Implementing a RESTful Web API with Python & Flask

Luis Rei
http://blog.luisrei.com

me@luisrei.com
@lmrei

github.com/lrei
linkedin/in/lmrei

02 May 2012

Example Code

Hacker News Discussion

Introduction

To begin:

```
sudo pip install flask
```

I'm assuming you already know the basics of REST. If not, or if you want a quick refresh, I've written an introduction to Designing a RESTful Web API.

Flask is a microframework for Python based on Werkzeug, a WSGI utility library.

Flask is a good choice for a REST API because it is:

- Written in Python (that can be an advantage);
- Simple to use;
- Flexible;
- Multiple good deployment options;
- **RESTful request dispatching**

I normally use curl to make test requests and there's a curl mini-reference at the end of this article. Another nice tool is REST Console for Google Chrome.

As a convention in this document, whenever a server response is presented, it is preceded by the HTTP request that was made to generate the particular response with any relevant parameters and headers. The request itself is not part of the response.

Resources

Let's begin by making a complete app that responds to requests at the root, /articles and /articles/:id.

```
from flask import Flask, url_for
app = Flask(__name__)

@app.route('/')
def api_root():
    return 'Welcome'

@app.route('/articles')
def api_articles():
    return 'List of ' + url_for('api_articles')

@app.route('/articles/<articleid>')
def api_article(articleid):
    return 'You are reading ' + articleid

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

You can use curl to make the requests using:

```
curl http://127.0.0.1:5000/
```

The responses will be, respectively,

```
GET /
Welcome

GET /articles
List of /articles

GET /articles/123
You are reading 123
```

Routes can use different converters in their definition,

```
@app.route('/articles/<articleid>')
```

Can be replaced by

```
@app.route('/articles/<int:articleid>')
@app.route('/articles/<float:articleid>')
@app.route('/articles/<path:articleid>')
```

The default is `string` which accepts any text without slashes.

Requests

Flask API Documentation: Incoming Request Data

GET Parameters

Lets begin by making a complete app that responds to requests at `/hello` and handles an optional GET parameter

```
from flask import request

@app.route('/hello')
def api_hello():
    if 'name' in request.args:
        return 'Hello ' + request.args['name']
    else:
        return 'Hello John Doe'
```

the server will reply in the following manner:

```
GET /hello
Hello John Doe

GET /hello?name=Luis
Hello Luis
```

Request Methods (HTTP Verbs)

Lets modify the to handle different HTTP verbs:

```
@app.route('/echo', methods = ['GET', 'POST', 'PATCH', 'PUT', 'DELETE'])
def api_echo():
    if request.method == 'GET':
        return "ECHO: GET\n"

    elif request.method == 'POST':
        return "ECHO: POST\n"

    elif request.method == 'PATCH':
        return "ECHO: PACTH\n"

    elif request.method == 'PUT':
        return "ECHO: PUT\n"

    elif request.method == 'DELETE':
        return "ECHO: DELETE"
```

To curl the -X option can be used to specify the request type:

```
curl -X PATCH http://127.0.0.1:5000/echo
```

The replies to the different request methods will be:

```
GET /echo
ECHO: GET

POST /ECHO
```

```
ECHO: POST
```

and so on.

Request Data & Headers

Usually POST and PATCH are accompanied by data. And sometimes that data can be in one of multiple formats: plain text, JSON, XML, your own data format, a binary file, …

Accessing the HTTP headers is done using the `request.headers` dictionary ("dictionary-like object") and the request data using the `request.data` string. As a convenience, if the mimetype is *application/json*, `request.json` will contain the parsed JSON.

```
from flask import json

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

    if request.headers['Content-Type'] == 'text/plain':
        return "Text Message: " + request.data

    elif request.headers['Content-Type'] == 'application/json':
        return "JSON Message: " + json.dumps(request.json)

    elif request.headers['Content-Type'] == 'application/octet-stream':
        f = open('./binary', 'wb')
        f.write(request.data)
                f.close()
        return "Binary message written!"

    else:
        return "415 Unsupported Media Type ;)"
```

To specify the content type with curl:

```
curl -H "Content-type: application/json" \
-X POST http://127.0.0.1:5000/messages -d '{"message":"Hello Data"}'
```

To send a file with curl:

```
curl -H "Content-type: application/octet-stream" \
-X POST http://127.0.0.1:5000/messages --data-binary @message.bin
```

The replies to the different content types will be:

