restkit Documentation

Release 4.2.2

Benoît Chesneau

Jul 14, 2017

Contents

Restkit is an HTTP resource kit for Python. It allows you to easily access to HTTP resource and build objects around it. It's the base of couchdbkit a Python CouchDB framework.

You can simply use *restkit.request()* function to do any HTTP requests.

Usage example, get a friendpaste paste:

```
>>> from restkit import request
>>> r = request('http://friendpaste.com/1ZSEoJeOarc3ULexzWOk5Y_633433316631/raw')
>>> r.body_string()
'welcome to friendpaste'
>>> r.headers
{'status': '200 OK', 'transfer-encoding': 'chunked', 'set-cookie':
'FRIENDPASTE_SID=b581975029d689119d3e89416e4c2f6480a65d96; expires=Sun,
14-Mar-2010 03:29:31 GMT; Max-Age=1209600; Path=/', 'server': 'nginx/0.7.62',
'connection': 'keep-alive', 'date': 'Sun, 28 Feb 2010 03:29:31 GMT',
'content-type': 'text/plain'}
```

of from a resource:

```
>>> from restkit import Resource
>>> res = Resource('http://friendpaste.com')
>>> r = res.get('/1ZSEoJeOarc3ULexzWOk5Y_633433316631/raw')
>>> r.body_string()
'welcome to friendpaste'
```

but you can do more like building object mapping HTTP resources, ...

Note: restkit source code is hosted on Github

CHAPTER 1

Features

- Full compatible HTTP client for HTTP 1.0 and 1.1
- Threadsafe
- Use pure socket calls and its own HTTP parser (It's not based on httplib or urllib2)
- Map HTTP resources to Python objects
- **Read** and **Send** on the fly
- Reuses connections
- Eventlet and Gevent support
- Support Chunked transfer encoding in both ways.
- Support Basic Authentification and OAuth.
- Multipart forms and url-encoded forms
- Streaming support
- Proxy handling
- HTTP Filters, you can hook requests in responses with your own callback
- Compatible with Python 2.x (>= 2.6)

CHAPTER 2

Content

Installation

Requirements

- **Python 2.6 or newer** (Python 3.x will be supported soon)
- setuptools >= 0.6c6
- nosetests (for the test suite only)

Installation

To install restkit using pip you must make sure you have a recent version of distribute installed:

```
$ curl -O http://python-distribute.org/distribute_setup.py
$ sudo python distribute_setup.py
$ easy_install pip
```

To install or upgrade to the latest released version of restkit:

```
$ pip install -r requirements.txt
$ pip install restkit
```

Note: if you get an error on MacOSX try to install with the following arguments:

```
$ env ARCHFLAGS="-arch i386 -arch x86_64" pip install http-parser
```

Installation from source

You can install Restkit from source as simply as you would install any other Python package. Restkit uses setuptools which will automatically fetch all dependencies (including setuptools itself).

Get a Copy

You can download a tarball of the latest sources from GitHub Downloads or fetch them with git:

```
$ git clone git://github.com/benoitc/restkit.git
```

Installation

```
$ python setup.py install
```

Note: If you don't use setuptools or distribute, make sure http-parser is installed first.

Build resource object

Building a resource object is easy using *restkit.resource.Resource* class. You just need too inherit this object and add your methods. Couchdbkit is using restkit to access to CouchDB. A resource object is an Python object associated to an URI. You can use *get*, *post*, *put*, *delete* or *head* method just like you do a request.

Create a simple Twitter Search resource

We use simplejson to handle deserialisation of data.

Here is the snippet:

```
from restkit import Resource

try:
    import simplejson as json
except ImportError:
    import json # py2.6 only

class TwitterSearch(Resource):

    def __init__(self, **kwargs):
        search_url = "http://search.twitter.com"
        super(TwitterSearch, self).__init__(search_url, follow_redirect=True,
                                            max_follow_redirect=10, **kwargs)

    def search(self, query):
        return self.get('search.json', q=query)

    def request(self, *args, **kwargs):
        resp = super(TwitterSearch, self).request(*args, **kwargs)
        return json.loads(resp.body_string())

if __name__ == "__main__":
    s = TwitterSearch()
    print s.search("gunicorn")
```

Reuses connections

Reusing connections is good. Restkit can maintain http connections for you and reuse them if the server allows it. To do that restkit uses the socketpool module

```
from restkit import *
from socketpool import ConnectionPool

pool = ConnectionPool(factory=Connection)

r = request("http://someurl", pool=pool)
```

Note: By default, restkit uses a generic session object that is globally available. You can change its settings by using the **restkit.sesssion.set_session** function.

Restkit also provides a Pool that works with eventlet or gevent.

Example of usage with Gevent:

```
from restkit import *
from socketpool import ConnectionPool

# set a pool with a gevent packend
pool = ConnectionPool(factory=Connection, backend="gevent")
```

Replace **gevent** by **eventlet** for eventlet support.

Authentication

Restkit support for now basic authentication and OAuth. But any other authentication schema can easily be added using http filters.

Basic authentication

Basic authentication is managed by the object **:api:'restkit.filters.BasicAuth'**. It's handled automatically in **:api:'restkit.request'** function and in **:api:'restkit.resource.Resource'** object if *basic_auth_url* property is True.

To use *basic authentication* in a *Resource object* you can do:

```
from restkit import Resource, BasicAuth

auth = BasicAuth("username", "password")
r = Resource("http://friendpaste.com", filters=[auth])
```

Or simply use an authentication url:

```
r = Resource("http://username:password@friendpaste.com")
```

OAuth

Restkit OAuth is based on simplegeo python-oauth2 module So you don't need other installation to use OAuth (you can also simply use **:api:'restkit.oauth2'** module in your applications).

