Designing a RESTful API with Python and Flask

May 20 2013

Posted by Miguel Grinberg under Python, Programming, REST, Flask.

In recent years REST (REpresentational State Transfer) has emerged as the standard architectural design for web services and web APIs.

In this article I'm going to show you how easy it is to create a RESTful web service using Python and the Flask microframework.

What is REST?

The characteristics of a REST system are defined by six design rules:

- **Client-Server**: There should be a separation between the server that offers a service, and the client that consumes it.
- **Stateless**: Each request from a client must contain all the information required by the server to carry out the request. In other words, the server cannot store information provided by the client in one request and use it in another request.
- **Cacheable**: The server must indicate to the client if requests can be cached or not.
- **Layered System**: Communication between a client and a server should be standardized in such a way that allows intermediaries to respond to requests instead of the end server, without the client having to do anything different.
- **Uniform Interface**: The method of communication between a client and a server must be uniform.
- **Code on demand**: Servers can provide executable code or scripts for clients to execute in their context. This constraint is the only one that is optional.

What is a RESTful web service?

The REST architecture was originally designed to fit the HTTP protocol that the world wide web uses.

Central to the concept of RESTful web services is the notion of resources. Resources are represented by URIs. The clients send requests to these URIs using the methods defined by the HTTP protocol, and possibly as a result of that the state of the affected resource changes.

The HTTP request methods are typically designed to affect a given resource in standard ways:

HTTP Method	Action	Examples

GET	Obtain information about a resource	http://example.com/api/orders (retrieve order list)
GET	Obtain information about a resource	http://example.com/api/orders/123 (retrieve order #123)
POST	Create a new resource	http://example.com/api/orders (create a new order, from data provided with the request)
PUT	Update a resource	http://example.com/api/orders/123 (update order #123, from data provided with the request)
DELETE	Delete a resource	http://example.com/api/orders/123 (delete order #123)

The REST design does not require a specific format for the data provided with the requests. In general data is provided in the request body as a JSON blob, or sometimes as arguments in the query string portion of the URL.

Designing a simple web service

The task of designing a web service or API that adheres to the REST guidelines then becomes an exercise in identifying the resources that will be exposed and how they will be affected by the different request methods.

Let's say we want to write a To Do List application and we want to design a web service for it. The first thing to do is to decide what is the root URL to access this service. For example, we could expose this service as:

```
http://[hostname]/todo/api/v1.0/
```

Here I have decided to include the name of the application and the version of the API in the URL. Including the application name in the URL is useful to provide a namespace that separates this service from others that can be running on the same system. Including the version in the URL can help with making updates in the future, since new and potentially incompatible functions can be added under a new version, without affecting applications that rely on the older functions.

The next step is to select the resources that will be exposed by this service. This is an extremely simple application, we only have tasks, so our only resource will be the tasks in our to do list.

Our tasks resource will use HTTP methods as follows:

HTTP Method	**URI**	**Action**
GET	http://[hostname]/todo/api/v1.0/tasks	Retrieve list of tasks
GET	http://[hostname]/todo/api/v1.0/tasks/[task_id]	Retrieve a task
POST	http://[hostname]/todo/api/v1.0/tasks	Create a new task
PUT	http://[hostname]/todo/api/v1.0/tasks/[task_id]	Update an existing task
DELETE	http://[hostname]/todo/api/v1.0/tasks/[task_id]	Delete a task

We can define a task as having the following fields:

- **id**: unique identifier for tasks. Numeric type.
- **title**: short task description. String type.
- **description**: long task description. Text type.
- **done**: task completion state. Boolean type.

And with this we are basically done with the design part of our web service. All that is left is to implement it!

A brief introduction to the Flask microframework

If you read my Flask Mega-Tutorial series you know that Flask is a simple, yet very powerful Python web framework.

Before we delve into the specifics of web services let's review how a regular Flask web application is structured.

