

Flask Blog Post API

Overview

This is a RESTful API built using Python Flask for managing blog posts. It includes features for creating, retrieving, updating, and deleting blog posts. The API also implements JWT-based authentication to secure these operations. SQLAlchemy is used for in-memory database management.

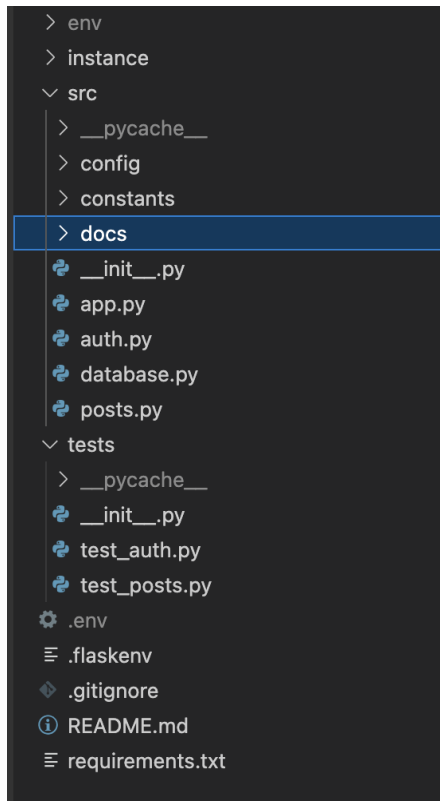
Features

- **JWT Authentication:** Users can sign up, sign in, and authenticate their requests.
- **CRUD Operations for Blog Posts:**
 - Create a new blog post
 - Retrieve a list of all blog posts
 - Retrieve a single blog post by its ID
 - Update an existing blog post
 - Delete a blog post

Technologies Used

- **Python Flask:** Web framework for building the API.
- **SQLAlchemy:** ORM for database management.
- **JWT:** Authentication mechanism.

Project Structure



Setup and Installation

- **Clone the repository:**
 - `git clone <repository-url>`
 - `cd <repository-directory>`
- **Create and activate a virtual environment:**
 - `python -m venv env`
 - `source env/bin/activate` (On Windows use `env\Scripts\activate`)
- **Install the dependencies:**
 - `pip install -r requirements.txt`
- **Set up environment variables for Mac:**

Create a `.flaskenv` file and add the following variables:

 - `export FLASK_APP=src`
 - `export FLASK_RUN_PORT=8000`
 - `export FLASK_ENV=development`

- `export FLASK_DEBUG=1`
- `export SQLALCHEMY_DB_URI=sqlite:///posts.db`
- `export JWT_SECRET_KEY=your_secret_key`

Create a `.env` file and add the following variables:

- `export SECRET_KEY=your_secret_key`

Running the Application

To run the application, use the following command:

```
flask run
```

Testing the Application

```
python -m unittest discover -s tests
```

Design Decisions and Trade-offs

Design Decisions

1. **Framework Choice:**
 - **Flask** was chosen for its simplicity and flexibility, which is suitable for building lightweight APIs quickly.
2. **Database Management:**
 - **SQLAlchemy** was selected as the ORM for its ease of integration with Flask and its capability to handle in-memory databases efficiently during development and testing.
3. **Authentication:**
 - **JWT (JSON Web Tokens)** was implemented for secure authentication, ensuring that each request is properly authenticated without maintaining session states on the server.
4. **Documentation:**
 - **Swagger** Add detailed API documentation using Swagger to make it easier for developers to understand and use the API.
5. **Testing**
 - **Unit test library** Used Unittest a built-in testing framework to write unit test cases.

Trade-offs

1. In-memory Database:

- Using an in-memory database with SQLAlchemy is efficient for development and testing but not suitable for production due to the lack of persistence. For production, a more robust database like PostgreSQL or MySQL should be used.

2. Simplicity vs. Scalability:

- Flask is excellent for simple applications but may require additional configuration and extensions for handling larger, more complex applications.

Potential Improvements with More Time

1. Database Integration:

- Integrate a robust database such as PostgreSQL or MongoDB to handle data persistence more effectively.

2. Enhanced Error Handling:

- Implement more comprehensive error handling and validation throughout the application to improve reliability and user experience.

3. Automated Testing:

- Expand the test suite to include more test cases, ensuring greater coverage and reliability of the API.

4. Deployment:

- Set up a deployment pipeline using Docker and a CI/CD tool to automate testing and deployment processes.

5. Rate Limiting:

- Implement rate limiting to prevent abuse and ensure fair usage of the API.

6. Testing

- Used inbuilt testing library, can shift to more robust and reliable testing tools like pytest.

7. Frontend

- We can create a website to show blog posts by integrating with the APIs created.