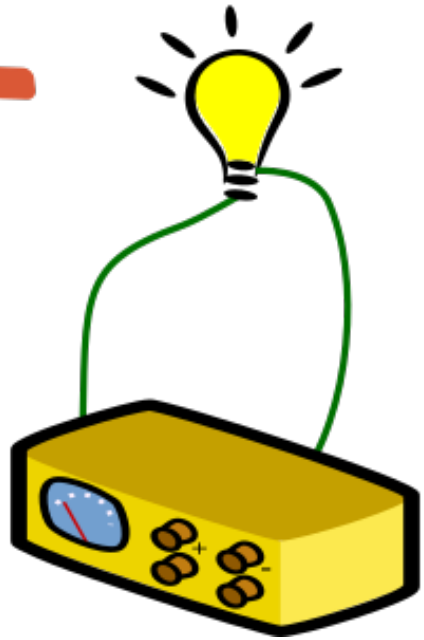
My Restful Lab

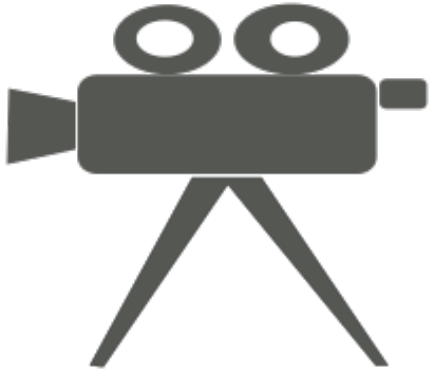
Hardware control through a REST API

Federico Ariza

December 4, 2016

My hardware



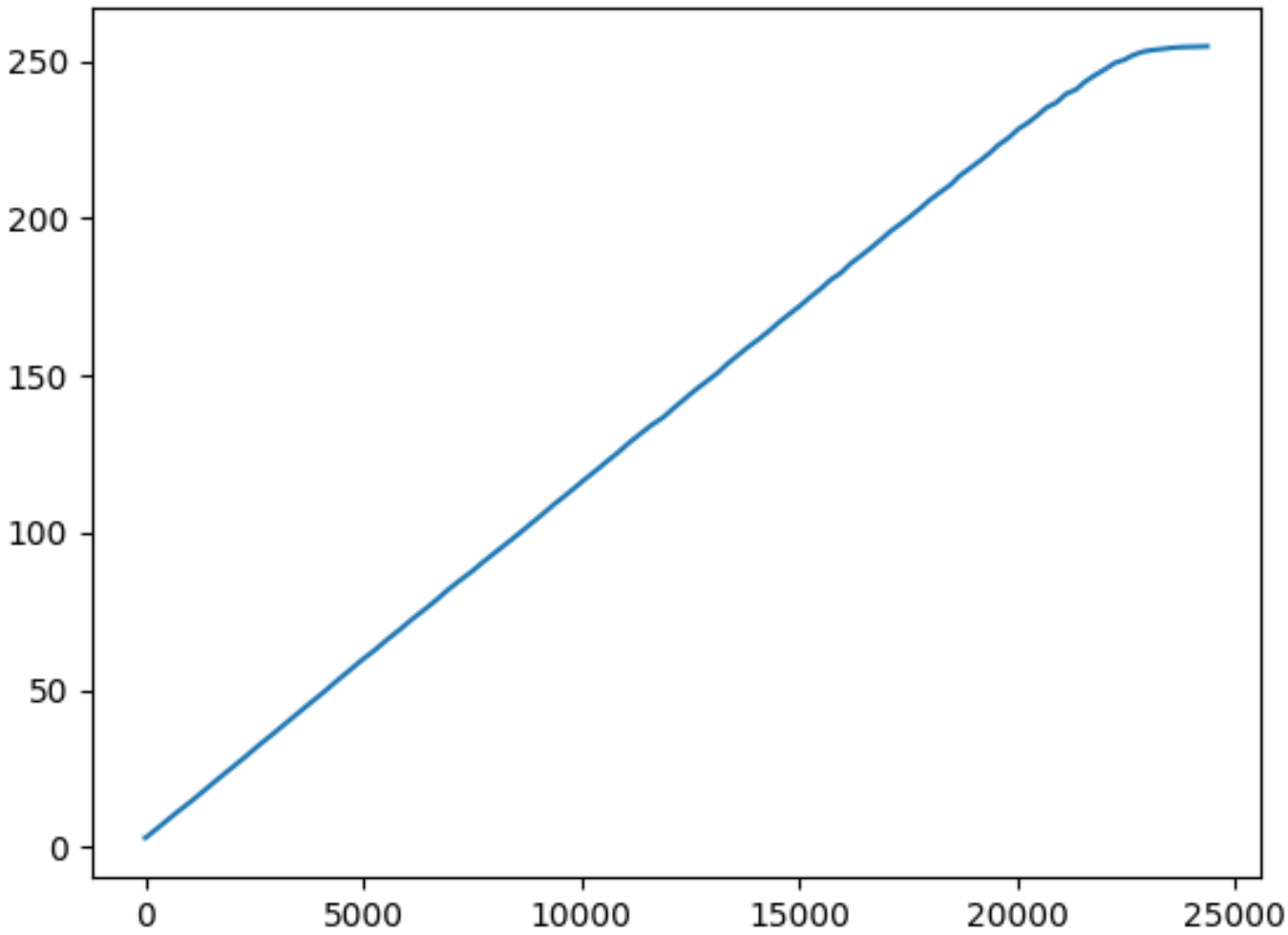


Typical test

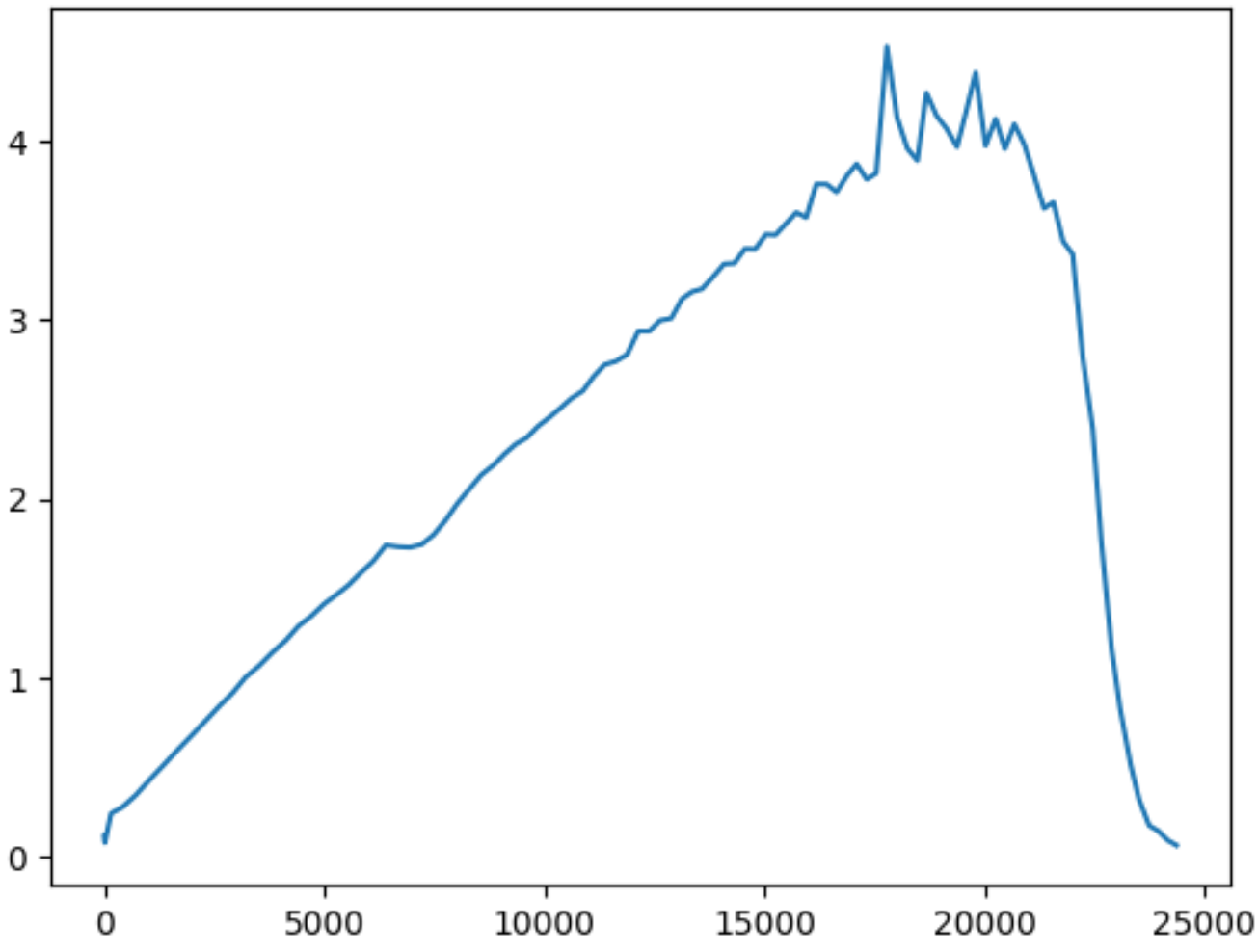
Simple test

```
import PSU, Radiometer, Camera
import numpy as np

psu = PSU()
rad = Radiometer()
cam = Camera()

x = []
y1 = []
y2 = []
for current in np.linspace(0, 0.35, 100):
    psu.current = current
    x.append(rad.radiance)
    img = camera.grab()
    y1.append(img.mean())
    y2.append(img.std())


# save into database
# process data
#....
```