I will assume you know the basics of working with Python in your platform. The examples I will show below are for a Unix-like operating system. In short, that means that they will work on Linux, Mac OS X and also on Windows if you use Cygwin. The commands are slightly different if you use the Wndows native version of Python.

Let's begin by installing Flask in a virtual environment. If you don't have `virtualenv.py` installed in your system, you can download it from https://pypi.python.org/pypi/virtualenv.

```
$ mkdir todo-api
$ cd todo-api
$ virtualenv.py flask
New python executable in flask/bin/python
Installing setuptools.............................done.
Installing pip...................done.
$ flask/bin/pip install flask
```

Now that we have Flask installed let's create a simple web application, which we will put in a file called `app.py`:

```
#!flask/bin/python
from flask import Flask

app = Flask(__name__)

@app.route('/')
def index():
    return "Hello, World!"

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

To run this application we have to execute `app.py`:

```
$ chmod a+x app.py
$ ./app.py
 * Running on http://127.0.0.1:5000/
 * Restarting with reloader
```

And now you can launch your web browser and type `http://localhost:5000` to see this tiny application in action.

Simple, right? Now we will convert this app into our RESTful service!

Implementing RESTful services in Python and Flask

Building web services with Flask is surprisingly simple, much simpler than building complete server side applications like the one I built in the Mega-Tutorial.

There are a couple of Flask extensions that help with building RESTful services with Flask, but the task is so simple that in my opinion there is no need to use an extension.

The clients of our web service will be asking the service to add, remove and modify tasks, so clearly we need to have a way to store tasks. The obvious way to do that is to build a small database, but because databases are not the topic of this article we are going to take a much simpler approach. To learn about proper use of databases with Flask once again I recommend that you read my Mega-Tutorial.

In place of a database we will store our task list in a memory structure. This will only work when the web server that runs our application is single process and single threaded. This is okay for Flask's own development web server. It is not okay to use this technique on a production web server, for that a proper database setup must be used.

Using the base Flask application we are now ready to implement the first entry point of our web service:

```
#!flask/bin/python
from flask import Flask, jsonify

app = Flask(__name__)

tasks = [
    {
        'id': 1,
        'title': u'Buy groceries',
        'description': u'Milk, Cheese, Pizza, Fruit, Tylenol',
        'done': False
    },
    {
        'id': 2,
        'title': u'Learn Python',
        'description': u'Need to find a good Python tutorial on the web',
        'done': False
    }
]

@app.route('/todo/api/v1.0/tasks', methods = ['GET'])
def get_tasks():
    return jsonify( { 'tasks': tasks } )

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

As you can see, not much has changed. We created a memory database of tasks, which is nothing more than a plain and simple array of dictionaries. Each entry in the array has the fields that we defined above for our tasks.

Instead of the `index` entry point we now have a `get_tasks` function that is associated with the `/todo/api/v1.0/tasks` URI, and only for the `GET` HTTP method.

The response of this function is not text, we are now replying with JSON data, which Flask's `jsonify` function generates for us from our data structure.

Using a web browser to test a web service isn't the best idea since web browsers cannot easily generate all types of HTTP requests. Instead, we will use curl. If you don't have `curl` installed, go ahead and install it now.

Start the web service in the same way we started the sample application, by running `app.py`. Then open a new console window and run the following command:

```
$ curl -i http://localhost:5000/todo/api/v1.0/tasks
HTTP/1.0 200 OK
Content-Type: application/json
Content-Length: 294
Server: Werkzeug/0.8.3 Python/2.7.3
Date: Mon, 20 May 2013 04:53:53 GMT

{
  "tasks": [
    {
      "description": "Milk, Cheese, Pizza, Fruit, Tylenol",
      "done": false,
      "id": 1,
      "title": "Buy groceries"
    },
    {
```

```
      "description": "Need to find a good Python tutorial on the web",
      "done": false,
      "id": 2,
      "title": "Learn Python"
    }
  ]
}
```

We just have invoked a function in our RESTful service!