The OAuth filter **:api:'restkit.oauth2.filter.OAuthFilter'** allow you to associate a consumer per resource (path). Initalize Oauth filter with:

```
path, consumer, token, signaturemethod)
```

token and *method signature* are optionnals. Consumer should be an instance of **:api:'restkit.oauth2.Consumer'**, token an instance of **:api:'restkit.oauth2.Token'** signature method an instance of **:api:'oauth2.SignatureMethod'** (**:api:'restkit.oauth2.Token'** is only needed for three-legged requests.

The filter is appleid if the path match. It allows you to maintain different authorization per path. A wildcard at the indicate to the filter to match all path behind.

Example the rule */some/resource/** will match */some/resource/other* and */some/resource/other2*, while the rule */some/resource* will only match the path */some/resource*.

Simple client example:

```
from restkit import OAuthFilter, request
import restkit.oauth2 as oauth

# Create your consumer with the proper key/secret.
consumer = oauth.Consumer(key="your-twitter-consumer-key",
  secret="your-twitter-consumer-secret")

# Request token URL for Twitter.
request_token_url = "http://twitter.com/oauth/request_token"

# Create our filter.
auth = oauth.OAuthFilter('*', consumer)

# The request.
resp = request(request_token_url, filters=[auth])
print resp.body_string()
```

If you want to add OAuth to your *TwitterSearch* resource:

```
# Create your consumer with the proper key/secret.
consumer = oauth.Consumer(key="your-twitter-consumer-key",
  secret="your-twitter-consumer-secret")

# Create our filter.
client = oauth.OAuthfilter('*', consumer)

s = TwitterSearch(filters=[client])
```

Twitter Three-legged OAuth Example:

Below is an example from python-oauth2 of how one would go through a three-legged OAuth flow to gain access to protected resources on Twitter. This is a simple CLI script, but can be easily translated to a web application:

```
import urlparse

from restkit import request
from restkit.filters import OAuthFilter
import restkit.util.oauth2 as oauth

consumer_key = 'my_key_from_twitter'
consumer_secret = 'my_secret_from_twitter'

request_token_url = 'http://twitter.com/oauth/request_token'
access_token_url = 'http://twitter.com/oauth/access_token'
authorize_url = 'http://twitter.com/oauth/authorize'

consumer = oauth.Consumer(consumer_key, consumer_secret)

auth = OAuthFilter('*', consumer)

# Step 1: Get a request token. This is a temporary token that is used for
# having the user authorize an access token and to sign the request to obtain
# said access token.



resp = request(request_token_url, filters=[auth])
if resp.status_int != 200:
    raise Exception("Invalid response %s." % resp.status_code)

request_token = dict(urlparse.parse_qsl(resp.body_string()))

print "Request Token:"
print "    - oauth_token        = %s" % request_token['oauth_token']
print "    - oauth_token_secret = %s" % request_token['oauth_token_secret']
print

# Step 2: Redirect to the provider. Since this is a CLI script we do not
# redirect. In a web application you would redirect the user to the URL
# below.

print "Go to the following link in your browser:"
print "%s?oauth_token=%s" % (authorize_url, request_token['oauth_token'])
print

# After the user has granted access to you, the consumer, the provider will
# redirect you to whatever URL you have told them to redirect to. You can
# usually define this in the oauth_callback argument as well.
accepted = 'n'
while accepted.lower() == 'n':
    accepted = raw_input('Have you authorized me? (y/n) ')
oauth_verifier = raw_input('What is the PIN? ')

# Step 3: Once the consumer has redirected the user back to the oauth_callback
# URL you can request the access token the user has approved. You use the
# request token to sign this request. After this is done you throw away the
# request token and use the access token returned. You should store this
# access token somewhere safe, like a database, for future use.
token = oauth.Token(request_token['oauth_token'],
    request_token['oauth_token_secret'])
token.set_verifier(oauth_verifier)
```

```
auth = OAuthFilter("*", consumer, token)

resp = request(access_token_url, "POST", filters=[auth])
access_token = dict(urlparse.parse_qsl(resp.body_string()))

print "Access Token:"
print "    - oauth_token        = %s" % access_token['oauth_token']
print "    - oauth_token_secret = %s" % access_token['oauth_token_secret']
print
print "You may now access protected resources using the access tokens above."
print
```

Stream your content

With Restkit you can easily stream your content to and from a server.

Stream to

To stream a content to a server, pass to your request a file (or file-like object) or an iterator as *body*. If you use an iterator or a file-like object and Restkit can't determine its size (by reading *Content-Length* header or fetching the size of the file), sending will be chunked and Restkit add *Transfer-Encoding: chunked* header to the list of headers.

Here is a quick snippet with a file:

```
from restkit import request

with open("/some/file", "r") as f:
  request("/some/url", 'POST', body=f)
```

Here restkit will put the file size in *Content-Length* header. Another example with an iterator:

```
from restkit import request

myiterator = ['line 1', 'line 2']
request("/some/url", 'POST', body=myiterator)
```

Sending will be chunked. If you want to send without TE: chunked, you need to add the *Content-Length* header:

```
request("/some/url", 'POST', body=myiterator,
    headers={'content-Length': 12})
```

Stream from

Each requests return a *restkit.wrappers.Response* object. If you want to receive the content in a streaming fashion you just have to use the *body_stream* member of the response. You can *iter* on it or just use as a file-like object (read, readline, readlines, ...).

Attention: Since 2.0, response.body are just streamed and aren't persistent. In previous version, the implementation may cause problem with memory or storage usage.

