22 Flask Web app

Flask is a powerful and flexible micro web framework for Python, ideal for both small and large web projects.

Get Started

Create a Virtual Environment

Before you continue working on your Flask project, it's a good idea to create and activate a virtual environment. That way, you're installing any project dependencies not system-wide but only in your project's virtual environment.

```
mkdir flask_board
cd flask_board

python -m venv venv

# On Windows
.\venv\Scripts\activate

# On Linux or Mac
source venv/bin/activate
```

Add Dependencies

```
(venv) $ pip install Flask
```

Initiate Your Flask Project

Initiate your project by start writing in `app.py`

Run the Flask Development Server

A Flask project can be as basic as one Python file, which is commonly named `app.py`

```
# app.py
# flask_board/app.py
from flask import Flask
```

```
app = Flask(__name__)

@app.route("/")
def home():
    return "Hello, World!"

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=8000, debug=True)
```

Functions like these are commonly referred to as **views** in Flask.
In this case, the `home` view returns a `"Hello, World!"` string that the browser can display.

Transform Your Project Into a Package

Since you're just at the start of the project, your to-do list to create the package has only three tasks:

1. Creating a package folder named `board/`
2. Moving `app.py` into `board/`
3. Renaming `app.py` to `__init__.py`

```
(venv) $ mkdir board
(venv) $ mv app.py board/__init__.py
```

With the commands above, you create the `board/` sub-folder. Then, you move `app.py` into `board/` and rename it to `__init__.py`.

By now, your Flask project structure should look like this:

```
flask_board/
│
└── board/
    └── __init__.py
```

With the new package structure in place, make sure that you stay in the `flask_board/` folder and run your Flask project with this command:

```
(venv) $ python -m flask --app board run --port 8000 --debug
```

With the command above, you're telling Flask to look for an app named `board` and serve it in debug mode on port `8000`.

When you visit `http://127.0.0.1:8000`, you see the same *Hello, World!* displayed as before.

Work With an Application Factory

An **application factory** in Flask is a design pattern that's highly beneficial for scaling Flask projects. With an application factory, you can create and configure your Flask project flexibly and efficiently, making it easier to develop, test, and maintain as it grows and evolves.

Instead of having your application's code at the root level of your `__init__.py` file, you'll work with a function that returns your application. To do so, replace the content of `__init__.py` with the content below:

```
# __init__.py
# flask_board/board/__init__.py
from flask import Flask

def create_app():
    app = Flask(__name__)

    return app
```

Leverage Blueprints

Flask Blueprints encapsulate **functionality**, such as views, templates, and other resources. Each Flask Blueprint is an **object** that works very similarly to a Flask application. They both can have resources, such as static files, templates, and views that are associated with routes.

However, a Flask Blueprint is not actually an application. It needs to be registered in an application before you can run it.

Blueprints contain related views that you can conveniently import in `__init__.py`

Create a file named `pages.py` in the `board/` folder and add the content below:

```
# pages.py
# flask_board/board/pages.py
from flask import Blueprint
bp = Blueprint("pages", __name__)

@bp.route("/")
def home():
    return "Hello, Home!"

@bp.route("/about")
```

```
def about():
    return "Hello, About!"
```

You define two routes, one as the `home` view and the other as the `about` view. Each of them returns a string to indicate on which page you are on.

Before you can visit the pages, you need to connect the `pages` blueprint with your Flask project:

```
# __init__.py
# flask_board/board/__init__.py
from flask import Flask

from board import pages

def create_app():
    app = Flask(__name__)

    app.register_blueprint(pages.bp)
    return app
```

When you visit `http://127.0.0.1:8000/about`, then you can see the *Hello, About!* string:

Introduce Templates

Templates are HTML files with the capability of rendering dynamic content sent over from your Flask views. For this, Flask uses the popular Jinja **template engine**.

Jinja is a template engine commonly used for web templates that receive dynamic content from the back end and render it as a static page in the front end. But you can use Jinja without a web framework running in the background.

With a template engine like Jinja, you can embed Python-like expressions into your templates to create dynamic web content. You can use loops, conditionals, and variables directly in your templates.

Jinja also supports **template inheritance**, allowing the creation of a **base template** that you can extend in **child templates**, promoting code reuse and a consistent layout across different pages of your project.

Build a Base Template

Create a new template named `base.html` in a `templates/` directory inside `board/`:

```
<!-- base.html -->
<!-- flask_board/board/templates/base.html -->
<!DOCTYPE html>
<html lang="en">
    <head>
        <title>Message Board - {% block title %}{% endblock title %}</title>
    </head>
    <body>
        <h1>Message Board</h1>
        <section>
            <header>
                {% block header %}{% endblock header %}
            </header>
            <main>
                {% block content %}<p>No message.</p>{% endblock content %}
            </main>
        </section>
    </body>
</html>
```

Commonly, the Jinja base template is named `base.html`, and it contains the main HTML structure of your web pages.