Now let's write the second version of the GET method for our tasks resource. If you look at the table above this will be the one that is used to return the data of a single task:

```
from flask import abort

@app.route('/todo/api/v1.0/tasks/<int:task_id>', methods = ['GET'])
def get_task(task_id):
    task = filter(lambda t: t['id'] == task_id, tasks)
    if len(task) == 0:
        abort(404)
    return jsonify( { 'task': task[0] } )
```

This second function is a little bit more interesting. Here we get the id of the task in the URL, and Flask translates it into the `task_id` argument that we receive in the function.

With this argument we search our `tasks` array. If the id that we were given does not exist in our database then we return the familiar error code 404, which according to the HTTP specification means "Resource Not Found", which is exactly our case.

If we find the task then we just package it as JSON with `jsonify` and send it as a response, just like we did before for the entire collection.

Here is how this function looks when invoked from `curl`:

```
$ curl -i http://localhost:5000/todo/api/v1.0/tasks/2
HTTP/1.0 200 OK
Content-Type: application/json
Content-Length: 151
Server: Werkzeug/0.8.3 Python/2.7.3
Date: Mon, 20 May 2013 05:21:50 GMT

{
  "task": {
    "description": "Need to find a good Python tutorial on the web",
    "done": false,
    "id": 2,
    "title": "Learn Python"
  }
}
$ curl -i http://localhost:5000/todo/api/v1.0/tasks/3
HTTP/1.0 404 NOT FOUND
Content-Type: text/html
Content-Length: 238
Server: Werkzeug/0.8.3 Python/2.7.3
Date: Mon, 20 May 2013 05:21:52 GMT

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">
<title>404 Not Found</title>
<h1>Not Found</h1>
<p>The requested URL was not found on the server.</p><p>If you     entered the URL manually please check your spelling and try again.</p>
```

When we ask for resource id #2 we get it, but when we ask for #3 we get back the 404 error. The odd thing about the error is that it came back with an HTML message instead of JSON, because that is how Flask generates the 404 response by default. Since this is a web service client applications will expect that we always respond with JSON, so we need to improve our 404

error handler:

```
from flask import make_response

@app.errorhandler(404)
def not_found(error):
    return make_response(jsonify( { 'error': 'Not found' } ), 404)
```

And we get a much more API friendly error response:

```
$ curl -i http://localhost:5000/todo/api/v1.0/tasks/3
HTTP/1.0 404 NOT FOUND
Content-Type: application/json
Content-Length: 26
Server: Werkzeug/0.8.3 Python/2.7.3
Date: Mon, 20 May 2013 05:36:54 GMT

{
  "error": "Not found"
}
```

Next in our list is the `POST` method, which we will use to insert a new item in our task database:

```
from flask import request

@app.route('/todo/api/v1.0/tasks', methods = ['POST'])
def create_task():
    if not request.json or not 'title' in request.json:
        abort(400)
    task = {
        'id': tasks[-1]['id'] + 1,
        'title': request.json['title'],
        'description': request.json.get('description', ""),
        'done': False
    }
    tasks.append(task)
    return jsonify( { 'task': task } ), 201
```

Adding a new task is also pretty easy. The `request.json` will have the request data, but only if it came marked as JSON. If the data isn't there, or if it is there, but we are missing a `title` item then we return an error code 400, which is the code for the bad request.

We then create a new task dictionary, using the id of the last task plus one (a cheap way to guarantee unique ids in our simple database). We tolerate a missing `description` field, and we assume the `done` field will always start set to `False`.

We append the new task to our `tasks` array, and then respond to the client with the added task and send back a status code 201, which HTTP defines as the code for "Created".

