CacheControl Documentation

Release 0.12.3

Eric Larson

May 25, 2017

Contents

1 Install **3**

2 Quick Start **5**

3 Tests **7**

4 Disclaimers **9**

4.1 Using CacheControl 9
4.2 Storing Cache Data 10
4.3 ETag Support 11
4.4 Custom Caching Strategies 13
4.5 Tips and Best Practices 16
4.6 Release Notes 16

5 Indices and tables **19**

CacheControl is a port of the caching algorithms in httplib2 for use with the requests session object.

It was written because httplib2's better support for caching is often mitigated by its lack of thread-safety. The same is true of requests in terms of caching.

CHAPTER 1

Install

CacheControl is available from PyPI. You can install it with pip

```
$ pip install CacheControl
```

Some of the included cache storage classes have external requirements. See *Storing Cache Data* for more info.

CHAPTER 2

Quick Start

For the impatient, here is how to get started using CacheControl:

```
import requests

from cachecontrol import CacheControl


sess = requests.session()
cached_sess = CacheControl(sess)

response = cached_sess.get('http://google.com')
```

This uses a thread-safe in-memory dictionary for storage.

CHAPTER 3

Tests

The tests are all in `cachecontrol/tests` and are runnable by `py.test`.

CHAPTER 4

Disclaimers

CacheControl is relatively new and might have bugs. I have made an effort to faithfully port the tests from httplib2 to CacheControl, but there is a decent chance that I've missed something. Please file bugs if you find any issues!

With that in mind, CacheControl has been used sucessfully in production environments, replacing httplib2's usage.

If you give it a try, please let me know of any issues.

Contents:

Using CacheControl

CacheControl assumes you are using a *requests.Session* for your requests. If you are making ad-hoc requests using *requests.get* then you probably are not terribly concerned about caching.

There are two way to use CacheControl, via the wrapper and the adapter.

Wrapper

The easiest way to use CacheControl is to utilize the basic wrapper. Here is an example:

```
import requests
import cachecontrol

sess = cachecontrol.CacheControl(requests.Session())
resp = sess.get('http://google.com')
```

This uses the default cache store, a thread safe in-memory dictionary.

Adapter

The other way to use CacheControl is via a requests Transport Adapter.

Here is how the adapter works:

```
import requests
import cachecontrol

sess = requests.Session()
sess.mount('http://', CacheControlAdapter())

resp = sess.get('http://google.com')
```

Under the hood, the wrapper method of using CacheControl mentioned above is the same as this example.

Use a Different Cache Store

Both the wrapper and adapter classes allow providing a custom cache store object for storing your cached data. Here is an example using the provided *FileCache* from CacheControl:

```
import requests

from cachecontrol import CacheControl

# NOTE: This requires lockfile be installed
from cachecontrol.caches import FileCache

sess = CacheControl(requests.Session(),
                    cache=FileCache('.webcache'))
```

The *FileCache* will create a directory called *.webcache* and store a file for each cached request.

Storing Cache Data

CacheControl comes with a few storage backends for storing your cache'd objects.

DictCache

The *DictCache* is the default cache used when no other is provided. It is a simple threadsafe dictionary. It doesn't try to do anything smart about deadlocks or forcing a busted cache, but it should be reasonably safe to use.

Also, the *DictCache* does not transform the request or response objects in anyway. Therefore it is unlikely you could persist the entire cache to disk. The converse is that it should be very fast.

FileCache

The *FileCache* is similar to the caching mechanism provided by httplib2. It requires lockfile be installed as it prevents multiple threads from writing to the same file at the same time.

Note: Note that you can install this dependency automatically with pip by requesting the *filecache* extra:

```
pip install cachecontrol[filecache]
```

Here is an example using the *FileCache*:

```
import requests
from cachecontrol import CacheControl
from cachecontrol.caches.file_cache import FileCache

sess = CacheControl(requests.Session(),
                    cache=FileCache('.web_cache'))
```

The *FileCache* supports a *forever* flag that disables deleting from the cache. This can be helpful in debugging applications that make many web requests that you don't want to repeat. It also can be helpful in testing. Here is an example of how to use it:

```
forever_cache = FileCache('.web_cache', forever=True)
sess = CacheControl(requests.Session(), forever_cache)
```

A Note About Pickle It should be noted that the *FileCache* uses pickle to store the cached response. Prior to requests 2.1, *requests.Response* objects were not 'pickleable' due to the use of *IOBase* base classes in *urllib3 HTTPResponse* objects. In CacheControl we work around this by patching the Response objects with the appropriate *__getstate__* and *__setstate__* methods when the requests version doesn't natively support Response pickling.

