

---

# **aiocache Documentation**

*Release 0.6.1*

**Manuel Miranda**

**Apr 03, 2018**



---

## Contents

---

<b>1</b>	<b>Installing</b>	<b>1</b>
<b>2</b>	<b>Usage</b>	<b>3</b>
<b>3</b>	<b>Contents</b>	<b>7</b>
3.1	Caches . . . . .	7
3.2	Serializers . . . . .	13
3.3	Plugins . . . . .	17
3.4	Configuration . . . . .	19
3.5	Decorators . . . . .	22
3.6	Testing . . . . .	24
<b>4</b>	<b>Indices and tables</b>	<b>27</b>
	<b>Python Module Index</b>	<b>29</b>



# CHAPTER 1

---

## Installing

---

```
pip install aiocache
```

If you don't need redis or memcached support, you can install as follows:

```
AIOCACHE_REDIS=no pip install aiocache      # Don't install redis client (aioredis)
AIOCACHE_MEMCACHED=no pip install aiocache  # Don't install memcached client
↪ (aiomcache)
```



Using a cache is as simple as

```
>>> import asyncio
>>> loop = asyncio.get_event_loop()
>>> from aiocache import SimpleMemoryCache
>>> cache = SimpleMemoryCache()
>>> loop.run_until_complete(cache.set('key', 'value'))
True
>>> loop.run_until_complete(cache.get('key'))
'value'
```

Here we are using the *SimpleMemoryCache* but you can use any other listed in *Caches*. All caches contain the same minimum interface which consists on the following functions:

- `add`: Only adds key/value if key does not exist. Otherwise raises `ValueError`.
- `get`: Retrieve value identified by key.
- `set`: Sets key/value.
- `multi_get`: Retrieves multiple key/values.
- `multi_set`: Sets multiple key/values.
- `exists`: Returns `True` if key exists `False` otherwise.
- `increment`: Increment the value stored in the given key.
- `delete`: Deletes key and returns number of deleted items.
- `clear`: Clears the items stored.
- `raw`: Executes the specified command using the underlying client.

You can also setup cache aliases like in Django settings:

```
1 import asyncio
```

```
2
```

(continues on next page)

(continued from previous page)

```
3 from aiocache import caches, SimpleMemoryCache, RedisCache
4 from aiocache.serializers import StringSerializer, PickleSerializer
5
6 caches.set_config({
7     'default': {
8         'cache': "aiocache.SimpleMemoryCache",
9         'serializer': {
10            'class': "aiocache.serializers.StringSerializer"
11        }
12    },
13    'redis_alt': {
14        'cache': "aiocache.RedisCache",
15        'endpoint': "127.0.0.1",
16        'port': 6379,
17        'timeout': 1,
18        'serializer': {
19            'class': "aiocache.serializers.PickleSerializer"
20        },
21        'plugins': [
22            {'class': "aiocache.plugins.HitMissRatioPlugin"},
23            {'class': "aiocache.plugins.TimingPlugin"}
24        ]
25    }
26 })
27
28
29 async def default_cache():
30     cache = caches.get('default') # This always returns the same instance
31     await cache.set("key", "value")
32
33     assert await cache.get("key") == "value"
34     assert isinstance(cache, SimpleMemoryCache)
35     assert isinstance(cache.serializer, StringSerializer)
36
37
38 async def alt_cache():
39     # This generates a new instance every time! You can also use `caches.create('alt
40     ↪')`
41     # or even `caches.create('alt', namespace="test", etc...)` to override extra args
42     cache = caches.create(**caches.get_alias_config('redis_alt'))
43     await cache.set("key", "value")
44
45     assert await cache.get("key") == "value"
46     assert isinstance(cache, RedisCache)
47     assert isinstance(cache.serializer, PickleSerializer)
48     assert len(cache.plugins) == 2
49     assert cache.endpoint == "127.0.0.1"
50     assert cache.timeout == 1
51     assert cache.port == 6379
52     await cache.close()
53
54 def test_alias():
55     loop = asyncio.get_event_loop()
56     loop.run_until_complete(default_cache())
57     loop.run_until_complete(alt_cache())
58
```

(continues on next page)



(continued from previous page)

```
59 cache = RedisCache()
60 loop.run_until_complete(cache.delete("key"))
61 loop.run_until_complete(cache.close())
62
63 loop.run_until_complete(caches.get('default').close())
64
65
66 if __name__ == "__main__":
67     test_alias()
```