To test this new function we can use the following `curl` command:

```
$ curl -i -H "Content-Type: application/json" -X POST -d '{"title":"Read a book"}' http://localhost:5000/todo/api/v1.0/tasks
HTTP/1.0 201 Created
Content-Type: application/json
Content-Length: 104
Server: Werkzeug/0.8.3 Python/2.7.3
Date: Mon, 20 May 2013 05:56:21 GMT

{
  "task": {
    "description": "",
    "done": false,
    "id": 3,
    "title": "Read a book"
```

```
  }
}
```

Note: if you are on Windows and use the Cygwin version of `curl` from `bash` then the above command will work just fine. However, if you are using the native version of `curl` from the regular command prompt there is a little dance that needs to be done to send double quotes inside the body of a request:

```
curl -i -H "Content-Type: application/json" -X POST -d "{""title"":""Read a book""}" http://localhost:5000/todo/api/v1.0/tasks
```

Essentially on Windows you have to use double quotes to enclose the body of the request, and then inside it you escape a double quote by writing three of them in sequence.

Of course after this request completed we can obtain the updated list of tasks:

```
$ curl -i http://localhost:5000/todo/api/v1.0/tasks
HTTP/1.0 200 OK
Content-Type: application/json
Content-Length: 423
Server: Werkzeug/0.8.3 Python/2.7.3
Date: Mon, 20 May 2013 05:57:44 GMT

{
  "tasks": [
    {
      "description": "Milk, Cheese, Pizza, Fruit, Tylenol",
      "done": false,
      "id": 1,
      "title": "Buy groceries"
    },
    {
      "description": "Need to find a good Python tutorial on the web",
      "done": false,
      "id": 2,
      "title": "Learn Python"
    },
    {
      "description": "",
      "done": false,
      "id": 3,
      "title": "Read a book"
    }
  ]
}
```

The remaining two functions of our web service are shown below:

```
@app.route('/todo/api/v1.0/tasks/<int:task_id>', methods = ['PUT'])
def update_task(task_id):
    task = filter(lambda t: t['id'] == task_id, tasks)
    if len(task) == 0:
        abort(404)
    if not request.json:
        abort(400)
    if 'title' in request.json and type(request.json['title']) != unicode:
        abort(400)
    if 'description' in request.json and type(request.json['description']) is not unicode:
        abort(400)
    if 'done' in request.json and type(request.json['done']) is not bool:
        abort(400)
    task[0]['title'] = request.json.get('title', task[0]['title'])
    task[0]['description'] = request.json.get('description', task[0]['description'])
    task[0]['done'] = request.json.get('done', task[0]['done'])
    return jsonify( { 'task': task[0] } )

@app.route('/todo/api/v1.0/tasks/<int:task_id>', methods = ['DELETE'])
def delete_task(task_id):
    task = filter(lambda t: t['id'] == task_id, tasks)
```

```
    if len(task) == 0:
        abort(404)
    tasks.remove(task[0])
    return jsonify( { 'result': True } )
```

The `delete_task` function should have no surprises. For the `update_task` function we are trying to prevent bugs by doing exhaustive checking of the input arguments. We need to make sure that anything that the client provided us is in the expected format before we incorporate it into our database.

A function call that updates task #2 as being done would be done as follows:

```
$ curl -i -H "Content-Type: application/json" -X PUT -d '{"done":true}' http://localhost:5000/todo/api/v1.0/tasks/2
HTTP/1.0 200 OK
Content-Type: application/json
Content-Length: 170
Server: Werkzeug/0.8.3 Python/2.7.3
Date: Mon, 20 May 2013 07:10:16 GMT

{
  "task": [
    {
      "description": "Need to find a good Python tutorial on the web",
      "done": true,
      "id": 2,
      "title": "Learn Python"
    }
  ]
}
```

Improving the web service interface

The article was originally published without this section, but a reader suggested a small improvement that I thought is interesting enough to share.

The problem he observed with my design is that clients are forced to construct URIs from the task identifiers that are returned. This is pretty easy in itself, but it indirectly forces clients to know how these URIs need to be built, and this will prevent us from making changes to URIs in the future.

