requests-cache Documentation

Release 0.4.13

Roman Haritonov

Nov 09, 2017

Contents

Requests-cache is a transparent persistent cache for requests (version >= 1.1.0) library.

Source code and issue tracking can be found at GitHub.

Contents:

CHAPTER 1

User guide

1.1 Installation

Install with pip or easy_install:

```
pip install --upgrade requests-cache
```

or download latest version from version control:

```
git clone git://github.com/reclosedev/requests-cache.git
cd requests-cache
python setup.py install
```

Warning: Version updates of `requests`, `urllib3` or `requests_cache` itself can break existing cache database (see https://github.com/reclosedev/requests-cache/issues/56). So if your code relies on cache, or is expensive in terms of time and traffic, please be sure to use something like `virtualenv` and pin your requirements.

1.2 Usage

There is two ways of using `requests_cache`:

- Using *`CachedSession`* instead `requests.Session`
- Monkey patching `requests` to use *`CachedSession`* by default

Monkey-patching allows to add caching to existent program by adding just two lines:

Import `requests_cache` and call *`install_cache()`*

```
import requests
import requests_cache
```

```
requests_cache.install_cache()
```

And you can use `requests`, all responses will be cached transparently!

For example, following code will take only 1-2 seconds instead 10:

```
for i in range(10):
    requests.get('http://httpbin.org/delay/1')
```

Cache can be configured with some options, such as cache filename, backend (sqlite, mongodb, redis, memory), expiration time, etc. E.g. cache stored in sqlite database (default format) named `'test_cache.sqlite'` with expiration set to 300 seconds can be configured as:

```
requests_cache.install_cache('test_cache', backend='sqlite', expire_after=300)
```

See also:

Full list of options can be found in *`requests_cache.install_cache()`* reference

Transparent caching is achieved by monkey-patching `requests` library It is possible to uninstall this patch with *`requests_cache.uninstall_cache()`*.

Also, you can use *`requests_cache.disabled()`* context manager for temporary disabling caching:

```
with requests_cache.disabled():
    print(requests.get('http://httpbin.org/ip').text)
```

If `Response` is taken from cache, `from_cache` attribute will be `True`:

```
>>> import requests
>>> import requests_cache
>>> requests_cache.install_cache()
>>> requests_cache.clear()
>>> r = requests.get('http://httpbin.org/get')
>>> r.from_cache
False
>>> r = requests.get('http://httpbin.org/get')
>>> r.from_cache
True
```

It can be used, for example, for request throttling with help of `requests` hook system:

```
import time
import requests
import requests_cache

def make_throttle_hook(timeout=1.0):
    """
    Returns a response hook function which sleeps for `timeout` seconds if
    response is not cached
    """
    def hook(response, *args, **kwargs):
        if not getattr(response, 'from_cache', False):
            print('sleeping')
            time.sleep(timeout)
        return response
    return hook
```

```
if __name__ == '__main__':
    requests_cache.install_cache('wait_test')
    requests_cache.clear()

    s = requests_cache.CachedSession()
    s.hooks = {'response': make_throttle_hook(0.1)}
    s.get('http://httpbin.org/delay/get')
    s.get('http://httpbin.org/delay/get')
```

See also:

example.py

Note: requests_cache prefetchs response content, be aware if your code uses streaming requests.

1.3 Persistence

`requests_cache` designed to support different backends for persistent storage. By default it uses `sqlite` database. Type of storage can be selected with `backend` argument of *`install_cache()`*.

List of available backends:

- `'sqlite'` - sqlite database (**default**)
- `'memory'` - not persistent, stores all data in Python `dict` in memory
- `'mongodb'` - (**experimental**) MongoDB database (`pymongo < 3.0` required)
- `'redis'` - stores all data on a redis data store (`redis` required)

You can write your own and pass instance to *`install_cache()`* or *`CachedSession`* constructor. See *Cache backends* API documentation and sources.

1.4 Expiration

If you are using cache with `expire_after` parameter set, responses are removed from the storage only when the same request is made. Since the store sizes can get out of control pretty quickly with expired items you can remove them using *`remove_expired_responses()`* or *`BaseCache.remove_old_entries(created_before)`*.

```
expire_after = timedelta(hours=1)
requests_cache.install_cache(expire_after=expire_after)
...
requests_cache.remove_expired_responses()
# or
remove_old_entries.get_cache().remove_old_entries(datetime.utcnow() - expire_after)
# when used as session
session = CachedSession(..., expire_after=expire_after)
...
session.cache.remove_old_entries(datetime.utcnow() - expire_after)
```

For more information see *API reference*.

CHAPTER 2

API

This part of the documentation covers all the interfaces of *requests-cache*

2.1 Public api

2.1.1 requests_cache.core

Core functions for configuring cache and monkey patching `requests`

class `requests_cache.core.`**`CachedSession`**(*cache_name='cache'*, *backend=None*, *expire_after=None*, *allowable_codes=(200,)*, *allowable_methods=('GET',)*, *old_data_on_error=False*, ***backend_options*)

Requests `Sessions` with caching support.