Quick snippet with iteration:

```
import os
from restkit import request
import tempfile

r = request("http://e-engura.com/images/logo.gif")
fd, fname = tempfile.mkstemp(suffix='.gif')

with r.body_stream() as body:
  with os.fdopen(fd, "wb") as f:
    for block in body:
        f.write(block)
```

Or if you just want to read:

```
with r.body_stream() as body:
  with os.fdopen(fd, "wb") as f:
    while True:
        data = body.read(1024)
        if not data:
            break
        f.write(data)
```

Tee input

While with body_stream you can only consume the input until the end, you may want to reuse this body later in your application. For that, restkit since the 3.0 version offer the *tee* method. It copy response input to standard output or a file if length > sock.MAX_BODY. When all the input has been read, connection is released:

```
from restkit import request
import tempfile

r = request("http://e-engura.com/images/logo.gif")
fd, fname = tempfile.mkstemp(suffix='.gif')
fd1, fname1 = tempfile.mkstemp(suffix='.gif')

body = t.tee()
# save first file
with os.fdopen(fd, "wb") as f:
   for chunk in body: f.write(chunk)

# reset
body.seek(0)
# save second file.
with os.fdopen(fd1, "wb") as f:
   for chunk in body: f.write(chunk)
```

Usage with Eventlet and Gevent

Restkit can be used with eventlet or gevent and provide specific connection manager to manage iddle connections for them.

Use it with gevent:

Here is a quick crawler example using Gevent:

```
import timeit

# patch python to use replace replace functions and classes with
# cooperative ones
from gevent import monkey; monkey.patch_all()

import gevent
from restkit import *
from socketpool import ConnectionPool

# set a pool with a gevent packend
pool = ConnectionPool(factory=Connection, backend="gevent")

urls = [
        "http://yahoo.fr",
        "http://google.com",
        "http://friendpaste.com",
        "http://benoitc.io",
        "http://couchdb.apache.org"]

allurls = []
for i in range(10):
    allurls.extend(urls)

def fetch(u):
    r = request(u, follow_redirect=True, pool=pool)
    print "RESULT: %s: %s (%s)" % (u, r.status, len(r.body_string()))

def extract():

    jobs = [gevent.spawn(fetch, url) for url in allurls]
    gevent.joinall(jobs)

t = timeit.Timer(stmt=extract)
print "%.2f s" % t.timeit(number=1)
```

You can also set a global pool and use it transparently in your application:

```
from restkit.session import set_session
set_session("gevent")
```

Use it with eventlet:

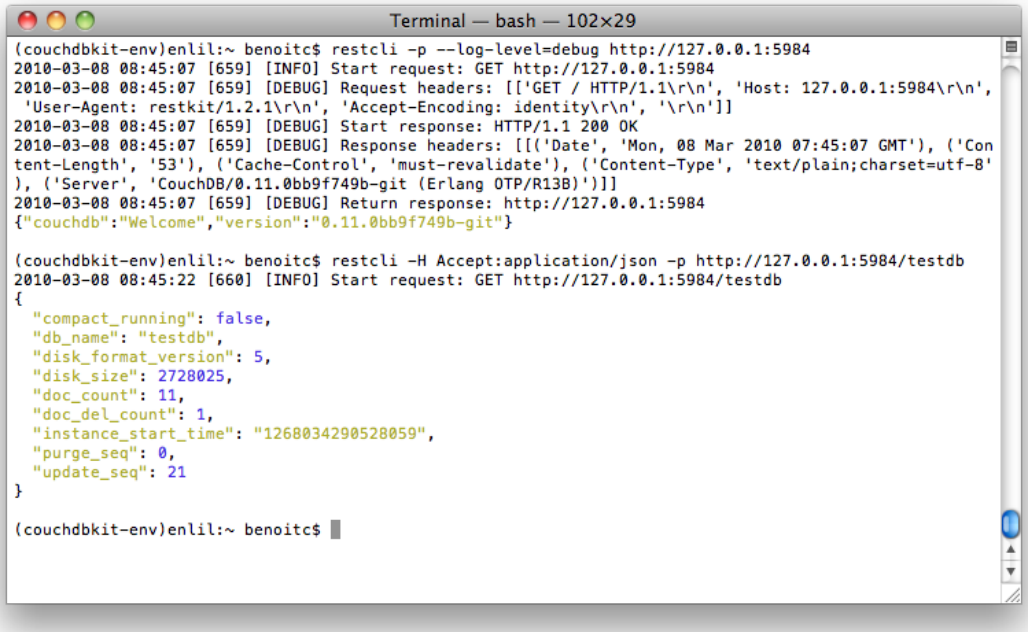
Same exemple as above but using eventlet:

```
import timeit

# patch python
import eventlet
eventlet.monkey_patch()

from restkit import *
from socketpool import ConnectionPool
```

```
# set a pool with a gevent packend
pool = ConnectionPool(factory=Connection, backend="eventlet")

epool = eventlet.GreenPool()

urls = [
        "http://yahoo.fr",
        "http://google.com",
        "http://friendpaste.com",
        "http://benoitc.io",
        "http://couchdb.apache.org"]

allurls = []
for i in range(10):
    allurls.extend(urls)

def fetch(u):
    r = request(u, follow_redirect=True, pool=pool)
    print "RESULT: %s: %s (%s)" % (u, r.status, len(r.body_string()))

def extract():
    for url in allurls:
        epool.spawn_n(fetch, url)
    epool.waitall()

t = timeit.Timer(stmt=extract)
print "%.2f s" % t.timeit(number=1)
```

Command Line

Restkit integrate a simple HTTP client in command line named *restcli* allowing you to perform requests.