Typical test



Typical test



Power Supply (PSU)

Basic PSU

```
class PSU:
    def __init__(self):
        self._voltage = 0
        self._on = False

    @property
    def voltage(self):
        return self._voltage * self._on

    @voltage.setter
    def voltage(self, value):
        self._voltage = value

    def turn_on(self):
        self._on = True
```

Direct API

Nice

- Natural interface
- It's my interface of choice
- It's my language of choice

Direct API

Nice

- Natural interface
- It's my interface of choice
- It's my language of choice

Not so nice

- User has to be phisically connected to device
- That guy over there wants to use C/lisp?

Remote API

Can I "remotisize" my API?

- Is my API simple enough?
- All methods inputs/outputs can be serialized (to strings)?

Remote API

Can I "remotisize" my API?

- Is my API simple enough?
- All methods inputs/outputs can be serialized (to strings)?

What are my options?

- Handmade socket protocol
- RPC
- ...
- REST

My choice REST

Rest basics

URL: **http://hostname:port/zzz/X1/X2/X3**

Rest basics

URL: **http://hostname:port/zzz/X1/X2/X3**

zzz: Base path (webserver config)

Rest basics

URL: **http://hostname:port/zzz/X1/X2/X3**

Can have verbs Associated to:

- : X1
- : X1/X2
- : X1/X2/X3

Rest basics

URL: **http://hostname:port/zzz/X1/X2/X3**

Can be variables

- X1
- X2
- X3

Rest basics

URL: **http://hostname:port/zzz/X1/X2/X3**

- Read current: .../current
- Read max current: .../psu/current/max
- Set current of 5th psu: ../psu/5/current/3.5

Rest basics

URL: **http://hostname:port/zzz/X1/X2/X3**

Pass Data

- Variable .../current/3.5
- Querystring: ?var1=xx&var2=yy...
- Data inside body: $data = \{var1 : x1, var2 : x2\}$

Rest basics

URL: **http://hostname:port/zzz/X1/X2/X3**

Verbs

- GET: Read
- POST: Create
- PUT: Update/Replace
- PATCH: Update/Modify
- DELETE: Delete

Rest basics

URL: **http://hostname:port/zzz/X1/X2/X3**

Return Codes

- 200: OK
- 201: Created
- 404: Not Found
- 409: Conflict

My REST API

- GET(http://hostname/voltage)
  $\rightarrow$ psu.voltage

My REST API

- GET(http://hostname/voltage)
  $\rightarrow$ psu.voltage
- PUT(http://hostname/voltage, {'data': value})
  $\rightarrow$ psu.voltage=value

My REST API

- GET(http://hostname/voltage)
  $\rightarrow$ psu.voltage
- PUT(http://hostname/voltage, {'data': value})
  $\rightarrow$ psu.voltage=value
- PUT(http://hostname/turn_on, {})
  $\rightarrow$ psu.turn_on

Rest interface for PSU

Flask boilerplate

```
from psu1 import PSU
from flask import Flask, request
from flask_restful import Resource, Api

psu = PSU()

app = Flask(__name__)
api = Api(app)
```

Rest interface for PSU

REST

```
class Voltage(Resource):
    def get(self):
        return psu.voltage

    def put(self):
        psu.voltage = request.form['data']
api.add_resource(Voltage, '/voltage')


class TurnOn(Resource):
    def put(self):
        psu.turn_on()
api.add_resource(TurnOn, '/turn_on')

if __name__ == "__main__":
    app.run(use_reloader=True, debug=True)
```

Rest client

Bash

```
#!/bin/bash
export no_proxy='127.0.0.1'
curl http://127.0.0.1:5000/voltage
curl http://127.0.0.1:5000/turn_on -d "data=" -X PUT
curl http://127.0.0.1:5000/voltage -d "data=4" -X PUT
curl http://127.0.0.1:5000/voltage
```

