# Features of HTTP

WE'LL COVER THE FOLLOWING

There are five important features which all HTTP clients should support.

## Caching #

The most important thing to understand about any type of web service is that network access is incredibly expensive. I don't mean "dollars and cents" expensive (although bandwidth ain't free). I mean that it takes an extraordinary long time to open a connection, send a request, and retrieve a response from a remote server. Even on the fastest broadband connection, *latency* (the time it takes to send a request and start retrieving data in a response) can still be higher than you anticipated. A router misbehaves, a packet is dropped, an intermediate proxy is under attack — there's never a dull moment on the public internet, and there may be nothing you can do about it.

Cache-Control: max-age means "don't bug me until next week."

HTTP is designed with caching in mind. There is an entire class of devices (called "caching proxies") whose only job is to sit between you and the rest of the world and minimize network access. Your company or ISP almost certainly maintains caching proxies, even if you're unaware of them. They work because caching is built into the HTTP protocol.

Here’s a concrete example of how caching works. You visit diveintomark.org in your browser. That page includes a background image, wearehugh.com/m.jpg. When your browser downloads that image, the server includes the following HTTP headers:

```
HTTP/1.1 200 OK
Date: Sun, 31 May 2009 17:14:04 GMT
Server: Apache
Last-Modified: Fri, 22 Aug 2008 04:28:16 GMT
ETag: "3075-ddc8d800"
Accept-Ranges: bytes
Content-Length: 12405
Cache-Control: max-age=31536000, public
Expires: Mon, 31 May 2010 17:14:04 GMT
Connection: close
Content-Type: image/jpeg
```

The `Cache-Control` and `Expires` headers tell your browser (and any caching proxies between you and the server) that this image can be cached for up to a year. A *year*! And if, in the next year, you visit another page which also includes a link to this image, your browser will load the image from its cache *without generating any network activity whatsoever*.

But wait, it gets better. Let’s say your browser purges the image from your local cache for some reason. Maybe it ran out of disk space; maybe you manually cleared the cache. Whatever. But the HTTP headers said that this data could be cached by public caching proxies. (Technically, the important thing is what the headers *don’t* say; the `Cache-Control` header doesn’t have the private keyword, so this data is cacheable by default.) Caching proxies are designed to have tons of storage space, probably far more than your local browser has allocated.

If your company or isp maintain a caching proxy, the proxy may still have the image cached. When you visit `diveintomark.org` again, your browser will look in its local cache for the image, but it won’t find it, so it will make a network request to try to download it from the remote server. But if the caching proxy still has a copy of the image, it will intercept that request and serve the image from *its* cache. That means that your request will never reach the remote server; in fact, it will never leave your company’s network. That makes for a faster download (fewer network hops) and saves your company money (less data being downloaded from the outside world).

HTTP caching only works when everybody does their part. On one side, servers need to send the correct headers in their response. On the other side, clients need to understand and respect those headers before they request the same data twice. The proxies in the middle are not a panacea; they can only be as smart as the servers and clients allow them to be.

Python's HTTP libraries do not support caching, but `httplib2` does.

## Last-Modified Checking #

Some data never changes, while other data changes all the time. In between, there is a vast field of data that *might* have changed, but hasn't. CNN.com's feed is updated every few minutes, but my weblog's feed may not change for days or weeks at a time. In the latter case, I don't want to tell clients to cache my feed for weeks at a time, because then when I do actually post something, people may not read it for weeks (because they're respecting my cache headers which said "don't bother checking this feed for weeks"). On the other hand, I don't want clients downloading my entire feed once an hour if it hasn't changed!

304: Not Modified means "same shit, different day."

HTTP has a solution to this, too. When you request data for the first time, the server can send back a `Last-Modified` header. This is exactly what it sounds like: the date that the data was changed. That background image referenced from `diveintomark.org` included a `Last-Modified` header.

```
HTTP/1.1 200 OK
Date: Sun, 31 May 2009 17:14:04 GMT
Server: Apache
Last-Modified: Fri, 22 Aug 2008 04:28:16 GMT
ETag: "3075-ddc8d800"
Accept-Ranges: bytes
Content-Length: 12405
Cache-Control: max-age=31536000, public
Expires: Mon, 31 May 2010 17:14:04 GMT
Connection: close
Content-Type: image/jpeg
```

When you request the same data a second (or third or fourth) time, you can send an `If-Modified-Since` header with your request, with the date you got back from the server last time. If the data has changed since then, then the server gives you the new data with a `200` status code. But if the data *hasn't* changed since then, the server sends back a special HTTP `304` status code, which means "this data hasn't changed since the last time you asked for it." You can test this on the command line, using curl:

```
you@localhost:~$ curl -I -H "If-Modified-Since: Fri, 22 Aug 2008 04:28:16 GMT" http://wearehu
HTTP/1.1 304 Not Modified
Date: Sun, 31 May 2009 18:04:39 GMT
Server: Apache
Connection: close
ETag: "3075-ddc8d800"
Expires: Mon, 31 May 2010 18:04:39 GMT
Cache-Control: max-age=31536000, public
```

Why is this an improvement? Because when the server sends a `304`, *it doesn't re-send the data*. All you get is the status code. Even after your cached copy has expired, last-modified checking ensures that you won't download the same data twice if it hasn't changed. (As an extra bonus, this `304` response also includes caching headers. Proxies will keep a copy of data even after it officially "expires," in the hopes that the data hasn't *really* changed and the next request responds with a `304` status code and updated cache information.)

