aspen Wilson

Unveiling the Power of Full Stack Development with React and Flask

In the ever-evolving landscape of web development, Full Stack Development has...

3 min read · Dec 21, 2023

See more recommendations