Falcon-Caching

Release 1.1.0

Apr 16, 2022

Contents

Version: 1.1.0

Falcon-Caching adds cache support to the Falcon web framework.

It is a port of the popular Flask-Caching library to Falcon.

The library aims to be compatible with CPython 3.7+ and PyPy 3.5+.

You can use this library both with a sync (WSGI) or an async (ASGI) app, by using the matching cache object (`Cache` or `AsyncCache`). Throughout the documentation we will be mostly be showcasing examples for the `Cache` object, but all those example could be used with the `AsyncCache` object too. The Quickstart example shows both `Cache` and `AsyncCache` side-by-side. Obviously you should never be mixing the two in a single app, use one or the other.

CHAPTER 1

Quickstart

WSGI (alias sync) example:

```
import falcon
from falcon_caching import Cache

# setup the cache instance
cache = Cache(config={'CACHE_TYPE': 'simple'})

class ThingsResource:

    # mark the method as cached
    @cache.cached(timeout=600)
    def on_get(self, req, resp):
        pass

# create the app with the cache middleware
# you can use falcon.API() instead of falcon.App() below Falcon 3.0.0
app = falcon.App(middleware=cache.middleware)

things = ThingsResource()

app.add_route('/things', things)
```

ASGI (alias async) example:

```
import falcon.asgi
from falcon_caching import AsyncCache

# setup the cache instance
cache = AsyncCache(config={'CACHE_TYPE': 'simple'})

class ThingsResource:

    # mark the method as cached
```

(continues on next page)

(continued from previous page)

```
    @cache.cached(timeout=600)
    async def on_get(self, req, resp):
        pass

app = falcon.asgi.App(middleware=cache.middleware)

things = ThingsResource()

app.add_route('/things', things)
```

Alternatively you could cache the **whole resource** (watch out for issues mentioned in *Resource level caching*):

```
# mark the whole resource as cached
@cache.cached(timeout=600)
class ThingsResource:

    def on_get(self, req, resp):
        pass

    def on_post(self, req, resp):
        pass
```

Warning: Be careful with the order of middlewares. The `cache.middleware` will short-circuit any further processing if a cached version of that resource is found. It will skip any remaining `process_request` and `process_resource` methods, as well as the `responder` method that the request would have been routed to. However, any `process_response` middleware methods will still be called.

This is why it is suggested that you add the `cache.middleware` **following** any authentication / authorization middlewares to avoid unauthorized access of records served directly from the cache.

CHAPTER 2

Installation

Install the extension with pip:

```
$ pip install Falcon-Caching
```

CHAPTER 3

Set Up

Cache is managed through a `Cache` and the `AsyncCache` instance:

```
import falcon
# import falcon.asgi
from falcon_caching import Cache, AsyncCache

# setup the cache instance
cache = Cache(  # could also be 'AsyncCache'
    config=
    {
        'CACHE_EVICTION_STRATEGY': 'time-based',  # how records are
                                                  # evicted
        'CACHE_TYPE': 'simple'  # backend used to store the cache
    })

class ThingsResource:
    # mark the method as cached for 600 seconds
    @cache.cached(timeout=600)
    def on_get(self, req, resp):   # this could also be an async function
        pass                       # if AsyncCache() is used

# create the app with the cache middleware
# you can use falcon.API() instead of falcon.App() below Falcon 3.0.0
app = falcon.App(middleware=cache.middleware)
# app = falcon.asgi.App(middleware=cache.middleware)


things = ThingsResource()

app.add_route('/things', things)
```

CHAPTER 4

Eviction strategies

Once a resource is cached, there is the question of how that cached record will be evicted from the cache - alias what *'eviction strategy'* is followed.

Below is the list of supported strategies:

4.1 'time-based'

The most well known *eviction strategy* is simply *time-based*, meaning that the cached record gets evicted based on a timeout (also called TTL, time-to-live) being reached. In this case the cached data is invalidated x seconds after it was generated. In our library this is called **'time-based'** eviction and it is the default *eviction strategy*.

4.2 'rest-based'

For REST APIs - which implement the RESTful methods closely - there is another possible option, to evict records based on the definition of the RESTful methods.

In this case GET requests are the only ones cached, but those are cached indefinitely. They only get removed from the cache when another request of the same resource of type PUT / PATCH / POST or DELETE arrives. This will invalidate/evict the cached record and force the next GET request to re-cache it. We call this **'rest-based'** eviction strategy.

4.3 'rest-and-time-based'

The third option is a combination of these two, where the eviction happens based on whichever of these two events occurs first - the time expires or a PUT/PATCH/POST/DELETE request arrives. We call this **'rest-and-time-based'** eviction strategy.

These eviction strategies can be set with the **CACHE_EVICTION_STRATEGY** config attribute - see *Configuring Falcon-Caching*.

```
from falcon_caching import Cache

cache = Cache(
    config={
        'CACHE_TYPE': 'simple',
        'CACHE_EVICTION_STRATEGY': 'rest-based'
    })
```

If no **CACHE_EVICTION_STRATEGY** is provided then the **'time-based'** strategy is used by default.

CHAPTER 5

Backends (alias 'CACHE_TYPE')

When you are caching you have the choice of what kind of backend to cache to, be that a Redis database, Memcached, the local process' memory or just files on the local filesystem.

The Falcon-Caching library offers you different backend options and made to be extendable, so additional backend options can be added.

The type of backend used is determined by the **CACHE_TYPE** attribute - see *Configuring Falcon-Caching*.

Below is an example of using *CACHE_TYPE* with value *'simple'* - which makes the cached records stored in the local process' memory (not 100% thread safe!):

```
from falcon_caching import Cache

cache = Cache(
    config={
        'CACHE_TYPE': 'simple',  # backend 'simple' will be used
        'CACHE_EVICTION_STRATEGY': 'time-based'
    })
```

Note: Credits must be given to the authors and maintainers of the Flask-Caching library, as the structure and much of the code of our backends was ported from their popular library.

Below is a list of available backends, alias the available *CACHE_TYPE* options:

5.1 'simple' (the default)

A simple memory cache for single process environments. This option exists mainly for the development server and is not 100% thread safe. It tries to use as many atomic operations as possible and no locks for simplicity, but it could happen under heavy load that keys are added multiple times. Do not use in production!

Example:

```
from falcon_caching import Cache

cache = Cache(
    config={
        'CACHE_TYPE': 'simple',  # backend 'simple' will be used
        'CACHE_EVICTION_STRATEGY': 'time-based'
    })
```

5.2 'null'

A cache that doesn't cache. This can be useful for unit testing.

5.3 'filesystem'

A cache that stores the items on the file system. This cache depends on being the only user of the '*cache_dir*'. Make absolutely sure that nobody but this cache stores files there or otherwise the cache will randomly delete files therein.

Example:

```
from falcon_caching import Cache

cache = Cache(
    config={
        'CACHE_TYPE': 'filesystem',
        'CACHE_EVICTION_STRATEGY': 'time-based',
        'CACHE_DIR': '/tmp/falcon-cache-dedicated/',
        'CACHE_THRESHOLD': 500  # the maximum number of items the
                                # cache stores before it starts
                                # deleting some. A threshold value
                                # of 0 indicates no threshold.
                                # default: 500
    })
```

5.4 'redis'

A cache that stores the items in the Redis key-value store or an object which is API compatible with the official Python Redis client (*redis-py*).

If you want to use an object which is API compatible with the official Python Redis client (*redis-py*), then just supply that as an initialized object to the *CACHE_REDIS_HOST* parameter.

