How to Build a REST API with Flask and SQLAlchemy

Updated at May 07, 2020

Flask is a great framework that enables you to build web applications quickly with Python. It's fast, small, and fun to work with. In this tutorial, we're going to build a RESTful API with Flask framework, and some other supporting tools.

The objective of this tutorial is to understand the concept of building a Flask server from the ground up, learn how to commuticate with SQL databases via object-relational mapper, as well as design a RESTful API with object-oriented design pattern.

By the end of this tutorial, you will be able to build a REST API that can manage blog posts data with create, read, update, and delete functionality.

Getting Started

I have published the finished project of this tutorial on my GitHub, you can check it out here or clone it into your machine by running the command below.

```
$ git clone https://github.com/rahmanfadhil/flask-rest-api.git
```

Then, create a new Python virtual environment and install the dependencies with Pip.

```
$ python -m venv env
$ source env/bin/activate
(env) $ pip install -r requirements.txt
```

Dependencies

According to its documentation, Flask is a micro-framework. This means, unlike any other web frameworks out there, Flask doesn't come with a lot of fancy stuff out of the box.

This makes Flask simpler and easier to learn than others. But at the same time, we often forced to find our own solutions to solve common problems. For instance, connecting to a database, implementing an authentication system, and so on.

Even though, the Flask community has provided some useful open-source extensions that allow us to rapidly solve those problems, only by installing and configuring them.

Here are some third-party packages we will use to build our API:

- Flask SQLAlchemy: A Flask extension that adds support for SQLAlchemy, an object-relational mapper which makes us easier to interact with SQL databases.
- Flask RESTful: Another Flask extension for quickly building REST APIs with object-oriented design pattern.
- Flask Marshmallow: Yet another Flask extension that integrates with Marshmallow, a python library for object serialization.

Flask project from ground up

We can start our project by initializing a new virtual environment with Python venv module.

```
$ python3 -m venv env
$ source env/bin/activate
```

Second, install our we can install our dependencies via Pip by running this command.

```
$ pip install Flask \
    Flask-SQLAlchemy \
    Flask-RESTful \
    flask-marshmallow
```

Finally, create a new Python file called `main.py` (or whatever you want to name it). Here, we setup our brand new Flask server.

```
from flask import Flask

app = Flask(__name__)

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

To test it out, we can execute our Python script with the command below.

```
(env) $ python main.py
```

Open http://localhost:5000, and you will see a 404 page.

Database setup

Here, we're going to use an SQL database to store our blog posts data.

For learning purposes, I'm going to use SQLite, a small SQL database implementation which very easy to get up and running. Keep in mind that you might want to consider a more reliable database like PostgreSQL or MySQL in production environment.

To setup SQLAlchemy in a Flask project, we can import the `flask_sqlalchemy` package (which we've installed earlier), and wrap our Flask `app` variable in a new `SQLAlchemy` object. We also want to setup `SQLALCHEMY_DATABASE_URI` in our flask app configuration to specify which database we want to use and how to access it.

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy # new

app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///test.db' # new
db = SQLAlchemy(app) # new

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

Database model

A model is nothing more than a representation of your database, where you can store, fetch, and manipulate your data from it. Think of it as the middleman of your app and database. And its usually represents a single table/collection.

It lets you do everything to that specific table directly from your python code. So that you don’t need to mess around with SQL queries anymore.

Let's create a model that represents our blog post data.

```
class Post(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    title = db.Column(db.String(50))
    content = db.Column(db.String(255))

    def __repr__(self):
        return '<Post %s>' % self.title
```

Here, we have an `id` property which is an auto-generated primary key field. Then, we also have a `title` and `content` field, just an ordinary string field with maximum length defined.

This part is optional, but to make sure that everything is obvious, we can set a `__repr__` method to make every single post object is printable to the console.

> Pro tip: overriding the default `__repr__` method allows you to display useful information about particular model using Python repr function. This makes your debuging process much more easier.

Then, we need to setup the schema for our model. This is necessary because we want to parse our post object(s) into a JSON response. In this case, we can make use of `flask_marshmallow` package.

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy
from flask_marshmallow import Marshmallow # new

app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///test.db'
db = SQLAlchemy(app)
ma = Marshmallow(app) # new
```

```
# ...

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

Create a new marshmallow schema based on our `Post` model.

```
class PostSchema(ma.Schema):
    class Meta:
        fields = ("id", "title", "content")
        model = Post

post_schema = PostSchema()
posts_schema = PostSchema(many=True)
```

In this schema, we can choose what fields to expose to our users. If your model has some sensitive data, you may want to exclude it here. Then, we also instantiate it in `posts_schema` and `post_schema`. Use `posts_schema` to serialize an array of posts. Otherwise, use `post_schema`.

The next important part which most people usually forget (including myself) is to initialize our database schema by invoking the `db.create_all` function from our SQLAchemy instance inside a Python interactive shell (REPL). The reason why we do this in a REPL is because we only need to initialize our schema once.

Also, make sure you are in the right Python environment.

