httplib2

Release 0.4

May 15, 2017

Contents

Author Joe Gregorio

Date Mar 8, 2007

Abstract

The *httplib2* module is a comprehensive HTTP client library that handles caching, keep-alive, compression, redirects and many kinds of authentication.

Contents:

CHAPTER 1

httplib2 A comprehensive HTTP client library.

The *httplib2* module is a comprehensive HTTP client library with the following features:

HTTP and HTTPS
HTTPS support is only available if the socket module was compiled with SSL support.

Keep-Alive
Supports HTTP 1.1 Keep-Alive, keeping the socket open and performing multiple requests over the same connection if possible.

Authentication
The following three types of HTTP Authentication are supported. These can be used over both HTTP and HTTPS.

- Digest
- Basic
- WSSE

Caching
The module can optionally operate with a private cache that understands the Cache-Control: header and uses both the ETag and Last-Modified cache validators.

All Methods
The module can handle any HTTP request method, not just GET and POST.

Redirects
Automatically follows 3XX redirects on GETs.

Compression
Handles both `deflate` and `gzip` types of compression.

Lost update support
Automatically adds back ETags into PUT requests to resources we have already cached. This implements Section 3.2 of Detecting the Lost Update Problem Using Unreserved Checkout

The *httplib2* module defines the following variables:

`httplib2.`**`debuglevel`**
The amount of debugging information to print. The default is 0.

`httplib2.`**`RETRIES`**
A request will be tried 'RETRIES' times if it fails at the socket/connection level. The default is 2.

The *httplib2* module may raise the following Exceptions. Note that there is an option that turns exceptions into normal responses with an HTTP status code indicating an error occured. See *Http.force_exception_to_status_code*

exception `httplib2.`**`HttpLib2Error`**
The Base Exception for all exceptions raised by httplib2.

exception `httplib2.`**`RedirectMissingLocation`**
A 3xx redirect response code was provided but no Location: header was provided to point to the new location.

exception `httplib2.`**`RedirectLimit`**
The maximum number of redirections was reached without coming to a final URI.

exception `httplib2.`**`ServerNotFoundError`**
Unable to resolve the host name given.

exception `httplib2.`**`RelativeURIError`**
A relative, as opposed to an absolute URI, was passed into request().

exception `httplib2.`**`FailedToDecompressContent`**
The headers claimed that the content of the response was compressed but the decompression algorithm applied to the content failed.

exception `httplib2.`**`UnimplementedDigestAuthOptionError`**
The server requested a type of Digest authentication that we are unfamiliar with.

exception `httplib2.`**`UnimplementedHmacDigestAuthOptionError`**
The server requested a type of HMACDigest authentication that we are unfamiliar with.

class `httplib2.`**`Http`**(*[cache=None][, timeout=None][, proxy_info==ProxyInfo.from_environment][, ca_certs=None][, disable_ssl_certificate_validation=False]*)
The class that represents a client HTTP interface. The *cache* parameter is either the name of a directory to be used as a flat file cache, or it must an object that implements the required caching interface. The *timeout* parameter is the socket level timeout. The *ca_certs* parameter is the filename of the CA certificates to use. If none is given a default set is used. The *disable_ssl_certificate_validation* boolean flag determines if ssl certificate validation is done. The *proxy_info* parameter is an object of type :class:ProxyInfo.

class `httplib2.`**`ProxyInfo`**(*proxy_type, proxy_host, proxy_port[, proxy_rdns=None][, proxy_user=None][, proxy_pass=None]*)
Collect information required to use a proxy. The parameter proxy_type must be set to one of socks.PROXY_TYPE_XXX constants. For example:

p = ProxyInfo(proxy_type=socks.PROXY_TYPE_HTTP, proxy_host='localhost', proxy_port=8000)

class `httplib2.`**`Response`**(*info*)
Response is a subclass of `dict` and instances of this class are returned from calls to Http.request. The *info* parameter is either an `rfc822.Message` or an `httplib.HTTPResponse` object.

class `httplib2.`**`FileCache`**(*dir_name[, safe=safename]*)
FileCache implements a Cache as a directory of files. The *dir_name* parameter is the name of the directory to use. If the directory does not exist then FileCache attempts to create the directory. The optional *safe* parameter is a funtion which generates the cache filename for each URI. A FileCache object is constructed and used for caching when you pass a directory name into the constructor of *Http*.

Http objects have the following methods:

Http Objects

`Http.request`(*uri*[, *method="GET"*, *body=None*, *headers=None*, *redirections=DEFAULT_MAX_REDIRECTS*, *connection_type=None*])

Performs a single HTTP request. The *uri* is the URI of the HTTP resource and can begin with either `http` or `https`. The value of *uri* must be an absolute URI.

The *method* is the HTTP method to perform, such as `GET`, `POST`, `DELETE`, etc. There is no restriction on the methods allowed.

The *body* is the entity body to be sent with the request. It is a string object.

Any extra headers that are to be sent with the request should be provided in the *headers* dictionary.

The maximum number of redirect to follow before raising an exception is *redirections*. The default is 5.

The *connection_type* is the type of connection object to use. The supplied class should implement the interface of httplib.HTTPConnection.

The return value is a tuple of (response, content), the first being and instance of the *Response* class, the second being a string that contains the response entity body.

`Http.add_credentials`(*name*, *password*[, *domain=None*])

Adds a name and password that will be used when a request requires authentication. Supplying the optional *domain* name will restrict these credentials to only be sent to the specified domain. If *domain* is not specified then the given credentials will be used to try to satisfy every HTTP 401 challenge.