If you use the same Redis database for other purposes too, then you are strongly advised to specify the *CACHE_KEY_PREFIX*, so keys would not accidentally collide and *cache.clean()* calls would only remove keys from the cache and not other records.

Example:

```
from falcon_caching import Cache

cache = Cache(
    config={
```

(continues on next page)

(continued from previous page)

```
        'CACHE_TYPE': 'redis',
        'CACHE_EVICTION_STRATEGY': 'time-based',
        'CACHE_REDIS_HOST': 'localhost',  # Redis host/client object
                                          # default: 'localhost'
        'CACHE_REDIS_PORT': 6379,  # default: 6379
        'CACHE_REDIS_PASSWORD': 'MyRedisPassword',  # default: None
        'CACHE_REDIS_DB': 0,  # default: 0
        'CACHE_KEY_PREFIX': 'mycache'  # default: None
    })
```

Alternatively you could also supply a Redis URL via the CACHE_REDIS_URL argument, like *redis://user:password@localhost:6379/2*.

5.5 'redis-sentinel'

A cache that stores the items in a Redis Sentinel, which is a high availability 'load-balancer' for a Redis cluster.

Just like for 'redis', if you use the same Redis database for other purposes too, then you are strongly advised to specify the *CACHE_KEY_PREFIX*, so keys would not accidentally collide and *cache.clean()* calls would only remove keys from the cache and not other records.

Example:

```
from falcon_caching import Cache

cache = Cache(
    config={
        'CACHE_TYPE': 'redissentinel'
        'CACHE_EVICTION_STRATEGY': 'time-based',
        'CACHE_REDIS_SENTINELS': [("127.0.0.1", 26379),
                                  ("10.0.0.1", 26379)],
        'CACHE_REDIS_SENTINEL_MASTER': 'mymaster',  # default: None
        'CACHE_REDIS_PASSWORD': 'MyRedisPassword',  # default: None
        'CACHE_REDIS_SENTINEL_PASSWORD': 'MyPsw',   # default: None
        'CACHE_REDIS_DB': 0,  # default: 0
        'CACHE_KEY_PREFIX': 'mycache'  # default: None
    })
```

5.6 'memcached'

A cache that stores the items in a Memcached instance or cluster. It supports the *pylibmc*, *memcache* and the *google app engine memcache* libraries.

You can supply one or more server addresses via *CACHE_MEMCACHED_SERVERS* or you can supply an already initialized client, an object that resembles the API of a *memcache.Client*. If you have supplied a server(s) address, then the library will pick the best memcached client library available to use.

Example:

```
from falcon_caching import Cache

cache = Cache(
    config={
```

(continues on next page)

(continued from previous page)

```
        'CACHE_TYPE': 'memcached',
        'CACHE_EVICTION_STRATEGY': 'time-based',
        'CACHE_MEMCACHED_SERVERS': ["127.0.0.1:11211",
                                    "127.0.0.1:11212"]
        'CACHE_KEY_PREFIX': 'cache'  # default: None
    })
```

Note: Flask-Caching does not pass additional configuration options to memcached backends. To add additional configuration to these caches, directly set the configuration options on the object after instantiation:

```
from falcon_caching import Cache

cache = Cache(
    config={
        'CACHE_TYPE': 'memcached',
        'CACHE_EVICTION_STRATEGY': 'time-based',
        'CACHE_MEMCACHED_SERVERS': ["127.0.0.1:11211",
                                    "127.0.0.1:11212"]
        'CACHE_KEY_PREFIX': 'cache'  # default: None
    })

# Break convention and set options on the _client object
# directly. For pylibmc behaviors:
cache.cache._client.behaviors["tcp_nodelay"] = True
```

5.7 'saslmemcached'

A cache that stores the items in an SASL-authentication protected Memcached instance or cluster.

Just like for *memcached* - you can supply one or more server addresses via *CACHE_MEMCACHED_SERVERS* or you can supply an already initialized client, an object that resembles the API of a *memcache.Client*.

Example:

```
from falcon_caching import Cache

cache = Cache(
    config={
        'CACHE_TYPE': 'saslmemcached',
        'CACHE_EVICTION_STRATEGY': 'time-based',
        'CACHE_MEMCACHED_SERVERS': ["127.0.0.1:11211",
                                    "127.0.0.1:11212"]
        'CACHE_MEMCACHED_USERNAME': 'myuser',  # default: None
        'CACHE_MEMCACHED_PASSWORD': 'MyPassword',  # default: None
        'CACHE_KEY_PREFIX': 'cache'  # default: None
    })
```

5.8 'spreadsaslmemcached'

A subclass of the *saslmemcached* backend that will spread the cached values across multiple records if they are bigger than the memcached treshold which by default is 1M.

Spreading requires using *pickle* to store the value, which can significantly impact the performance.

5.9 'uwsgi'

Implements the cache using uWSGI's caching framework.

To set the uwsgi caching instance to connect to, for example: *mycache@localhost:3031*, use the *CACHE_UWSGI_NAME* argument, which defaults to an empty string, in which case uWSGI will cache in the local instance.

This backend cannot be used when running under PyPy, because the uWSGI API implementation for PyPy is lacking the required functionality.

Example:

```
from falcon_caching import Cache

cache = Cache(
    config={
        'CACHE_TYPE': 'uwsgi',
        'CACHE_UWSGI_NAME': 'mycache@localhost:3031',  # default: ''
        'CACHE_KEY_PREFIX': 'cache'  # default: None
    })
```

CHAPTER 6

What gets cached

You might ask the question that what (what data) is getting cached when a *responder* is cached.

By default two things are cached: the `response body` and the response's `Content-Type` header.

To be able to store these two things in the cache backend under one object, we use msgpack to serialize and then deserialize when loading the record back from the cache. While *msgpack* is a fast serializer, this does take some time.

Note: If you know that all of your cached responders are using the `` `Content-Type`= `application/json` `` header - which is very typical for basic APIs in these days - then you don't need the `` `Content-Type` `` header to be cached. This is because the `` `Content-Type` = `application/json` `` is the default in Falcon and it is added to the response when no other value is specified.

So in case your application only generates responses with the `` `Content-Type` = `application/json `` header, then you can turn off this serialization storing the `Content-Type` header and benefit from the performance boost of not needing to serialize and deserialize messages.

You can turn off the serialization by setting **'CACHE_CONTENT_TYPE_JSON_ONLY = True'** in the config - see *Configuring Falcon-Caching*.

New in version 0.2.

CHAPTER 7

Memoization

New in version 0.3.

See *Cache.memoize()*

Using the @memoize decorator you are able to cache the result of other non-view related functions. In memoization, the functions arguments are also included into the cache_key.

Note: Credits must be given to the authors and maintainers of the Flask-Caching library, as much of the code of our memoize method was ported from their popular library.

Outside just simple function, memoize is also designed for methods, since it will take into account the identity. of the 'self' or 'cls' argument as part of the cache key.

The theory behind memoization is that if you have a function you need to call several times in one request, it would only be calculated the first time that function is called with those arguments. For example, an sqlalchemy object that determines if a user has a role. You might need to call this function many times during a single request. To keep from hitting the database every time this information is needed you might do something like the following:

```
class Person(db.Model):
    @cache.memoize(50)
    def has_membership(self, role_id):
        return Group.query.filter_by(user=self, role_id=role_id).count() >= 1
```

Warning: Using mutable objects (classes, etc) as part of the cache key can become tricky. It is suggested to not pass in an object instance into a memoized function. However, the memoize does perform a repr() on the passed in arguments so that if the object has a __repr__ function that returns a uniquely identifying string for that object, that will be used as part of the cache key.

