Introducing httplib2

WE'LL COVER THE FOLLOWING

Before you can use `httplib2`, you'll need to install it. Visit code.google.com/p/httplib2/ and download the latest version. `httplib2` is available for Python 2.x and Python 3.x; make sure you get the Python 3 version, named something like `httplib2-python3-0.5.0.zip`.

Unzip the archive, open a terminal window, and go to the newly created `httplib2` directory. On Windows, open the `Start` menu, select `Run`..., type `cmd.exe` and press `ENTER`.

```
c:\Users\pilgrim\Downloads> dir
 Volume in drive C has no label.
 Volume Serial Number is DED5-B4F8

 Directory of c:\Users\pilgrim\Downloads

07/28/2009  12:36 PM    <DIR>          .
07/28/2009  12:36 PM    <DIR>          ..
07/28/2009  12:36 PM    <DIR>          httplib2-python3-0.5.0
07/28/2009  12:33 PM            18,997 httplib2-python3-0.5.0.zip
               1 File(s)         18,997 bytes
               3 Dir(s)  61,496,684,544 bytes free

c:\Users\pilgrim\Downloads> cd httplib2-python3-0.5.0
c:\Users\pilgrim\Downloads\httplib2-python3-0.5.0> c:\python31\python.exe setup.py install
running install
running build
running build_py
running install_lib
creating c:\python31\Lib\site-packages\httplib2
copying build\lib\httplib2\iri2uri.py -> c:\python31\Lib\site-packages\httplib2

copying build\lib\httplib2\__init__.py -> c:\python31\Lib\site-packages\httplib2
byte-compiling c:\python31\Lib\site-packages\httplib2\iri2uri.py to iri2uri.pyc
byte-compiling c:\python31\Lib\site-packages\httplib2\__init__.py to __init__.pyc
running install_egg_info
Writing c:\python31\Lib\site-packages\httplib2-python3_0.5.0-py3.1.egg-info
```

On Mac OS X, run the `Terminal.app` application in your `/Applications/Utilities/` folder. On Linux, run the `Terminal` application, which is usually in your `Applications` menu under `Accessories` or `System`.

```
you@localhost:~/Desktop$ unzip httplib2-python3-0.5.0.zip
Archive:  httplib2-python3-0.5.0.zip
  inflating: httplib2-python3-0.5.0/README
  inflating: httplib2-python3-0.5.0/setup.py
  inflating: httplib2-python3-0.5.0/PKG-INFO
  inflating: httplib2-python3-0.5.0/httplib2/__init__.py
  inflating: httplib2-python3-0.5.0/httplib2/iri2uri.py
you@localhost:~/Desktop$ cd httplib2-python3-0.5.0/
you@localhost:~/Desktop/httplib2-python3-0.5.0$ sudo python3 setup.py install
running install
running build
running build_py
creating build
creating build/lib.linux-x86_64-3.1
creating build/lib.linux-x86_64-3.1/httplib2
copying httplib2/iri2uri.py -> build/lib.linux-x86_64-3.1/httplib2
copying httplib2/__init__.py -> build/lib.linux-x86_64-3.1/httplib2
running install_lib
creating /usr/local/lib/python3.1/dist-packages/httplib2
copying build/lib.linux-x86_64-3.1/httplib2/iri2uri.py -> /usr/local/lib/python3.1/dist-packa
copying build/lib.linux-x86_64-3.1/httplib2/__init__.py -> /usr/local/lib/python3.1/dist-pack
byte-compiling /usr/local/lib/python3.1/dist-packages/httplib2/iri2uri.py to iri2uri.pyc
byte-compiling /usr/local/lib/python3.1/dist-packages/httplib2/__init__.py to __init__.pyc
running install_egg_info
Writing /usr/local/lib/python3.1/dist-packages/httplib2-python3_0.5.0.egg-info
```

To use `httplib2`, create an instance of the `httplib2.Http` class.

```
import httplib2
h = httplib2.Http('.cache')                                                #①
response, content = h.request('http://diveintopython3.org/examples/feed.xml')  #②
print (response.status)                                                    #③
#200
print (content[:52])                                                       #④
#b"<?xml version='1.0' encoding='utf-8'?>\r\n<feed xmlns="

print (len(content))
#303
```

① The primary interface to `httplib2` is the `Http` object. For reasons you'll see in the next section, you should always pass a directory name when you create an `Http` object. The directory does not need to exist; `httplib2` will create it if necessary.

