Jaime Gil de Sagredo

Software development, DevOps, Python, JavaScript, BDD, ...

A python RESTful API consumer

by *jaimegil* on *December 26, 2012*

Web APIs, and more particularly RESTful APIs, have become very popular in the last few years by the hand of large sites like Fakebook, Twitter or Github, who give developers the opportunity to extend their services with a wide variety of applications and services.

This huge growth in the development and use of REST APIs has been mainly because of the ease of consuming data from different services to create applications, tools or whatever you imagine, such as another APIs! on nearly any device, operating system or programming language. And what has made this possible? The answer is the HTTP protocol.

Consuming REST APIs

Right, let me show you how easy is to start playing with, for example, the Github API, using Python and the awesome Requests http library.

```
import requests

response = requests.get('https://api.github.com/users/jaimegildesagredo/repos')
```

```
    assert response.status_code == 200

    for repo in response.json():
        print '[{}] {}'.format(repo['language'], repo['name'])
```

gistfile1.py hosted with by GitHub view raw

It's so easy! We only have needed to make a GET request to the user repositories endpoint and wait for an OK response to display a list of repository names and programming languages.

We also could add a new repository to the list making an authenticated POST request to the repositories endpoint and including some data.

```
import json
import requests

response = requests.post('https://api.github.com/user/repos',
    data=json.dumps({'name': 'foo'}), auth=('user', 'pass'))

assert response.status_code == 201
```

gistfile1.py hosted with by GitHub view raw

These two previous examples shown very simple use cases, but serve to show the simplicity of consuming REST APIs.

But unfortunately, in the real world, client applications tend to become more and more complex and begins to be necessary to write some boilerplate code for stuff like *prepare requests*, *validate data* and *parse responses*.

Introducing Finch

Finch is an asynchronous RESTful API consumer I'm developing for Python.

The idea is to develop a general purpose, asynchronous http API consumer, specially focused on remove all of this boilerplate code and provide a high level abstraction layer on top of any API.

Metadata driven clients

In general, a REST API client can be divided into two different parts: resources definition and http related stuff. This separation let us put http code in a high level abstraction and resources details in application metadata.

> *"Put Abstractions in Code, Details in Metadata"* *The Pragmatic Programmer*

See the Finch code example below to understand what I'm talking about.

```
from finch import Model, Collection, StringField

class Repo(Model):
    name = StringField()
    language = StringField()



class Repos(Collection):
    url = 'https://api.github.com/users/jaimegildesagredo/repos'
    model = Repo

repos = Repos()

for repo in repos:
    print '[{}] {}'.format(repo.language, repo.name)
```

gistfile1.py hosted with by GitHub view raw

Here, we have only described the `Repo` model and `Repos` collection. All of this code is only *metadata*, declaratively defined using Python, that Finch will use to perform all the operations needed to get a list of repositories from Github.

I think this is really interesting because you can dedicate exclusively to define the peculiarities of the API that you are going to consume and your business logic, and leave Finch to do all the repetitive work.

And do it asynchronously

The last example shows you Finch working synchronously. That's fine when your application does not make an intensive use of API request, like in our example.

But imagine you have to build a highly scalable service that combines several services behind an unique API. You should not have to wait for each response to perform the next request. Here is where asynchronous programming comes into play.

To allow asynchronous requests, Finch will be initially built on top of the Tornadoasynchronous HTTP Client. This way you could do something like `repos.all(callback)`.

Let's see the example where we added a new Github repo, but now using Finch asynchronously.

```
    def on_repo(repo, error):
        if error:
            raise error

        print 'Repo "{}" added'.format(repo.name)

    foo = Repo(name=u'foo')

    repos.add(foo, on_repo)

    # Do something here
```

gistfile1.py hosted with by GitHub view raw

Sounds good, I want to start using it!

Sorry, not yet. Finch is still under active development. Although the examples shown here already work, there are some things I want to finish before release the first public version. For this first release I want to provide at least complete CRUD support, resources definition, authentication and asynchronous requests. I think I can have this version for the beginning of 2013.

And finally, if you are interested in the project you can track progress on the Github repo, on this blog or through my Twitter account. And of course, I would like to know your thoughts ;)

Discussion