```
POST /messages {"message": "Hello Data"}
Content-type: application/json
JSON Message: {"message": "Hello Data"}

POST /message <message.bin>
Content-type: application/octet-stream
Binary message written!
```

Also note that Flask can handle files POSTed via an HTML form using `request.files` and curl can simulate that behavior with the `-F` flag.

Responses

Responses are handled by Flask's Response class:

```
from flask import Response

@app.route('/hello', methods = ['GET'])
def api_hello():
    data = {
        'hello'  : 'world',
        'number' : 3
    }
    js = json.dumps(data)

    resp = Response(js, status=200, mimetype='application/json')
    resp.headers['Link'] = 'http://luisrei.com'

    return resp
```

To view the response HTTP headers using curl, specify the `-i` option:

```
curl -i http://127.0.0.1:5000/hello
```

The response returned by the server, with headers included, will be:

```
GET /hello
HTTP/1.0 200 OK
Content-Type: application/json
Content-Length: 31
Link: http://luisrei.com
Server: Werkzeug/0.8.2 Python/2.7.1
Date: Wed, 25 Apr 2012 16:40:27 GMT
{"hello": "world", "number": 3}
```

Mimetype is just the content-type without the additional information (e.g. charset, encoding, language,…). If possible, return the the full content type information.

The previous example can be further simplified by using a Flask convenience method for generating JSON responses:

```
   from flask import jsonify
```

and replacing

```
 resp = Response(js, status=200, mimetype='application/json')
```

with

```
resp = jsonify(data)
```

```
resp.status_code = 200
```

which will generate the exact same response as the previous code.

Specifying the mime type is particularly useful when using a custom mime type e.g. *application/vnd.example.v2+json*.

Status Codes & Errors

Note that *200* is the default status code reply for `GET` requests, in both of these examples, specifying it was just for the sake of illustration. There are certain cases where overriding the defaults is necessary. Such is the case with error handling:

```
@app.errorhandler(404)
def not_found(error=None):
    message = {
            'status': 404,
            'message': 'Not Found: ' + request.url,
    }
    resp = jsonify(message)
    resp.status_code = 404

    return resp

@app.route('/users/<userid>', methods = ['GET'])
def api_users(userid):
    users = {'1':'john', '2':'steve', '3':'bill'}

    if userid in users:
        return jsonify({userid:users[userid]})
    else:
        return not_found()
```

This produces:

```
GET /users/2
HTTP/1.0 200 OK
{
    "2": "steve"
}

GET /users/4
HTTP/1.0 404 NOT FOUND
{
"status": 404,
"message": "Not Found: http://127.0.0.1:5000/users/4"
}
```

Default Flask error messages can be overwritten using either the `@error_handler` decorator or

```
app.error_handler_spec[None][404] = not_found
```

Even if the API does not need custom error messages, if supports different mime types (JSON, XML, …) this feature is important because Flask defaults to HTML errors.

There's a snippet by Pavel Repin that shows how to automatically replace all the default error messages with their JSON equivalents.

Authorization

Another very useful snippet by Armin Ronacher shows how to handle HTTP Basic Authentication and can be easily modified to handle other schemes. I have slightly modified it:

```
from functools import wraps

def check_auth(username, password):
    return username == 'admin' and password == 'secret'

def authenticate():
    message = {'message': "Authenticate."}
    resp = jsonify(message)

    resp.status_code = 401
    resp.headers['WWW-Authenticate'] = 'Basic realm="Example"'

    return resp

def requires_auth(f):
    @wraps(f)
    def decorated(*args, **kwargs):
        auth = request.authorization
        if not auth:
            return authenticate()

        elif not check_auth(auth.username, auth.password):
            return authenticate()
        return f(*args, **kwargs)

    return decorated
```

And using it is a matter of replacing the `check_auth` function and using the `requires_auth` decorator:

```
@app.route('/secrets')
@requires_auth
def api_hello():
    return "Shhh this is top secret spy stuff!"
```

So now, and unauthenticated request:

```
GET /secrets
HTTP/1.0 401 UNAUTHORIZED
WWW-Authenticate: Basic realm="Example"
{
  "message": "Authenticate."
}
```

While an authenticated request which can be made with curl using the `-u` option to use HTTP basic authentication and the `-v`option to look at the headers in the request

```
curl -v -u "admin:secret" http://127.0.0.1:5000/secrets
```

results in the expected response