② Once you have an `Http` object, retrieving data is as simple as calling the `request()` method with the address of the data you want. This will issue an HTTP `GET` request for that URL. (Later in this chapter, you'll see how to issue other HTTP requests, like `POST`.)

③ The `request()` method returns two values. The first is an `httplib2.Response` object, which contains all the HTTP headers the server returned. For example, a `status` code of `200` indicates that the request was successful.

④ The `content` variable contains the actual data that was returned by the HTTP server. The data is returned as a `bytes` object, not a string. If you want it as a string, you'll need to determine the character encoding and convert it yourself.

> *You probably only need one **httplib2.Http** object. There are valid reasons for creating more than one, but you should only do so if you know why you need them. "I need to request data from two different URLs" is not a valid reason. Re-use the **Http** object and just call the **request()** method twice.*

A Short Digression To Explain Why httplib2 Returns Bytes Instead of Strings

Bytes. Strings. What a pain. Why can't `httplib2` "just" do the conversion for you? Well, it's complicated, because the rules for determining the character encoding are specific to what kind of resource you're requesting. How could `httplib2` know what kind of resource you're requesting? It's usually listed in the `Content-Type` HTTP header, but that's an optional feature of HTTP and not all HTTP servers include it. If that header is not included in the HTTP response, it's left up to the client to guess. (This is commonly called "content sniffing," and it's never perfect.)

If you know what sort of resource you're expecting (an XML document in this case), perhaps you could "just" pass the returned bytes object to the

xml.etree.ElementTree.parse() function. That'll work as long as the XML document includes information on its own character encoding (as this one does), but that's an optional feature and not all XML documents do that. If an XML document doesn't include encoding information, the client is supposed to look at the enclosing transport — i.e. the Content-Type http header, which can include a charset parameter.

But it's worse than that. Now character encoding information can be in two places: within the XML document itself, and within the `Content-Type` HTTP header. If the information is in *both* places, which one wins? According to RFC 3023 (I swear I am not making this up), if the media type given in the `Content-Type` HTTP header is `application/xml`, `application/xml-dtd`, `application/xml-external-parsed-entity`, or any one of the subtypes of `application/xml` such as `application/atom+xml` or `application/rss+xml` or even `application/rdf+xml`, then the encoding is

1. the encoding given in the `charset` parameter of the `Content-Type` HTTP header, or
2. the encoding given in the `encoding` attribute of the XML declaration within the document, or
3. UTF-8

On the other hand, if the media type given in the `Content-Type` HTTP header is `text/xml`, `text/xml-external-parsed-entity`, or a subtype like `text/AnythingAtAll+xml`, then the encoding attribute of the XML declaration within the document is ignored completely, and the encoding is

1. the encoding given in the charset parameter of the `Content-Type` HTTP header, or
2. us-ascii

And that's just for XML documents. For HTML documents, web browsers have constructed such byzantine rules for content-sniffing [pdf] that we're still trying to figure them all out.

"Patches welcome."

How httplib2 Handles Caching

How httplib2 Handles Caching

Remember in the previous section when I said you should always create an `httplib2.Http` object with a directory name? Caching is the reason.

```
# continued from the previous example
import httplib2
h = httplib2.Http('.cache')
response2, content2 = h.request('http://diveintopython3.org/examples/feed.xml')  #①
print (response2.status)                                                         #②
#200

print (content2[:52])                                                            #③
#b"<?xml version='1.0' encoding='utf-8'?>\r\n<feed xmlns="

print (len(content2))
#303
```

① This shouldn't be terribly surprising. It's the same thing you did last time, except you're putting the result into two new variables.

② The HTTP `status` is once again `200`, just like last time.

③ The downloaded content is the same as last time, too.

So… who cares? Quit your Python interactive shell and relaunch it with a new session, and I'll show you.

```
# NOT continued from previous example!
# Please exit out of the interactive shell
# and launch a new one.
import httplib2
httplib2.debuglevel = 1                                                        #①
h = httplib2.Http('.cache')                                                    #②
response, content = h.request('http://diveintopython3.org/examples/feed.xml')  #③
print (len(content))                                                           #④
#303

print (response.status)                                                        #⑤
#200

print (response.fromcache )                                                    #⑥
#False
```

① Let’s turn on debugging and see what’s on the wire. This is the `httplib2` equivalent of turning on debugging in `http.client`. `httplib2` will print all the data being sent to the server and some key information being sent back.