Usage:

```
$ restcli --help
Usage: 'restcli [options] url [METHOD] [filename]'

Options:
  -H HEADERS, --header=HEADERS
                        http string header in the form of Key:Value. For
                        example: "Accept: application/json"
  -X METHOD, --request=METHOD
                        http request method
  --follow-redirect
  -S, --server-response
                        print server response
  -p, --prettify        Prettify display
  --log-level=LOG_LEVEL
                        Log level below which to silence messages. [info]
  -i FILE, --input=FILE
                        the name of the file to read from
  -o OUTPUT, --output=OUTPUT
                        the name of the file to write to
  --version             show program's version number and exit
  -h, --help            show this help message and exit
```

To have better prettyfication, make sure you have pygments, tidy and simplejson (or python2.6) installed. They may be already installed on your machine.

restkit shell

restkit come with a IPython based shell to help you to debug your http apps. Just run:

```
$ restkit --shell http://benoitc.github.com/restkit/
```

HTTP Methods

::

```
>>> delete([req|url|path_info])                                    # send a HTTP␣
↪delete
>>> get([req|url|path_info], **query_string)                       # send a HTTP get
>>> head([req|url|path_info], **query_string)                      # send a HTTP head
>>> post([req|url|path_info], [Stream()|**query_string_body])     # send a HTTP post
>>> put([req|url|path_info], stream)                               # send a HTTP put
```

Helpers

```
>>> req    # request to play with. By default http methods will use this one
<Request at 0x18fdb70 GET http://benoitc.github.com/restkit/>

>>> stream # Stream() instance if you specified a -i in command line
None

>>> ctypes # Content-Types helper with headers properties
<ContentTypes(['application_atom_xml', 'application_json',
'application_rss_xml', 'application_xhtml_xml', 'application_xml',
'application_xsl_xml', 'application_xslt_xml', 'image_svg_xml',
'text_html', 'text_xml'])>

restkit shell 1.2.1
1) restcli$
```

Here is a sample session:

```
1) restcli$ req
----------> req()
GET /restkit/ HTTP/1.0
Host: benoitc.github.com
2) restcli$ get()
200 OK
Content-Length: 10476
Accept-Ranges: bytes
Expires: Sat, 03 Apr 2010 12:25:09 GMT
Server: nginx/0.7.61
Last-Modified: Mon, 08 Mar 2010 07:53:16 GMT
Connection: keep-alive
Cache-Control: max-age=86400
Date: Fri, 02 Apr 2010 12:25:09 GMT
Content-Type: text/html
         2) <Response at 0x19333b0 200 OK>
3) restcli$ resp.status
         3) '200 OK'
4) restcli$ put()
405 Not Allowed
Date: Fri, 02 Apr 2010 12:25:28 GMT
Content-Length: 173
```

```
Content-Type: text/html
Connection: keep-alive
Server: nginx/0.7.61

<html>
<head><title>405 Not Allowed</title></head>
<body bgcolor="white">
<center><h1>405 Not Allowed</h1></center>
<hr><center>nginx/0.7.61</center>
</body>
</html>

4) <Response at 0x1933330 405 Not Allowed>
5) restcli$ resp.status
         5) '405 Not Allowed'
6) restcli$ req.path_info = '/restkit/api/index.html'
7) restcli$ get
----------> get()
200 OK
Content-Length: 10476
Accept-Ranges: bytes
Expires: Sat, 03 Apr 2010 12:26:18 GMT
Server: nginx/0.7.61
Last-Modified: Mon, 08 Mar 2010 07:53:16 GMT
Connection: keep-alive
Cache-Control: max-age=86400
Date: Fri, 02 Apr 2010 12:26:18 GMT
Content-Type: text/html
         7) <Response at 0x19300f0 200 OK>
8) restcli$ get('/restkit')
301 Moved Permanently
Location: http://benoitc.github.com/restkit/

<html>
<head><title>301 Moved Permanently</title></head>
<body bgcolor="white">
<center><h1>301 Moved Permanently</h1></center>
<hr><center>nginx/0.7.61</center>
</body>
</html>

8) <Response at 0x1930410 301 Moved Permanently>
9) restcli$ resp.location
9) 'http://benoitc.github.com/restkit/'
```

wsgi_proxy

Restkit version 1.2 introduced a WSGI proxy extension written by Gael Pasgrimaud. This extension proxies WSGI requests to a remote server.

Here is a quick example. You can read full post here .

In this example, we create a simple proxy for CouchDB. We use webob and gunicorn to launch it:

```
import urlparse
```

```
from webob import Request
from restkit.conn import TConnectionManager
from restkit.ext.wsgi_proxy import HostProxy

mgr = TConnectionManager(nb_connections=10)
proxy = HostProxy("http://127.0.0.1:5984", pool=mgr)


def application(environ, start_response):
    req = Request(environ)
    if 'RAW_URI' in req.environ:
        # gunicorn so we can use real path non encoded
        u = urlparse.urlparse(req.environ['RAW_URI'])
        req.environ['PATH_INFO'] = u.path

    # do smth like adding oauth headers ..
    resp = req.get_response(proxy)

    # rewrite response
    # do auth ...
    return resp(environ, start_response)
```

And then launch your application:

```
gunicorn -w 12 -a "egg:gunicorn#eventlet" couchdbproxy:application
```

And access to your couchdb at *http://127.0.0.1:8000*.