```
GET /secrets Authorization: Basic YWRtaW46c2VjcmV0
Shhh this is top secret spy stuff!
```

Flask uses a MultiDict to store the headers. To present clients with multiple possible authentication schemes it is possible to simply add more *WWW-Authenticate* lines to the header

```
resp.headers['WWW-Authenticate'] = 'Basic realm="Example"'
resp.headers.add('WWW-Authenticate', 'Bearer realm="Example"')
```

or use a single line with multiple schemes (the standard allows both).

Simple Debug & Logging

Activating pretty (HTML) debug messages during development can be done simply by passing an argument

```
app.run(debug=True)
```

Flask uses python logging off the box - *some configuration required*:

```
import logging
file_handler = logging.FileHandler('app.log')
app.logger.addHandler(file_handler)
app.logger.setLevel(logging.INFO)

@app.route('/hello', methods = ['GET'])
def api_hello():
    app.logger.info('informing')
    app.logger.warning('warning')
    app.logger.error('screaming bloody murder!')

    return "check your logs\n"
```

Mini-Reference: curl options

option	purpose
-X	specify HTTP request method e.g. POST
-H	specify request headers e.g. "Content-type: application/json"
-d	specify request data e.g. '{"message":"Hello Data"}'
--data-binary	specify binary request data e.g. @file.bin
-i	shows the response headers
-u	specify username and password e.g. "admin:secret"

-v	enables verbose mode which outputs info such as request and response headers and errors

Bibliography

Flask documentation

Flask snippets

Werkzeug documentation

curl manual

Tagged: rest python

---

39 Comments blog.luisrei.com 1 Login

Recommend 36 Tweet Share Sort by Best

Join the discussion…

LOG IN WITH OR SIGN UP WITH DISQUS ?

Name

Bennarunga • 6 years ago

Great! tutorial many thanks Luis

9 • Reply • Share ›

ЮOpa • 4 years ago

Really simple to understand tutorial. Thank you!

2 • Reply • Share ›

userK • 2 years ago

Straight to the point! +1

1 • Reply • Share ›

Kejsarmakten • 5 years ago

Thanks. Nice tutorial!

1 • Reply • Share ›

Vaibhav Bansal • 3 months ago

Clear. To the point.
A very simple and helpful article. Thanks Luis Rei!

• Reply • Share ›

iraq2010 • 6 months ago

great tutorial thanks for sharing knowledge for all ...

• Reply • Share ›

noemi_quezada • 9 months ago

Probably my fave Flask tutorial/introduction. Was able to create a service-based api by just your guidelines. I feel so much more confident to read and be able to understand more complex Flask concepts and solutions.

• Reply • Share ›

Musa Rayy • a year ago

Really it's Greats! thanks

• Reply • Share ›

Oscar Calles • a year ago

Thanks!!

• Reply • Share ›

DarkWizard • 2 years ago

Salvou a minha vida. Thanks

• Reply • Share ›

Palash Gupta • 2 years ago

how to get input from user in a python api

• Reply • Share ›

boris runakov • 2 years ago

Thank you for this great tutorial! Can you please post an example of json data sent by the browser , with jquery for example ?

• Reply • Share ›

Sayan Misra • 2 years ago

Thanks for the tutorial!
I created one REST API using the above tutorial. However, when I am trying to access the

I created one REST API using the above tutorial. However, when I am trying to access the content of the API using python requests module or Curl, I need to refresh the API manually from my browser. If I dont refresh, the curl or the requests module does not return anything. Can you guys please help me here?

• Reply • Share ›

Shreeraj Dabholkar • 2 years ago

Great and to the point tutorial!

• Reply • Share ›

lapisan langit • 2 years ago

very helpful...and great tutorial....i hope you will add how to connect flask with database for example mysql, and return json file? i think this is very important...thanks..

• Reply • Share ›

ANSHU ADITYA • 2 years ago

Best tutorial I have ever found in this. Thank you so much.

• Reply • Share ›

Phoenix Song • 2 years ago

That's really a great tutorial! Huge thanks.

I've been crawling around to understand how the API server works and how to built one from a scratch, for days.... And this fantastic guide totally saved me!

• Reply • Share ›

Ashu kumar • 2 years ago

I have written one piece of code with basic auth as described in the article. Now I am trying to access that function with authentication, but everytime code returns 401 error.