Add Your Child Templates

Start with the template for your `home` view by creating a `home.html` file in a `pages/` subfolder of your `templates/` folder:

```
<!-- home.html -->
<!-- flask_board/board/templates/pages/home.html -->
{% extends 'base.html' %}

{% block header %}
    <h2>{% block title %}Home{% endblock title %}</h2>
{% endblock header %}

{% block content %}
    <p>
        Learn more about this project by visiting the <a href="{{
url_for('pages.about')}}"></a>
    </p>
{% endblock content %}
```

Child templates also contain `{% block %}` tags. By providing the block's name as an argument, you're connecting the blocks from the child template with the blocks from the parent template.

When you use `url_for()`, Flask creates the full URL to the given view for you. This means that if you change the route in your Python code, then the URL in your templates updates automatically.

Continue by creating a file named `about.html` in the same `pages/` directory:

```
<!-- about.html -->
<!-- flask_board/board/templates/pages/about.html -->
{% extends 'base.html' %}

{% block header %}
    <h2>{% block title %}About{% endblock title %}</h2>
{% endblock header %}

{% block content %}
    <p>This is a message board for friendly messages.</p>
{% endblock content %}
```

With both child templates in place, you need to adjust your views to return the templates instead of the plain strings.

```
# pages.py
# flask_board/board/pages.py
from flask import Blueprint, render_template

bp = Blueprint("pages", __name__)

@bp.route("/")
def home():
    return render_template("pages/home.html")

@bp.route("/about")
def about():
    return render_template("pages/about.html")
```

By default, Flask expects your templates to be in a `templates/` directory. Therefore, you don't need to include `templates/` in the path of the templates.

Improve the User Experience

A good user experience is crucial in any web application. It ensures that users find the application not only convenient to use, but also enjoyable.

Include a Navigation Menu

The navigation of a website is usually displayed on every page. With your base and child template structure, it's best to add the code for the navigation menu into `base.html`.

Instead of adding the navigation menu code directly into `base.html`, you can leverage the `{% include %}` tag. By referencing another template with `{% include %}`, you're loading the whole template into that position.

Included templates are partials that contain a fraction of the full HTML code. To indicate that a template is meant to be included, you can prefix its name with an underscore (`_`).

Follow the prefix-based naming scheme and create a new template named `_navigation.html` in your `templates/` folder:

```
<!-- _navigation.html -->
<!-- flask_board/board/templates/_navigation.html -->
<nav>
    <ul>
        <li><a href="{{ url_for('pages.home') }}">Home</a></li>
        <li><a href="{{ url_for('pages.about') }}">About</a></li>
    </ul>
</nav>
```

Include `_navigation.html` in `base.html` to display your navigation menu on all of your pages:

```
<!-- base.html -->
<!-- flask_board/board/templates/base.html -->
<!DOCTYPE html>
<html lang="en">
    <head>
        <title>Message Board - {% block title %}{% endblock title %}</title>
    </head>
    <body>
        <h1>Message Board</h1>
        {% include("_navigation.html") %}
        <section>
          <header>
            {% block header %}{% endblock header %}
          </header>
```

```
          <main>
            {% block content %}<p>No messages.</p>{% endblock content %}
          </main>
        </section>
    </body>
</html>
```

Since both `home.html` and `about.html` extend `base.html`, you didn't need to make any changes in these templates to make the navigation menu appear.

Make Your Project Look Nice

The templates in your project provide the logic and structure for the **front end** of your project. With the help of **Cascading Style Sheets (CSS)**, you can style the content.

In Flask projects, you commonly save CSS files in a `static/` directory.

Create a new directory named `static/` inside of `board/` and place a file named `styles.css` in it with the **CSS declarations** below:

```
/* styles.css */
/* flask_board/board/static/styles.css */
* {
  box-sizing: border-box;
}

body {
  font-family: sans-serif;
  font-size: 20px;
  margin: 0 auto;
  text-align: center;
}

a,
a:visited {
  color: #007bff;
}

a:hover {
  color: #0056b3;
}

nav ul {
  list-style-type: none;
  padding: 0;
```

```
}

nav ul li {
  display: inline;
  margin: 0 5px;
}

main {
  width: 80%;
  margin: 0 auto;
}
```

With the style sheet in place, you need to add a link to `styles.css` inside `<head>` of `base.html`:

```
<!-- base.html -->
<!-- flask_board/board/templates/base.html -->
<!DOCTYPE html>
<html lang="en">
<head>
  <title>Message Board - {% block title %}{% endblock title %}</title>
  <link rel="stylesheet" href="{{ url_for('static', filename='styles.css')
}}">
</head>
<body>
<!-- ... -->
</body>
</html>
```

By adding the CSS file reference to `base.html`, you're again taking advantage of the inheritance mechanism that Jinja templates provide. You only need to add `styles.css` in your base template to make it present in your child templates.