You can also use a Paste configuration:

```
[app:proxy]
use = egg:restkit#host_proxy
uri = http://www.example.com/example_db
strip_script_name = false
allowed_methods = get
```

Here is a more advanced example to show how to use the Proxy class to build a distributed proxy. */a/db* will proxify *http://a.mypool.org/db*:

```
import urlparse

from webob import Request
from restkit.conn import TConnectionManager
from restkit.ext.wsgi_proxy import Proxy

mgr = TConnectionManager(nb_connections=10)

proxy = Proxy(pool=mgr, strip_script_name=True)


def application(environ, start_response):
    req = Request(environ).copy()
    req.path_info_pop()
    req.environ['SERVER_NAME'] = '%s.mypool.org:80' % req.script_name.strip('/')
    resp = req.get_response(Proxy)
    return resp(environ, start_response)
```

CHAPTER 3

API

The changelog is available here .

restkit API

restkit Package

restkit.**request**(*url*, *method='GET'*, *body=None*, *headers=None*, ***kwargs*)

Quick shortcut method to pass a request

- **url**: str, url string
- **method**: str, by default GET. http verbs
- **body**: the body, could be a string, an iterator or a file-like object
- **headers**: dict or list of tupple, http headers
- **follow_redirect**: follow redirection, by default False
- **max_follow_redirect**: number of redirections available
- **filters** http filters to pass
- **decompress**: allows the client to decompress the response body
- ** max_status_line_garbage**: defines the maximum number of ignorable lines before we expect a HTTP response's status line. With HTTP/1.1 persistent connections, the problem arises that broken scripts could return a wrong Content-Length (there are more bytes sent than specified). Unfortunately, in some cases, this cannot be detected after the bad response, but only before the next one. So the client is abble to skip bad lines using this limit. 0 disable garbage collection, None means unlimited number of tries.
- **max_header_count**: determines the maximum HTTP header count allowed. by default no limit.
- manager: the manager to use. By default we use the global one.
- **response_class**: the response class to use

- **timeout**: the default timeout of the connection (SO_TIMEOUT)
- **max_tries**: the number of tries before we give up a connection
- **wait_tries**: number of time we wait between each tries.
- **ssl_args**: ssl named arguments, See http://docs.python.org/library/ssl.html informations

restkit.**set_logging**(*level*, *handler=None*)
Set level of logging, and choose where to display/save logs (file or standard output).

client Module

class restkit.client.**Client**(*follow_redirect=False*, *force_follow_redirect=False*, *max_follow_redirect=5*, *filters=None*, *decompress=True*, *max_status_line_garbage=None*, *max_header_count=0*, *pool=None*, *response_class=None*, *timeout=None*, *use_proxy=False*, *max_tries=3*, *wait_tries=0.3*, *pool_size=10*, *backend='thread'*, ***ssl_args*)
Bases: object

A client handle a connection at a time. A client is threadsafe, but an handled shouldn't be shared between threads. All connections are shared between threads via a pool.

```
>>> from restkit import *
>>> c = Client()
>>> r = c.request("http://google.com")
>>> r.status
'301 Moved Permanently'
>>> r.body_string()
'<HTML><HEAD><meta http-equiv="content-type [...]'
>>> c.follow_redirect = True
>>> r = c.request("http://google.com")
>>> r.status
'200 OK'
```

get_connection(*request*)
get a connection from the pool or create new one.

get_response(*request*, *connection*)
return final respons, it is only accessible via peform method

load_filters()
Populate filters from self.filters. Must be called each time self.filters is updated.

make_headers_string(*request*, *extra_headers=None*)
create final header string

perform(*request*)
perform the request. If an error happen it will first try to restart it

proxy_connection(*request*, *req_addr*, *is_ssl*)
do the proxy connection

redirect(*location*, *request*)
reset request, set new url of request and perform it

request(*url*, *method='GET'*, *body=None*, *headers=None*)
perform immediatly a new request

response_class
alias of Response

version = (1, 1)

conn Module

class restkit.conn.**Connection**(*host*, *port*, *backend_mod=None*, *pool=None*, *is_ssl=False*, *extra_headers=[]*, *proxy_pieces=None*, *timeout=None*, ***ssl_args*)
Bases: socketpool.conn.Connector

close()

get_lifetime()

handle_exception(*exception*)

invalidate()

is_connected()

matches(***match_options*)

recv(*size=1024*)

release(*should_close=False*)

send(*data*, *chunked=False*)

send_chunk(*data*)

sendfile(*data*, *chunked=False*)
send a data from a FileObject

sendlines(*lines*, *chunked=False*)

socket()

errors Module

exception classes.

exception restkit.errors.**AlreadyRead**
Bases: exceptions.Exception

raised when a response have already been read

exception restkit.errors.**BadStatusLine**
Bases: exceptions.Exception

Exception returned by the parser when the status line is invalid

exception restkit.errors.**ChunkMissingTerminator**(*term*)
Bases: *restkit.errors.ParseException*

exception restkit.errors.**HeaderLimit**
Bases: *restkit.errors.ParseException*

exception raised when we gore more headers than max_header_count

exception restkit.errors.**InvalidChunkSize**(*data*)
Bases: *restkit.errors.ParseException*

exception restkit.errors.**InvalidHTTPStatus**(*status*)
Bases: *restkit.errors.ParseException*

exception restkit.errors.**InvalidHTTPVersion**(*version*)
Bases: *restkit.errors.ParseException*

exception restkit.errors.**InvalidHeader**(*hdr*)
Bases: *restkit.errors.ParseException*

exception restkit.errors.**InvalidHeaderName**(*hdr*)
Bases: *restkit.errors.ParseException*

exception restkit.errors.**InvalidRequestLine**(*req*)
Bases: *restkit.errors.ParseException*

exception restkit.errors.**InvalidRequestMethod**(*method*)
Bases: *restkit.errors.ParseException*

exception restkit.errors.**InvalidUrl**
Bases: exceptions.Exception

Not a valid url for use with this software.

exception restkit.errors.**NoMoreData**(*buf=None*)
Bases: *restkit.errors.ParseException*

exception restkit.errors.**ParseException**
Bases: exceptions.Exception

exception restkit.errors.**ParserError**
Bases: exceptions.Exception

Generic exception returned by the parser

exception restkit.errors.**ProxyError**
Bases: exceptions.Exception

exception restkit.errors.**RedirectLimit**
Bases: exceptions.Exception

Exception raised when the redirection limit is reached.

exception restkit.errors.**RequestError**
Bases: exceptions.Exception

Exception raised when a request is malformed

exception restkit.errors.**RequestFailed**(*msg=None*, *http_code=None*, *response=None*)
Bases: *restkit.errors.ResourceError*

Exception raised when an unexpected HTTP error is received in response to a request.