RedisCache

The *RedisCache* uses a Redis database to store values. The values are stored as strings in redis, which means the get, set and delete actions are used. It requires the redis library to be installed.

Note: Note that you can install this dependency automatically with pip by requesting the *redis* extra:

```
pip install cachecontrol[redis]
```

The *RedisCache* also provides a clear method to delete all keys in a database. Obviously, this should be used with caution as it is naive and works iteratively, looping over each key and deleting it.

Here is an example using a *RedisCache*:

```
import redis
import requests
from cachecontrol import CacheControl
from cachecontrol.caches.redis_cache import RedisCache


pool = redis.ConnectionPool(host='localhost', port=6379, db=0)
r = redis.Redis(connection_pool=pool)
sess = CacheControl(requests.Session(), RedisCache(r))
```

This is primarily a proof of concept, so please file bugs if there is a better method for utilizing redis as a cache.

ETag Support

CacheControl's support of ETags is slightly different than httplib2. In httplib2, an ETag is considered when using a cached response when the cache is considered stale. When a cached response is expired and it has an ETag header, it returns a response with the appropriate *If-None-Match* header. We'll call this behavior a **Time Priority** cache as the ETag support only takes effect when the time has expired.

In CacheControl the default behavior when an ETag is sent by the server is to cache the response. We'll refer to this pattern as a **Equal Priority** cache as the decision to cache is either time base or due to the presense of an ETag.

The spec is not explicit what takes priority when caching with both ETags and time based headers. Therefore, CacheControl supports the different mechanisms via configuration where possible.

Turning Off Equal Priority Caching

The danger in Equal Priority Caching is that a server that returns ETag headers for every request may fill up your cache. You can disable Equal Priority Caching and utilize a Time Priority algorithm like httplib2.

```
import requests
from cachecontrol import CacheControl

sess = CacheControl(requests.Session(), cache_etags=False)
```

This will only utilize ETags when they exist within the context of time based caching headers. If a response has time base caching headers that are valid along with an ETag, we will still attempt to handle a 304 Not Modified even though the cached value as expired. Here is a simple example.

```
# Server response
GET /foo.html
Date: Tue, 26 Nov 2013 00:50:49 GMT
Cache-Control: max-age=3000
ETag: JAsUYM8K
```

On a subsequent request, if the cache has expired, the next request will still include the *If-None-Match* header. The cached response will remain in the cache awaiting the response.

```
# Client request
GET /foo.html
If-None-Match: JAsUYM8K
```

If the server returns a *304 Not Modified*, it will use the stale cached value, updating the headers from the most recent request.

```
# Server response
GET /foo.html
Date: Tue, 26 Nov 2013 01:30:19 GMT
Cache-Control: max-age=3000
ETag: JAsUYM8K
```

If the server returns a *200 OK*, the cache will be updated accordingly.

Equal Priority Caching Benefits

The benefits of equal priority caching is that you have two orthogonal means of introducing a cache. The time based cache provides an effective way to reduce the load on requests that can be eventually consistent. Static resource are a great example of when time based caching is effective.

The ETag based cache is effective for working with documents that are larger and/or need to be correct immediately after changes. For example, if you exported some data from a large database, the file could be 10 GBs. Being able to send an ETag with this sort of request an know the version you have locally is valid saves a ton of bandwidth and time.

Likewise, if you have a resource that you want to update, you can be confident there will not be a lost update because you have local version that is stale.

Endpoint Specific Caching

It should be pointed out that there are times when an endpoint is specifically tailored for different caching techniques. If you have a RESTful service, there might be endpoints that are specifically meant to be cached via time based caching techniques where as other endpoints should focus on using ETags. In this situation it is recommended that you use the *CacheControlAdapter* directly.

```
import requests
from cachecontrol import CacheControlAdapter
from cachecontrol.caches import RedisCache

# using django for an idea on where you might get a
# username/password.
from django.conf import settings

# a function to return a redis connection all the instances of the
# app may use. this allows updates to the API (ie PUT) to invalidate
# the cache for other users.
from myapp.db import redis_connection


# create our session
client = sess.Session(auth=(settings.user, settings.password))

# we have a gettext like endpoint. this doesn't get updated very
# often so a time based cache is a helpful way to reduce many small
# requests.
client.mount('http://myapi.foo.com/gettext/',
             CacheControlAdapter(cache_etags=False))


# here we have user profile endpoint that lets us update information
# about users. we need this to be consistent immediately after a user
# updates some information because another node might handle the
# request. It uses the global redis cache to coordinate the cache and
# uses the equal priority caching to be sure etags are used by default.
redis_cache = RedisCache(redis_connection())
client.mount('http://myapi.foo.com/user_profiles/',
             CacheControlAdapter(cache=redis_cache))
```