When you reach here your project structure would look like this:

```
flask_board/
│
└── board/
    │
    ├── static/
    │   └── styles.css
    │
    ├── templates/
```

```
    │   │
    │   ├── pages/
    │   │   ├── about.html
    │   │   └── home.html
    │   │
    │   ├── _navigation.html
    │   └── base.html
    │
    ├── __init__.py
    └── pages.py
```

Prepare Your Development Environment

You also need `click`, which you'll use for a script to initialize the database. Generally, `click` comes with your Flask installation. But to be sure, you can run the `pip install` command explicitly:

```
(venv) $ pip install click
```

Other than Flask and `click`, you'll work with `python-dotenv`. With the help of `python-dotenv`, you'll be able to read key-value pairs from an external file and keep sensitive information out of your codebase.

```
(venv) $ pip install python-dotenv
```

Prepare Your App for Interaction

So far, your Flask project is quite static. Now you'll add two new pages to the web app so users can interact with it. One page will be for posting a message and another one for displaying any posts in the future.

Start With the Templates

Create a subfolder named `posts/` inside `templates/`. Then, create a template for your form in the `posts/` subfolder of your `templates/` folder:

```
<!-- create.html -->
<!-- flask_board/board/templates/posts/create.html -->
{% extends 'base.html' %}

{% block header %}
```

```
    <h2>{% block title %}Add Post{% endblock title %}</h2>
{% endblock header %}

{% block content %}
  <form method="post">
    <div class="form-group">
        <label for="author">Your Name</label>
        <input type="text" name="author" id="author" value="{{
request.form['author'] }}" class="form-control">
    </div>
    <div class="form-group">
        <label for="message">Your Message</label>
        <textarea name="message" id="message" class="form-control">{{
request.form['message'] }}</textarea>
    </div>
    <div class="form-group">
        <input type="submit" value="Post Message" class="submit-btn">
    </div>
</form>
{% endblock content %}
```

To connect the child template with its parent template, you must add an `{% extends %}` tag at the top of the file.

Child templates also contain `{% block %}` tags. By providing the block's name as an argument, you're connecting the blocks from the child template with the blocks from the parent template.

For *Posts* page, create a template named `posts.html` in the `posts/` directory:

```
<!-- posts.html -->
<!-- flask_board/board/templates/posts/posts.html -->
{% extends 'base.html' %}

{% block header %}
    <h2>{% block title %}Posts{% endblock title %}</h2>
{% endblock header %}

{% block content %}
    {% for post in posts %}
    <article>
      <p class="message">{{ post["message"] }}<p>
      <aside>Posted by {{ post["author"] }} on {{
post["created"].strftime("%b %d, %Y at %I:%M%p") }}</aside>
    </article>
```

```
    {% endfor %}
{% endblock content %}
```

Just like in `create.html`, you extend the base template. Then, you add the headline of the page inside the `header` block. By nesting the headline in the `title` block, you also add the headline as the title of the page.

Inside the `content` block, you loop through all posts and show the message, the author, and a formatted timestamp.

Register a New Blueprint

In Flask, **blueprints** are modules that contain related views that you can conveniently import in your `__init__.py` file. You do this already for the *Pages* module. Now, you'll create a blueprint for your *Posts* functionality.

Create a file named `posts.py` in the `board/` folder and add the content below:

```
# posts.py
# flask_board/board/posts.py
from flask import Blueprint, render_template

bp = Blueprint("posts", __name__)

@bp.route("/create", methods=("GET", "POST"))
def create():
    return render_template("posts/create.html")

@bp.route("/posts")
def posts():
    posts = []
    return render_template("posts/posts.html", posts=posts)
```

Next, you need to connect the `posts` blueprint with your Flask project:

```
# __init__.py
# flask_board/board/__init__.py
from flask import Flask

from board import pages, posts

def create_app():
    app = Flask(__name__)
```

```
    app.register_blueprint(pages.bp)
    app.register_blueprint(posts.bp)
    return app
```

When you visit `http://127.0.0.1:8000/create`, then you can see the form to add posts.

When you fill out the form and press the *Post Message* button, the page reloads. However, you don't save any posts in the database yet.

Hide Your Secrets

You're currently developing your Flask project on your local computer. It makes sense to use different settings for different environments. Your local development environment may use a differently named database than an online production environment.

On top of that, you'll work with sensitive information like passwords or database names that nobody should be able to see. Still, your Flask project needs to be able to obtain this information. That's when **environment variables** come into play.

Define Environment Variables

To be flexible, you store this information in special variables that you can adjust for each environment.

Start by creating a `.env` file in the root directory of your project. For now, only add an `ENVIRONMENT` variable to it:

```
# .env
# flask_board/.env
ENVIRONMENT="Development"
```

You can think of environment variables as constants that store important information that you'll read but not overwrite in your code.

By saving your environment variables externally, you're following the twelve-factor app methology. The **twelve-factor app methodology** defines twelve principles to enable developers to build portable and scalable web applications.