The request failed, meaning the remote HTTP server returned a code other than success, unauthorized, or Not-Found.

The exception message attempts to extract the error

You can get the status code by e.status_int, or see anything about the response via e.response. For example, the entire result body (which is probably an HTML error page) is e.response.body.

exception restkit.errors.**RequestTimeout**
Bases: exceptions.Exception

Exception raised on socket timeout

exception restkit.errors.**ResourceError**(*msg=None*, *http_code=None*, *response=None*)
Bases: exceptions.Exception

default error class

message

status_int = None

exception restkit.errors.**ResourceGone**(*msg=None*, *http_code=None*, *response=None*)
Bases: *restkit.errors.ResourceError*

http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html#sec10.4.11

status_int = 410

exception restkit.errors.**ResourceNotFound**(*msg=None*, *http_code=None*, *response=None*)
Bases: *restkit.errors.ResourceError*

Exception raised when no resource was found at the given url.

status_int = 404

exception restkit.errors.**ResponseError**
Bases: exceptions.Exception

Error raised while getting response or decompressing response stream

exception restkit.errors.**Unauthorized**(*msg=None*, *http_code=None*, *response=None*)
Bases: *restkit.errors.ResourceError*

Exception raised when an authorization is required to access to the resource specified.

exception restkit.errors.**UnexpectedEOF**
Bases: exceptions.Exception

exception raised when remote closed the connection

filters Module

class restkit.filters.**BasicAuth**(*username*, *password*)
Bases: object

Simple filter to manage basic authentification

on_request(*request*)

class restkit.filters.**OAuthFilter**(*path*, *consumer*, *token=None*, *method=None*, *realm=''*)
Bases: object

oauth filter

on_path(*request*)

on_request(*request*)

restkit.filters.**validate_consumer**(*consumer*)
validate a consumer agains oauth2.Consumer object

restkit.filters.**validate_token**(*token*)
validate a token agains oauth2.Token object

forms Module

class restkit.forms.**BoundaryItem**(*name*, *value*, *fname=None*, *filetype=None*, *filesize=None*, *quote=<function url_quote>*)

Bases: object

encode(*boundary*)
Returns the string encoding of this parameter

encode_hdr(*boundary*)
Returns the header of the encoding of this parameter

encode_unreadable_value(*value*)

iter_encode(*boundary*, *blocksize=16384*)

class restkit.forms.**MultipartForm**(*params*, *boundary*, *headers*, *bitem_cls=<class 'restkit.forms.BoundaryItem'>*, *quote=<function url_quote>*)

Bases: object

get_size(*recalc=False*)

restkit.forms.**form_encode**(*obj*, *charset='utf8'*)

restkit.forms.**multipart_form_encode**(*params*, *headers*, *boundary*, *quote=<function url_quote>*)

Creates a tuple with MultipartForm instance as body and dict as headers

params dict with fields for the body

headers dict with fields for the header

boundary string to use as boundary

quote (default: url_quote) some callable expecting a string an returning a string. Use for quoting of boundary and form-data keys (names).

oauth2 Module

class restkit.oauth2.**Consumer**(*key*, *secret*)

Bases: object

A consumer of OAuth-protected services.

The OAuth consumer is a "third-party" service that wants to access protected resources from an OAuth service provider on behalf of an end user. It's kind of the OAuth client.

Usually a consumer must be registered with the service provider by the developer of the consumer software. As part of that process, the service provider gives the consumer a *key* and a *secret* with which the consumer software can identify itself to the service. The consumer will include its key in each request to identify itself, but will use its secret only when signing requests, to prove that the request is from that particular registered consumer.

Once registered, the consumer can then use its consumer credentials to ask the service provider for a request token, kicking off the OAuth authorization process.

key = None

secret = None

exception restkit.oauth2.**Error**(*message='OAuth error occurred.'*)

Bases: exceptions.RuntimeError

Generic exception class.

message
A hack to get around the deprecation errors in 2.6.

exception restkit.oauth2.**MissingSignature**(*message='OAuth error occurred.'*)
Bases: *restkit.oauth2.Error*

class restkit.oauth2.**Request**(*method='GET'*, *url=None*, *parameters=None*, *body=''*, *is_form_encoded=False*)
Bases: dict

The parameters and information for an HTTP request, suitable for authorizing with OAuth credentials.

When a consumer wants to access a service's protected resources, it does so using a signed HTTP request identifying itself (the consumer) with its key, and providing an access token authorized by the end user to access those resources.

classmethod from_consumer_and_token(*consumer*, *token=None*, *http_method='GET'*, *http_url=None*, *parameters=None*, *body=''*, *is_form_encoded=False*)

classmethod from_request(*http_method*, *http_url*, *headers=None*, *parameters=None*, *query_string=None*)
Combines multiple parameter sources.

classmethod from_token_and_callback(*token*, *callback=None*, *http_method='GET'*, *http_url=None*, *parameters=None*)

get_nonoauth_parameters()
Get any non-OAuth parameters.

get_normalized_parameters()
Return a string that contains the parameters that must be signed.

get_parameter(*parameter*)

classmethod make_nonce()
Generate pseudorandom number.

classmethod make_timestamp()
Get seconds since epoch (UTC).

method

sign_request(*signature_method*, *consumer*, *token*)
Set the signature parameter to the result of sign.

to_header(*realm=''*)
Serialize as a header for an HTTPAuth request.

to_postdata()
Serialize as post data for a POST request.

to_url()
Serialize as a URL for a GET request.

url

version = '1.0'

class restkit.oauth2.**SignatureMethod**
Bases: object

A way of signing requests.