In examples folder you can check different use cases:

- Using cached decorator.
- Using multi\_cached decorator.
- Configuring cache class default args
- Simple LRU plugin for memory
- Using marshmallow as a serializer
- TimingPlugin and HitMissRatioPlugin demos
- Storing a python object in Redis
- Creating a custom serializer class that compresses data
- Integrations with frameworks like Sanic, Aiohttp and Tornado

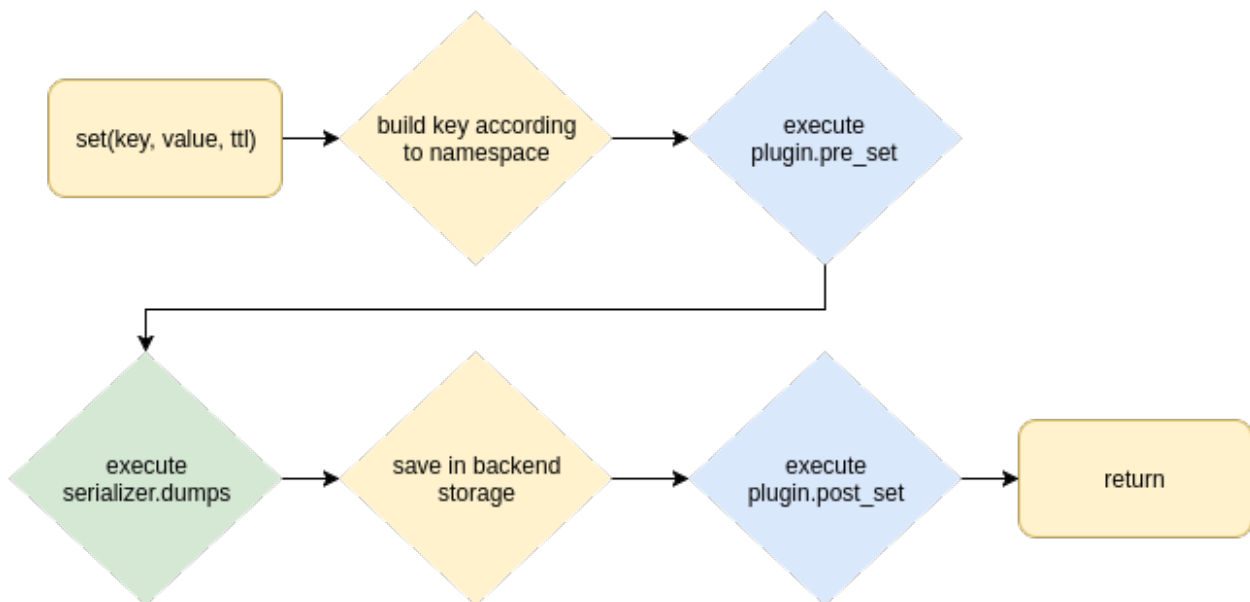


### 3.1 Caches

You can use different caches according to your needs. All the caches implement the same interface.

Caches are always working together with a serializer which transforms data when storing and retrieving from the backend. It may also contain plugins that are able to enrich the behavior of your cache (like adding metrics, logs, etc).

This is the flow of the `set` command:



Let's go with a more specific case. Let's pick Redis as the cache with namespace "test" and PickleSerializer as the serializer:

1. We receive `set("key", "value")`.

2. Hook `pre_set` of all attached plugins (none by default) is called.
3. “key” will become “test:key” when calling `build_key`.
4. “value” will become an array of bytes when calling `serializer.dumps` because of `PickleSerializer`.
5. the byte array is stored together with the key using `set` cmd in Redis.
6. Hook `post_set` of all attached plugins is called.

By default, all commands are covered by a timeout that will trigger an `asyncio.TimeoutError` in case of timeout. Timeout can be set at instance level or when calling the command.

The supported commands are:

- `add`
- `get`
- `set`
- `multi_get`
- `multi_set`
- `delete`
- `exists`
- `increment`
- `expire`
- `clear`
- `raw`

If you feel a command is missing here do not hesitate to [open an issue](#)

### 3.1.1 BaseCache

**class** `aiocache.base.BaseCache` (*serializer=None, plugins=None, namespace=None, timeout=5*)

Base class that agregates the common logic for the different caches that may exist. Cache related available options are:

#### Parameters

- **serializer** – obj derived from `aiocache.serializers.StringSerializer`. Default is `aiocache.serializers.StringSerializer`.
- **plugins** – list of `aiocache.plugins.BasePlugin` derived classes. Default is empty list.
- **namespace** – string to use as default prefix for the key used in all operations of the back-end. Default is None
- **timeout** – int or float in seconds specifying maximum timeout for the operations to last. By default its 5. Use 0 or None if you want to disable it.