For example, an sqlalchemy person object that returns the database id as part of the unique identifier:

```
class Person(db.Model):
    def __repr__(self):
        return "%s(%s)" % (self.__class__.__name__, self.id)
```

7.1 Deleting memoize cache

See *Cache.delete_memoized()*

New in version 0.3.

You might need to delete the cache on a per-function bases. Using the above example, lets say you change the users permissions and assign them to a role, but now you need to re-calculate if they have certain memberships or not. You can do this with the `delete_memoized()` function:

```
cache.delete_memoized(user_has_membership)
```

Note: If only the function name is given as parameter, all the memoized versions of it will be invalidated. However, you can delete specific cache by providing the same parameter values as when caching. In following example only the `user`-role cache is deleted:

```
user_has_membership('demo', 'admin')
user_has_membership('demo', 'user')

cache.delete_memoized(user_has_membership, 'demo', 'user')
```

CHAPTER 8

Configuring Falcon-Caching

The following configuration values exist for Falcon-Caching:

CACHE_EVICTION_STRATEGY	The *eviction strategy* determines when a cached resource is removed from cache. Available eviction strategies: • **time-based**: records are removed once time expires (default) • **rest-based**: records are removed once a PUT/POST/PATCH/DELETE call is made against the resource • **rest-and-time-based**: records are removed either by time or request method (whichever happens first) See more at *Eviction strategies*
CACHE_TYPE	Specifies which type of caching object to use. This is an import string that will be imported and instantiated. It is assumed that the import object is a function that will return a cache object that adheres to the cache API. For falcon_caching.backends objects, you do not need to specify the entire import string, just one of the following names. Built-in cache types: • **null**: NullCache (default) • **simple**: SimpleCache • **filesystem**: FileSystemCache • **redis**: RedisCache (redis required) • **redissentinel**: RedisSentinelCache (redis required) • **uwsgi**: UWSGICache (uwsgi required) • **memcached**: MemcachedCache (pylibmc or memcache required) • **gaememcached**: same as memcached (for backwards compatibility) • **saslmemcached**: SASLMemcachedCache (pylibmc required) • **spreadsaslmemcached**: SpreadSASLMemcachedCache (pylibmc required)
CACHE_CONTENT_TYPE_JSON_ONLY	Set to `True` if all your cached responders use the `application/json` `Content-Type`, which will turn off serialization and provide a performance boost. Defaults to `False`.
CACHE_NO_NULL_WARNING	Silence the warning message when using cache type of 'null'.
CACHE_ARGS	Optional list to unpack and pass during the cache class instantiation.
CACHE_OPTIONS	Optional dictionary to pass during the cache class instantiation.
CACHE_DEFAULT_TIMEOUT	The default timeout that is used if no timeout is specified. Unit of time is seconds.
CACHE_IGNORE_ERRORS	If set to any errors that occurred during the deletion process will be ignored. However, if it is set to `False` it will stop on the first error. This option is only relevant for the backends **filesystem** and **simple**. Defaults to `False`.
CACHE_THRESHOLD	The maximum number of items the cache will store before it starts deleting some. Used only for SimpleCache and FileSystemCache
CACHE_KEY_PREFIX	A prefix that is added before all keys. This makes it possible to use the same memcached server for different apps. Used only for RedisCache and MemcachedCache
CACHE_UWSGI_NAME	The name of the uwsgi caching instance to connect to, for example: mycache@localhost:3031, defaults to an empty string, which means uWSGI will cache in the local instance.
CACHE_MEMCACHED_SERVERS	A list or a tuple of server addresses. Used only for Mem-

CHAPTER 9

Resource level caching

In Falcon-Caching you mark individual methods or resources to be cached by adding the `@cache.cached()` decorator to them.

It is possible to add this decorator on the resource (class) level to mark the whole resource - and so all of its `'on_'` methods - as cached:

```
# mark the whole resource as cached
# which will decorate all the on_...() methods of this class
@cache.cached(timeout=600)
class ThingsResource:

    def on_get(self, req, resp):
        pass

    def on_post(self, req, resp):
        pass
```

BUT if any of those `'on_'` methods are supposed to modify the data or have some other non-cachable actions, then that will NOT be executed when the response is returned from the cache - assuming the **CACHE_EVICTION_STRATEGY** is set to **'time-based'** - which is the default.

The **CACHE_EVICTION_STRATEGY** values of **'rest-based'** and **'rest-and-time-based'** are safe, as those invalidate the cache for any PUT/PATCH/POST/DELETE calls and do NOT serve the response from the cache for those methods.

This happens because the *cache.middleware* short-circuits any further processing **if a cached version of that item is found**. If a cached version is found then it will skip any remaining *process_request* and *process_resource* methods, as well as the *responder* method that the request would have been routed to. However, any *process_response* middleware methods will still be called.

We suggest that you only use the resource level (eg class) decorator if you use the **CACHE_EVICTION_STRATEGY** of **'rest-based'** or **'rest-and-time-based'** and NOT if you use the **'time-based'** strategy. The only exception to this rule of thumb could be if (1) you are certain that **all** the methods of that resource can be served from the cache or (2) all the actions for those methods are taken in *process_response* phase.

CHAPTER 10

Explicitly Caching Data

Data can be cached explicitly by using the proxy methods like `Cache.set()`, and `Cache.get()` directly. There are many other proxy methods available via the `Cache` class - see them listed below.

For example:

```
from falcon_caching import Cache

cache = Cache(
    config={
        'CACHE_TYPE': 'simple',
        'CACHE_EVICTION_STRATEGY': 'rest-based'
    })

...

def test(foo=None):
    if foo is not None:
        cache.set("foo", foo)  # saving a value into the cache
    bar = cache.get("foo")  # retrieving the value from the cache
```

Supported methods:

```
cache.set("foo", "bar")
cache.has("foo")
cache.get("foo")
cache.add("foo", "bar")  # like set, except it doesn't overwrite
cache.set_many({"foo": "bar", "foo2": "bar2"})
cache.get_many(["foo", "foo2"])  # returns a list
cache.get_dict(["foo", "foo2"])  # returns a dict
cache.delete("foo")
cache.delete_many("foo", "foo2")
cache.set("foo3", 1)
cache.inc("foo3")  # increment, only supported by Redis&Redis Sentinel
cache.dec("foo3")  # decrement, only supported by Redis&Redis Sentinel
```

(continues on next page)

(continued from previous page)

```
cache.clear()  # clears all cache - not supported by all backends
               # WARNING: some implementations (Redis) will flush
               # the whole database!!!
```

CHAPTER 11

Query String

Currently the query string is NOT used in the cache key, so two requests which only differ in the query string will be cached against the same key.

CHAPTER 12

Recipes

12.1 Multiple decorators

For scenarios where there is a need for multiple decorators and the `@cache.cached()` cannot be the topmost one, we need to register the decorators a special way.

This scenario is complicated because our `@cache.cached()` just marks the fact that the given method is decorated with a cache, which later gets picked up by the middleware and triggers caching. If the `@cache.cached()` is the topmost decorator then it is easy to pick that up, but if there are other decorators 'ahead' it, then those will 'hide' the `@cache.cached()`. This is because decorators in Python are just syntactic sugar for nested function calls.

To be able to tell if the given endpoint was decorated by the `@cache.cached()` decorator when that is NOT the topmost decorator, you need to decorate your method by registering your decorators using the `@register()` helper decorator.