Instead of returning task ids we can return the full URI that controls the task. For this we can write a small helper function that generates a "public" version of a task to send to the client:

```
from flask import url_for

def make_public_task(task):
    new_task = {}
    for field in task:
        if field == 'id':
            new_task['uri'] = url_for('get_task', task_id = task['id'], _external = True)
        else:
            new_task[field] = task[field]
    return new_task
```

All we are doing here is taking a task from our database and creating a new task that has all the fields except `id`, which gets replaced with another field called `uri`, generated with Flask's `url_for`.

When we return the list of tasks we pass them through this function before sending them to the client:

```
@app.route('/todo/api/v1.0/tasks', methods = ['GET'])
def get_tasks():
    return jsonify( { 'tasks': map(make_public_task, tasks) } )
```

So now this is what the client gets when it retrieves the list of tasks:

```
$ curl -i http://localhost:5000/todo/api/v1.0/tasks
HTTP/1.0 200 OK
Content-Type: application/json
Content-Length: 406
Server: Werkzeug/0.8.3 Python/2.7.3
Date: Mon, 20 May 2013 18:16:28 GMT

{
  "tasks": [
    {
      "title": "Buy groceries",
      "done": false,
      "description": "Milk, Cheese, Pizza, Fruit, Tylenol",
      "uri": "http://localhost:5000/todo/api/v1.0/tasks/1"
    },
    {
      "title": "Learn Python",
      "done": false,
      "description": "Need to find a good Python tutorial on the web",
      "uri": "http://localhost:5000/todo/api/v1.0/tasks/2"
    }
  ]
}
```

We apply this technique to all the other functions and with this we ensure that the client always sees URIs instead of ids.

Securing a RESTful web service

Can you believe we are done? Well, we are done with the functionality of our service, but we still have a problem. Our service is open to anybody, and that is a bad thing.

We have a complete web service that can manage our to do list, but the service in its current state is open to any clients. If a stranger figures out how our API works he or she can write a new client that can access our service and mess with our data.

Most entry level tutorials ignore security and stop here. In my opinion this is a serious problem that should always be addressed.

The easiest way to secure our web service is to require clients to provide a username and a password. In a regular web application you would have a login form that posts the credentials, and at that point the server would create a session for the logged in user to continue working, with the session id stored in a cookie in the client browser. Unfortunately doing that here would violate the stateless requirement of REST, so instead we have to ask clients to send their authentication information with every request they send to us.

With REST we always try to adhere to the HTTP protocol as much as we can. Now that we need to implement authentication we should do so in the context of HTTP, which provides two forms of authentication called Basic and Digest.

There is a small Flask extension that can help with this, written by no other than yours truly. So let's go ahead and install Flask-HTTPAuth:

```
$ flask/bin/pip install flask-httpauth
```

Let's say we want our web service to only be accessible to username `miguel` and password `python`. We can setup a Basic HTTP authentication as follows:

```
from flask.ext.httpauth import HTTPBasicAuth
auth = HTTPBasicAuth()

@auth.get_password
def get_password(username):
```

```
    if username == 'miguel':
        return 'python'
    return None

@auth.error_handler
def unauthorized():
    return make_response(jsonify( { 'error': 'Unauthorized access' } ), 401)
```

The `get_password` function is a callback function that the extension will use to obtain the password for a given user. In a more complex system this function could check a user database, but in this case we just have a single user so there is no need for that.

The `error_handler` callback will be used by the extension when it needs to send the unauthorized error code back to the client. Like we did with other error codes, here we customize the response so that is contains JSON instead of HTML.