② Create an `httplib2.Http` object with the same directory name as before.

③ Request the same URL as before. *Nothing appears to happen*. More precisely, nothing gets sent to the server, and nothing gets returned from the server. There is absolutely no network activity whatsoever.

④ Yet we did “receive” some data — in fact, we received all of it.

⑤ We also “received” an HTTP status code indicating that the “request” was successful.

⑥ Here’s the rub: this “response” was generated from `httplib2’s` local cache. That directory name you passed in when you created the `httplib2.Http` object — that directory holds `httplib2`’s cache of all the operations it’s ever performed.

What’s on the wire? Absolutely nothing.

> *If you want to turn on* ***httplib2*** *debugging, you need to set a module-level constant* ***(httplib2.debuglevel)****, then create a new* ***httplib2.Http*** *object. If you want to turn off debugging, you need to change the same module-level constant, then create a new* ***httplib2.Http object****.*

You previously requested the data at this URL. That request was successful (`status: 200`). That response included not only the feed data, but also a set of caching headers that told anyone who was listening that they could cache this resource for up to 24 hours (`Cache-Control: max-age=86400`, which is 24 hours measured in seconds). `httplib2` understand and respects those caching headers, and it stored the previous response in the `.cache` directory (which you passed in when you create the `Http` object). That cache hasn’t expired yet, so the second time you request the data at this URL, `httplib2` simply returns the cached result without ever hitting the network.

I say “simply,” but obviously there is a lot of complexity hidden behind that simplicity. `httplib2` handles HTTP caching *automatically* and *by default*. If for some reason you need to know whether a response came from the cache, you can check `response.fromcache`. Otherwise, it Just Works.

Now, suppose you have data cached, but you want to bypass the cache and re-request it from the remote server. Browsers sometimes do this if the user specifically requests it. For example, pressing `F5` refreshes the current page, but pressing `Ctrl+F5` bypasses the cache and re-requests the current page from the remote server. You might think “oh, I’ll just delete the data from my local cache, then request it again.” You could do that, but remember that there may be more parties involved than just you and the remote server. What about those intermediate proxy servers? They’re completely beyond your control, and they may still have that data cached, and will happily return it to you because (as far as they are concerned) their cache is still valid.

Instead of manipulating your local cache and hoping for the best, you should use the features of HTTP to ensure that your request actually reaches the remote server.

```
# continued from the previous example
response2, content2 = h.request('http://diveintopython3.org/examples/feed.xml',
     headers={'cache-control':'no-cache'})      #①
#connect: (diveintopython3.org, 80)             #②
#send: b'GET /examples/feed.xml HTTP/1.1
#Host: diveintopython3.org
#user-agent: Python-httplib2/$Rev: 259 $
#accept-encoding: deflate, gzip
#cache-control: no-cache'
#reply: 'HTTP/1.1 200 OK'
#…further debugging information omitted…

response2.status
#200

response2.fromcache                             #③
#False

dict(response2.items())                         #④
#{'status': '200',
# 'content-length': '3070',
# 'content-location': 'http://diveintopython3.org/examples/feed.xml',
# 'accept-ranges': 'bytes',
# 'expires': 'Wed, 03 Jun 2009 00:40:26 GMT',
# 'vary': 'Accept-Encoding',
# 'server': 'Apache',
# 'last-modified': 'Sun, 31 May 2009 22:51:11 GMT',
# 'connection': 'close',
# '-content-encoding': 'gzip',

# 'etag': '"bfe-255ef5c0"',
# 'cache-control': 'max-age=86400',
# 'date': 'Tue, 02 Jun 2009 00:40:26 GMT',
# 'content-type': 'application/xml'}
```

① `httplib2` allows you to add arbitrary HTTP headers to any outgoing request. In order to bypass *all* caches (not just your local disk cache, but also any caching proxies between you and the remote server), add a `no-cache` header in the `headers` dictionary.

② Now you see `httplib2` initiating a network request. `httplib2` understands and respects caching headers *in both directions* — as part of the incoming response and *as part of the outgoing request*. It noticed that you added the `no-cache` header, so it bypassed its local cache altogether and then had no choice but to hit the network to request the data.

