

# Implementing a RESTful Web API with Python & Flask

**Luis Rei**

<http://blog.luisrei.com>

[me@luisrei.com](mailto:me@luisrei.com)

[@lmrei](#)

[github.com/lrei](https://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](#)

37 Comments

blog.luisrei.com


1

Login

Recommend 34

Share

Sort by Best




Join the discussion...

LOG IN WITH

OR SIGN UP WITH DISQUS ?


Name



Bennarunga • 5 years ago

Great! tutorial many thanks Luis


8 ^ | v • Reply • Share ›



IOpa • 4 years ago

Really simple to understand tutorial. Thank you!


2 ^ | v • Reply • Share ›



userK • a year ago

Straight to the point! +1

1 ^ | v • Reply • Share ›



Kejsarmakten • 5 years ago

Thanks. Nice tutorial!

1 ^ | v • Reply • Share ›