The OAuth protocol lets consumers and service providers pick a way to sign requests. This interface shows the methods expected by the other *oauth* modules for signing requests. Subclass it and implement its methods to provide a new way to sign requests.

check(*request*, *consumer*, *token*, *signature*)
Returns whether the given signature is the correct signature for the given consumer and token signing the given request.

sign(*request*, *consumer*, *token*)
Returns the signature for the given request, based on the consumer and token also provided.

You should use your implementation of *signing_base()* to build the message to sign. Otherwise it may be less useful for debugging.

signing_base(*request*, *consumer*, *token*)
Calculates the string that needs to be signed.

This method returns a 2-tuple containing the starting key for the signing and the message to be signed. The latter may be used in error messages to help clients debug their software.

class restkit.oauth2.**SignatureMethod_HMAC_SHA1**
Bases: *restkit.oauth2.SignatureMethod*

name = 'HMAC-SHA1'

sign(*request*, *consumer*, *token*)
Builds the base signature string.

signing_base(*request*, *consumer*, *token*)

class restkit.oauth2.**SignatureMethod_PLAINTEXT**
Bases: *restkit.oauth2.SignatureMethod*

name = 'PLAINTEXT'

sign(*request*, *consumer*, *token*)

signing_base(*request*, *consumer*, *token*)
Concatenates the consumer key and secret with the token's secret.

class restkit.oauth2.**Token**(*key*, *secret*)
Bases: object

An OAuth credential used to request authorization or a protected resource.

Tokens in OAuth comprise a *key* and a *secret*. The key is included in requests to identify the token being used, but the secret is used only in the signature, to prove that the requester is who the server gave the token to.

When first negotiating the authorization, the consumer asks for a *request token* that the live user authorizes with the service provider. The consumer then exchanges the request token for an *access token* that can be used to access protected resources.

callback = None

callback_confirmed = None

static from_string(*s*)
Deserializes a token from a string like one returned by *to_string()*.

get_callback_url()

key = None

secret = None

set_callback(*callback*)

set_verifier(*verifier=None*)

to_string()
Returns this token as a plain string, suitable for storage.

The resulting string includes the token's secret, so you should never send or store this string where a third party can read it.

verifier = None

restkit.oauth2.**build_authenticate_header**(*realm=''*)
Optional WWW-Authenticate header (401 error)

restkit.oauth2.**build_xoauth_string**(*url*, *consumer*, *token=None*)
Build an XOAUTH string for use in SMTP/IMPA authentication.

restkit.oauth2.**escape**(*s*)
Escape a URL including any /.

restkit.oauth2.**generate_nonce**(*length=8*)
Generate pseudorandom number.

restkit.oauth2.**generate_timestamp**()
Get seconds since epoch (UTC).

restkit.oauth2.**generate_verifier**(*length=8*)
Generate pseudorandom number.

restkit.oauth2.**setter**(*attr*)

restkit.oauth2.**to_unicode**(*s*)
Convert to unicode, raise exception with instructive error message if s is not unicode, ascii, or utf-8.

restkit.oauth2.**to_unicode_if_string**(*s*)

restkit.oauth2.**to_unicode_optional_iterator**(*x*)
Raise TypeError if x is a str containing non-utf8 bytes or if x is an iterable which contains such a str.

restkit.oauth2.**to_utf8**(*s*)

restkit.oauth2.**to_utf8_if_string**(*s*)

restkit.oauth2.**to_utf8_optional_iterator**(*x*)
Raise TypeError if x is a str or if x is an iterable which contains a str.

resource Module

restkit.resource

This module provide a common interface for all HTTP request.

class restkit.resource.**Resource**(*uri*, ***client_opts*)
Bases: object

A class that can be instantiated for access to a RESTful resource, including authentication.

basic_auth_url = True

charset = 'utf-8'

clone()
if you want to add a path to resource uri, you can do:

```
resr2 = res.clone()
```

delete(*path=None*, *headers=None*, *params_dict=None*, ***params*)
HTTP DELETE

see GET for params description.

encode_keys = True

get(*path=None*, *headers=None*, *params_dict=None*, ***params*)
HTTP GET

- path: string additionnal path to the uri
- **headers: dict, optionnal headers that will** be added to HTTP request.
- params: Optionnal parameterss added to the request.

head(*path=None*, *headers=None*, *params_dict=None*, ***params*)
HTTP HEAD

see GET for params description.

make_headers(*headers*)

make_params(*params*)

post(*path=None*, *payload=None*, *headers=None*, *params_dict=None*, ***params*)
HTTP POST

- payload: string passed to the body of the request
- path: string additionnal path to the uri
- **headers: dict, optionnal headers that will** be added to HTTP request.
- params: Optionnal parameterss added to the request

put(*path=None*, *payload=None*, *headers=None*, *params_dict=None*, ***params*)
HTTP PUT

see POST for params description.

request(*method*, *path=None*, *payload=None*, *headers=None*, *params_dict=None*, ***params*)
HTTP request

This method may be the only one you want to override when subclassing *restkit.rest.Resource*.

- payload: string or File object passed to the body of the request
- path: string additionnal path to the uri
- **headers: dict, optionnal headers that will** be added to HTTP request.