③ This response was *not* generated from your local cache. You knew that, of course, because you saw the debugging information on the outgoing request. But it's nice to have that programmatically verified.

④ The request succeeded; you downloaded the entire feed again from the remote server. Of course, the server also sent back a full complement of HTTP headers along with the feed data. That includes caching headers, which `httplib2` uses to update its local cache, in the hopes of avoiding network access the next time you request this feed. Everything about http caching is designed to maximize cache hits and minimize network access. Even though you bypassed the cache this time, the remote server would really appreciate it if you would cache the result for next time.

How httplib2 Handles Last-Modified and ETag Headers

The `Cache-Control` and `Expires` caching headers are called *freshness indicators*. They tell caches in no uncertain terms that you can completely avoid all network access until the cache expires. And that's exactly the behavior you saw in the previous section: given a freshness indicator, `httplib2` *does not generate a single byte of network activity* to serve up cached data (unless you explicitly bypass the cache, of course).

But what about the case where the data *might* have changed, but hasn't? HTTP defines Last-Modified and Etag headers for this purpose. These headers are called *validators*. If the local cache is no longer fresh, a client can send the validators with the next request to see if the data has actually changed. If the data hasn't changed, the server sends back a `304` status code *and no data*. So there's still a round-trip over the network, but you end up downloading fewer bytes.

```
import httplib2
httplib2.debuglevel = 1

response, content = h.request('http://diveintopython3.org/')  #①
#connect: (diveintopython3.org, 80)
#send: b'GET / HTTP/1.1
#Host: diveintopython3.org
#accept-encoding: deflate, gzip
#user-agent: Python-httplib2/$Rev: 259 $'
#reply: 'HTTP/1.1 200 OK'

dict(response.items())                                         #②
#{'-content-encoding': 'gzip',
# 'accept-ranges': 'bytes',
# 'connection': 'close',
# 'content-length': '6657',
# 'content-location': 'http://diveintopython3.org/',
# 'content-type': 'text/html',
# 'date': 'Tue, 02 Jun 2009 03:26:54 GMT',
# 'etag': '"7f806d-1a01-9fb97900"',
# 'last-modified': 'Tue, 02 Jun 2009 02:51:48 GMT',
# 'server': 'Apache',
# 'status': '200',
# 'vary': 'Accept-Encoding,User-Agent'}

print (len(content))                                                  # ③
#6657
```

① Instead of the feed, this time we're going to download the site's home page, which is HTML. Since this is the first time you've ever requested this page, `httplib2` has little to work with, and it sends out a minimum of headers with the request.

② The response contains a multitude of HTTP headers… but no caching information. However, it does include both an `ETag` and `Last-Modified` header.

③ At the time I constructed this example, this page was 6657 bytes. It's probably changed since then, but don't worry about it.

```
# continued from the previous example
response, content = h.request('http://diveintopython3.org/')  #①
#connect: (diveintopython3.org, 80)
#send: b'GET / HTTP/1.1
#Host: diveintopython3.org
#if-none-match: "7f806d-1a01-9fb97900"                         #②
#if-modified-since: Tue, 02 Jun 2009 02:51:48 GMT             #③
#accept-encoding: deflate, gzip
#user-agent: Python-httplib2/$Rev: 259 $'
#reply: 'HTTP/1.1 304 Not Modified'                           #④

response.fromcache                                            #⑤
#True

response.status                                               #⑥
#200

response.dict['status']                                       #⑦
#'304'

print (len(content))                                          #⑧
#6657
```

① You request the same page again, with the same `Http` object (and the same local cache).

② `httplib2` sends the `ETag` validator back to the server in the `If-None-Match` header.

③ `httplib2` also sends the `Last-Modified` validator back to the server in the `If-Modified-Since` header.

④ The server looked at these validators, looked at the page you requested, and determined that the page has not changed since you last requested it, so it sends back a `304` status code and no data.

⑤ Back on the client, `httplib2` notices the `304` status code and loads the content of the page from its cache.

⑥ This might be a bit confusing. There are really *two* status codes — `304` (returned from the server this time, which caused `httplib2` to look in its cache), and `200` (returned from the server *last time*, and stored in `httplib2`'s cache along with the page data). `response.status` returns the status from the cache.

⑦ If you want the raw status code returned from the server, you can get that by looking in `response.dict`, which is a dictionary of the actual headers

returned from the server.

