Build a Flask CRUD Application with MVC Architecture

How to implement a CRUD App with Flask Blueprint

Felipe Silveira Follow
Apr 1 · 4 min read

I have been working with Node.js in both development and production applications for several years and recently I have started specialization in Full Stack Web Development where Flask is used.

I remember my first hands-on Node.js project and it was tremendously faster to get something up when comparing now with Flask. As it happens, with Flask I got stuck a lot of times while implementing a simple CRUD application, mainly in how to architecture the project using Python.

In particular, I have implemented many machine learning applications using Python. In my opinion, it is the best language for that. However, when it comes to implementing CRUD and/or API projects I still would prefer Node.js.

Overall, note that this tutorial is not focused on explaining what is a CRUD App neither its methods. It is simply a guide in how to implement an application following an MVC Architecture.

I tried to implement it as related as possible to a NodeJS application and I hope you like it. Please, feel free to contact me with any comments or suggestions.

In order to build this application, we first need to understand the basic idea of CLIENT / SERVER.

- The Client sends a request to the server.
- After manipulating the Database response, the Server then sends a response to the Client.

In this approach, the Server response returns the entire page requested by the Client. For example, requesting the page `/users` returns all the HTML, CSS, and JS for that. A practical understanding of it is the `render_template` method from flask.

Bear in mind that this format is limited to a specific Client. As the Server renders HTML, CSS, and JS, it is impossible for an Android or iOS Client to handle it. As a solution, RESTful APIs became often used, removing the backend responsibility to render templates.

The project structure

This tutorial assumes that you already have installed Python, PostgreSQL, and Flask on your computer. After that, please create the following structure in your project folder.

```
project/
|
├── templates/
|   └── index.html
|
├── migrations/ **
|   └── ...
|
├── routes/
|
|   └── user_bp.py
|
├── models/
|   └── User.py
|
├── controllers/
|   └── UserController.py
|
├── app.py
└── config.py
```

- ** automatically created from the command* `flask db init`.

If you come from another framework you might have noticed the Flask CLI is used to set things such as `FLASK_APP=app.py` and `FLASK_ENV=development` but not to automatically create this structure for us.

Now, let's start writing code.

In *src/app.py:*

```python
from flask import Flask, render_template
from flask_migrate import Migrate

from models.User import db
from routes.user_bp import user_bp

app = Flask(__name__)
app.config.from_object('config')

db.init_app(app)
migrate = Migrate(app, db)

app.register_blueprint(user_bp, url_prefix='/users')

@app.route('/')
def index():
    return render_template('index.html')

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

In src/config.py:

```
import os

SECRET_KEY = os.urandom(32)

# Grabs the folder where the script runs.
basedir = os.path.abspath(os.path.dirname(__file__))

# Enable debug mode.
DEBUG = True

# Connect to the database
SQLALCHEMY_DATABASE_URI = 'your psycopg2 URI connection'

# Turn off the Flask-SQLAlchemy event system and warning
SQLALCHEMY_TRACK_MODIFICATIONS = False
```

In src/models/User.py:

```
from flask_sqlalchemy import SQLAlchemy

db = SQLAlchemy()

class User(db.Model):
    __tablename__ = 'users'

    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String)
    age = db.Column(db.String(120))
    address = db.Column(db.String(120))

    @property
    def serialize(self):
        return {
            'id': self.id,
            'name': self.name,
            'city': self.city,
            'state': self.state,
            'address': self.address
        }
```

After that, you can run the commands:

- `flask db init` : it will create the migrations folder with a version subfolder.
- `flask db migrate` : it will detect the model changes with an upgrade and downgrade logic set up.
- `flask db upgrade` : it will apply the model changes you have implemented.
- `flask db downgrade` : if something goes wrong, you can use this command to unapply changes you have done on your model file.

Routes:

Routes are usually written with the controller function as you can see

```
@app.route('/', methods=['GET'])
def index():
    ...

@app.route('/create', methods=['POST'])
def store():
    ...

...
```

However, as we want to organize it following an MVC architecture, we need some changes. To do that, `Blueprint` can be used as a solution. It is responsible to split the code into different modules resulting in better maintenance and improving scalability.

In *src/routes/user_bp.py:*

```
from flask import Blueprint

from controllers.UserController import index, store, show, update,
destroy

user_bp = Blueprint('user_bp', __name__)
```

```
user_bp.route('/', methods=['GET'])(index)
user_bp.route('/create', methods=['POST'])(store)
user_bp.route('/<int:user_id>', methods=['GET'])(show)
user_bp.route('/<int:user_id>/edit', methods=['POST'])(update)
user_bp.route('/<int:user_id>', methods=['DELETE'])(destroy)
```

Returning to the `app.py` , you can notice this blueprint being registered in `app.register_blueprint(user_bp, url_prefix='/users')`

In src/controllers/UserController.py:

```
# pseudo code

import sys
from flask import render_template, redirect, url_for, request, abort

from models.User import User

from flask_sqlalchemy import SQLAlchemy
db = SQLAlchemy()

def index():
    ...

def store():
    ...

def show(userId):
    ...

def update(userId):
    ...

def delete(userId):
    ...
```

After coding all the necessary methods, you can run the project using the command `flask run` or `python app.py` . Finally, you can check if your project is correctly running at http://localhost:5000/

This tutorial is focused to guide you on the backend implementation, if you want details on the frontend part, please leave a comment for a second tutorial.

I hope I've shown you how easy it can be to implement a CRUD Application using Flask as it is with Node.js. Thanks for reading!

Flask Python Programming Software Development Sql