One recommendation is to have different `.env` files for different environments. Also, you should never add the `.env` file to your version control system, as your environment variables may store sensitive information.

Note: If you're sharing your code with other developers, then you may want to show in your repository what their `.env` files should look like. In that case, you can add `.env_sample` to your version control system. In `.env_sample`, you can store the keys with placeholder values. To help yourself and your fellow developers, don't forget to write instructions in your `README.md` file on how to copy `.env_sample` and store the correct values in the file.

To check if Python can load your environment variables, start a Python interactive shell and perform the commands below:

```
>>> import os
>>> from dotenv import load_dotenv
>>> load_dotenv()
True

>>> print(os.getenv("ENVIRONMENT"))
Development
```

Create Environment Variables for Flask

While the purpose of the `ENVIRONMENT` variable in your `.env` file is mainly for testing, it's time to add two environment variables that you'll need in your Flask project:

1. `FLASK_SECRET_KEY`: A secure identifier allowing the server to validate the integrity of your data. You'll create the secret key in a moment.
2. `FLASK_DATABASE`: The name of the database that you'll use in your development environment. In this tutorial, the database name is `board.sqlite`.
   `FLASK_SECRET_KEY` will be a random string of characters that Flask uses to encrypt and decrypt data, such as tokens or session data. If you need to create cryptographically secure data, like a Flask secret key, then you can use Python's `secrets` module:

```
>>> import secrets
>>> secrets.token_hex()
'c874b801a7caf1be80c5ff8900a3dad08c7de89c'
```

The `.token_hex()` method returns a hexadecimal string containing random numbers and letters from `0` to `9` and `a` to `f`. Copy the value that `secrets.token_hex()` outputs and add it, as well as the name of your database, to your `.env` file:

```
# .env
# flask_board/.env
ENVIRONMENT="Development"
```

```
FLASK_SECRET_KEY=c874b801a7caf1be80c5ff8900a3dad08c7de89c
FLASK_DATABASE=board.sqlite
```

By choosing exactly the names `FLASK_SECRET_KEY` and `FLASK_DATABASE`, you work with builtin configuration variables that Flask provides.

Load Environment Variables Into Flask

You can load environment variables with `python-dotenv`. Now you can confidently implement a environment variables functionality into your Flask project by adding the highlighted code below to your `__init__.py` file:

```
# __init__.py
# flask_board/board/__init__.py
import os
from dotenv import load_dotenv
from flask import Flask

from board import pages, posts

load_dotenv()

def create_app():
    app = Flask(__name__)
    app.config.from_prefixed_env()

    app.register_blueprint(pages.bp)
    app.register_blueprint(posts.bp)
    print(f"Current Environment: {os.getenv('ENVIRONMENT')}")
    print(f"Using Database: {app.config.get('DATABASE')}")
    return app
```

Like before, you first import `os` and `load_dotenv`. Then, you use `load_dotenv()` to load the environment variables from a `.env` file into the application's environment. These environment variables are then accessible throughout the application once you run `app.config.from_prefixed_env()`.

You can see the contents of `ENVIRONMENT` and `FLASK_DATABASE` when you restart your Flask development server:

```
$ python -m flask --app board run --port 8000 --debug
```

Work With a Database in Flask

A database provides a structured and efficient way to store, retrieve, and manage data.

The database engine that you'll use in this tutorial is SQLite. SQLite stores the entire database as a single file on the disk, simplifying database management and reducing the complexity of deployment. Moreover, Python comes with a built-in SQLite module that's ready for you to use out of the box.

Define the Database Schema

A database schema is basically a recipe for how you want the database to be structured. It contains a **model** for all the tables and columns of your database.

Note: The model that you'll define below matches the message board project.

Create a new file named `schema.sql` in your `board/` directory and add the SQL code below:

```
/* schema.sql */
/* flask_board/board/schema.sql */
DROP TABLE IF EXISTS post;

CREATE TABLE post (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    created TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
    author TEXT NOT NULL,
    message TEXT NOT NULL
);
```

Initialize Your Database

When you run your Flask project, you want to either initialize the database if it doesn't exist yet or connect to it if the database exists.

Create a new file named `database.py` in your `board/` directory. This module will contain the functionality for setting up and connecting to your database:

```
# database.py
# flask_board/board/database.py
import sqlite3
import click
from flask import current_app, g

def init_app(app):
    app.cli.add_command(init_db_command)
```

```
@click.command("init-db")
def init_db_command():
    db = get_db()

    with current_app.open_resource("schema.sql") as f:
        db.executescript(f.read().decode("utf-8"))

    click.echo("You successfully initialized the database!")

def get_db():
    if "db" not in g:
        g.db = sqlite3.connect(
            current_app.config["DATABASE"],
            detect_types=sqlite3.PARSE_DECLTYPES
        )
        g.db.row_factory = sqlite3.Row

    return g.db
```

In the first three lines, you import:

- The `sqlite3` module, a built-in Python module for interacting with SQLite databases
- `click`, a Python package for creating **command-line interfaces (CLI)**
- `current_app` and `g`, two specific Flask objects that help you interact with your application and its storage

The `current_app` object points to the Flask application handling the current activity. You can use it to access application-specific data like your environment variables. The use of `current_app` makes your code more flexible, especially when you work with an application factory, as you do in your `__init__.py` file.