```
import requests;
from requests.auth import HTTPBasicAuth
from requests.packages.urllib3.exceptions import InsecureRequestWarning
requests.packages.urllib3.disable_warnings(InsecureRequestWarning)
res = requests.get(<url>, verify=False, auth=HTTPBasicAuth('admin', 'secret'))
```

I tried pycurl as well

```
myCurlPut = pycurl.Curl()
myCurlPut.setopt(pycurl.URL, '<url>')
myCurlPut.setopt(pycurl.HTTPAUTH, pycurl.HTTPAUTH_BASIC)
myCurlPut.setopt(pycurl.USERPWD, "%s:%s" % ('admin', 'secret'))
```

myCurlPut.setopt(pycurl.SSL_VERIFYPEER, 0)
myCurlPut.perform()

Can you please suggest me, why authentication is not happening.

• Reply • Share ›

Dileep • 2 years ago

Hi Nice tutorial. I created the rest api and ran. But after closing the session I am not able to access the URL. I would like to how to deploy these in web server where we can access at any time

• Reply • Share ›

luisrei Mod Dileep • 2 years ago

Hi Dileep: see the Flask deployment options here: http://flask.pocoo.org/docs...

• Reply • Share ›

Ankush Thakur • 3 years ago

Very, very nice tutorial! You've helped me out of months of painful search and confusion. REST is now less of a mystery for me, and I can now look forward to digging deeper into the topic.

• Reply • Share ›

Sidhant hasija • 3 years ago

That was really helpful and simple reference. Thanks

• Reply • Share ›

Vijayenthiran Subramaniam • 3 years ago

I am new to RESTapi and Flask. '/hello' url request worked fine for me. But for '/echo' and '/messages' when I typed the given curl command in terminal, I am getting the following error:

```
<title>500 Internal Server Error</title>

<h1>Internal Server Error</h1>

The server encountered an internal error and was unable to complete your request. Either the server is overloaded or there is an error in the application.
```

where as `curl http://localhost:5000` returns the message as 'Welcome'

• Reply • Share ›

Kata Vijayenthiran Subramaniam • 3 years ago

You need to import the global "request" object with from flask import request

• Reply • Share ›

Ashutosh Jain Kata • 3 years ago

is it "request" or "Request" that we need to import?

• Reply • Share ›

Kata Ashutosh Jain • 3 years ago

I don't remember writing this, but I wrote explicitly "from flask import request". So it's "request" not "Request".

• Reply • Share ›

AlexCaranha • 4 years ago

Very very good! Great tutorial.

• Reply • Share ›

Krys Allen • 4 years ago

I have this working from localhost, but I cant access from an external device, I just get connection refused.

• Reply • Share ›

Phil Krys Allen • 4 years ago

app.run(host="0.0.0.0", debug=True)

1 • Reply • Share ›

luisrei Mod Krys Allen • 4 years ago

Check http://flask.pocoo.org/docs... - i'm not sure the normal development mini-server you get with simply running python flask_app.py allows connection from an external IP. Might allow with some config.

• Reply • Share ›

Rafael V. Gonçalves • 4 years ago

Thanks!!! It's help me a lot

• Reply • Share ›

zhkzyth • 5 years ago

love the basic auth~

• Reply • Share ›

João Aparício • 5 years ago

Obrigado!
Deixaste de actualizar o teu blog? O teu RSS tem só 3 artigos...

• Reply • Share ›

luisrei Mod João Aparício • 5 years ago

http://luisrei.com/rss (full blog/tumblr) vs http://blog.luisrei.com/rss (long articles)

• Reply • Share ›

João Aparício luisrei • 5 years ago

Fixe! Já aprendi uma coisa (a existência de backbone.js :P)

• Reply • Share ›

wiesson • 5 years ago

Thanks for the brief introduction!

• Reply • Share ›

Hector Simosa • 5 years ago

Great tutorial. Very clear and consice. Thanks

• Reply • Share ›

Haukur Kristinsson • 5 years ago

Very helpful. Thanks!

• Reply • Share ›

l1905 • 5 years ago

Great。very helpful for me。thanks

• Reply • Share ›

ALSO ON **BLOG.LUISREI.COM**

Client-Side Web Application Development with JavaScript & Backbone.js

2 comments • 6 years ago

luisrei — No :(I guess I should've done that...

Designing a RESTful Web API

4 comments • 6 years ago

Guest — Fantastic resource; no pun intended!

Subscribe Add Disqus to your siteAdd DisqusAdd Disqus' Privacy PolicyPrivacy PolicyPrivacy

RSS
home blog