Hopefully this more indepth example reveals how to configure a *requests.Session* to better utilize ETag based caching vs. Time Priority Caching.

Custom Caching Strategies

There are times when a server provides responses that are logically cacheable, but they lack the headers necessary to cause CacheControl to cache the response. The HTTP Caching Spec does allow for caching systems to cache requests that lack caching headers. In these situations, the caching system can use heuristics to determine an appropriate amount of time to cache a response.

By default, in CacheControl the decision to cache must be explicit by default via the caching headers. When there is a need to cache responses that wouldn't normally be cached, a user can provide a heuristic to adjust the response in order to make it cacheable.

For example when running a test suite against a service, caching all responses might be helpful speeding things up while still making real calls to the API.

Caching Heuristics

A cache heuristic allows specifying a caching strategy by adjusting response headers before the response is considered for caching.

For example, if we wanted to implement a caching strategy where every request should be cached for a week, we can implement the strategy in a *cachecontrol.heuristics.Heuristic*.

```
import calendar
from cachecontrol.heuristics import BaseHeuristic
from datetime import datetime, timedelta
from email.utils import parsedate, formatdate


class OneWeekHeuristic(BaseHeuristic):

    def update_headers(self, response):
        date = parsedate(response.headers['date'])
        expires = datetime(*date[:6]) + timedelta(weeks=1)
        return {
            'expires' : formatdate(calendar.timegm(expires.timetuple())),
            'cache-control' : 'public',
        }

    def warning(self, response):
        msg = 'Automatically cached! Response is Stale.'
        return '110 - "%s"' % msg
```

When a response is received and we are testing for whether it is cacheable, the heuristic is applied before checking its headers. We also set a warning header to communicate why the response might be stale. The original response is passed into the warning header in order to use its values. For example, if the response has been expired for more than 24 hours a Warning 113 should be used.

In order to use this heuristic, we pass it to our *CacheControl* constructor.

```
from requests import Session
from cachecontrol import CacheControl


sess = CacheControl(Session(), heuristic=OneWeekHeuristic())
sess.get('http://google.com')
r = sess.get('http://google.com')
assert r.from_cache
```

The google homepage specifically uses a negative expires header and private cache control header to avoid caches. We've managed to work around that aspect and cache the response using our heuristic.

Best Practices

Cache heuristics are still a new feature, which means that the support is somewhat rudimentary. There likely to be best practices and common heuristics that can meet the needs of many use cases. For example, in the above heuristic it is important to change both the *expires* and *cache-control* headers in order to make the response cacheable.

If you do find a helpful best practice or create a helpful heuristic, please consider sending a pull request or opening a issue.

Expires After

CacheControl bundles an *ExpiresAfter* heuristic that is aimed at making it relatively easy to automatically cache responses for a period of time. Here is an example

```
import requests
from cachecontrol import CacheControlAdapter
from cachecontrol.heuristics import ExpiresAfter

adapter = CacheControlAdapter(heuristic=ExpiresAfter(days=1))

sess = requests.Session()
sess.mount('http://', adapter)
```

The arguments are the same as the *datetime.timedelta* object. *ExpiresAfter* will override or add the *Expires* header and override or set the *Cache-Control* header to *public*.

Last Modified

CacheControl bundles an *LastModified* heuristic that emulates the behavior of Firefox, following RFC7234. Roughly stated, this sets the expiration on a page to 10% of the difference between the request timestamp and the last modified timestamp. This is capped at 24-hr.

```
import requests
from cachecontrol import CacheControlAdapter
from cachecontrol.heuristics import LastModified

adapter = CacheControlAdapter(heuristic=LastModified())

sess = requests.Session()
sess.mount('http://', adapter)
```

Site Specific Heuristics

If you have a specific domain that you want to apply a specific heuristic to, use a separate adapter.

```
import requests
from cachecontrol import CacheControlAdapter
from mypkg import MyHeuristic


sess = requests.Session()
sess.mount(
    'http://my.specific-domain.com',
    CacheControlAdapter(heuristic=MyHeuristic())
)
```

In this way you can limit your heuristic to a specific site.

