requests-cache Documentation

Release 0.2.1

Roman Haritonov

January 13, 2013

CONTENTS

Requests-cache is a transparent persistent cache for requests (version <= 0.14.2) library.

Source code and issue tracking can be found at GitHub.

Contents:

CHAPTER

ONE

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
```

1.2 Usage

Just import `requests_cache` and call `configure()`

```
import requests
import requests_cache

requests_cache.configure()
```

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, memory), expiration time, etc. E.g. cache stored in sqlite database (default format) named `'test_cache.sqlite'` with expiration set to 5 minutes can be configured as:

```
requests_cache.configure('test_cache', backend='sqlite', expire_after=5)
```

See Also:

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

Transparent caching is achieved by monkey-patching `requests` library (it can be disabled, see `monkey_patch` argument for `configure()`) , It is possible to undo this patch, and redo it again with `undo_patch()` and `redo_patch()`. But preferable way is to use `requests_cache.disabled()` and `requests_cache.enabled` context managers for temporary disabling and enabling caching:

```
with requests_cache.disabled():
    for i in range(3):
        print(requests.get('http://httpbin.org/ip').text)

with requests_cache.enabled():
    for i in range(10):
        print(requests.get('http://httpbin.org/delay/1').text)
```

Also, you can check if url is present in cache with `requests_cache.has_url()` and delete it with `requests_cache.delete_url()`:

```
>>> import requests
>>> import requests_cache
>>> requests_cache.configure()
>>> requests.get('http://httpbin.org/get')
>>> requests_cache.has_url('http://httpbin.org/get')
True
>>> requests_cache.delete_url('http://httpbin.org/get')
>>> requests_cache.has_url('http://httpbin.org/get')
False
```

New in version 0.1.4: If `Response` is taken from cache, it will have `from_cache` attribute:

```
>>> import requests
>>> import requests_cache
>>> requests_cache.configure()
>>> requests_cache.clear()
>>> r = requests.get('http://httpbin.org/get')
>>> hasattr(r, 'from_cache')
False
>>> r = requests.get('http://httpbin.org/get')
>>> hasattr(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):
        if not hasattr(response, 'from_cache'):
            time.sleep(timeout)
        return response
    return hook

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

    s = requests.Session(hooks={'response': make_throttle_hook(2.0)})
    s.get('http://httpbin.org/get')
    s.get('http://httpbin.org/get')
```

See Also:

example.py

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 `configure()`.

List of available backends:

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

Note: `pymongo` doesn't work fine with gevent which powers `requests.async`, but there is some workarounds, see question on StackOverflow.

Also, you can write your own. See *Cache backends* API documentation and sources.

For more information see *API reference* .

CHAPTER

TWO

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`

`requests_cache.core.`**`configure`**(*cache_name='cache'*, *backend='sqlite'*, *expire_after=None*, *allowable_codes=(200,)*, *allowable_methods=('GET',)*, *monkey_patch=True*, ***backend_options*)

Configure cache storage and patch `requests` library to transparently cache responses

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
- **backend** – cache backend e.g `'sqlite'`, `'mongodb'`, `'memory'`. See *Persistence*
- **expire_after** (*int, float or None*) – number of minutes 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')
- **monkey_patch** – patch `requests.Request.send` if *True* (default), otherwise cache will not work until calling `redo_patch()` or using `enabled()` context manager
- **backend_options** – options for chosen backend. See corresponding *sqlite* and *mongo* backends API documentation

`requests_cache.core.`**`has_url`**(*url*)

Returns *True* if cache has *url*, *False* otherwise

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

Context manager for temporary disabling cache

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

requests_cache.core.**enabled**(**args*, ***kwds*)
Context manager for temporary enabling cache

```
>>> with requests_cache.enabled():
...     request.get('http://httpbin.org/ip')
...     request.get('http://httpbin.org/get')
```

requests_cache.core.**clear**()
Clear cache

requests_cache.core.**undo_patch**()
Undo `requests` monkey patch

requests_cache.core.**redo_patch**()
Redo `requests` monkey patch

requests_cache.core.**get_cache**()
Returns internal cache object

requests_cache.core.**delete_url**(*url*)
Deletes all cache for *url*

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**(*location='memory'*, **args*, ***kwargs*)
Base class for cache implementations, can be used as in-memory cache.

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

url_map = None
url -> key_in_cache mapping

responses = None
key_in_cache -> response mapping

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

Parameters

- **url** – url for this response

Note: urls from history saved automatically

- **response** – response to save

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

---

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

Parameters

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

Returns tuple (response, datetime)

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

---

del_cached_url(*url*)
Delete *url* from cache. Also deletes all urls from response history

clear()
Clear cache

has_url(*url*)
Returns *True* if cache has *url*, *False* otherwise

reduce_response(*response*)
Reduce response object to make it compatible with `pickle`

restore_response(*response*)
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`

2.3 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.dbdict.`**`DbDict`**(*filename*, *table_name='data'*, *fast_save=False*)

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.dbdict.`**`DbPickleDict`**(*filename*, *table_name='data'*, *fast_save=False*)

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.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.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

CHAPTER

THREE

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX

r

requests_cache.backends.base, ??
requests_cache.backends.dbdict, ??
requests_cache.backends.mongo, ??
requests_cache.backends.mongodict, ??
requests_cache.backends.sqlite, ??
requests_cache.core, ??