Params_dict Options parameters added to the request as a dict

- params: Optionnal parameterss added to the request

response_class
alias of `Response`

safe = '/:'

unauthorized(*response*)

update_uri(*path*)
to set a new uri absolute path

wrappers Module

class restkit.wrappers.**BodyWrapper**(*resp*, *connection*)
Bases: object

close()
release connection

next()

read(*n=-1*)

readline(*limit=-1*)

readlines(*hint=None*)

restkit.wrappers.**ClientResponse**
alias of *Response*

class restkit.wrappers.**Request**(*url*, *method='GET'*, *body=None*, *headers=None*)
Bases: object

body
request body

headers

host

is_chunked()

is_ssl()

maybe_rewind(*msg=''*)

parsed_url
parsed url

path

class restkit.wrappers.**Response**(*connection*, *request*, *resp*)
Bases: object

body_stream()
stream body

body_string(*charset=None*, *unicode_errors='strict'*)
return body string, by default in bytestring

can_read()

charset = 'utf8'

close()

skip_body()
skip the body and release the connection

tee()
copy response input to standard output or a file if length > sock.MAX_BODY. This make possible to reuse it in your appplication. When all the input has been read, connection is released

unicode_errors = 'strict'

contrib Package

console Module

restkit.contrib.console.**as_bool**(*value*)

restkit.contrib.console.**external**(*cmd*, *data*)

restkit.contrib.console.**indent**(*mimetype*, *data*)

restkit.contrib.console.**indent_json**(*data*)

restkit.contrib.console.**indent_xml**(*data*)

restkit.contrib.console.**main**()
function to manage restkit command line

restkit.contrib.console.**options**()
build command lines options

restkit.contrib.console.**prettify**(*response*, *cli=True*)

restkit.contrib.console.**update_defaults**(*defaults*)

ipython_shell Module

class restkit.contrib.ipython_shell.**ContentTypes**
Bases: object

class restkit.contrib.ipython_shell.**JSON**(*value*)
Bases: *restkit.contrib.ipython_shell.Stream*

class restkit.contrib.ipython_shell.**Request**(*environ*, *charset=None*, *unicode_errors=None*, *decode_param_names=None*, ***kw*)
Bases: *restkit.contrib.webob_api.Request*

ResponseClass
alias of *Response*

get_response(**args*, ***kwargs*)

class restkit.contrib.ipython_shell.**Response**(*body=None*, *status=None*, *headerlist=None*, *app_iter=None*, *content_type=None*, *conditional_response=None*, *charset=<object object>*, ***kw*)
Bases: webob.response.Response

class restkit.contrib.ipython_shell.**RestShell**(*user_ns={}*)
Bases: IPython.terminal.embed.InteractiveShellEmbed

class restkit.contrib.ipython_shell.**ShellClient**(*url='/'*, *options=None*, ***kwargs*)
Bases: object

help()

methods = {'put': '[req|url|path_info], stream', 'head': '[req|url|path_info], **query_string', 'delete': '[req|url|path_info

request(*meth*, **args*, ***kwargs*)
forward to restkit.request

request_meth(*k*)

update_ns(*ns*)

class restkit.contrib.ipython_shell.**Stream**(*buf=''*)
Bases: StringIO.StringIO

restkit.contrib.ipython_shell.**main**(**args*, ***kwargs*)

webob_api Module

Subclasses of webob.Request who use restkit to get a webob.Response via restkit.ext.wsgi_proxy.Proxy.

Example:

```
>>> req = Request.blank('http://pypi.python.org/pypi/restkit')
>>> resp = req.get_response()
>>> print resp
200 OK
Date: ...
Transfer-Encoding: chunked
Content-Type: text/html; charset=utf-8
Server: Apache/2...

<?xml version="1.0" encoding="UTF-8"?>
...
```

class restkit.contrib.webob_api.**Method**(*name*)
Bases: property

class restkit.contrib.webob_api.**Request**(*environ*, *charset=None*, *unicode_errors=None*, *decode_param_names=None*, ***kw*)
Bases: webob.request.Request

delete

get

get_response()

head

post

put

set_url(*url*)

webob_helper Module

exception restkit.contrib.webob_helper.**WebobResourceError**(*msg=None*, *http_code=None*, *response=None*)
Bases: webob.exc.WSGIHTTPException

Wrapper to return webob exceptions instead of restkit errors. Usefull for those who want to build WSGI applications speaking directly to others via HTTP.

To do it place somewhere in your application the function *wrap_exceptions*:

```
wrap_exceptions()
```

It will automatically replace restkit errors by webob exceptions.

message

status_int
The status as an integer

restkit.contrib.webob_helper.**wrap_exceptions**()
wrap restkit exception to return WebBob exceptions

wsgi_proxy Module

class restkit.contrib.wsgi_proxy.**HostProxy**(*uri*, ***kwargs*)
Bases: *restkit.contrib.wsgi_proxy.Proxy*

A proxy to redirect all request to a specific uri

extract_uri(*environ*)

class restkit.contrib.wsgi_proxy.**Proxy**(*manager=None, allowed_methods=['GET', 'HEAD', 'POST', 'PUT', 'DELETE'], strip_script_name=True, **kwargs*)
Bases: object

A proxy wich redirect the request to SERVER_NAME:SERVER_PORT and send HTTP_HOST header

extract_uri(*environ*)

class restkit.contrib.wsgi_proxy.**TransparentProxy**(*manager=None, allowed_methods=['GET', 'HEAD', 'POST', 'PUT', 'DELETE'], strip_script_name=True, **kwargs*)
Bases: *restkit.contrib.wsgi_proxy.Proxy*

A proxy based on HTTP_HOST environ variable

extract_uri(*environ*)

restkit.contrib.wsgi_proxy.**get_config**(*local_config*)
parse paste config

restkit.contrib.wsgi_proxy.**make_host_proxy**(*global_config*, *uri=None*, ***local_config*)
HostProxy entry_point

restkit.contrib.wsgi_proxy.**make_proxy**(*global_config*, ***local_config*)
TransparentProxy entry_point

Python Module Index

r

Index

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W