**add** (*key, value, ttl=None, dumps\_fn=None, namespace=None, \_conn=None*)

Stores the value in the given key with ttl if specified. Raises an error if the key already exists.

#### Parameters

- **key** – str
- **value** – obj

- **t<sub>ttl</sub>** – int the expiration time in seconds. Due to memcached restrictions if you want compatibility use int. In case you need miliseconds, redis and memory support float t<sub>tls</sub>
- **dumps<sub>fn</sub>** – callable alternative to use as dumps function
- **namespace** – str alternative namespace to use
- **timeout** – int or float in seconds specifying maximum timeout for the operations to last

**Returns** True if key is inserted

**Raises**

- ValueError if key already exists
- `asyncio.TimeoutError` if it lasts more than `self.timeout`

**clear** (*namespace=None, \_conn=None*)

Clears the cache in the cache namespace. If an alternative namespace is given, it will clear those ones instead.

**Parameters**

- **namespace** – str alternative namespace to use
- **timeout** – int or float in seconds specifying maximum timeout for the operations to last

**Returns** True

**Raises** `asyncio.TimeoutError` if it lasts more than `self.timeout`

**close** (*\*args, \_conn=None, \*\*kwargs*)

Perform any resource clean up necessary to exit the program safely. After closing, cmd execution is still possible but you will have to close again before exiting.

**Raises** `asyncio.TimeoutError` if it lasts more than `self.timeout`

**delete** (*key, namespace=None, \_conn=None*)

Deletes the given key.

**Parameters**

- **key** – Key to be deleted
- **namespace** – str alternative namespace to use
- **timeout** – int or float in seconds specifying maximum timeout for the operations to last

**Returns** int number of deleted keys

**Raises** `asyncio.TimeoutError` if it lasts more than `self.timeout`

**exists** (*key, namespace=None, \_conn=None*)

Check key exists in the cache.

**Parameters**

- **key** – str key to check
- **namespace** – str alternative namespace to use
- **timeout** – int or float in seconds specifying maximum timeout for the operations to last

**Returns** True if key exists otherwise False

**Raises** `asyncio.TimeoutError` if it lasts more than `self.timeout`

**expire** (*key, ttl, namespace=None, \_conn=None*)

Set the ttl to the given key. By setting it to 0, it will disable it

#### Parameters

- **key** – str key to expire
- **ttl** – int number of seconds for expiration. If 0, ttl is disabled
- **namespace** – str alternative namespace to use
- **timeout** – int or float in seconds specifying maximum timeout for the operations to last

**Returns** True if set, False if key is not found

**get** (*key, default=None, loads\_fn=None, namespace=None, \_conn=None*)

Get a value from the cache. Returns default if not found.

#### Parameters

- **key** – str
- **default** – obj to return when key is not found
- **loads\_fn** – callable alternative to use as loads function
- **namespace** – str alternative namespace to use
- **timeout** – int or float in seconds specifying maximum timeout for the operations to last

**Returns** obj loaded

**Raises** `asyncio.TimeoutError` if it lasts more than `self.timeout`

**increment** (*key, delta=1, namespace=None, \_conn=None*)

Increments value stored in key by delta (can be negative). If key doesn't exist, it creates the key with delta as value.

#### Parameters

- **key** – str key to check
- **delta** – int amount to increment/decrement
- **namespace** – str alternative namespace to use
- **timeout** – int or float in seconds specifying maximum timeout for the operations to last

**Returns** Value of the key once incremented. -1 if key is not found.

**Raises** `asyncio.TimeoutError` if it lasts more than `self.timeout`

**Raises** `TypeError` if value is not incrementable

**multi\_get** (*keys, loads\_fn=None, namespace=None, \_conn=None*)

Get multiple values from the cache, values not found are Nones.