See more about this issue at https://stackoverflow.com/questions/3232024/introspection-to-get-decorator-names-on-a-method

```
import falcon
from falcon_caching import Cache
from falcon_caching.utils import register

limiter = Limiter(
    key_func=get_remote_addr,
    default_limits=["10 per hour", "2 per minute"]
)

cache = Cache(config={'CACHE_TYPE': 'simple'})

class ThingsResource:
    # this is fine, as the @cache.cached() is the topmost (eg the first) decorator:
    @cache.cached(timeout=600)
    @another_decorator
    def on_get(self, req, resp):
        pass
```

(continues on next page)

(continued from previous page)

```
class ThingsResource2:
    # the @cache.cached() is NOT the topmost decorator, so
    # this would NOT work - the cache decorator would be ignored!!!!
    # DO NOT DO THIS:
    @another_decorator
    @cache.cached(timeout=600)
    def on_get(self, req, resp):
        pass

class ThingsResource3:
    # use your decorators this way:
    @register(another_decorator, cache.cached(timeout=600))
    def on_get(self, req, resp):
        pass
```

12.2 Development

For development guidelines see https://github.com/zoltan-fedor/falcon-caching#development

12.3 API Reference

If you are looking for information on a specific function, class or method of a service, then this part of the documentation is for you.

12.3.1 API Reference Guide

Cache API

Falcon-Caching - a caching module for the Falcon web framework

class falcon_caching.**AsyncCache**(*config: Dict[str, Any]*)

This is the central class for the caching

You need to initialize this object to setup the attributes of the caching and then supply the object's middleware to the Falcon app.

Parameters **(dict of str** (*config*) – str): Cache config settings

cache

An initialized 'CACHE_TYPE' cache from the backends.

Type BaseCache

cache_args

Optional list passed during the cache class instantiation.

Type list of str

cache_options (dict of str

str): Optional dictionary passed during the cache class instantiation.

config (dict of str

str): Cache config settings

add(**args*, ***kwargs*) → bool
It adds a given key and value to the cache, but only if no record which such key already exists.

cached(*timeout: int*)
This is the decorator used to decorate a resource class or the requested method of the resource class

clear() → bool
It clears all cache - if the *CACHE_KEY_PREFIX* config attribute is used then it only removes key starting with that prefix, otherwise it flushes the whole database.

dec(**args*, ***kwargs*) → Optional[int]
It decrements and returns the value of a numerical cache record. Only works for Redis and Redis Sentinel!

delete(**args*, ***kwargs*) → bool
It deletes the cached record based on the provided key.

delete_many(**args*, ***kwargs*) → bool
It deletes all cached record matching the list of keys provided.

delete_memoized(*f*, **args*, ***kwargs*)
Deletes the specified functions caches, based by given parameters. If parameters are given, only the functions that were memoized with them will be erased. Otherwise all versions of the caches will be forgotten. Example:

```
@cache.memoize(50)
def random_func():
    return random.randrange(1, 50)
@cache.memoize()
def param_func(a, b):
    return a+b+random.randrange(1, 50)
```

::

```
>>> random_func()
43
>>> random_func()
43
>>> cache.delete_memoized(random_func)
>>> random_func()
16
>>> param_func(1, 2)
32
>>> param_func(1, 2)
32
>>> param_func(2, 2)
47
>>> cache.delete_memoized(param_func, 1, 2)
>>> param_func(1, 2)
13
>>> param_func(2, 2)
47
```

Delete memoized is also smart about instance methods vs class methods. When passing a instancemethod, it will only clear the cache related to that instance of that object. (object uniqueness can be overridden by defining the __repr__ method, such as user id). When passing a classmethod, it will clear all caches related across all instances of that class. Example:

```
class Adder(object):
    @cache.memoize()
    def add(self, b):
        return b + random.random()
```

::

```
>>> adder1 = Adder()
>>> adder2 = Adder()
>>> adder1.add(3)
3.23214234
>>> adder2.add(3)
3.60898509
>>> cache.delete_memoized(adder1.add)
>>> adder1.add(3)
3.01348673
>>> adder2.add(3)
3.60898509
>>> cache.delete_memoized(Adder.add)
>>> adder1.add(3)
3.53235667
>>> adder2.add(3)
3.72341788
```

Parameters

- **fname** – The memoized function.
- ***args** – A list of positional parameters used with memoized function.
- ****kwargs** – A dict of named parameters used with memoized function.

Note: Falcon-Caching uses inspect to order kwargs into positional args when the function is memoized. If you pass a function reference into `fname`, Falcon-Caching will be able to place the args/kwargs in the proper order, and delete the positional cache. However, if `delete_memoized` is just called with the name of the function, be sure to pass in potential arguments in the same order as defined in your function as args only, otherwise Falcon-Caching will not be able to compute the same cache key and delete all memoized versions of it.

Note: Falcon-Caching maintains an internal random version hash for the function. Using delete_memoized will only swap out the version hash, causing the memoize function to recompute results and put them into another key. This leaves any computed caches for this memoized function within the caching backend. It is recommended to use a very high timeout with memoize if using this function, so that when the version hash is swapped, the old cached results would eventually be reclaimed by the caching backend.

---

delete_memoized_verhash (*f*, **args*)

Delete the version hash associated with the function. .. warning:

```
Performing this operation could leave keys behind that have
been created with this version hash. It is up to the application
to make sure that all keys that may have been created with this
```

(continues on next page)

(continued from previous page)

```
version hash at least have timeouts so they will not sit orphaned
in the cache backend.
```

get (**args*, ***kwargs*) → Any
It returns the value for the given key from the cache.

get_dict (**args*, ***kwargs*) → Dict[Any, Any]
It returns the keys and values as dictionary for all requested keys.

get_many (**args*, ***kwargs*) → List[Any]
It returns the list of values matching the list of keys.

has (**args*, ***kwargs*) → bool
It determines if the given key is in the cache.

inc (**args*, ***kwargs*) → Optional[int]
It increments and returns the value of a numerical cache record. Only works for Redis and Redis Sentinel!

memoize (*timeout=None*, *make_name=None*, *unless=None*, *forced_update=None*, *response_filter=None*, *hash_method=<built-in function openssl_md5>*, *cache_none=False*)
Use this to cache the result of a function, taking its arguments into account in the cache key. Information on Memoization. Example:

```
@cache.memoize(timeout=50)
def big_foo(a, b):
    return a + b + random.randrange(0, 1000)
```

::

```
>>> big_foo(5, 2)
753
>>> big_foo(5, 3)
234
>>> big_foo(5, 2)
753
The returned decorated function now has three function attributes
assigned to it.
    **uncached**
        The original undecorated function. readable only
    **cache_timeout**
        The cache timeout value for this function.
        For a custom value to take affect, this must be
        set before the function is called.
        readable and writable
    **make_cache_key**
        A function used in generating the cache_key used.
        readable and writable
```

Parameters

- **timeout** – Default None. If set to an integer, will cache for that amount of time. Unit of time is in seconds.
- **make_name** – Default None. If set this is a function that accepts a single argument, the function name, and returns a new string to be used as the function name. If not set then the function name is used.