With `g`, Flask provides you with a global namespace object that you can use as temporary storage when a user makes a **request**, such as sending a form. Each request has its own `g` object, which resets at the end of the request. This is useful for storing data that might be accessed multiple times during the request, such as a database connection, avoiding the need to create new connections for every single database operation within the same request.

- The `init_app()` function, carries the core logic for database initialization, and integrates a new command into the Flask application's CLI by adding `init_db_command()` as a CLI command by decorating the function with `click.command()`.
- init_db_command() reads and executes the SQL commands from `schema.sql`. As this file contains SQL statements for creating tables and setting up the database schema, it'll prepare the database for the application's data.

- `get_db()` returns a database connection with the conditional statement either return an existing connection or establish a new one first.
- `sqlite3.connect` creates a database connection, pointing to the database name that you defined in your environment variables.
- `g.db.row_factory` to `sqlite3.Row` allows you to access the column by name. This means you can interact with your database similarly to interacting with a dictionary.

With this code, you set up a command-line interface for your Flask project. When you execute `init-db` in the terminal, you either initialize the database or reset an existing database.

Before you can perform the `init-db` command, you need to register the database in your Flask application factory:

```
# __init__.py
# flask_board/board/__init__.py
import os
from dotenv import load_dotenv
from flask import Flask

from board import pages, posts, database

load_dotenv()

def create_app():
    app = Flask(__name__)
    app.config.from_prefixed_env()

    database.init_app(app)

    app.register_blueprint(pages.bp)
    app.register_blueprint(posts.bp)
    print(f"Current Environment: {os.getenv('ENVIRONMENT')}")
    print(f"Using Database: {app.config.get('DATABASE')}")
    return app
```

When you run your `create_app()` application factory, you execute `database.init_app()` with your current app as an argument. Since you only registered `init_db_command` as a command in `init_app()`, Flask doesn't automatically perform the database initialization when you start the development server. Instead, you can now execute the command manually in the terminal:

```
(venv) $ python -m flask --app board init-db
```

When you run `init-db`, you create a brand-new database in the root directory of your Flask project. You should find a file named `board.sql` next to your `board/` folder.

Note: You must use this command with caution, as you delete the content of any existing database when you execute `init-db`.

Close Database Connections

After any database interaction, you need to make sure to close the database connection again. You do this by registering a teardown function to your app context:

```
# database.py
# flask_board/board/database.py
# ...

def init_app(app):
    app.teardown_appcontext(close_db)
    app.cli.add_command(init_db_command)

# ...

def close_db(e=None):
    db = g.pop("db", None)

    if db is not None:
        db.close()
```

By passing `close_db` into `.teardown_appcontext()`, you make sure to call `close_db()` when the application context closes. The application context in Flask is created for each request, and it's where the Flask app handles all the request processing. When the context is torn down, it means the application is done processing the request, and you can close the database connection.

Post and Display Messages

To work with `POST` methods, add the code below:

```
# posts.py
# flask_board/board/posts.py
from flask import (
    Blueprint,
    redirect,
    render_template,
    request,
```

```
    url_for,
)

from board.database import get_db

bp = Blueprint("posts", __name__)

@bp.route("/create", methods=("GET", "POST"))
def create():
    if request.method == "POST":
        author = request.form["author"] or "Anonymous"
        message = request.form["message"]

        if message:
            db = get_db()
            db.execute(
                "INSERT INTO post (author, message) VALUES (?, ?)",
                (author, message),
            )
            db.commit()
            return redirect(url_for("posts.posts"))

    return render_template("posts/create.html")

@bp.route("/posts")
def posts():
    posts = []
    return render_template("posts/posts.html", posts=posts)
```

Finally, you redirect the request to the URL of your posts. That way, the author can see their post go public once you've made a small change to the `posts` view:

```
# posts.py
# flask_board/board/posts.py
from flask import (
    Blueprint,
    redirect,
    render_template,
    request,
    url_for,
)

from board.database import get_db

bp = Blueprint("posts", __name__)
```

```
# ...

@bp.route("/posts")
def posts():
    db = get_db()
    posts = db.execute(
        "SELECT author, message, created FROM post ORDER BY created DESC"
    ).fetchall()
    return render_template("posts/posts.html", posts=posts)
```

To create posts, open your browser and visit `http://127.0.0.1:8000/create`. After creating a few posts, you'll see all your messages listed on `http://127.0.0.1:8000/posts`

Add Some Final Touches

Add some CSS declarations to your style sheet to make your web app look cleaner.

