cache$_{r}equestsDocumentation$

Release 4.0.0

Manu Phatak

December 26, 2015

Contents

CHAPTER 1

Contents:

1.1 cache_requests

Simple. Powerful. Persistent LRU caching for the requests library.

1.1.1 Features

- Documentation: https://cache_requests.readthedocs.org
- Open Source: https://github.com/bionikspoon/cache_requests
- Python version agnostic: tested against Python 2.7, 3.3, 3.4, 3.5 and Pypy
- MIT license
- Drop in decorator for the requests library.
- Automatic timer based expiration on stored items (optional).
- Backed by yahoo's powerful `redislite`.
- Scalable with redis. Optionally accepts a `redis` connection.
- Exposes the powerful underlying `Memoize` decorator to decorate any function.
- Tested with high coverage.
- Lightweight. Simple logic.
- Lightning fast.
- Jump start your development cycle.
- Collect and reuse entire response objects.

1.1.2 Credits

Tools used in rendering this package:

- Cookiecutter
- bionikspoon/cookiecutter-pypackage forked from audreyr/cookiecutter-pypackage

1.2 Installation

At the command line either via easy_install or pip:

```
$ pip install cache_requests
```

```
$ easy_install cache_requests
```

Or, if you have virtualenvwrapper installed:

```
$ mkvirtualenv cache_requests
$ pip install cache_requests
```

Uninstall:

```
$ pip uninstall cache_requests
```

1.3 Usage

To use cache_requests in a project:

```
import cache_requests
```

1.3.1 Quick Start

To use `cache_requests` in a project:

```
>>> from cache_requests import Session()

requests = Session()

# from python-requests.org
>>> r = requests.get('https://api.github.com/user', auth=('user', 'pass'))
>>> r.status_code
200
>>> r.headers['content-type']
'application/json; charset=utf8'
>>> r.encoding
'utf-8'
>>> r.text
u'{"type":"User"...'
>>> r.json()
{u'private_gists': 419, u'total_private_repos': 77, ...}
```

1.3.2 Config Options

Decorated Methods

`method.ex` sets the default expiration (seconds) for new cache entries.

`method.redis` creates the connection to the `redis` or `redislite` database. By default this is a `redislite` connection. However, a redis connection can be dropped in for easy scalability.

cache_requests.Session

- `ex` is shared between request methods. They can be accessed by `Session.cache.ex` or `Session.get.ex`, where `get` is the `requests.get` method
- By default requests that return and error will not be cached. This can be overridden by overriding the `Session.cache.set_cache_cb` to return `False`. The callback takes the response object as an argument:

```
from cache_requests import Session

requests = Session()

requests.cache.set_cache_db = lambda _:False
```

- By default only autonomous methods are cached (`get`, `head`, `options`). Each method can be setup to be cached using the `Session.cache` config option.

These methods are accessed through the Session objects `Session.cache.[method name]`. They can be overridden with the `Session.cache.all` setting.

For example:

```
from cache_requests import Session

requests = Session()

requests.cache.delete = True

# cached, only called once.
requests.delete('http://google.com')
requests.delete('http://google.com')

requests.cache.delete = True

# not cached, called twice.
requests.delete('http://google.com')
requests.delete('http://google.com')

# cache ALL methods
requests.cache.all = True

# don't cache any methods
requests.cache.all = False

# Use individual method cache options.
requests.cache.all = None
```

Default settings

Method	Cached
get	True
head	True
options	True
post	False
put	False
patch	False
delete	False
all	None

Function Level Config

Cache Busting Use keyword `bust_cache=True` in a memoized function to force reevaluation.

Conditionally Set Cache Use keyword `set_cache` to provide a callback. The callback takes the results of function as an argument and must return a `bool`. Alternatively, `True` and `False` can be used.

1.3.3 Use Case Scenarios

Development: 3rd Party APIs

Scenario: Working on a project that uses a 3rd party API or service.

Things you want:

- A cache that persists between sessions and is lightning fast.
- Ability to rapidly explore the API and it's parameters.
- Ability to inspect and debug response content.
- Ability to focus on progress.
- Perfect transition to a production environment.

Things you don't want:

- Dependency on network and server stability for development.
- Spamming the API. Especially APIs with limits.
- Responses that change in non-meaningful ways.
- Burning energy with copypasta or fake data to run piece of your program.
- Slow. Responses.