- **unless** – Default None. Cache will *always* execute the caching facilities unless this callable is true. This will bypass the caching entirely.
- **forced_update** – Default None. If this callable is true, cache value will be updated regardless cache is expired or not. Useful for background renewal of cached functions.
- **response_filter** – Default None. If not None, the callable is invoked after the cached funtion evaluation, and is given one arguement, the response content. If the callable returns False, the content will not be cached. Useful to prevent caching of code 500 responses.
- **hash_method** – Default hashlib.md5. The hash method used to generate the keys for cached results.
- **cache_none** – Default False. If set to True, add a key exists check when cache.get returns None. This will likely lead to wrongly returned None values in concurrent situations and is not recommended to use.

middleware
Falcon middleware integration

set(**args*, ***kwargs*) → bool
It stores the given key and value in the cache.

set_many(**args*, ***kwargs*) → bool
It stores multiple records based on the dictionary of keys and values provided.

class falcon_caching.**Cache**(*config: Dict[str, Any]*)
This is the central class for the caching

You need to initialize this object to setup the attributes of the caching and then supply the object's middleware to the Falcon app.

Parameters **(dict of str** (*config*) – str): Cache config settings

cache
An initialized 'CACHE_TYPE' cache from the backends.

Type BaseCache

cache_args
Optional list passed during the cache class instantiation.

Type list of str

cache_options (dict of str
str): Optional dictionary passed during the cache class instantiation.

config (dict of str
str): Cache config settings

add(**args*, ***kwargs*) → bool
It adds a given key and value to the cache, but only if no record which such key already exists.

static cached(*timeout: int*)
This is the decorator used to decorate a resource class or the requested method of the resource class

clear() → bool
It clears all cache - if the *CACHE_KEY_PREFIX* config attribute is used then it only removes key starting with that prefix, otherwise it flushes the whole database.

dec(**args*, ***kwargs*) → Optional[int]
It decrements and returns the value of a numerical cache record. Only works for Redis and Redis Sentinel!

delete(**args*, ***kwargs*) → bool
It deletes the cached record based on the provided key.

delete_many(**args*, ***kwargs*) → bool
It deletes all cached record matching the list of keys provided.

delete_memoized(*f*, **args*, ***kwargs*)
Deletes the specified functions caches, based by given parameters. If parameters are given, only the functions that were memoized with them will be erased. Otherwise all versions of the caches will be forgotten. Example:

```
@cache.memoize(50)
def random_func():
    return random.randrange(1, 50)
@cache.memoize()
def param_func(a, b):
    return a+b+random.randrange(1, 50)
```

::

```
>>> random_func()
43
>>> random_func()
43
>>> cache.delete_memoized(random_func)
>>> random_func()
16
>>> param_func(1, 2)
32
>>> param_func(1, 2)
32
>>> param_func(2, 2)
47
>>> cache.delete_memoized(param_func, 1, 2)
>>> param_func(1, 2)
13
>>> param_func(2, 2)
47
```

Delete memoized is also smart about instance methods vs class methods. When passing a instancemethod, it will only clear the cache related to that instance of that object. (object uniqueness can be overridden by defining the __repr__ method, such as user id). When passing a classmethod, it will clear all caches related across all instances of that class. Example:

```
class Adder(object):
    @cache.memoize()
    def add(self, b):
        return b + random.random()
```

::

```
>>> adder1 = Adder()
>>> adder2 = Adder()
>>> adder1.add(3)
3.23214234
>>> adder2.add(3)
```

(continues on next page)

(continued from previous page)

```
3.60898509
>>> cache.delete_memoized(adder1.add)
>>> adder1.add(3)
3.01348673
>>> adder2.add(3)
3.60898509
>>> cache.delete_memoized(Adder.add)
>>> adder1.add(3)
3.53235667
>>> adder2.add(3)
3.72341788
```

Parameters

- **fname** – The memoized function.
- ***args** – A list of positional parameters used with memoized function.
- ****kwargs** – A dict of named parameters used with memoized function.

Note: Falcon-Caching uses inspect to order kwargs into positional args when the function is memoized. If you pass a function reference into `fname`, Falcon-Caching will be able to place the args/kwargs in the proper order, and delete the positional cache. However, if `delete_memoized` is just called with the name of the function, be sure to pass in potential arguments in the same order as defined in your function as args only, otherwise Falcon-Caching will not be able to compute the same cache key and delete all memoized versions of it.

Note: Falcon-Caching maintains an internal random version hash for the function. Using delete_memoized will only swap out the version hash, causing the memoize function to recompute results and put them into another key. This leaves any computed caches for this memoized function within the caching backend. It is recommended to use a very high timeout with memoize if using this function, so that when the version hash is swapped, the old cached results would eventually be reclaimed by the caching backend.

---

delete_memoized_verhash (*f*, **args*)
Delete the version hash associated with the function. .. warning:

```
Performing this operation could leave keys behind that have
been created with this version hash. It is up to the application
to make sure that all keys that may have been created with this
version hash at least have timeouts so they will not sit orphaned
in the cache backend.
```

get (**args*, ***kwargs*) → Any
It returns the value for the given key from the cache.

get_dict (**args*, ***kwargs*) → Dict[Any, Any]
It returns the keys and values as dictionary for all requested keys.

get_many (**args*, ***kwargs*) → List[Any]
It returns the list of values matching the list of keys.

has (**args*, ***kwargs*) → bool
It determines if the given key is in the cache.

inc(**args*, ***kwargs*) → Optional[int]
It increments and returns the value of a numerical cache record. Only works for Redis and Redis Sentinel!

memoize(*timeout=None*, *make_name=None*, *unless=None*, *forced_update=None*, *response_filter=None*, *hash_method=<built-in function openssl_md5>*, *cache_none=False*)
Use this to cache the result of a function, taking its arguments into account in the cache key. Information on Memoization. Example:

```
@cache.memoize(timeout=50)
def big_foo(a, b):
    return a + b + random.randrange(0, 1000)
```

::

```
>>> big_foo(5, 2)
753
>>> big_foo(5, 3)
234
>>> big_foo(5, 2)
753
The returned decorated function now has three function attributes
assigned to it.
    **uncached**
        The original undecorated function. readable only
    **cache_timeout**
        The cache timeout value for this function.
        For a custom value to take affect, this must be
        set before the function is called.
        readable and writable
    **make_cache_key**
        A function used in generating the cache_key used.
        readable and writable
```

Parameters

- **timeout** – Default None. If set to an integer, will cache for that amount of time. Unit of time is in seconds.
- **make_name** – Default None. If set this is a function that accepts a single argument, the function name, and returns a new string to be used as the function name. If not set then the function name is used.
- **unless** – Default None. Cache will *always* execute the caching facilities unless this callable is true. This will bypass the caching entirely.
- **forced_update** – Default None. If this callable is true, cache value will be updated regardless cache is expired or not. Useful for background renewal of cached functions.
- **response_filter** – Default None. If not None, the callable is invoked after the cached funtion evaluation, and is given one arguement, the response content. If the callable returns False, the content will not be cached. Useful to prevent caching of code 500 responses.
- **hash_method** – Default hashlib.md5. The hash method used to generate the keys for cached results.
- **cache_none** – Default False. If set to True, add a key exists check when cache.get returns None. This will likely lead to wrongly returned None values in concurrent situations and is not recommended to use.

middleware
Falcon middleware integration

set(**args*, ***kwargs*) → bool
It stores the given key and value in the cache.

set_many(**args*, ***kwargs*) → bool
It stores multiple records based on the dictionary of keys and values provided.

Backends

BaseCache

class falcon_caching.backends.base.**BaseCache**(*default_timeout=300*)
Baseclass for the cache systems. All the cache systems implement this API or a superset of it.

Parameters **default_timeout** – The default timeout (in seconds) that is used if no timeout is specified on *set()*. A timeout of 0 indicates that the cache never expires.

add(*key*, *value*, *timeout=None*)
Works like *set()* but does not overwrite the values of already existing keys.

