Onwuka Gideon FOLLOW
Full-stack Problem Solver.

How to Build RESTful APIs with Python and Flask

Published Jan 05, 2018

For some time now I have been working with Python but I just got to try out Flask recently, so I felt it would be nice to write about it. In this aritcle I'll discuss about Flask and how you can use it to build RESTfull APIs.

Flask is a Python-based microframework that enables you to quickly build web applications; the “micro” in microframework simply means Flask aims to keep the core simple but extensible.

What is REST?

REST is acronym for **RE**presentational **S**tate **T**ransfer. It is an architectural style, and an approach to communications that is often used in the development of Web services.

REST web services are a way of providing interoperability between computer systems on the Internet. REST-compliant Web services allow requesting systems to access and manipulate textual representations of Web resources using a uniform and predefined set of stateless operations.

This guide will walk you through building a Restful APi with Flask coupled with other extensions - Flask-RESTful, flask_migrate, marshmallow etc. At the end, you'll build a simple Commenting API. The API will have endpoints that can be used to `add category`, `view categories`, `update category`, `delete category`, `add comments` and `view comments`.

> *Flask does not come out of the box with all the extension needed to buld a full fledge API. However there are lot's of extension that can be pluged into Flask to enable you build awesome APIs.*

Enjoy this post? ♡ 58

- GET - `/api/Category` - Retrieve all categories
- POST - `/api/Category` - Add a new category
- PUT - `/api/Category` - Update a category
- DELETE - `/api/Category` - Delete a category
- GET - `/api/Comment` - Retrieve all the stored comments
- POST - `/api/Comment` - Add new comment

Before You Begin

- Make sure you have Python3.x installed on your system.
- We'll use PostgreSQL database for this tutorial. Install Postgres database if you don't have it installed already.
- Make sure virtualenv is installed on your system.

Step 0: Setting up the application

- First, create the structure of the app at any location on your system:

```
project/
├── app.py
├── migrate.py
├── Model.py
├── requirements.txt
├── resources
│   └── Hello.py
└── run.py
```

If you prefer the command line to create the folders and files, you can use the below command:

Enjoy this post?

58

```
# Create folders
$ mkdir project && cd project && mkdir resources

# Create files
$ touch app.py migrate.py Model.py requirements.txt run.py config.py && cd reso
```

- Then open up your terminal and CD (change directory) into the app root folder - `project` .
- Next, create and activate a virtual environment.

> *virtualenv is a tool to create isolated Python environments. virtualenv creates a folder which contains all the necessary executables to use the packages that a Python project would need.*

Creates virtual environment

```
$ python3.6 -m venv env
```

OR:

```
$ python -m venv env
```

The command to use depends on which associates with your python3 installation.

Activates the virtual environment

Enjoy this post? 58

```
$ source env/bin/activate
```

Add the folowing extensions to `requirements.txt` :

```
flask==0.12.2
flask_restful==0.3.6
flask_script==2.0.6
flask_migrate==2.1.1
marshmallow==2.14.0
flask_sqlalchemy==2.3.2
flask_marshmallow==0.8.0
marshmallow-sqlalchemy==0.13.2
psycopg2==2.7.5
```

The above file contains all python extensions our API will use.

- flask - This is a microframework for Python
- flask_restful - This is an extension for Flask that adds support for quickly building of REST APIs.
- flask_script - This is an extension that provides support for writing external scripts in Flask.
- flask_migrate - This is an extension that handles SQLAlchemy database migrations for Flask applications using Alembic.
- marshmallow - This is an ORM/ODM/framework-agnostic library for converting complex datatypes, such as objects, to and from native Python datatypes. We'll use this for validation. This is used to Serializing and Deserializing Objects.
- flask_sqlalchemy - This is an extension for Flask that adds support for SQLAlchemy.
- flask_marshmallow - This is an ntegration layer for Flask and marshmallow (an object serialization/deserialization library) that adds additional features to marshmallow.
- marshmallow-sqlalchemy - This adds additional features to marshmallow.
- psycopg - This is a PostgreSQL adapter for the Python programming language

Enjoy this post?

♡ 58

Step 1: Install all dependencies:

From your terminal, make sure you are in the root folder of the project then run the below command:

```
$ pip install -r requirements.txt
```

This will download and install all the extensions in `requirements.txt` .

Step 2: Setting up configuration

Add the following code to `config.py` :

```
import os

# You need to replace the next values with the appropriate values for your conf

basedir = os.path.abspath(os.path.dirname(__file__))
SQLALCHEMY_ECHO = False
SQLALCHEMY_TRACK_MODIFICATIONS = True
SQLALCHEMY_DATABASE_URI = "postgresql://username:password@localhost/database_na
```

Here, we have defined some configuration that the API will be using. We are also using the PostgreSQL database. If you prefer other databases, you just have to modify the value accordingly.