Improve the Navigation

With the help of the existing template inheritance, adding the `create` view and the `posts` view to the navigation only requires two additional lines in your `_navigation.html` file:

```
<!-- _navigation.html -->
<!-- flask_board/board/templates/_navigation.html -->
<nav>
    <ul>
        <li><a href="{{ url_for('pages.home') }}">Home</a></li>
        <li><a href="{{ url_for('pages.about') }}">About</a></li>
        <li><a href="{{ url_for('posts.posts') }}">All Posts</a></li>
        <li><a href="{{ url_for('posts.create') }}">Add Post</a></li>
    </ul>
</nav>
```

Since you include the navigation in your base template, the navigation updates on all your pages.

Enhance the Design

In your base template, you reference the navigation template. That's why it's rendered on all pages. You similarly distribute the styling over your web app. In `base.html`, you're linking to a CSS stylesheet in your `static/` folder.

Open `styles.css` and add some **CSS declarations** for form elements and your messages:

```css
/* styles.css */
/* board/static/styles.css */
/* ... */

article,
form {
  text-align: left;
  min-width: 200px;
  padding: 20px;
  margin-bottom: 20px;
  box-shadow: 0px 0px 10px #ccc;
  vertical-align: top;
}

aside {
  color: #ccc;
  text-align: right;
}

form {
  display: flex;
  flex-direction: column;
  margin-top: 20px;
}

.form-group {
  margin-bottom: 20px;
}

.form-control {
  width: 100%;
  padding: 10px;
  border: 1px solid #ccc;
  font-size: 1em;
}

.submit-btn {
  background-color: #007bff;
  color: #fff;
  border: none;
  border-radius: 4px;
  padding: 10px 20px;
  cursor: pointer;
  font-size: 1em;
}
```

```
.submit-btn:hover {
  background-color: #0056b3;
}

.message {
  font-size: 2.5em;
  font-family: serif;
  margin: 0;
  padding: 0;
}
```

Open your browser and visit `http://127.0.0.1:8000/create`, and adore your new design.

When you reach here you should end up with a folder structure that looks like this:

```
flask_board/
├── board/
│   │
│   ├── static/
│   │   └── styles.css
│   │
│   ├── templates/
│   │   │
│   │   ├── pages/
│   │   │   ├── about.html
│   │   │   └── home.html
│   │   │
│   │   ├── posts/
│   │   │   ├── create.html
│   │   │   └── posts.html
│   │   │
│   │   ├── _navigation.html
│   │   └── base.html
│   │
│   ├── __init__.py
│   ├── database.py
│   ├── pages.py
│   ├── posts.py
│   └── schema.sql
│
└── board.sqlite
```

Add a Friendly Error Page

Add another page to your Flask web project that you can display when a URL doesn't exist.

Explore the Current Error Page

Flask comes with an error page that the development server displays when you enter a wrong URL. When a user enters a nonexistent page URL for your web app, then they'll see a generic error page.

Create an Error Handler

To work with your custom error page, you can add an **error handler** to your project. An error handler is basically a view for a specific error type. That means you can implement an error handler for a `404` error similarly to regular views.

Create a new file named `errors.py` in your `board/` folder and add the code below:

```
# errors.py
# flask_board/board/errors.py
from flask import render_template

def page_not_found(e):
    return render_template("errors/404.html"), 404
```

Note: When Flask calls the error handler, you also receive the error object, which is commonly abbreviated to the letter `e`.

Design the 404 Error Template

our `page_not_found()` error handler wants to render a template named `404.html`. Go ahead and create the `404.html` template inside an `errors/` subfolder in `templates/` and add the code below:

```
<!-- 404.html -->
<!-- flask_board/board/templates/errors/404.html -->
<!DOCTYPE html>
<html lang="en">
    <head>
        <title>{% block title %}{% endblock title %}</title>
        <link rel="stylesheet" href="{{ url_for('static',
filename='styles.css') }}">
    </head>
    <body>
```

```
        <h1>Page not found</h1>
        <p>Please go to one of the pages below:</p>
          {% include("_navigation.html") %}
    </body>
</html>
```

Register the Error Handler

```
# __init__.py
# flask_board/board/__init__.py
import os
from dotenv import load_dotenv
from flask import Flask

from board import (
    database,
    errors,
    pages,
    posts,
)

load_dotenv()

def create_app():
    app = Flask(__name__)
    app.config.from_prefixed_env()

    database.init_app(app)

    app.register_blueprint(pages.bp)
    app.register_blueprint(posts.bp)
    app.register_error_handler(404, errors.page_not_found)
    print(f"Current Environment: {os.getenv('ENVIRONMENT')}")
    print(f"Using Database: {app.config.get('DATABASE')}")
    return app
```

The first argument of register_error_handler() is the error status code you want to register, which is `404`. This status code must match the status code returned by the second argument, which is your `page_not_found` view.