Parameters

- **key** – the key to set
- **value** – the value for the key
- **timeout** – the cache timeout for the key in seconds (if not specified, it uses the default timeout). A timeout of 0 idicates that the cache never expires.

Returns Same as *set()*, but also `False` for already existing keys.

Return type boolean

clear()
Clears the cache. Keep in mind that not all caches support completely clearing the cache.

Returns Whether the cache has been cleared.

Return type boolean

dec(*key*, *delta=1*)
Decrements the value of a key by *delta*. If the key does not yet exist it is initialized with *-delta*.

For supporting caches this is an atomic operation.

Parameters

- **key** – the key to increment.
- **delta** – the delta to subtract.

Returns The new value or *None* for backend errors.

delete(*key*)
Delete *key* from the cache.

Parameters **key** – the key to delete.

Returns Whether the key existed and has been deleted.

Return type boolean

delete_many(**keys*)
Deletes multiple keys at once.

Parameters **keys** – The function accepts multiple keys as positional arguments.

Returns Whether all given keys have been deleted.

Return type boolean

get(*key*)
Look up key in the cache and return the value for it.

Parameters **key** – the key to be looked up.

Returns The value if it exists and is readable, else `None`.

get_dict(**keys*)
Like *get_many()* but return a dict:

```
d = cache.get_dict("foo", "bar")
foo = d["foo"]
bar = d["bar"]
```

Parameters **keys** – The function accepts multiple keys as positional arguments.

get_many(**keys*)
Returns a list of values for the given keys. For each key an item in the list is created:

```
foo, bar = cache.get_many("foo", "bar")
```

Has the same error handling as *get()*.

Parameters **keys** – The function accepts multiple keys as positional arguments.

has(*key*)
Checks if a key exists in the cache without returning it. This is a cheap operation that bypasses loading the actual data on the backend.

This method is optional and may not be implemented on all caches.

Parameters **key** – the key to check

inc(*key*, *delta=1*)
Increments the value of a key by *delta*. If the key does not yet exist it is initialized with *delta*.

For supporting caches this is an atomic operation.

Parameters

- **key** – the key to increment.
- **delta** – the delta to add.

Returns The new value or `None` for backend errors.

set(*key*, *value*, *timeout=None*)
Add a new key/value to the cache (overwrites value, if key already exists in the cache).

Parameters

- **key** – the key to set
- **value** – the value for the key

- **timeout** – the cache timeout for the key in seconds (if not specified, it uses the default timeout). A timeout of 0 indicates that the cache never expires.

Returns `True` if key has been updated, `False` for backend errors. Pickling errors, however, will raise a subclass of `pickle.PickleError`.

Return type boolean

set_many(*mapping*, *timeout=None*)
Sets multiple keys and values from a mapping.

Parameters

- **mapping** – a mapping with the keys/values to set.
- **timeout** – the cache timeout for the key in seconds (if not specified, it uses the default timeout). A timeout of 0 idicates that the cache never expires.

Returns Whether all given keys have been set.

Return type boolean

NullCache

class falcon_caching.backends.**NullCache**(*default_timeout=300*)
A cache that doesn't cache. This can be useful for unit testing.

Parameters **default_timeout** – a dummy parameter that is ignored but exists for API compatibility with other caches.

has(*key*)
Checks if a key exists in the cache without returning it. This is a cheap operation that bypasses loading the actual data on the backend.

This method is optional and may not be implemented on all caches.

Parameters **key** – the key to check

SimpleCache

class falcon_caching.backends.**SimpleCache**(*threshold=500*, *default_timeout=300*, *ignore_errors=False*)
Simple memory cache for single process environments. This class exists mainly for the development server and is not 100% thread safe. It tries to use as many atomic operations as possible and no locks for simplicity but it could happen under heavy load that keys are added multiple times.

Parameters

- **threshold** – the maximum number of items the cache stores before it starts deleting some.
- **default_timeout** – the default timeout that is used if no timeout is specified on `set()`. A timeout of 0 indicates that the cache never expires.
- **ignore_errors** – If set to `True` the `delete_many()` method will ignore any errors that occured during the deletion process. However, if it is set to `False` it will stop on the first error. Defaults to `False`.

add(*key*, *value*, *timeout=None*)
Works like *set()* but does not overwrite the values of already existing keys.

Parameters

- **key** – the key to set
- **value** – the value for the key
- **timeout** – the cache timeout for the key in seconds (if not specified, it uses the default timeout). A timeout of 0 idicates that the cache never expires.

Returns Same as *set()*, but also `False` for already existing keys.

Return type boolean

delete(*key*)
Delete *key* from the cache.

Parameters **key** – the key to delete.

Returns Whether the key existed and has been deleted.

Return type boolean

get(*key*)
Look up key in the cache and return the value for it.

Parameters **key** – the key to be looked up.

Returns The value if it exists and is readable, else `None`.

has(*key*)
Checks if a key exists in the cache without returning it. This is a cheap operation that bypasses loading the actual data on the backend.

This method is optional and may not be implemented on all caches.

Parameters **key** – the key to check

set(*key*, *value*, *timeout=None*)
Add a new key/value to the cache (overwrites value, if key already exists in the cache).

Parameters

- **key** – the key to set
- **value** – the value for the key
- **timeout** – the cache timeout for the key in seconds (if not specified, it uses the default timeout). A timeout of 0 indicates that the cache never expires.

Returns `True` if key has been updated, `False` for backend errors. Pickling errors, however, will raise a subclass of `pickle.PickleError`.

Return type boolean

FileSystemCache

class falcon_caching.backends.**FileSystemCache**(*cache_dir*, *threshold=500*, *default_timeout=300*, *mode=384*, *hash_method=<built-in function openssl_md5>*, *ignore_errors=False*)
A cache that stores the items on the file system. This cache depends on being the only user of the *cache_dir*. Make absolutely sure that nobody but this cache stores files there or otherwise the cache will randomly delete files therein.

Parameters

- **cache_dir** – the directory where cache files are stored.
- **threshold** – the maximum number of items the cache stores before it starts deleting some. A threshold value of 0 indicates no threshold.
- **default_timeout** – the default timeout that is used if no timeout is specified on `set()`. A timeout of 0 indicates that the cache never expires.
- **mode** – the file mode wanted for the cache files, default 0600
- **hash_method** – Default hashlib.md5. The hash method used to generate the filename for cached results.
- **ignore_errors** – If set to `True` the `delete_many()` method will ignore any errors that occured during the deletion process. However, if it is set to `False` it will stop on the first error. Defaults to `False`.

add(*key*, *value*, *timeout=None*)
Works like *set()* but does not overwrite the values of already existing keys.

Parameters

- **key** – the key to set
- **value** – the value for the key
- **timeout** – the cache timeout for the key in seconds (if not specified, it uses the default timeout). A timeout of 0 idicates that the cache never expires.

Returns Same as *set()*, but also `False` for already existing keys.

Return type boolean

clear()
Clears the cache. Keep in mind that not all caches support completely clearing the cache.

Returns Whether the cache has been cleared.

Return type boolean

delete(*key*, *mgmt_element=False*)
Delete *key* from the cache.

Parameters **key** – the key to delete.

Returns Whether the key existed and has been deleted.

Return type boolean

get(*key*)
Look up key in the cache and return the value for it.

Parameters **key** – the key to be looked up.

Returns The value if it exists and is readable, else `None`.

has(*key*)
Checks if a key exists in the cache without returning it. This is a cheap operation that bypasses loading the actual data on the backend.

This method is optional and may not be implemented on all caches.