```
(env) $ python
>>> from main import db
>>> db.create_all()
>>> exit()
```

RESTful Routes

Finally, we can start to define our RESTful handler. We will make use of Flask-RESTful package, a set of tools that help us to construct a RESTful routes with object-oriented design.

We need to setup Flask-RESTful extension to get up and running in our Flask server.

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy
from flask_restful import Api, Resource # new

app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///test.db'
db = SQLAlchemy(app)
api = Api(app) # new

# ...

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

Create a new RESTful resource, in that way, we define our business logic like how to fetch our data from the database, how to do authentication, and so on.

```
class PostListResource(Resource):
    def get(self):
        posts = Post.query.all()
        return posts_schema.dump(posts)

api.add_resource(PostListResource, '/posts')
```

Here, we accept a `GET` request and make query to fetch all posts with `Post` model. Then, we take advantage of `posts_schema` to serialize the data from database and return it as a response to the client.

Finally, we register our resource by using `api.add_resource` method and define the rouute endpoint.

Start the server, send a request to `/posts` endpoint, and you will get an empty array.

```
$ curl http://localhost:5000/posts
[]
```

Cool, now it's time to work with the create operation.

```
# ...
from flask import Flask, request # change

# ...

class PostListResource(Resource):
    def get(self):
        posts = Post.query.all()
        return posts_schema.dump(posts)

    # new
    def post(self):
        new_post = Post(
            title=request.json['title'],
            content=request.json['content']
        )
        db.session.add(new_post)
        db.session.commit()
        return post_schema.dump(new_post)
```

```
# ...
```

We create a new `post` method in our resource, instantiate a new post object with the request data, and save the record to the database. Finally, we return the post data as the response to the client.

Try to send a POST request to `/todos` endpoint with a post data.

```
$ curl http://localhost:5000/posts \
    -X POST \
    -H "Content-Type: application/json" \
    -d '{"title":"Post 1", "content":"Lorem ipsum"}'
{
    "content": "Lorem ipsum",
    "id": 1,
    "title": "Post 1"
}
```

Let's create a new resource to fetch individual post.

```
class PostResource(Resource):
    def get(self, post_id):
        post = Post.query.get_or_404(post_id)
        return post_schema.dump(post)

api.add_resource(PostResource, '/posts/<int:post_id>')
```

Here, we also have accept a `GET` request, but instead of querying all posts, we just fetch a single post with the given id. If it not exist, it will raise a 404 error.

Try to fetch `/todos/<int:id>`, and you will get the post that we're just created.

```
$ curl http://localhost:5000/posts/1
{
    "title": "Post 1",
    "content": "Lorem ipsum",
    "id": 1
}
```

And, if you request a post with an id that doesn't exist, you will get a 404 error.

```
$ curl http://localhost:5000/posts/12 -I
HTTP/1.0 404 NOT FOUND
...
```

Now, let's add the remaining CRUD operations, update and delete.

```
class PostResource(Resource):
    # ...

    def patch(self, post_id):
        post = Post.query.get_or_404(post_id)

        if 'title' in request.json:
            post.title = request.json['title']
        if 'content' in request.json:
            post.content = request.json['content']

        db.session.commit()
        return post_schema.dump(post)

    def delete(self, post_id):
        post = Post.query.get_or_404(post_id)
        db.session.delete(post)
```

```
        db.session.commit()
        return '', 204
```

In the `patch` method, we first get the post object if exist, then we update the properties which defined in the request body (`request.json`). That's why we need to check both properties with `in` expression. Save the changes to the database by using the `db.session.commit()` and send the update data to the client.

In the `delete` method, we also get the post object. But this time, we just delete the object with `delete` method from the post object. Save the changes and return nothing to the client (because there's nothing to show for).

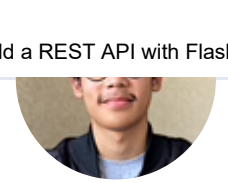
Now, we have all the functionalities up and running, let's test it out!

Update post:

```
$ curl http://localhost:5000/posts/1 \
    -X PATCH \
    -H "Content-Type: application/json" \
    -d '{"title":"Updated Post", "content":"Updated post content"}'
{
    "content": "Updated post content",
    "id": 1,
    "title": "Updated Post"
}
```

Delete post:

```
$ curl http://localhost:5000/posts/1 -X DELETE -I
HTTP/1.0 204 NO CONTENT
...
```



Rahman Fadhil

I'm a software engineer specialized in iOS and full-stack web development. I can help you to learn new skills and solve your coding problems in Codementor.

Find me on codementor