⑧ However, you still get the data in the `content` variable. Generally, you don't need to know why a response was served from the cache. (You may not even care that it was served from the cache at all, and that's fine too. `httplib2` is smart enough to let you act dumb.) By the time the `request()` method returns to the caller, `httplib2` has already updated its cache and returned the data to you.

How http2lib Handles Compression

"We have both kinds of music, country AND western."

HTTP supports several types of compression; the two most common types are gzip and deflate. `httplib2` supports both of these.

```
response, content = h.request('http://diveintopython3.org/')
#connect: (diveintopython3.org, 80)
#send: b'GET / HTTP/1.1
#Host: diveintopython3.org
#accept-encoding: deflate, gzip                         #①
#user-agent: Python-httplib2/$Rev: 259 $'
#reply: 'HTTP/1.1 200 OK'

dict(response.items())
#{'-content-encoding': 'gzip',                           #②
# 'accept-ranges': 'bytes',
# 'connection': 'close',
# 'content-length': '6657',
# 'content-location': 'http://diveintopython3.org/',
# 'content-type': 'text/html',
# 'date': 'Tue, 02 Jun 2009 03:26:54 GMT',
# 'etag': '"7f806d-1a01-9fb97900"',
# 'last-modified': 'Tue, 02 Jun 2009 02:51:48 GMT',
# 'server': 'Apache',
# 'status': '304',
# 'vary': 'Accept-Encoding,User-Agent'}
```

① Every time `httplib2` sends a request, it includes an `Accept-Encoding` header to tell the server that it can handle either `deflate` or `gzip` compression.

② In this case, the server has responded with a gzip-compressed payload. By the time the `request()` method returns, `httplib2` has already decompressed the body of the response and placed it in the content variable. If you're curious about whether or not the response was compressed, you can check `response['-content-encoding']`; otherwise, don't worry about it.

How httplib2 Handles Redirects

HTTP defines two kinds of redirects: temporary and permanent. There's nothing special to do with temporary redirects except follow them, which `httplib2` does automatically.

```
import httplib2
httplib2.debuglevel = 1
h = httplib2.Http('.cache')
response, content = h.request('http://diveintopython3.org/examples/feed-302.xml')      #①
#connect: (diveintopython3.org, 80)
#send: b'GET /examples/feed-302.xml HTTP/1.1                                           #②
#Host: diveintopython3.org
#accept-encoding: deflate, gzip
#user-agent: Python-httplib2/$Rev: 259 $'
#reply: 'HTTP/1.1 302 Found'                                                           #③
#send: b'GET /examples/feed.xml HTTP/1.1                                               #④
#Host: diveintopython3.org
#accept-encoding: deflate, gzip
#user-agent: Python-httplib2/$Rev: 259 $'
#reply: 'HTTP/1.1 200 OK'
```

① There is no feed at this URL. I've set up my server to issue a temporary redirect to the correct address.

② There's the request.

③ And there's the response: `302 Found`. Not shown here, this response also includes a `Location` header that points to the real URL.

④ `httplib2` immediately turns around and "follows" the redirect by issuing another request for the URL given in the `Location` header: `http://diveintopython3.org/examples/feed.xml`

"Following" a redirect is nothing more than this example shows. `httplib2` sends a request for the URL you asked for. The server comes back with a response that says "No no, look over there instead." `httplib2` sends another request for the new URL.

```
# continued from the previous example
response                                                                  #①
```

```
#{'status': '200',
# 'content-length': '3070',
# 'content-location': 'http://diveintopython3.org/examples/feed.xml',  #②
# 'accept-ranges': 'bytes',
# 'expires': 'Thu, 04 Jun 2009 02:21:41 GMT',
# 'vary': 'Accept-Encoding',
# 'server': 'Apache',
# 'last-modified': 'Wed, 03 Jun 2009 02:20:15 GMT',
# 'connection': 'close',
# '-content-encoding': 'gzip',                                         #③
# 'etag': '"bfe-4cbbf5c0"',
# 'cache-control': 'max-age=86400',                                    #④
# 'date': 'Wed, 03 Jun 2009 02:21:41 GMT',
# 'content-type': 'application/xml'}
```

① The `response` you get back from this single call to the `request()` method is the response from the final url.

② `httplib2` adds the final URL to the `response` dictionary, as `content-location`. This is not a header that came from the server; it's specific to `httplib2`.