Python's HTTP libraries do not support last-modified date checking, but `httplib2` does.

## ETag Checking #

ETags are an alternate way to accomplish the same thing as the last-modified checking. With Etags, the server sends a hash code in an `ETag` header along with the data you requested. (Exactly how this hash is determined is entirely up to the server. The only requirement is that it changes when the data changes.) That background image referenced from `diveintomark.org` had an `ETag` header.

```
HTTP/1.1 200 OK
Date: Sun, 31 May 2009 17:14:04 GMT
Server: Apache
Last-Modified: Fri, 22 Aug 2008 04:28:16 GMT
ETag: "3075-ddc8d800"
Accept-Ranges: bytes
```

```
Content-Length: 12405
Cache-Control: max-age=31536000, public
Expires: Mon, 31 May 2010 17:14:04 GMT

Connection: close
Content-Type: image/jpeg
```

The second time you request the same data, you include the ETag hash in an `If-None-Match` header of your request. If the data hasn't changed, the server will send you back a `304` status code. As with the last-modified date checking, the server sends back only the `304` status code; it doesn't send you the same data a second time. By including the ETag hash in your second request, you're telling the server that there's no need to re-send the same data if it still matches this hash, since you still have the data from the last time.

ETag means "there's nothing new under the sun."

Again with the `curl`:

```
you@localhost:~$ curl -I -H "If-None-Match: \"3075-ddc8d800\"" http://wearehugh.com/m.jpg  ①
HTTP/1.1 304 Not Modified
Date: Sun, 31 May 2009 18:04:39 GMT
Server: Apache
Connection: close
ETag: "3075-ddc8d800"
Expires: Mon, 31 May 2010 18:04:39 GMT
Cache-Control: max-age=31536000, public
```

① ETags are commonly enclosed in quotation marks, but *the quotation marks are part of the value*. That means you need to send the quotation marks back to the server in the `If-None-Match` header.

Python's HTTP libraries do not support ETags, but `httplib2` does.

## Compression

When you talk about HTTP web services, you're almost always talking about moving text-based data back and forth over the wire. Maybe it's XML, maybe it's JSON, maybe it's just plain text. Regardless of the format, text compresses well. The example feed in the XML chapter is 3070 bytes uncompressed, but would be 941 bytes after gzip compression. That's just 30% of the original size!

HTTP supports several compression algorithms. The two most common types are gzip and deflate. When you request a resource over HTTP, you can ask the are gzip and deflate. When you request a resource over HTTP, you can ask the server to send it in compressed format. You include an `Accept-encoding`

header in your request that lists which compression algorithms you support. If the server supports any of the same algorithms, it will send you back compressed data (with a `Content-encoding` header that tells you which algorithm it used). Then it’s up to you to decompress the data.

> *Important tip for server-side developers: make sure that the compressed version of a resource has a different Etag than the uncompressed version. Otherwise, caching proxies will get confused and may serve the compressed version to clients that can’t handle it. Read the discussion of Apache bug 39727 for more details on this subtle issue.*

Python’s HTTP libraries do not support compression, but `httplib2` does.

## Redirects #

Cool URIs don’t change, but many URIs are seriously uncool. Web sites get reorganized, pages move to new addresses. Even web services can reorganize. A syndicated feed at `http://example.com/index.xml` might be moved to `http://example.com/xml/atom.xml`. Or an entire domain might move, as an organization expands and reorganizes; `http://www.example.com/index.xml` becomes `http://server-farm-1.example.com/index.xml`.

**Location** means “look over there!”

Every time you request any kind of resource from an HTTP server, the server includes a status code in its response. Status code `200` means “everything’s normal, here’s the page you asked for”. Status code `404` means “page not found”. (You’ve probably seen `404` errors while browsing the web.) Status codes in the 300’s indicate some form of redirection.

HTTP has several different ways of signifying that a resource has moved. The two most common techiques are status codes `302` and `301`. Status code `302` is a *temporary redirect*; it means “oops, that got moved over here temporarily” (and then gives the temporary address in a `Location` header). Status code `301` is a *permanent redirect*; it means “oops, that got moved permanently” (and then gives the new address in a `Location` header). If you get a `302` status code and a new address, the HTTP specification says you should use the new

address to get what you asked for, but the next time you want to access the same resource, you should retry the old address. But if you get a `301` status code and a new address, you're supposed to use the new address from then on.

The `urllib.request` module automatically "follow" redirects when it receives the appropriate status code from the HTTP server, but it doesn't tell you that it did so. You'll end up getting data you asked for, but you'll never know that the underlying library "helpfully" followed a redirect for you. So you'll continue pounding away at the old address, and each time you'll get redirected to the new address, and each time the `urllib.request` module will "helpfully" follow the redirect. In other words, it treats permanent redirects the same as temporary redirects. That means two round trips instead of one, which is bad for the server and bad for you.

`httplib2` handles permanent redirects for you. Not only will it tell you that a permanent redirect occurred, it will keep track of them locally and automatically rewrite redirected URLs before requesting them.