`Http.add_certificate`(*key*, *cert*, *domain*)

Add a *key* and *cert* that will be used for an SSL connection to the specified domain. *keyfile* is the name of a PEM formatted file that contains your private key. *certfile* is a PEM formatted certificate chain file.

`Http.clear_credentials`()

Remove all the names and passwords used for authentication.

`Http.follow_redirects`

If `True`, which is the default, safe redirects are followed, where safe means that the client is only doing a `GET` or `HEAD` on the URI to which it is being redirected. If `False` then no redirects are followed. Note that a False 'follow_redirects' takes precedence over a True 'follow_all_redirects'. Another way of saying that is for 'follow_all_redirects' to have any affect, 'follow_redirects' must be True.

`Http.follow_all_redirects`

If `False`, which is the default, only safe redirects are followed, where safe means that the client is only doing a `GET` or `HEAD` on the URI to which it is being redirected. If `True` then all redirects are followed. Note that a False 'follow_redirects' takes precedence over a True 'follow_all_redirects'. Another way of saying that is for 'follow_all_redirects' to have any affect, 'follow_redirects' must be True.

`Http.forward_authorization_headers`

If `False`, which is the default, then Authorization: headers are stripped from redirects. If `True` then Authorization: headers are left in place when following redirects. This parameter only applies if following redirects is turned on. Note that turning this on could cause your credentials to leak, so carefully consider the consequences.

`Http.force_exception_to_status_code`

If `True` then no *httplib2* exceptions will be thrown. Instead, those error conditions will be turned into *Response* objects that will be returned normally.

If `False`, which is the default, then exceptions will be thrown.

`Http.optimistic_concurrency_methods`

By default a list that only contains "PUT", this attribute controls which methods will get 'if-match' headers attached to them from cached responses with etags. You can append new items to this list to add new methods that should get this support, such as "PATCH".

`Http.`**`ignore_etag`**
: Defaults to `False`. If `True`, then any etags present in the cached response are ignored when processing the current request, i.e. httplib2 does **not** use 'if-match' for PUT or 'if-none-match' when GET or HEAD requests are made. This is mainly to deal with broken servers which supply an etag, but change it capriciously.

If you wish to supply your own caching implementation then you will need to pass in an object that supports the following methods. Note that the `memcache` module supports this interface natively.

Cache Objects

`Cache.`**`get`**(*key*)
: Takes a string *key* and returns the value as a string.

`Cache.`**`set`**(*key*, *value*)
: Takes a string *key* and *value* and stores it in the cache.

`Cache.`**`delete`**(*key*)
: Deletes the cached value stored at *key*. The value of *key* is a string.

Response objects are derived from `dict` and map header names (lower case with the trailing colon removed) to header values. In addition to the dict methods a Response object also has:

Response Objects

`Response.`**`fromcache`**
: If `true` the the response was returned from the cache.

`Response.`**`version`**
: The version of HTTP that the server supports. A value of 11 means '1.1'.

`Response.`**`status`**
: The numerical HTTP status code returned in the response.

`Response.`**`reason`**
: The human readable component of the HTTP response status code.

`Response.`**`previous`**
: If redirects are followed then the *`Response`* object returned is just for the very last HTTP request and *previous* points to the previous *`Response`* object. In this manner they form a chain going back through the responses to the very first response. Will be `None` if there are no previous respones.

The Response object also populates the header `content-location`, that contains the URI that was ultimately requested. This is useful if redirects were encountered, you can determine the ultimate URI that the request was sent to. All Response objects contain this key value, including `previous` responses so you can determine the entire chain of redirects. If *`Http.force_exception_to_status_code`* is `True` and the number of redirects has exceeded the number of allowed number of redirects then the *`Response`* object will report the error in the status code, but the complete chain of previous responses will still be in tact.

To do a simple `GET` request just supply the absolute URI of the resource:

Examples

```
import httplib2
h = httplib2.Http()
resp, content = h.request("http://bitworking.org/")
assert resp.status == 200
assert resp['content-type'] == 'text/html'
```

Here is more complex example that does a PUT of some text to a resource that requires authentication. The Http instance also uses a file cache in the directory `.cache`.

```
import httplib2
h = httplib2.Http(".cache")
h.add_credentials('name', 'password')
resp, content = h.request("https://example.org/chap/2",
    "PUT", body="This is text",
    headers={'content-type':'text/plain'} )
```

Here is an example that connects to a server that supports the Atom Publishing Protocol.

```
import httplib2
h = httplib2.Http()
h.add_credentials(myname, mypasswd)
h.follow_all_redirects = True
headers = {'Content-Type': 'application/atom+xml'}
body    = """<?xml version="1.0" ?>
    <entry xmlns="http://www.w3.org/2005/Atom">
      <title>Atom-Powered Robots Run Amok</title>
      <id>urn:uuid:1225c695-cfb8-4ebb-aaaa-80da344efa6a</id>
      <updated>2003-12-13T18:30:02Z</updated>
      <author><name>John Doe</name></author>
      <content>Some text.</content>
</entry>
"""
uri     = "http://www.example.com/collection/"
resp, content = h.request(uri, "POST", body=body, headers=headers)
```

Here is an example of providing data to an HTML form processor. In this case we presume this is a POST form. We need to take our data and format it as "application/x-www-form-urlencoded" data and use that as a body for a POST request.

```
>>> import httplib2
>>> import urllib
>>> data = {'name': 'fred', 'address': '123 shady lane'}
>>> body = urllib.urlencode(data)
>>> body
'name=fred&address=123+shady+lane'
>>> h = httplib2.Http()
>>> resp, content = h.request("http://example.com", method="POST", body=body)
```

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

h

Index