Warning!

Caching is hard and while HTTP does a reasonable job defining rules for freshness, overriding those rules should be done with caution. Many have been frustrated by over aggressive caches, so please carefully consider your use case before utilizing a more aggressive heuristic.

Tips and Best Practices

Caching is hard! It is considered one of the great challenges of computer science. Fortunatley, the HTTP spec helps to navigate some pitfalls of invalidation using stale responses. Below are some suggestions and best practices to help avoid the more subtle issues that can crop up using CacheControl and HTTP caching.

If you have a suggetions please create a new issue in github and let folks know what you ran into and how you fixed it.

Timezones

It is important to remember that the times reported by a server may or may not be timezone aware. If you are using CacheControl with a service you control, make sure any timestamps are used consistently, especially if requests might cross timezones.

Cached Responses

We've done our best to make sure cached responses act like a normal response, but there are aspects that are different for somewhat obvious reasons.

- Cached responses are never streaming
- Cached responses have *None* for the *raw* attribute

Obviously, when you cache a response, you have downloaded the entire body. Therefore, there is never a use case for streaming a cached response.

With that in mind, you should be aware that if you try to cache a very large response on a network store, you still might have some latency tranferring the data from the network store to your application. Another consideration is storing large responses in a *FileCache*. If you are caching using ETags and the server is extremely specific as to what constitutes an equivalent request, it could provide many different responses for essentially the same data within the context of your application.

Query String Params

If you are caching requests that use a large number of query string parameters, consider sorting them to ensure that the request is properly cached.

Requests supports passing both dictionaries and lists of tuples as the param argument in a request. For example:

```
requests.get(url, params=sorted([('foo', 'one'), ('bar', 'two')]))
```

By ordering your params, you can be sure the cache key will be consistent across requests and you are caching effectively.

Release Notes

0.12.0

Rather than using compressed JSON for caching values, we are now using MessagePack (http://msgpack.org/). MessagePack has the advantage that that serialization and deserialization is faster, especially for caching large binary payloads.

0.11.2

This release introduces the *cachecontrol.heuristics.LastModified* heuristic. This uses the same behaviour as many browsers to base expiry on the *Last-Modified* header when no explicit expiry is provided.

0.11.0

The biggest change is the introduction of using compressed JSON rather than pickle for storing cached values. This allows Python 3.4 and Python 2.7 to use the same cache store. Previously, if a cache was created on 3.4, a 2.7 client would fail loading it, causing an invalid cache miss. Using JSON also avoids the exec call used in pickle, making the cache more secure by avoiding a potential code injection point. Finally, the compressed JSON is a smaller payload, saving a bit of space.

In order to support arbitrary binary data in the JSON format, base64 encoding is used to turn the data into strings. It has to do some encoding dances to make sure that the bytes/str types are correct, so **please** open a new issue if you notice any issues.

This release also introduces the *cachecontrol.heuristics.ExpiresAfter* heuristic. This allows passing in arguments like a *datetime.timedelta* in order to configure that all responses are cached for the specific period of time.

0.10.0

This is an important release as it changes what is actually cached. Rather than caching requests' Response objects, we are now caching the underlying urllib3 response object. Also, the response will not be cached unless the response is actually consumed by the user.

These changes allowed the reintroduction of .raw support.

Huge thanks goes out to @dstufft for these excellent patches and putting so much work into CacheControl to allow cached responses to behave exactly as a normal response.

- FileCache Updates (via @dstufft)
  - files are now hashed via sha-2
  - files are stored in a namespaced directory to avoid hitting os limits on the number of files in a directory.
  - use the io.BytesIO when reading / writing (via @alex)
- #19 Allow for a custom controller via @cournape
- #17 use highest protocol version for pickling via @farwayer
- #16 FileCache: raw field serialization via @farwayer

0.9.3

- #16: All cached responses get None for a raw attribute.
- #13 Switched to md5 encoded keys in file cache (via @mxjeff)
- #11 Fix timezones in tests (via @kaliko)

CHAPTER 5

Indices and tables

- genindex
- modindex
- search