③ Apropos of nothing, this feed is compressed.

④ And cacheable. (This is important, as you'll see in a minute.)

The `response` you get back gives you information about the *final* URL. What if you want more information about the intermediate URLs, the ones that eventually redirected to the final URL? `httplib2` lets you do that, too.

```
# continued from the previous example
response.previous                                                     #①
#{'status': '302',
# 'content-length': '228',
# 'content-location': 'http://diveintopython3.org/examples/feed-302.xml',
# 'expires': 'Thu, 04 Jun 2009 02:21:41 GMT',
# 'server': 'Apache',
# 'connection': 'close',
# 'location': 'http://diveintopython3.org/examples/feed.xml',
# 'cache-control': 'max-age=86400',
# 'date': 'Wed, 03 Jun 2009 02:21:41 GMT',
# 'content-type': 'text/html; charset=iso-8859-1'}

type(response)                                                        #②
#<class 'httplib2.Response'>

type(response.previous)
#<class 'httplib2.Response'>

response.previous.previous                                          #③
```

① The `response.previous` attribute holds a reference to the previous `response` object that `httplib2` followed to get to the current response object.

② Both `response` and `response.previous` are `httplib2.Response` objects.

③ That means you can check `response.previous.previous` to follow the redirect chain backwards even further. (Scenario: one URL redirects to a second URL which redirects to a third URL. It could happen!) In this case, we've already reached the beginning of the redirect chain, so the attribute is `None`.

What happens if you request the same URL again?

```
# continued from the previous example
response2, content2 = h.request('http://diveintopython3.org/examples/feed-302.xml')   #①
#connect: (diveintopython3.org, 80)
#send: b'GET /examples/feed-302.xml HTTP/1.1                                           #②
#Host: diveintopython3.org
#accept-encoding: deflate, gzip
#user-agent: Python-httplib2/$Rev: 259 $'
#reply: 'HTTP/1.1 302 Found'                                                           #③

content2 == content                                                                    #④
#True
```

① Same URL, same `httplib2.Http` object (and therefore the same cache).

② The `302` response was not cached, so `httplib2` sends another request for the same URL.

③ Once again, the server responds with a `302`. But notice what *didn't* happen: there wasn't ever a second request for the final URL, `http://diveintopython3.org/examples/feed.xml`. That response was cached (remember the `Cache-Control` header that you saw in the previous example). Once `httplib2` received the `302 Found` code, *it checked its cache before issuing another request*. The cache contained a fresh copy of `http://diveintopython3.org/examples/feed.xml`, so there was no need to re-request it.

④ By the time the `request()` method returns, it has read the feed data from the cache and returned it. Of course, it's the same as the data you received last time.

In other words, you don't have to do anything special for temporary redirects.

httplib2 will follow them automatically, and the fact that one URL redirects

to another has no bearing on httplib2's support for compression, caching, ETags, or any of the other features of HTTP.

Permanent redirects are just as simple.

```
# continued from the previous example
response, content = h.request('http://diveintopython3.org/examples/feed-301.xml')  #①
#connect: (diveintopython3.org, 80)
#send: b'GET /examples/feed-301.xml HTTP/1.1
#Host: diveintopython3.org
#accept-encoding: deflate, gzip
#user-agent: Python-httplib2/$Rev: 259 $'
#reply: 'HTTP/1.1 301 Moved Permanently'                                           #②

response.fromcache                                                                 #③
#True
```

① Once again, this URL doesn't really exist. I've set up my server to issue a permanent redirect to http://diveintopython3.org/examples/feed.xml.

② And here it is: status code 301. But again, notice what didn't happen: there was no request to the redirect URL. Why not? Because it's already cached locally.

③ httplib2 "followed" the redirect right into its cache.

But wait! There's more!

```
# continued from the previous example
response2, content2 = h.request('http://diveintopython3.org/examples/feed-301.xml') #①
response2.fromcache                                                                 #②
#True

content2 == content                                                                 #③
#True
```

① Here's the difference between temporary and permanent redirects: once httplib2 follows a permanent redirect, all further requests for that URL will transparently be rewritten to the target URL without hitting the network for the original URL. Remember, debugging is still turned on, yet there is no output of network activity whatsoever.

② Yep, this response was retrieved from the local cache.

③ Yep, you got the entire feed (from the cache).

HTTP. It works.