Parameters **key** – the key to check

set(*key*, *value*, *timeout=None*, *mgmt_element=False*)
Add a new key/value to the cache (overwrites value, if key already exists in the cache).

Parameters

- **key** – the key to set
- **value** – the value for the key
- **timeout** – the cache timeout for the key in seconds (if not specified, it uses the default timeout). A timeout of 0 indicates that the cache never expires.

Returns `True` if key has been updated, `False` for backend errors. Pickling errors, however, will raise a subclass of `pickle.PickleError`.

Return type boolean

RedisCache

class falcon_caching.backends.**Redis**(*host='localhost'*, *port=6379*, *password=None*, *db=0*, *default_timeout=300*, *key_prefix=None*, ***kwargs*)

Uses the Redis key-value store as a cache backend.

The first argument can be either a string denoting address of the Redis server or an object resembling an instance of a redis.Redis class.

Note: Python Redis API already takes care of encoding unicode strings on the fly.

Parameters

- **host** – address of the Redis server or an object which API is compatible with the official Python Redis client (redis-py).
- **port** – port number on which Redis server listens for connections.
- **password** – password authentication for the Redis server.
- **db** – db (zero-based numeric index) on Redis Server to connect.
- **default_timeout** – the default timeout that is used if no timeout is specified on `set()`. A timeout of 0 indicates that the cache never expires.
- **key_prefix** – A prefix that should be added to all keys.

Any additional keyword arguments will be passed to `redis.Redis`.

add(*key*, *value*, *timeout=None*)

Works like *set()* but does not overwrite the values of already existing keys.

Parameters

- **key** – the key to set
- **value** – the value for the key
- **timeout** – the cache timeout for the key in seconds (if not specified, it uses the default timeout). A timeout of 0 idicates that the cache never expires.

Returns Same as *set()*, but also `False` for already existing keys.

Return type boolean

clear()

Clears the cache. Keep in mind that not all caches support completely clearing the cache.

Returns Whether the cache has been cleared.

Return type boolean

dec(*key*, *delta=1*)
Decrements the value of a key by *delta*. If the key does not yet exist it is initialized with *-delta*.

For supporting caches this is an atomic operation.

Parameters

- **key** – the key to increment.
- **delta** – the delta to subtract.

Returns The new value or *None* for backend errors.

delete(*key*)
Delete *key* from the cache.

Parameters **key** – the key to delete.

Returns Whether the key existed and has been deleted.

Return type boolean

delete_many(**keys*)
Deletes multiple keys at once.

Parameters **keys** – The function accepts multiple keys as positional arguments.

Returns Whether all given keys have been deleted.

Return type boolean

dump_object(*value*)
Dumps an object into a string for redis. By default it serializes integers as regular string and pickle dumps everything else.

get(*key*)
Look up key in the cache and return the value for it.

Parameters **key** – the key to be looked up.

Returns The value if it exists and is readable, else `None`.

get_many(**keys*)
Returns a list of values for the given keys. For each key an item in the list is created:

```
foo, bar = cache.get_many("foo", "bar")
```

Has the same error handling as *get()*.

Parameters **keys** – The function accepts multiple keys as positional arguments.

has(*key*)
Checks if a key exists in the cache without returning it. This is a cheap operation that bypasses loading the actual data on the backend.

This method is optional and may not be implemented on all caches.

Parameters **key** – the key to check

inc(*key*, *delta=1*)
Increments the value of a key by *delta*. If the key does not yet exist it is initialized with *delta*.

For supporting caches this is an atomic operation.

Parameters

- **key** – the key to increment.

- **delta** – the delta to add.

Returns The new value or `None` for backend errors.

load_object(*value*)
The reversal of *dump_object()*. This might be called with None.

set(*key*, *value*, *timeout=None*)
Add a new key/value to the cache (overwrites value, if key already exists in the cache).

Parameters

- **key** – the key to set
- **value** – the value for the key
- **timeout** – the cache timeout for the key in seconds (if not specified, it uses the default timeout). A timeout of 0 indicates that the cache never expires.

Returns `True` if key has been updated, `False` for backend errors. Pickling errors, however, will raise a subclass of `pickle.PickleError`.

Return type boolean

set_many(*mapping*, *timeout=None*)
Sets multiple keys and values from a mapping.

Parameters

- **mapping** – a mapping with the keys/values to set.
- **timeout** – the cache timeout for the key in seconds (if not specified, it uses the default timeout). A timeout of 0 idicates that the cache never expires.

Returns Whether all given keys have been set.

Return type boolean

unlink(**keys*)
when redis-py >= 3.0.0 and redis > 4, support this operation

RedisSentinelCache

class falcon_caching.backends.**RedisSentinel**(*sentinels=None*, *master=None*, *password=None*, *db=0*, *default_timeout=300*, *key_prefix=None*, ***kwargs*)
Uses the Redis key-value store as a cache backend.

The first argument can be either a string denoting address of the Redis server or an object resembling an instance of a redis.Redis class.

Note: Python Redis API already takes care of encoding unicode strings on the fly.

Parameters

- **sentinels** – A list or a tuple of Redis sentinel addresses.
- **master** – The name of the master server in a sentinel configuration.
- **password** – password authentication for the Redis server.
- **db** – db (zero-based numeric index) on Redis Server to connect.
- **default_timeout** – the default timeout that is used if no timeout is specified on `set()`. A timeout of 0 indicates that the cache never expires.

- **key_prefix** – A prefix that should be added to all keys.

Any additional keyword arguments will be passed to `redis.sentinel.Sentinel`.

UWSGICache

class falcon_caching.backends.**UWSGICache**(*default_timeout=300*, *cache=''*)

Implements the cache using uWSGI's caching framework.

Note: This class cannot be used when running under PyPy, because the uWSGI API implementation for PyPy is lacking the needed functionality.

Parameters

- **default_timeout** – The default timeout in seconds.
- **cache** – The name of the caching instance to connect to, for example: mycache@localhost:3031, defaults to an empty string, which means uWSGI will cache in the local instance. If the cache is in the same instance as the werkzeug app, you only have to provide the name of the cache.

add(*key*, *value*, *timeout=None*)

Works like *set()* but does not overwrite the values of already existing keys.

Parameters

- **key** – the key to set
- **value** – the value for the key
- **timeout** – the cache timeout for the key in seconds (if not specified, it uses the default timeout). A timeout of 0 idicates that the cache never expires.

Returns Same as *set()*, but also `False` for already existing keys.

Return type boolean

clear()

Clears the cache. Keep in mind that not all caches support completely clearing the cache.

Returns Whether the cache has been cleared.

Return type boolean

delete(*key*)

Delete *key* from the cache.

Parameters **key** – the key to delete.

Returns Whether the key existed and has been deleted.

Return type boolean

get(*key*)

Look up key in the cache and return the value for it.

Parameters **key** – the key to be looked up.

Returns The value if it exists and is readable, else `None`.

has(*key*)
Checks if a key exists in the cache without returning it. This is a cheap operation that bypasses loading the actual data on the backend.

This method is optional and may not be implemented on all caches.

Parameters **key** – the key to check

set(*key*, *value*, *timeout=None*)
Add a new key/value to the cache (overwrites value, if key already exists in the cache).

Parameters

- **key** – the key to set
- **value** – the value for the key
- **timeout** – the cache timeout for the key in seconds (if not specified, it uses the default timeout). A timeout of 0 indicates that the cache never expires.

Returns `True` if key has been updated, `False` for backend errors. Pickling errors, however, will raise a subclass of `pickle.PickleError`.