Make a request one time. Cache the results for the rest of your work session.:

```
import os

if os.environ.get('ENV') == 'DEVELOP':
    from cache_requests import Session

    request = Session(ex=60 * 60 )  # Set expiration, 60 min
else:
    import requests
```

```
# strange, complicated request you might make
headers = {"accept-encoding": "gzip, deflate, sdch", "accept-language": "en-US,en;q=0.8"}
payload = dict(sourceid="chrome-instant", ion="1", espv="2", ie="UTF-8", client="ubuntu",
               q="hash%20a%20dictionary%20python")
response = requests.get('http://google.com/search', headers=headers, params=payload)

# spam to prove a point
response = requests.get('http://google.com/search', headers=headers, params=payload)
response = requests.get('http://google.com/search', headers=headers, params=payload)
response = requests.get('http://google.com/search', headers=headers, params=payload)
response = requests.get('http://google.com/search', headers=headers, params=payload)
response = requests.get('http://google.com/search', headers=headers, params=payload)
response = requests.get('http://google.com/search', headers=headers, params=payload)
response = requests.get('http://google.com/search', headers=headers, params=payload)

# tweak your query, we're exploring here
payload = dict(sourceid="chrome-instant", ion="1", espv="2", ie="UTF-8", client="ubuntu",
               q="hash%20a%20dictionary%20python2")
# do you see what changed? the caching tool did.
response = requests.get('http://google.com/search', headers=headers, params=payload)
response = requests.get('http://google.com/search', headers=headers, params=payload)
response = requests.get('http://google.com/search', headers=headers, params=payload)
```

Production: Web Scraping

Automatically expire old content.

- How often? After a day? A week? A Month? etc. 100% of this logic is built in with the `Session.cache.ex` setting.
- Effectively it can manage all of the time-based rotation.
- Perfect if you theres more data then what your API caps allow.

One line of code to use a `redis` full database.

- Try `redislite`; it can handle quite a bit. The `redislite` api used by this module is 1:1 with the redis package. Just replace the connection parameter/config value.
- `redis` is a drop in::

```
    connection  = redis.StrictRedis(host='localhost', port=6379, db=0)
    requests = Session(connection=connection)
```

- Everything else just works. There's no magic required.:

```
    from cache_requests import Session

    connection  = redis.StrictRedis(host='localhost', port=6379, db=0)
    ex = 7 * 24 * 60 * 60 # 1 week

    requests = Session(ex=ex, connection=connection)

    for i in range(1000)
        payload = dict(q=i)
        response = requests.get('http://google.com/search', params=payload)
        print(response.text)
```

Usage: memoize

```
from cache_requests import Memoize

@Memoize(ex=15 * 60)  # 15 min, default, 60 min
def amazing_but_expensive_function(*args, **kwargs)
    print("You're going to like this")
```

1.4 Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

1.4.1 Types of Contributions

Report Bugs

Report bugs at https://github.com/bionikspoon/cache_requests/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with “bug” is open to whoever wants to implement it.

Implement Features

Look through the GitHub issues for features. Anything tagged with “feature” is open to whoever wants to implement it.

Write Documentation

cache_requests could always use more documentation, whether as part of the official cache_requests docs, in docstrings, or even on the web in blog posts, articles, and such.

Submit Feedback

The best way to send feedback is to file an issue at https://github.com/bionikspoon/cache_requests/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

1.4.2 Get Started!

Ready to contribute? Here's how to set up *cache_requests* for local development.

1. Fork the *cache_requests* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/cache_requests.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv cache_requests
$ cd cache_requests/
$ python setup.py develop
```

4. Create a branch for local development:

```
$ git checkout -b feature/name-of-your-feature
$ git checkout -b hotfix/name-of-your-bugfix
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ flake8 cache_requests tests
$ python setup.py test
$ tox
```

   To get flake8 and tox, just pip install them into your virtualenv.

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

1.4.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 2.7, 3.3, 3.4, 3.5, and PyPy. Check https://travis-ci.org/bionikspoon/cache_requests/pull_requests and make sure that the tests pass for all supported Python versions.

1.4.4 Tips

To run a subset of tests:

```
$ py.test tests/test_cache_requests.py
```

1.5 Credits

1.5.1 Development Lead

- Manu Phatak <bionikspoon@gmail.com>

1.5.2 Contributors

None yet. Why not be the first?

1.6 History

1.6.1 Next Release

- Stay tuned.

1.6.2 4.0.0 (2015-12-25)

- Fix: Use MD5 for hash to avoid PYTHONHASHSEED issue.
- Fix: Give default dbfilename a more unique name, based on caller.
- BREAKING:Move `Session.ex` and `Session.connection` to `Session.cache` config object.
- Updated examples. New example demonstrates Memoize decorator.
- Updated requirements.

1.6.3 3.0.0 (2015-12-22)

- Feature: Cache busting! Use keyword argument `bust_cache=True` to force reevaluation.
- Feature: Session automatically skips caching error responses.
- Feature: Callback argument to decide if results should be cached.
- Feature: Decorated Session methods share a centralized configuration per session.
- BREAKING: Remove global config, in favor component level config. Reasoning: Global config adds way too much complexity and adds too little value. (Everything needs to lazy load the config at the last moment)
- Fix: Unique cache per function in shared db.
- Fix: Tweaks to keep the classes sub classable.
- Fix: Cleaned up tests.
- Updated requirements.

1.6.4 2.0.0 (2015-12-12)

- API completely rewritten
- New API extends `requests` internals as opposed to monkeypatching.
- Entire package is redesigned to be more maintainable, more modular, and more usable.
- Dependencies are pinned.
- Tests are expanded.
- PY26 and PY32 support is dropped, because of dependency constraints.
- PY35 support is added.
- Docs are rewritten.
- Move towards idiomatic code.
- 2.0.6 Fix broken coverage, broken rst render.

1.6.5 1.0.0 (2015-04-23)

- First real release.
- Feature/ Unit test suite, very high coverage.
- Feature/ `redislite` integration.
- Feature/ Documentation. https://cache-requests.readthedocs.org.
- Feature/ Exposed the beefed up `Memoize` decorator.
- **Feature/ Upgraded compatibility to:**
  - PY26
  - PY27
  - PY33
  - PY34
  - PYPY
- Added examples and case studies.

1.6.6 0.1.0 (2015-04-19)

- First release on PyPI.

1.7 cache_requests package

1.7.1 `cache_requests`

Simple. Powerful. Persistent LRU caching for the requests library.

class `cache_requests.`**`Session`**(*ex=None*, *connection=None*)

Bases: `requests.sessions.Session`

`requests.Session` with memoized methods.

class `cache_requests.`**`Memoize`**(*func=None*, *ex=None*, *connection=None*)

Bases: `object`

Decorator class. Implements LRU cache pattern that syncs cache with `redislite` storage.

`put_cache_results`(*key*, *func_akw*, *set_cache_cb*)

Put function results into cache.

`redis`

Provide access to the redis connection handle.

1.7.2 Submodules

`cache_requests._compat`

Python 2to3 compatibility handling.

class `cache_requests._compat.`**`NullHandler`**(*level=0*)

Bases: `logging.Handler`

This handler does nothing. It's intended to be used to avoid the "No handlers could be found for logger XXX" one-off warning. This is important for library code, which may contain code to log events. If a user of the library does not configure logging, the one-off warning might be produced; to avoid this, the library developer simply needs to instantiate a NullHandler and add it to the top-level logger of the library module or package.

`createLock`()

`emit`(*record*)

`handle`(*record*)

`cache_requests.memoize`

`Memoize` cache decorator.

Public Api

- *`Memoize`*

Source

class `cache_requests.memoize.`**`Memoize`**(*func=None*, *ex=None*, *connection=None*)

Bases: `object`

Decorator class. Implements LRU cache pattern that syncs cache with `redislite` storage.

`put_cache_results`(*key*, *func_akw*, *set_cache_cb*)

Put function results into cache.

`redis`

Provide access to the redis connection handle.

`cache_requests.sessions`

Extend `requests` with cache decorator.

Public Api

- *`Session`*

Private API

- *`MemoizeRequest`*
- *`CacheConfig`*

Source

class `cache_requests.sessions.`**`MemoizeRequest`**(*func=None*, ***kwargs*)
Bases: *`cache_requests.memoize.Memoize`*

Cache session method calls.

`ex`

`redis`

`use_cache`

class `cache_requests.sessions.`**`CacheConfig`**(***kwargs*)
Bases: *`cache_requests.utils.AttributeDict`*

A strict dict with attribute access.

class `cache_requests.sessions.`**`Session`**(*ex=None*, *connection=None*)
Bases: `requests.sessions.Session`

`requests.Session` with memoized methods.

`cache_requests.utils`

Package utilities.

Private API

- *`AttributeDict`*
- *`deep_hash()`*
- *`normalize_signature()`*

Source

class `cache_requests.utils.`**`AttributeDict`**(***kwargs*)

Bases: `object`

Strict dict with attribute access

`cache_requests.utils.`**`deep_hash`**(**args*, ***kwargs*)

`cache_requests.utils.`**`normalize_signature`**(*func*)

Decorator. Combine args and kwargs. Unpack single item tuples.

`cache_requests.utils.`**`make_callback`**(*value*)

Convert bool values to callback

`cache_requests.utils.`**`temp_file`**(*name*)

CHAPTER 2

Feedback

If you have any suggestions or questions about **cache_requests** feel free to email me at bionikspoon@gmail.com.

If you encounter any errors or problems with **cache_requests**, please let me know! Open an Issue at the GitHub https://github.com/bionikspoon/cache_requests main repository.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Index