Example: if you want to use `SQLite` , you should modify this line as:

Enjoy this post?

58

```
SQLALCHEMY_DATABASE_URI = "sqlite:///app.db"
```

> *Make sure you change the above settings with the appropriate values for your configuration. Change the `username` and `password` to your correct database credentials.*

Step 3: Create the API entry points

Add the following code to `app.py` :

```
from flask import Blueprint
from flask_restful import Api
from resources.Hello import Hello

api_bp = Blueprint('api', __name__)
api = Api(api_bp)

# Route
api.add_resource(Hello, '/Hello')
```

Here we imported `Blueprint` from flask, and also `Api` from flask_restful. The imported `Api` will add some functionality to flask which will help us to add routes and simplify some processes.

`api_bp = Blueprint('api', __name__)` - creates a Blueprint which we'll register to the app later.

`api.add_resource(Hello, '/Hello')` - creates a route - `/Hello` . add_resource accepts two parameter - `Hello` and `/Hello` , where `Hello` is the class we have imported and `/Hello` is the route we defined for that Resource.

Now let's create the `Hello` class we mentioned above. Add the following code to -

Enjoy this post? ♡ 58

```
from flask_restful import Resource


class Hello(Resource):
    def get(self):
        return {"message": "Hello, World!"}
```

In the Hello class, we defined a function - `get` . This means that any GET Request on the `/Hello` endpoint will be hitting this function.

So if you need a POST method, you will have something like:

```
    def post(self):
        return {"message": "Hello, World!"}
```

Finally, add the following code to `run.py` :

Enjoy this post?

♡ 58

```
from flask import Flask

def create_app(config_filename):
    app = Flask(__name__)
    app.config.from_object(config_filename)

    from app import api_bp
    app.register_blueprint(api_bp, url_prefix='/api')

    from Model import db
    db.init_app(app)

    return app


if __name__ == "__main__":
    app = create_app("config")
    app.run(debug=True)
```

Step 4: Starting the Server

From your terminal, make sure you are on the root folder of the app then run this command:

```
$ python run.py
```

If everything goes fine at this point, you should have an output like the below:

```
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 136-695-873
```

Enjoy this post?

♡ 58

```
{
    "hello": "world"
}
```

Awesome! 😄

Step 5: Creating Models.

Up next is to write our models. Flask provides developers with the power and flexibility of the SQL Alchemy ORM to manage and query the application data. ORM stands for Object Relational Mapper, it is a tool that allows developers to store and retrieve data using object oriented approaches and solves the object-relational impedance mismatch.

Also, Flask-SQLAlchemy is an extension that provides support for SQLAlchemy ORM in the Flask framework.

In the - `Model.py` file, add the following code to it:

```
from flask import Flask
from marshmallow import Schema, fields, pre_load, validate
from flask_marshmallow import Marshmallow
from flask_sqlalchemy import SQLAlchemy


ma = Marshmallow()
db = SQLAlchemy()


class Comment(db.Model):
    __tablename__ = 'comments'
    id = db.Column(db.Integer, primary_key=True)
    comment = db.Column(db.String(250), nullable=False)
    creation_date = db.Column(db.TIMESTAMP, server_default=db.func.current_time
    category_id = db.Column(db.Integer, db.ForeignKey('categories.id', ondelete
    category = db.relationship('Category', backref=db.backref('comments', lazy=

    def __init__(self, comment, category_id):
```

Enjoy this post? ♡ 58

```
class Category(db.Model):
    __tablename__ = 'categories'
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(150), unique=True, nullable=False)

    def __init__(self, name):
        self.name = name


class CategorySchema(ma.Schema):
    id = fields.Integer()
    name = fields.String(required=True)


class CommentSchema(ma.Schema):
    id = fields.Integer(dump_only=True)
    category_id = fields.Integer(required=True)
    comment = fields.String(required=True, validate=validate.Length(1))
    creation_date = fields.DateTime()
```

Here, we have two models - `Comment` and `Category` class. These Class defines our table structure.
`__tablename__ = 'categories'` is used to declare the table name associated with that Model.

We've also defined columns for our table using eg:

```
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(150), unique=True, nullable=False)
```

The `id` and `name` are the columns for the table and the the value after the `=` defines column structure. You can read more on creating table schema here

Enjoy this post? ♡ 58

Step 6: Running migrations

In the `migrate.py` file add the following code to it:

```
from flask_script import Manager
from flask_migrate import Migrate, MigrateCommand
from Model import db
from run import create_app

app = create_app('config')

migrate = Migrate(app, db)
manager = Manager(app)
manager.add_command('db', MigrateCommand)


if __name__ == '__main__':
    manager.run()
```

In the code above, we created a command for running migration from our command prompt.