Now that you've registered the `404` error, you can check out the error for any nonexistent URL of your project.

Give Feedback to the User

When you send a form to the message board, you can see that everything worked fine by finding your message posted to the board. You can use notification messages that pop up when a user submits a form.

Find a Spot for Notifications

First, you need to find a spot on your pages where you want to display notification messages when they appear. When they appear, it's important that the users notice them. That's why the top of the page is usually a good location to place your notification messages.

In your Flask project, the perfect spot is the `header` that you can find in the base template. Navigate to your `templates/` folder, open `base.html`, and add three message containers inside the `<header>` element:

```
<!-- base.html -->
<!-- flask_board/board/templates/base.html  -->
<!DOCTYPE html>
<html lang="en">
    <head>
        <title>Message Board - {% block title %}{% endblock title %}</title>
        <link rel="stylesheet" href="{{ url_for('static',
filename='styles.css') }}">
    </head>
    <body>
        <h1>Message Board</h1>
        {% include("_navigation.html") %}
        <section>
            <header>
                {% block header %}{% endblock header %}
                <div class="flash success">Everything worked!</div>
                <div class="flash error">Oh no, something went wrong.</div>
            </header>
            <main>
                {% block content %}<p>No messages.</p>{% endblock content %}
            <main>
        </section>
    </body>
</html>
```

Next you'll work with two message types:

- **Success**: A friendly notification that reassures the user that everything worked fine.
- **Error**: An informative notification to indicate that something went wrong.
  Besides the success and error categories, you often use warning and info messages.

Style Your Notifications

Following the colors of most traffic lights, the success notification should be green, and the error notification should be red. Hop over to your CSS file in your `static/` folder and add the **CSS declarations** below:

```
/* styles.css */
/* flask_board/board/static/styles.css */
/* ... */

.flash {
  padding: 20px;
  margin: 20px;
}

.flash.error {
  background-color: #ffa7ae;
  color: #a02a36;
}

.flash.success {
  background-color: #96faad;
  color: #1c7530;
}
```

All of your notifications share the `.flash` class. Open your browser, visit `http://127.0.0.1:8000/`, and check out how your notification messages look.

Leverage Flask's Notification System

Displaying notification messages to users is vital for the look and feel of web applications. That's why Flask comes with a message flashing system that you can leverage as a developer.

To display a notification message with Flask, you need to call the `flash()` function in suitable views. In your project, it's a good idea to display notification messages after the user submits a form.

You can write the logic to handle forms in `posts.py`

```
# posts.py
# flask_board/board/posts.py
from flask import (
    Blueprint,
    flash,
```

```
    redirect,
    render_template,
    request,
    url_for,
)

from board.database import get_db

bp = Blueprint("posts", __name__)

@bp.route("/create", methods=("GET", "POST"))
def create():
    if request.method == "POST":
        author = request.form["author"] or "Anonymous"
        message = request.form["message"]

        if message:
            db = get_db()
            db.execute(
                "INSERT INTO post (author, message) VALUES (?, ?)",
                (author, message),
            )
            db.commit()
            flash(f"Thanks for posting, {author}!", category="success")
            return redirect(url_for("posts.posts"))
        else:
            flash("You need to post a message.", category="error")

    return render_template("posts/create.html")

# ...
```

Before you can see the messages in action, you need to make an adjustment in `base.html` to work with Flask's flash system:

```
<!-- base.html -->
<!-- flask_board/board/templates/base.html  -->
<!DOCTYPE html>
<html lang="en">
    <head>
        <title>Message Board - {% block title %}{% endblock title %}</title>
        <link rel="stylesheet" href="{{ url_for('static',
filename='styles.css') }}">
    </head>
    <body>
```

```
        <h1>Message Board</h1>
        {% include("_navigation.html") %}
        <section>
            <header>
                {% block header %}{% endblock header %}
                {% for category, message in
get_flashed_messages(with_categories=true) %}
                <div class="flash {{ category }}">{{ message }}</div>
                {% endfor %}
            </header>
            <main>
                {% block content %}<p>No messages.</p>{% endblock content %}
            <main>
        </section>
    </body>
</html>
```

To receive messages from your Flask view, you use `get_flashed_messages()`. By default, you'd only receive the message text from Flask. If you also pass in the `with_categories` argument and set it to `true`, then Flask also sends the message category along. Having `category` and `message` present enables you to set the notification message text and CSS class dynamically.

To see the notification messages in action, hop over to `http://127.0.0.1:8000/create` and post a message.

As long as a posted message contains text, you receive the success notification. Next, try posting a message without text. You receive a red error notification message when you leave the message input empty when sending the form.

Use Flask's Logging Message to Receive Terminal Info

With **logging**, you can get a better understanding of your Python code. Flask uses Python's standard `logging` module for the output that you already see in the terminal when you interact with your Flask project in the browser.