With the authentication system setup, all that is left is to indicate which functions need to be protected, by adding the `@auth.login_required` decorator. For example:

```
@app.route('/todo/api/v1.0/tasks', methods = ['GET'])
@auth.login_required
def get_tasks():
    return jsonify( { 'tasks': tasks } )
```

If we now try to invoke this function with `curl` this is what we get:

```
$ curl -i http://localhost:5000/todo/api/v1.0/tasks
HTTP/1.0 401 UNAUTHORIZED
Content-Type: application/json
Content-Length: 36
WWW-Authenticate: Basic realm="Authentication Required"
Server: Werkzeug/0.8.3 Python/2.7.3
Date: Mon, 20 May 2013 06:41:14 GMT

{
  "error": "Unauthorized access"
}
```

To be able to invoke this function we have to send our credentials:

```
$ curl -u miguel:python -i http://localhost:5000/todo/api/v1.0/tasks
HTTP/1.0 200 OK
Content-Type: application/json
Content-Length: 316
Server: Werkzeug/0.8.3 Python/2.7.3
Date: Mon, 20 May 2013 06:46:45 GMT

{
  "tasks": [
    {
      "title": "Buy groceries",
      "done": false,
      "description": "Milk, Cheese, Pizza, Fruit, Tylenol",
      "uri": "http://localhost:5000/todo/api/v1.0/tasks/1"
    },
    {
      "title": "Learn Python",
      "done": false,
      "description": "Need to find a good Python tutorial on the web",
      "uri": "http://localhost:5000/todo/api/v1.0/tasks/2"
    }
  ]
}
```

The authentication extension gives us the freedom to choose which functions in the service are open and which are protected.

To ensure the login information is secure the web service should be exposed in a HTTP Secure server (i.e. https://...) as this encrypts all the communications between client and server and prevents a third party from seeing the authentication credentials in transit.

Unfortunately web browsers have the nasty habit of showing an ugly login dialog box when a request comes back with a 401 error code. This happens even for background requests, so if we were to implement a web browser client with our current web server we would need to jump through hoops to prevent browsers from showing their authentication dialogs and let our client application handle the login.

A simple trick to distract web browsers is to return an error code other than 401. An alternative error code favored by many is 403, which is the "Forbidden" error. While this is a close enough error, it sort of violates the HTTP standard, so it is not the proper thing to do if full compliance is necessary. In particular this would be a bad idea if the client application is not a web browser. But for cases where server and client are developed together it saves a lot of trouble. The simple change that we can make to implement this trick is to replace the 401 with a 403:

```
@auth.error_handler
def unauthorized():
    return make_response(jsonify( { 'error': 'Unauthorized access' } ), 403)
```

Of course if we do this we will need the client application to look for 403 errors as well.

Possible improvements

There are a number of ways in which this little web service we have built today can be improved.

For starters, a real web service should be backed by a real database. The memory data structure that we are using is very limited in functionality and should not be used for a real application.

Another area in which an improvement could be made is in handling multiple users. If the system supports multiple users the authentication credentials sent by the client could be used to obtain user specific to do lists. In such a system we would have a second resource, which would be the users. A `POST` request on the users resource would represent a new user registering for the service. A `GET` request would return user information back to the client. A `PUT` request would update the user information, maybe updating an email address. A `DELETE` request would delete the user account.

The `GET` request that retrieves the task list could be expanded in a couple of ways. First, this request could take optional pagination arguments, so that a client can request a portion of the list. Another way to make this function more useful would be to allow filtering by certain criteria. For example, a client might want to see only completed tasks, or only tasks with a title that begins with the letter A. All these elements can be added to the URL as arguments.

Conclusion

The complete code for the To Do List web service is here: https://gist.github.com/miguelgrinberg/5614326.

I hope this was a simple and friendly introduction to RESTful APIs. If there is enough interest I could write a second part to this article in which we can develop a simple web client that uses this service for a complete To Do List application. Let me know what you think below in the comments!

UPDATE: A follow up to this tutorial is now online: Writing a Javascript REST client.

UPDATE #2: I have written yet another follow-up: Designing a RESTful API using Flask-RESTful.

Miguel

46 comments

#1 Jason said 10 months ago:

Thanks for taking the time to share this. I'm going to go back and read the Mega tutorials before I give this a try. I'm new to Python and not familiar at all with Flask. I'm looking to build off of what I'm learning and this a a great way to do so. Thank again, this is awesome.