Migrations

Migrations are like version control for your database, allowing your team to easily modify and share the application's database schema. If you have ever had to tell a teammate to manually add a column to their local database schema, you've faced the problem that database migrations solve. In this tutorial, we are using a flask extension - flask migrate to acheive this.

- Run migrations initialization, using the db init command as follows:

Enjoy this post?

58

```
$ python migrate.py db init
```

```
Creating directory /home/dongido/Desktop/pylinone/migrations ... done
Creating directory /home/dongido/Desktop/pylinone/migrations/versions ... don
Generating /home/dongido/Desktop/pylinone/migrations/alembic.ini ... done
Generating /home/dongido/Desktop/pylinone/migrations/README ... done
Generating /home/dongido/Desktop/pylinone/migrations/env.py ... done
Generating /home/dongido/Desktop/pylinone/migrations/script.py.mako ... done
Please edit configuration/connection/logging settings in '/home/dongido/Deskt
proceeding.
```

The script also generated a new migrations sub-folder within the folder with a versions subfolder and many other files.

Run the second script that populates the migration script with the detected changes in the
models. In this case, it is the first time we populate the migration script, and therefore, the
migration script will generate the tables that will persist our two models: Category and Comment:

Run the migration:

Enjoy this post? 58

```
$ python migrate.py db migrate
```

```
INFO  [alembic.runtime.migration] Context impl SQLiteImpl.
INFO  [alembic.runtime.migration] Will assume non-transactional DDL.
INFO  [alembic.autogenerate.compare] Detected added table 'category'
INFO  [alembic.autogenerate.compare] Detected added table 'comment'
  Generating /home/dongido/Desktop/pylinone/migrations/versions/015c8b601c6b_.p
```

- Then apply the migration to the database. Run the upgrade command:
- Run the upgrade command:

```
python migrate.py db upgrade
```

```
INFO  [alembic.runtime.migration] Context impl SQLiteImpl.
INFO  [alembic.runtime.migration] Will assume non-transactional DDL.
INFO  [alembic.runtime.migration] Running upgrade  -> 015c8b601c6b, empty messa
```

Each time the database models change repeat the `migrate` and `upgrade` commands.

Step 7: Adding Category resources

Create a new file `resources/Category.py` . This will take care of all Category related Logic. Add the below code to `resources/Category.py` :

Enjoy this post? 58

```python
from flask import request
from flask_restful import Resource
from Model import db, Category, CategorySchema

categories_schema = CategorySchema(many=True)
category_schema = CategorySchema()

class CategoryResource(Resource):
    def get(self):
        categories = Category.query.all()
        categories = categories_schema.dump(categories).data
        return {'status': 'success', 'data': categories}, 200
```

Here, we have created a `get` method for fetching categories. We now have a new endpoint - `http://127.0.0.1:5000/api/Category` .

Using `Category.query.all()` , we fetched all categories in the database and using `categories_schema.dump(categories).data` , we deserialized the data fetched.

Next, lets add a POST method for creating new category. Add the following code to `resources/Category.py` :

Enjoy this post?

58

```
...
    def post(self):
        json_data = request.get_json(force=True)
        if not json_data:
               return {'message': 'No input data provided'}, 400
        # Validate and deserialize input
        data, errors = category_schema.load(json_data)
        if errors:
            return errors, 422
        category = Category.query.filter_by(name=data['name']).first()
        if category:
            return {'message': 'Category already exists'}, 400
        category = Category(
            name=json_data['name']
            )

        db.session.add(category)
        db.session.commit()

        result = category_schema.dump(category).data

        return { "status": 'success', 'data': result }, 201
```

Next, lets add a PUT method for updating category. Update `resources/Category.py` as below:

Enjoy this post? 58

```
...
    def put(self):
        json_data = request.get_json(force=True)
        if not json_data:
               return {'message': 'No input data provided'}, 400
        # Validate and deserialize input
        data, errors = category_schema.load(json_data)
        if errors:
            return errors, 422
        category = Category.query.filter_by(id=data['id']).first()
        if not category:
            return {'message': 'Category does not exist'}, 400
        category.name = data['name']
        db.session.commit()

        result = category_schema.dump(category).data

        return { "status": 'success', 'data': result }, 204
```

Finally, lets add a DELETE method for deleting category. Add the following code to `resources/Category.py` :

```
...
    def delete(self):
        json_data = request.get_json(force=True)
        if not json_data:
               return {'message': 'No input data provided'}, 400
        # Validate and deserialize input
        data, errors = category_schema.load(json_data)
        if errors:
            return errors, 422
        category = Category.query.filter_by(id=data['id']).delete()
        db.session.commit()

        result = category_schema.dump(category).data

        return { "status": 'success', 'data': result}, 204
```