Parameters

- **`cache_name`** – for `sqlite` backend: cache file will start with this prefix, e.g `cache.sqlite`

  for `mongodb`: it's used as database name

  for `redis`: it's used as the namespace. This means all keys are prefixed with `'cache_name:'`

- **`backend`** – cache backend name e.g `'sqlite'`, `'mongodb'`, `'redis'`, `'memory'`. (see *Persistence*). Or instance of backend implementation. Default value is `None`, which means use `'sqlite'` if available, otherwise fallback to `'memory'`.

- **`expire_after`** (*`float`*) – `timedelta` or number of seconds after cache will be expired or *None* (default) to ignore expiration

- **`allowable_codes`** (*`tuple`*) – limit caching only for response with this codes (default: 200)

- **`allowable_methods`** (*`tuple`*) – cache only requests of this methods (default: 'GET')

- **backend_options** – options for chosen backend. See corresponding *sqlite*, *mongo* and *redis* backends API documentation
- **include_get_headers** – If *True* headers will be part of cache key. E.g. after get('some_link', headers={'Accept':'application/json'}) get('some_link', headers={'Accept':'application/xml'}) is not from cache.
- **ignored_parameters** – List of parameters to be excluded from the cache key. Useful when requesting the same resource through different credentials or access tokens, passed as parameters.
- **old_data_on_error** – If *True* it will return expired cached response if update fails

cache_disabled(**args*, ***kwds*)

Context manager for temporary disabling cache

```
>>> s = CachedSession()
>>> with s.cache_disabled():
...     s.get('http://httpbin.org/ip')
```

remove_expired_responses()

Removes expired responses from storage

requests_cache.core.**install_cache**(*cache_name='cache'*, *backend=None*, *expire_after=None*, *allowable_codes=(200,)*, *allowable_methods=('GET',)*, *session_factory=<class 'requests_cache.core.CachedSession'>*, ***backend_options*)

Installs cache for all `Requests` requests by monkey-patching `Session`

Parameters are the same as in *CachedSession*. Additional parameters:

Parameters **session_factory** – Session factory. It must be class which inherits *CachedSession* (default)

requests_cache.core.**configure**(*cache_name='cache'*, *backend=None*, *expire_after=None*, *allowable_codes=(200,)*, *allowable_methods=('GET',)*, *session_factory=<class 'requests_cache.core.CachedSession'>*, ***backend_options*)

Installs cache for all `Requests` requests by monkey-patching `Session`

Parameters are the same as in *CachedSession*. Additional parameters:

Parameters **session_factory** – Session factory. It must be class which inherits *CachedSession* (default)

requests_cache.core.**uninstall_cache**()

Restores `requests.Session` and disables cache

requests_cache.core.**disabled**(**args*, ***kwds*)

Context manager for temporary disabling globally installed cache

Warning: not thread-safe

```
>>> with requests_cache.disabled():
...     requests.get('http://httpbin.org/ip')
...     requests.get('http://httpbin.org/get')
```

requests_cache.core.**enabled**(**args*, ***kwds*)

Context manager for temporary installing global cache.

Accepts same arguments as *install_cache()*

Warning: not thread-safe

```
>>> with requests_cache.enabled('cache_db'):
...     requests.get('http://httpbin.org/get')
```

requests_cache.core.**get_cache**()
Returns internal cache object from globally installed CachedSession

requests_cache.core.**clear**()
Clears globally installed cache

requests_cache.core.**remove_expired_responses**()
Removes expired responses from storage

2.2 Cache backends

2.2.1 requests_cache.backends.base

Contains BaseCache class which can be used as in-memory cache backend or extended to support persistence.

class requests_cache.backends.base.**BaseCache**(**args*, ***kwargs*)
Base class for cache implementations, can be used as in-memory cache.

To extend it you can provide dictionary-like objects for *keys_map* and *responses* or override public methods.

keys_map = None
key -> *key_in_responses* mapping

responses = None
key_in_cache -> *response* mapping

save_response(*key*, *response*)
Save response to cache

Parameters

- **key** – key for this response
- **response** – response to save

Note: Response is reduced before saving (with *reduce_response()*) to make it picklable

add_key_mapping(*new_key*, *key_to_response*)
Adds mapping of *new_key* to *key_to_response* to make it possible to associate many keys with single response

Parameters

- **new_key** – new key (e.g. url from redirect)
- **key_to_response** – key which can be found in *responses*

Returns

get_response_and_time(*key*, *default=(None*, *None)*)
Retrieves response and timestamp for *key* if it's stored in cache, otherwise returns *default*

Parameters

- **key** – key of resource
- **default** – return this if *key* not found in cache

Returns tuple (response, datetime)

Note: Response is restored after unpickling with *restore_response()*

---

delete(*key*)
Delete *key* from cache. Also deletes all responses from response history

delete_url(*url*)
Delete response associated with *url* from cache. Also deletes all responses from response history. Works only for GET requests

clear()
Clear cache

remove_old_entries(*created_before*)
Deletes entries from cache with creation time older than `created_before`

has_key(*key*)
Returns *True* if cache has *key*, *False* otherwise

has_url(*url*)
Returns *True* if cache has *url*, *False* otherwise. Works only for GET request urls

reduce_response(*response*, *seen=None*)
Reduce response object to make it compatible with `pickle`

restore_response(*response*, *seen=None*)
Restore response object after unpickling

2.2.2 requests_cache.backends.sqlite

`sqlite3` cache backend

class `requests_cache.backends.sqlite.`**DbCache**(*location='cache'*, *fast_save=False*, *extension='.sqlite'*, ***options*)
sqlite cache backend.