#### Parameters

- **keys** – list of str
- **loads\_fn** – callable alternative to use as loads function
- **namespace** – str alternative namespace to use
- **timeout** – int or float in seconds specifying maximum timeout for the operations to last

**Returns** list of objs

**Raises** `asyncio.TimeoutError` if it lasts more than `self.timeout`

**multi\_set** (*pairs, ttl=None, dumps\_fn=None, namespace=None, \_conn=None*)

Stores multiple values in the given keys.

**Parameters**

- **pairs** – list of two element iterables. First is key and second is value
- **t1** – int the expiration time in seconds. Due to memcached restrictions if you want compatibility use int. In case you need milliseconds, redis and memory support float t1s
- **dumps\_fn** – callable alternative to use as dumps function
- **namespace** – str alternative namespace to use
- **timeout** – int or float in seconds specifying maximum timeout for the operations to last

**Returns** True**Raises** `asyncio.TimeoutError` if it lasts more than `self.timeout`**raw** (*command, \*args, \_conn=None, \*\*kwargs*)

Send the raw command to the underlying client. Note that by using this CMD you will lose compatibility with other backends.

Due to limitations with aiomcache client, args have to be provided as bytes. For rest of backends, str.

**Parameters**

- **command** – str with the command.
- **timeout** – int or float in seconds specifying maximum timeout for the operations to last

**Returns** whatever the underlying client returns**Raises** `asyncio.TimeoutError` if it lasts more than `self.timeout`**set** (*key, value, ttl=None, dumps\_fn=None, namespace=None, \_conn=None*)

Stores the value in the given key with ttl if specified

**Parameters**

- **key** – str
- **value** – obj
- **t1** – int the expiration time in seconds. Due to memcached restrictions if you want compatibility use int. In case you need milliseconds, redis and memory support float t1s
- **dumps\_fn** – callable alternative to use as dumps function
- **namespace** – str alternative namespace to use
- **timeout** – int or float in seconds specifying maximum timeout for the operations to last

**Returns** True**Raises** `asyncio.TimeoutError` if it lasts more than `self.timeout`

### 3.1.2 RedisCache

**class** `aiocache.RedisCache` (*\*\*kwargs*)**Redis cache implementation with the following components as defaults:**

- serializer: `aiocache.serializers.StringSerializer`
- plugins: []

Config options are:

**Parameters**

- **serializer** – obj derived from *aiocache.serializers.StringSerializer*.
- **plugins** – list of *aiocache.plugins.BasePlugin* derived classes.
- **namespace** – string to use as default prefix for the key used in all operations of the back-end. Default is None.
- **timeout** – int or float in seconds specifying maximum timeout for the operations to last. By default its 5.
- **endpoint** – str with the endpoint to connect to. Default is “127.0.0.1”.
- **port** – int with the port to connect to. Default is 6379.
- **db** – int indicating database to use. Default is 0.
- **password** – str indicating password to use. Default is None.
- **pool\_min\_size** – int minimum pool size for the redis connections pool. Default is 1
- **pool\_max\_size** – int maximum pool size for the redis connections pool. Default is 10

### 3.1.3 SimpleMemoryCache

**class** `aiocache.SimpleMemoryCache (**kwargs)`

Memory cache implementation with the following components as defaults:

- serializer: *aiocache.serializers.StringSerializer*
- plugins: None

Config options are:

#### Parameters

- **serializer** – obj derived from *aiocache.serializers.StringSerializer*.
- **plugins** – list of *aiocache.plugins.BasePlugin* derived classes.
- **namespace** – string to use as default prefix for the key used in all operations of the back-end. Default is None.
- **timeout** – int or float in seconds specifying maximum timeout for the operations to last. By default its 5.

### 3.1.4 MemcachedCache

**class** `aiocache.MemcachedCache (**kwargs)`

Memcached cache implementation with the following components as defaults:

- serializer: *aiocache.serializers.StringSerializer*
- plugins: []

Config options are:

#### Parameters

- **serializer** – obj derived from *aiocache.serializers.StringSerializer*.
- **plugins** – list of *aiocache.plugins.BasePlugin* derived classes.



- **namespace** – string to use as default prefix for the key used in all operations of the backend. Default is None
- **timeout** – int or float in seconds specifying maximum timeout for the operations to last. By default its 5.
- **endpoint** – str with the endpoint to connect to. Default is 127.0.0.1.
- **port** – int with the port to connect to. Default is 11211.
- **pool\_size** – int size for memcached connections pool. Default is 2.