#2 Michael Tiller said 10 months ago:

Overall, I think you've done a nice job putting together this material. My one beef is that it perpetuates an unfortunately common pattern. You constantly refer to an 'id' when it would be much better to simply use a URI. The issue is that any client working with your API will have to know how to construct URLs for requests they want to make. This is completely unnecessary and brittle. I'm referring to the so-called "HATEOAS" approach here. Using that discipline, your ids would be replaced by URIs and your POST methods would return status code 201 (CREATED) and provide the URI of the resource created in the "Location" header.

#3 Miguel Grinberg said 10 months ago:

@Michael: I don't fully agree with the self-discovery ideas of HATEOAS. The proposers of this make it sound like a client should be allowed to explore your web service and treat it like a user clicking through links in an HTML would. For an API that is not a good idea, in my opinion. APIs serve a specific purpose and should be documented accordingly. That said, your suggestions are valid ones, preventing the client from having to construct URIs is a good goal, so I'm updating the article to show how you would do that. Thanks!

#4 vannen said 10 months ago:

Nice tutorial miguel! I use MethodView class from Flask to build my API, in my opinion it is more structured.

http://flask.pocoo.org/docs/views/#method-based-dispatching

#5 Evan said 9 months ago:

Thanks for another great tutorial. The mega-tutorial got me up and running with flask, and this one is nice as well. I appreciate the time you put into sharing knowledge.

#6 Leandro Guerra said 9 months ago:

Nice tutorial miguel! Could you help me please? Im trying to use the POST method, which we will use to insert a new item in our task database, but I got bach this error: Bad Request The browser(or proxy) sent a request that this server could not understand. I dont know how to fix it.

Thank you a lot.

#7 Miguel Grinberg said 9 months ago:

@Leandro: You need to look at your server code and figure out why this POST route returns bad request. It's probably validating your request and finding it is invalid in some way.

#8 Leandro Guerra said 9 months ago:

Nice Miguel, tks again.

#9 Brent said 9 months ago:

Excellent article! Thanks Miguel

#10 JayKim said 9 months ago:

I am developing REST api that require authorization. I am going to use flask-HTTPAuth with HTTPS. Do I request api call with ID and password for authorization? If I have to save ID and password, it is so dangerous. how can I do?

- ««
- «
- »»
- »

Leave a Comment

Name *

Website

Email *

Comment *

Type the two words

Privacy & Terms

Post Comment Note: all comments are screened before they are published. Thank you for your patience!

About Miguel

Welcome to my blog!

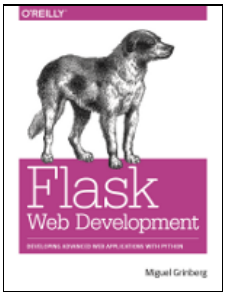
I'm a software engineer, photographer and filmmaker in Portland, Oregon, USA.

You can also find me on Facebook, Google+, LinkedIn and Twitter.

Thank you for visiting!

Flask Web Development Book

I'm currently writing a book on web development with Python and Flask, to be published by O'Reilly Media in 2014. The official site for this book is http://flaskbook.com.



Flask Tutorial at PyCon 2014

If you want to learn Flask in a class setting then I hope you will consider my upcoming tutorial session at PyCon 2014 in Montréal on April 10th, 2014. Visit the tutorial page for more information. I hope to see you there!

Categories

- Arduino (7)
- Blog (1)
- C++ (5)
- Cloud (1)
- Filmmaking (6)
- Flask (28)
- Games (1)
- Gear (6)
- HTML5 (1)
- Heroku (1)
- Javascript (3)
- Movie Reviews (5)
- Netflix (5)
- Node.js (1)
- Personal (1)
- Photography (6)
- Product Reviews (1)
- Programming (36)
- Project Management (1)
- Python (29)
- REST (4)
- Raspberry Pi (4)
- RhinoSteady (6)
- Robotics (6)
- Video (2)

This page was generated in 0.575 seconds.