Return type boolean

MemcachedCache

class falcon_caching.backends.**MemcachedCache**(*servers=None*, *default_timeout=300*, *key_prefix=None*)
A cache that uses memcached as backend.

The first argument can either be an object that resembles the API of a `memcache.Client` or a tuple/list of server addresses. In the event that a tuple/list is passed, Werkzeug tries to import the best available memcache library.

This cache looks into the following packages/modules to find bindings for memcached:

- `pylibmc`
- `google.appengine.api.memcached`
- `memcached`
- `libmc`

Implementation notes: This cache backend works around some limitations in memcached to simplify the interface. For example unicode keys are encoded to utf-8 on the fly. Methods such as `get_dict()` return the keys in the same format as passed. Furthermore all get methods silently ignore key errors to not cause problems when untrusted user data is passed to the get methods which is often the case in web applications.

Parameters

- **servers** – a list or tuple of server addresses or alternatively a `memcache.Client` or a compatible client.
- **default_timeout** – the default timeout that is used if no timeout is specified on `set()`. A timeout of 0 indicates that the cache never expires.
- **key_prefix** – a prefix that is added before all keys. This makes it possible to use the same memcached server for different applications. Keep in mind that `clear()` will also clear keys with a different prefix.

add(*key*, *value*, *timeout=None*)
Works like *`set()`* but does not overwrite the values of already existing keys.

Parameters

- **key** – the key to set
- **value** – the value for the key
- **timeout** – the cache timeout for the key in seconds (if not specified, it uses the default timeout). A timeout of 0 idicates that the cache never expires.

Returns Same as *set()*, but also `False` for already existing keys.

Return type boolean

clear()
Clears the cache. Keep in mind that not all caches support completely clearing the cache.

Returns Whether the cache has been cleared.

Return type boolean

dec(*key*, *delta=1*)
Decrements the value of a key by *delta*. If the key does not yet exist it is initialized with *-delta*.

For supporting caches this is an atomic operation.

Parameters

- **key** – the key to increment.
- **delta** – the delta to subtract.

Returns The new value or *None* for backend errors.

delete(*key*)
Delete *key* from the cache.

Parameters **key** – the key to delete.

Returns Whether the key existed and has been deleted.

Return type boolean

delete_many(**keys*)
Deletes multiple keys at once.

Parameters **keys** – The function accepts multiple keys as positional arguments.

Returns Whether all given keys have been deleted.

Return type boolean

get(*key*)
Look up key in the cache and return the value for it.

Parameters **key** – the key to be looked up.

Returns The value if it exists and is readable, else `None`.

get_dict(**keys*)
Like *get_many()* but return a dict:

```
d = cache.get_dict("foo", "bar")
foo = d["foo"]
bar = d["bar"]
```

Parameters **keys** – The function accepts multiple keys as positional arguments.

get_many(**keys*)
Returns a list of values for the given keys. For each key an item in the list is created:

```
foo, bar = cache.get_many("foo", "bar")
```

Has the same error handling as *get()*.

Parameters **keys** – The function accepts multiple keys as positional arguments.

has(*key*)
Checks if a key exists in the cache without returning it. This is a cheap operation that bypasses loading the actual data on the backend.

This method is optional and may not be implemented on all caches.

Parameters **key** – the key to check

import_preferred_memcache_lib(*servers*)
Returns an initialized memcache client. Used by the constructor.

inc(*key*, *delta=1*)
Increments the value of a key by *delta*. If the key does not yet exist it is initialized with *delta*.

For supporting caches this is an atomic operation.

Parameters

- **key** – the key to increment.
- **delta** – the delta to add.

Returns The new value or `None` for backend errors.

set(*key*, *value*, *timeout=None*)
Add a new key/value to the cache (overwrites value, if key already exists in the cache).

Parameters

- **key** – the key to set
- **value** – the value for the key
- **timeout** – the cache timeout for the key in seconds (if not specified, it uses the default timeout). A timeout of 0 indicates that the cache never expires.

Returns `True` if key has been updated, `False` for backend errors. Pickling errors, however, will raise a subclass of `pickle.PickleError`.

Return type boolean

set_many(*mapping*, *timeout=None*)
Sets multiple keys and values from a mapping.

Parameters

- **mapping** – a mapping with the keys/values to set.
- **timeout** – the cache timeout for the key in seconds (if not specified, it uses the default timeout). A timeout of 0 idicates that the cache never expires.

Returns Whether all given keys have been set.

Return type boolean

SASLMemcachedCache

class falcon_caching.backends.**SASLMemcachedCache**(*servers=None*, *default_timeout=300*, *key_prefix=None*, *username=None*, *password=None*, ***kwargs*)

SpreadSASLMemcachedCache

class falcon_caching.backends.**SpreadSASLMemcachedCache**(**args*, ***kwargs*)

Simple Subclass of SASLMemcached client that will spread the value across multiple keys if they are bigger than a given treshold.

Spreading requires using pickle to store the value, which can significantly impact the performance.

delete(*key*)

Delete *key* from the cache.

Parameters **key** – the key to delete.

Returns Whether the key existed and has been deleted.

Return type boolean

delete_many(**keys*)

Deletes multiple keys at once.

Parameters **keys** – The function accepts multiple keys as positional arguments.

Returns Whether all given keys have been deleted.

Return type boolean

get(*key*, *chunk=True*)

Get a cached value.

Parameters **chunk** – If set to `False`, it will return a cached value that is spread across multiple keys.

has(*key*)

Checks if a key exists in the cache without returning it. This is a cheap operation that bypasses loading the actual data on the backend.

This method is optional and may not be implemented on all caches.

Parameters **key** – the key to check

set(*key*, *value*, *timeout=None*, *chunk=True*)

Set a value in cache, potentially spreading it across multiple key.

Parameters

- **key** – The cache key.
- **value** – The value to cache.
- **timeout** – The timeout after which the cache will be invalidated.
- **chunk** – If set to *False*, then spreading across multiple keys is disabled. This can be faster, but it will fail if the value is bigger than the chunks. It requires you to get back the object by specifying that it is not spread.

AsyncBackends

BaseCache

NullCache

SimpleCache

FileSystemCache

RedisCache

RedisSentinelCache

MemcachedCache

12.4 Additional Information

12.4.1 Changelog

Version 1.1.0

- *coredis* Python dependency (used in async) upgrade to version 3.0+ - Thanks @alisaifee!
- Dropping support for Python 3.6

Version 1.0.1

- Documentation fix - AsyncBackend API reference was missing

Version 1.0.0

- Async support has been added
- Switching CI from Travis to GitHub Actions

Version 0.3.4

- Falcon 3.0.0 has renamed the *Response.body* to *Response.text*

Version 0.3.3

- Fixing the issue with multiple decorators when *@cache.cached()* is not the topmost one
- Fixing document readability issues in Sphinx

Version 0.3.1

- Added a new memoize() method to cache arbitrary methods with their arguments

Version 0.3.0

- *Cache.memoize()* and *Cache.delete_memoized()* methods were added to allow you to cache the result of other non-resource related functions with their argurments.

Version 0.2.0

- The `Content-Type` header is cached now, except when this is turned off by the `CACHE_CONTENT_TYPE_JSON_ONLY` setting, see and in the docs
- Added a safer method to identify the *on_* methods to decorate, see
- The *request_body* is no longer included in the cache key, see

Version 0.1.0

- Initial public release

12.4.2 License

```
MIT License

Copyright (c) 2019 Zoltan Fedor

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.
```

- search

Python Module Index

f

Index

F

G

H

I

L

M

N

R

S

U