## 3.2 Serializers

Serializers can be attached to backends in order to serialize/deserialize data sent and retrieved from the backend. This allows to apply transformations to data in case you want it to be saved in a specific format in your cache backend. For example, imagine you have your `Model` and want to serialize it to something that Redis can understand (Redis can't store python objects). This is the task of a serializer.

To use a specific serializer:

```
>>> from aiocache import SimpleMemoryCache
>>> from aiocache.serializers import PickleSerializer
cache = SimpleMemoryCache(serializer=PickleSerializer())
```

Currently the following are built in:

- `StringSerializer`: stores data casting it to str. Won't return the same type if the data stored is not a str.
- `PickleSerializer`: ideal for storing any Python object or keeping types.
- `JsonSerializer`: ideal for storing in json format.

In case the current serializers are not covering your needs, you can always define your custom serializer as shown in `examples/serializer_class.py`:

```
1 import sys
2 import asyncio
3 import zlib
4
5 from aiocache import RedisCache
6 from aiocache.serializers import StringSerializer
7
8
9 class CompressionSerializer(StringSerializer):
10
11     # This is needed because zlib works with bytes.
12     # this way the underlying backend knows how to
13     # store/retrieve values
14     encoding = None
15
16     def dumps(self, value):
17         print("I've received:\n{}".format(value))
18         compressed = zlib.compress(value.encode())
19         print("But I'm storing:\n{}".format(compressed))
20         return compressed
21
22     def loads(self, value):
```

(continues on next page)

(continued from previous page)

```

23     print("I've retrieved:\n{}".format(value))
24     decompressed = zlib.decompress(value).decode()
25     print("But I'm returning:\n{}".format(decompressed))
26     return decompressed
27
28
29 cache = RedisCache(serializer=CompressionSerializer(), namespace="main")
30
31
32 async def serializer():
33     text = (
34         "Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod_
↵tempor incididunt"
35         "ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud_
↵exercitation"
36         "ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure_
↵dolor in"
37         "reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur.
↵ Excepteur"
38         "sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt_
↵mollit"
39         "anim id est laborum.")
40     await cache.set("key", text)
41     print("-----")
42     real_value = await cache.get("key")
43
44     compressed_value = await cache.raw("get", "main:key")
45     assert sys.getsizeof(compressed_value) < sys.getsizeof(real_value)
46
47
48 def test_serializer():
49     loop = asyncio.get_event_loop()
50     loop.run_until_complete(serializer())
51     loop.run_until_complete(cache.delete("key"))
52     loop.run_until_complete(cache.close())
53
54
55 if __name__ == "__main__":
56     test_serializer()

```

You can also use marshmallow as your serializer (examples/marshmallow\_serializer\_class.py):

```

1  import random
2  import string
3  import asyncio
4
5  from marshmallow import fields, Schema, post_load
6
7  from aiocache import SimpleMemoryCache
8  from aiocache.serializers import StringSerializer
9
10
11 class RandomModel:
12     MY_CONSTANT = "CONSTANT"
13
14     def __init__(self, int_type=None, str_type=None, dict_type=None, list_type=None):
15         self.int_type = int_type or random.randint(1, 10)

```

(continues on next page)

(continued from previous page)

```

16     self.str_type = str_type or random.choice(string.ascii_lowercase)
17     self.dict_type = dict_type or {}
18     self.list_type = list_type or []
19
20     def __eq__(self, obj):
21         return self.__dict__ == obj.__dict__
22
23
24 class MarshmallowSerializer(Schema, StringSerializer):
25     int_type = fields.Integer()
26     str_type = fields.String()
27     dict_type = fields.Dict()
28     list_type = fields.List(fields.Integer())
29
30     def dumps(self, *args, **kwargs):
31         # dumps returns (data, errors), we just want to save data
32         return super().dumps(*args, **kwargs).data
33
34     def loads(self, *args, **kwargs):
35         # dumps returns (data, errors), we just want to return data
36         return super().loads(*args, **kwargs).data
37
38     @post_load
39     def build_my_type(self, data):
40         return RandomModel(**data)
41
42     class Meta:
43         strict = True
44
45
46 cache = SimpleMemoryCache(serializer=MarshmallowSerializer(), namespace="main")
47
48
49 async def serializer():
50     model = RandomModel()
51     await cache.set("key", model)
52
53     result = await cache.get("key")
54
55     assert result.int_type == model.int_type
56     assert result.str_type == model.str_type
57     assert result.dict_type == model.dict_type
58     assert result.list_type == model.list_type
59
60
61 def test_serializer():
62     loop = asyncio.get_event_loop()
63     loop.run_until_complete(serializer())
64     loop.run_until_complete(cache.delete("key"))
65
66
67 if __name__ == "__main__":
68     test_serializer()

```

By default cache backends assume they are working with `str` types. If your custom implementation transform data to bytes, you will need to set the class attribute `encoding` to `None`.