If you heavily rely on logging in a Flask project, then Flask offers you plenty of logging configuration options. But for your use case, it's best to use the default logger of your Flask app.

Open `__init__.py` and replace the `print()` calls with debugging logs:

```
# __init__.py
# flask_board/board/__init__.py
```

```
# ...

def create_app():
    app = Flask(__name__)
    app.config.from_prefixed_env()

    database.init_app(app)

    app.register_blueprint(pages.bp)
    app.register_blueprint(posts.bp)
    app.register_error_handler(404, errors.page_not_found)
    app.logger.debug(f"Current Environment: {os.getenv('ENVIRONMENT')}")
    app.logger.debug(f"Using Database: {app.config.get('DATABASE')}")
    return app
```

If you restart your Flask development server, then the output looks similar to the output that you saw before. But instead of just seeing your messages in the output, you can additionally see a timestamp and the file that triggered the log message:

```
(venv) $ python -m flask --app board run --port 8000 --debug
```

In Flask, using `app.logger.debug()` instead of the `print()` function call offers another major advantage. You can work with **logging levels**.

Logging levels allow you to categorize messages according to their severity and importance. For example, using `app.logger.debug()` for debug messages ensures that you only see these messages when you start the development server in the `debug` mode.

With `debug` in your `flask` command, you can see both logging messages that you added to `__init__.py`. Now, stop the Flask development server by pressing Ctrl+C in your terminal and start the server again with this command:

```
(venv) $ python -m flask --app board run --port 8000
```

Without the `debug` flag, you run the server on the `warning` log level. As this level is higher than `debug`, Flask doesn't output both of your messages anymore. That's cool because it allows you to see debugging messages during your development and other logging messages in normal production.

Show Some Info

For now, you're using logging when your application factory runs. But you can also log when a user interacts with your forms or runs into an error. For this, you'll use the `info` log level.

Start by expanding the `create()` view that handles your forms in `posts.py`:

```
# posts.py
# flask_board/board/posts.py
from flask import (
    Blueprint,
    current_app,
    flash,
    redirect,
    render_template,
    request,
    url_for,
)

from board.database import get_db

bp = Blueprint("posts", __name__)

@bp.route("/create", methods=("GET", "POST"))
def create():
    if request.method == "POST":
        author = request.form["author"] or "Anonymous"
        message = request.form["message"]

        if message:
            db = get_db()
            db.execute(
                "INSERT INTO post (author, message) VALUES (?, ?)",
                (author, message),
            )
            db.commit()
            current_app.logger.info(f"New post by {author}")
            flash(f"Thanks for posting, {author}!", category="success")
            return redirect(url_for("posts.posts"))
        else:
            flash("You need to post a message.", category="error")

    return render_template("posts/create.html")

# ...
```

After importing `current_app` you can access your app's logger in your views. With `current_app.logger.info` you're logging when a user submits a form. You use the `info` log level for this message to see it even when you run the server without the `debug` flag.

Another opportunity to introduce some logging in your Flask project is the error handler that you created in the former section. For you as an admin, it can be helpful to know what pages your user wants to visit.

Open `errors.py` and adjust the code as shown below:

```
# errors.py
# flask_board/board/errors.py
from flask import current_app, render_template, request

def page_not_found(e):
    current_app.logger.info(f"'{e.name}' error ({e.code}) at {request.url}")
    return render_template("errors/404.html"), 404
```

Remember the parameter `e` that you added to `page_not_found()`, With the error object available, you can access more information about an error.

You import the `request` object, with that you can log the URL that the user entered in the browser.

To see the info-level log messages, you need to adjust the log level that Flask outputs:

```
# __init__.py
# flask_board/board/__init__.py
# ...

def create_app():
    app = Flask(__name__)
    app.config.from_prefixed_env()
    app.logger.setLevel("INFO")

    database.init_app(app)

    app.register_blueprint(pages.bp)
    app.register_blueprint(posts.bp)
    app.register_error_handler(404, errors.page_not_found)
    app.logger.debug(f"Current Environment: {os.getenv('ENVIRONMENT')}")
    app.logger.debug(f"Using Database: {app.config.get('DATABASE')}")
    return app
```

You can try out your new info log messages by visiting a page that doesn't exist and sending a form. If you do, then your server logs will look something like this

```
...
[2023-11-20 20:10:10,694] INFO in posts: New post by Philipp
127.0.0.1 - - [20/Nov/2023 20:10:10] "POST /create HTTP/1.1" 302 -
127.0.0.1 - - [20/Nov/2023 20:10:10] "GET /posts HTTP/1.1" 200 -
[2023-11-20 20:10:31,425] INFO in errors: 'Not Found' error (404) at
http://127.0.0.1:8000/part-4
127.0.0.1 - - [20/Nov/2023 20:10:31] "GET /part-4 HTTP/1.1" 404 -
```

By incorporating logging into your application factory, you’re now able to track when users interact with your forms or encounter errors.