Reading is fast, saving is a bit slower. It can store big amount of data with low memory usage.

Parameters

- **location** – database filename prefix (default: `'cache'`)
- **fast_save** – Speedup cache saving up to 50 times but with possibility of data loss. See *backends.DbDict* for more info
- **extension** – extension for filename (default: `'.sqlite'`)

2.2.3 requests_cache.backends.mongo

`mongo` cache backend

class `requests_cache.backends.mongo.`**`MongoCache`**(*db_name='requests-cache'*, ***options*)
`mongo` cache backend.

Parameters

- **`db_name`** – database name (default: `'requests-cache'`)
- **`connection`** – (optional) `pymongo.Connection`

2.2.4 requests_cache.backends.redis

`redis` cache backend

class `requests_cache.backends.redis.`**`RedisCache`**(*namespace='requests-cache'*, ***options*)
`redis` cache backend.

Parameters

- **`namespace`** – redis namespace (default: `'requests-cache'`)
- **`connection`** – (optional) `redis.StrictRedis`

2.3 Internal modules which can be used outside

2.3.1 requests_cache.backends.dbdict

Dictionary-like objects for saving large data sets to *sqlite* database

class `requests_cache.backends.storage.dbdict.`**`DbDict`**(*filename*, *table_name='data'*, *fast_save=False*, ***options*)
DbDict - a dictionary-like object for saving large datasets to *sqlite* database

It's possible to create multiply DbDict instances, which will be stored as separate tables in one database:

```
d1 = DbDict('test', 'table1')
d2 = DbDict('test', 'table2')
d3 = DbDict('test', 'table3')
```

all data will be stored in `test.sqlite` database into correspondent tables: `table1`, `table2` and `table3`

Parameters

- **`filename`** – filename for database (without extension)
- **`table_name`** – table name
- **`fast_save`** – If it's True, then sqlite will be configured with "PRAGMA synchronous = 0;" to speedup cache saving, but be careful, it's dangerous. Tests showed that insertion order of records can be wrong with this option.

`can_commit` = None
Transactions can be commited if this property is set to *True*

commit (*force=False*)
Commits pending transaction if *can_commit* or *force* is *True*

Parameters **force** – force commit, ignore *can_commit*

bulk_commit (**args*, ***kwds*)
Context manager used to speedup insertion of big number of records

```
>>> d1 = DbDict('test')
>>> with d1.bulk_commit():
...     for i in range(1000):
...         d1[i] = i * 2
```

class requests_cache.backends.storage.dbdict.**DbPickleDict** (*filename*, *table_name='data'*, *fast_save=False*, ***options*)
Same as *DbDict*, but pickles values before saving

Parameters

- **filename** – filename for database (without extension)
- **table_name** – table name
- **fast_save** – If it's True, then sqlite will be configured with "PRAGMA synchronous = 0;" to speedup cache saving, but be careful, it's dangerous. Tests showed that insertion order of records can be wrong with this option.

2.3.2 requests_cache.backends.mongodict

Dictionary-like objects for saving large data sets to `mongodb` database

class requests_cache.backends.storage.mongodict.**MongoDict** (*db_name*, *collection_name='mongo_dict_data'*, *connection=None*)
MongoDict - a dictionary-like interface for `mongo` database

Parameters

- **db_name** – database name (be careful with production databases)
- **collection_name** – collection name (default: mongo_dict_data)
- **connection** – `pymongo.Connection` instance. If it's `None` (default) new connection with default options will be created

class requests_cache.backends.storage.mongodict.**MongoPickleDict** (*db_name*, *collection_name='mongo_dict_data'*, *connection=None*)
Same as *MongoDict*, but pickles values before saving

Parameters

- **db_name** – database name (be careful with production databases)
- **collection_name** – collection name (default: mongo_dict_data)
- **connection** – `pymongo.Connection` instance. If it's `None` (default) new connection with default options will be created

2.3.3 requests_cache.backends.redisdict

Dictionary-like objects for saving large data sets to `redis` key-store

class `requests_cache.backends.storage.redisdict.`**`RedisDict`**(*namespace*, *collection_name='redis_dict_data'*, *connection=None*)

RedisDict - a dictionary-like interface for `redis` key-stores

The actual key name on the redis server will be `namespace:collection_name`

In order to deal with how redis stores data/keys, everything, i.e. keys and data, must be pickled.

Parameters

- **`namespace`** – namespace to use
- **`collection_name`** – name of the hash map stored in redis (default: redis_dict_data)
- **`connection`** – `redis.StrictRedis` instance. If it's `None` (default), a new connection with default options will be created

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

r

Index

S

U