Enjoy this post? 58

Finally, import these resources to `app.py` . Update `app.py` exactly as below:

```
from flask import Blueprint
from flask_restful import Api
from resources.Hello import Hello
from resources.Category import CategoryResource


api_bp = Blueprint('api', __name__)
api = Api(api_bp)

# Routes

api.add_resource(Hello, '/Hello')
api.add_resource(CategoryResource, '/Category')
```

In the Category resources, we have creating 4 endpoints:

GET - http://127.0.0.1:5000/api/Category

POST - http://127.0.0.1:5000/api/Category

PUT - http://127.0.0.1:5000/api/Category

DELETE - http://127.0.0.1:5000/api/Category

Step 8: Adding Comment resources

Create a new file `resources/Comment.py` . This will take care of all Comments related Logic.

Enjoy this post?

58

```
from flask import jsonify, request
from flask_restful import Resource
from Model import db, Comment, Category, CommentSchema

comments_schema = CommentSchema(many=True)
comment_schema = CommentSchema()

class CommentResource(Resource):
    def get(self):
        comments = Comment.query.all()
        comments = comments_schema.dump(comments).data
        return {"status":"success", "data":comments}, 200

    def post(self):
        json_data = request.get_json(force=True)
        if not json_data:
               return {'message': 'No input data provided'}, 400
        # Validate and deserialize input
        data, errors = comment_schema.load(json_data)
        if errors:
            return {"status": "error", "data": errors}, 422
        category_id = Category.query.filter_by(id=data['category_id']).first()
        if not category_id:
            return {'status': 'error', 'message': 'comment category not found'}
        comment = Comment(
            category_id=data['category_id'],
            comment=data['comment']
            )
        db.session.add(comment)
        db.session.commit()

        result = comment_schema.dump(comment).data

        return {'status': "success", 'data': result}, 201

    # You can add the other methods here...
```

Finally, import these resources to `app.py` . Update `app.py` exactly as below:

Enjoy this post?

58

```
from flask import Blueprint
from flask_restful import Api
from resources.Hello import Hello
from resources.Category import CategoryResource
from resources.Comment import CommentResource


api_bp = Blueprint('api', __name__)
api = Api(api_bp)

# Routes

api.add_resource(Hello, '/Hello')
api.add_resource(CategoryResource, '/Category')
api.add_resource(CommentResource, '/Comment')
```

Step 9: Test Enpoints

We now have the following endpoints available:

- POST - http://127.0.0.1:5000/api/category - for adding categories.

Enjoy this post?

58

- GET - http://127.0.0.1:5000/api/category - for listing categories.

Enjoy this post?

58

- POST - http://127.0.0.1:5000/api/comment - for adding comment.

Enjoy this post?

58

- GET - http://127.0.0.1:5000/api/comment - for listing comments.

Enjoy this post?

♡ 58

Conclusion

Building web application with flask is always fun! In this guide we have seen how we can build an API with Flask. Do you have any questions? or you think there is something missing, please feel free to add your comments below, I'll be more than happy to respond to them.

Enjoy this post?

58

Python Flask Rest Web Services Onwuka gideon Api

Enjoy this post?

58

Enjoy this post? Give **Onwuka Gideon** a like if it's helpful.

58 25 SHARE

Onwuka Gideon

Full-stack Problem Solver.

Software Developer, Security Professional, I build stuff for the web and beyond.

FOLLOW

25 Replies

Leave a reply

Accelerate Dllr a month ago

Hi, first of all, thank you for the very detailed tutorial. I tried to put this into practice, but i encountered a problem that i seem to find no solution for, when i wanna write this :

```
    from flask_restful import Api, Resource
```

Show more

Reply

Accelerate Dllr a month ago

I think i found the problem. i was still in (env) when i used pip install -r

Enjoy this post? 58

Rahul 2 months ago

Hi, I completed al the steps but not getting the last part . How are we getting GET.POST things...its giving me "Failed to Open page"
When I run, run.py

Traceback (most recent call last):
File "run.py", line 17, in <module>

Show more

Reply

Mahendra Rajdhami 3 months ago

i have successfully managed to run my app in local. will you please guide me to deploy in production server

Reply

Show more replies

Mathias Gatti

Python implementation of Normalized Google Distance (Simple web scraping example)

Enjoy this post?

58

Based on the count of google results we can infer the popularity of a word. Also the relationship between the frequency of two words together with respect to its individual frequency is a useful measure of how much two words are related.

READ MORE

Enjoy this post?

58