Rest client

C++

```
#include "restclient-cpp/connection.h"
#include "restclient-cpp/restclient.h"

string BasePSU::get(string url_attr){
  RestClient::Response r = conn->get(url_attr);
  return r.body;
}

string BasePSU::put(string url_attr, string payload){
  RestClient::Response r = conn->put(url_attr, payload);
  return r.body;
}

float BasePSU::get_voltage(){
   string url_attr ("voltage/");
   string r = get(url_attr);
  return atof(r.c_str());
}

void BasePSU::set_voltage(float voltage){
 string url_attr ("voltage/");
 string r = put(url_attr,"{\"voltage\":" + to_string(voltage) + "}");
}
```

Rest client

Python

```
import os
os.environ['no_proxy'] = 'localhost,127.0.0.1'
from requests import get, put


class PSU:
    def __init__(self, url='http://localhost:5000/'):
        self._url = url

    @property
    def voltage(self):
        return get(self._url + 'voltage').json()

    @voltage.setter
    def voltage(self, value):
        put(self._url + 'voltage', data={'data': value})

    def turn_on(self):
        put(self._url + 'turn_on')
```

Nice

- Controller and user can be separated
- Natural interface
- Remote and local interface are the same
- User choice of language

Nice

- Controller and user can be separated
- Natural interface
- Remote and local interface are the same
- User choice of language

Not so much

- THREE (or more)! times the same code

DRY and SSOT

- Implementation includes extra info (metadata)
- Description auto-extracted from Implementation
- Server auto-generates from Description
- Client auto-generates from Server

Add some metadata

PSU with extra info

```
class PSU:
    def __init__(self):
        self._voltage = 0
        self._on = False

    @property
    def voltage(self) -> float:
        return self._voltage * self._on

    @voltage.setter
    def voltage(self, value: float):
        self._voltage = value

    def turn_on(self):
        self._on = True
```

Use the metadata

Get api from code

```
def get_api(cls):
    api = {'set': {},
           'get': {},
           'methods': {}
           }

    for name in dir(cls):
        if name.startswith('_'):
            continue

        attr = getattr(cls, name)
        if isinstance(attr, property):
            if attr.fget:
                api['get'][name] = attr.fget.__annotations__['return']
            if attr.fset:
                api['set'][name] = attr.fset.__annotations__['value']
        else:
            api['methods'][name] = {
                    'args': [],
                    'kwargs': {},
                    'return': attr.__annotations__.get('return')}
    return api
```

Auto server

Flask boilerplate

```
from psu2 import PSU
from flask import Flask, request
from flask_restful import Resource, Api
from extractor import get_api
import json

psu = PSU()
d_api = get_api(PSU)
app = Flask(__name__)
api = Api(app)


class TypeAwareJSONEncoder(json.JSONEncoder):
    def default(self, obj):
        try:
            r = json.JSONEncoder.default(self, obj)
        except TypeError:
            r = obj.__name__
        return r
```

Auto server

Rest resources

```
class DApi(Resource):
    def get(self):
        return json.dumps(d_api, cls=TypeAwareJSONEncoder)
api.add_resource(DApi, '/api')


class RestPSU(Resource):
    def get(self, attr):
        return getattr(psu, attr)

    def put(self, attr):
        if attr in d_api['set']:
            value = d_api['set'][attr](request.form['data'])
            setattr(psu, attr, value)
        elif attr in d_api['methods']:
            getattr(psu, attr)()

api.add_resource(RestPSU, '/<string:attr>')


if __name__ == "__main__":
    app.run(use_reloader=True, debug=True)
```

Auto client

Init class

```
import os
import json
os.environ['no_proxy'] = 'localhost,127.0.0.1'
from requests import get, put


class PSU:
    def __init__(self, url='http://localhost:5000/'):
        self._url = url
        self._api = json.loads(self._get('api'))
        self._add_properties(self._api)
        self._add_methods(self._api)

    def _get(self, attr):
        return get(self._url + attr).json()

    def _put(self, attr, value):
        put(self._url + attr, data={'data': value})

    def _call(self, attr, *args, **kwargs):
        return put(self._url + attr, data={'args': args,
                                           'kwargs': kwargs})
```

Auto client

Auto attributes and methods

```
@classmethod
def _add_properties(cls, api):
    for attr in api['get'].keys():
        def fget(self):
            return self._get(attr)

        def fset(self, value):
            self._put(attr, value)
        setattr(cls, attr, property(fget, fset))

@classmethod
def _add_methods(cls, api):
    for attr in api['methods'].keys():
        def method(self):
            self._call(attr)
        setattr(cls, attr, method)
```

What about C/C++ client?

Piece of cake

Just call jinja!

Thanks!