### 3.2.1 StringSerializer

**class** aiocache.serializers.**StringSerializer** (\*args, \*\*kwargs)

Converts all input values to str. All return values are also str. Be careful because this means that if you store an int (1), you will get back '1'.

The transformation is done by just casting to str in the dumps method.

If you want to keep python types, use PickleSerializer. JsonSerializer may also be useful to keep type of simple python types.

**classmethod** dumps (value)

Serialize the received value casting it to str.

**Parameters** value – obj Anything support cast to str

**Returns** str

encoding = 'utf-8'

**classmethod** loads (value)

Returns value back without transformations

### 3.2.2 PickleSerializer

**class** aiocache.serializers.**PickleSerializer** (\*args, \*\*kwargs)

Transform data to bytes using pickle.dumps and pickle.loads to retrieve it back.

**classmethod** dumps (value)

Serialize the received value using pickle.dumps.

**Parameters** value – obj

**Returns** bytes

**classmethod** loads (value)

Deserialize value using pickle.loads.

**Parameters** value – bytes

**Returns** obj

### 3.2.3 JsonSerializer

**class** aiocache.serializers.**JsonSerializer** (\*args, \*\*kwargs)

Transform data to json string with json.dumps and json.loads to retrieve it back. Check <https://docs.python.org/3/library/json.html#py-to-json-table> for how types are converted.

**classmethod** dumps (value)

Serialize the received value using json.dumps.

**Parameters** value – dict

**Returns** str

**classmethod** loads (value)

Deserialize value using json.loads.

**Parameters** value – str

**Returns** output of json.loads.

### 3.3 Plugins

Plugins can be used to enrich the behavior of the cache. By default all caches are configured without any plugin but can add new ones in the constructor or after initializing the cache class:

```
>>> from aiocache import SimpleMemoryCache
>>> from aiocache.plugins import TimingPlugin
cache = SimpleMemoryCache(plugings=[HitMissRatioPlugin()])
cache.plugings += [TimingPlugin()]
```

You can define your custom plugin by inheriting from *BasePlugin* and overriding the needed methods (the overrides NEED to be async). All commands have `pre_<command_name>` and `post_<command_name>` hooks.

A complete example of using plugins:

```
1 import asyncio
2 import random
3 import logging
4
5 from aiocache import SimpleMemoryCache
6 from aiocache.plugins import HitMissRatioPlugin, TimingPlugin, BasePlugin
7
8
9 logger = logging.getLogger(__name__)
10
11
12 class MyCustomPlugin(BasePlugin):
13
14     async def pre_set(self, *args, **kwargs):
15         logger.info("I'm the pre_set hook being called with %s %s" % (args, kwargs))
16
17     async def post_set(self, *args, **kwargs):
18         logger.info("I'm the post_set hook being called with %s %s" % (args, kwargs))
19
20
21 cache = SimpleMemoryCache(
22     plugings=[HitMissRatioPlugin(), TimingPlugin(), MyCustomPlugin()],
23     namespace="main")
24
25
26 async def run():
27     await cache.set("a", "1")
28     await cache.set("b", "2")
29     await cache.set("c", "3")
30     await cache.set("d", "4")
31
32     possible_keys = ["a", "b", "c", "d", "e", "f"]
33
34     for t in range(1000):
35         await cache.get(random.choice(possible_keys))
36
37     assert cache.hit_miss_ratio["hit_ratio"] > 0.5
38     assert cache.hit_miss_ratio["total"] == 1000
39
40     assert cache.profiling["get_min"] > 0
41     assert cache.profiling["set_min"] > 0
42     assert cache.profiling["get_max"] > 0
```

(continues on next page)

```
43     assert cache.profiling["set_max"] > 0
44
45     print (cache.hit_miss_ratio)
46     print (cache.profiling)
47
48
49 def test_run():
50     loop = asyncio.get_event_loop()
51     loop.run_until_complete(run())
52     loop.run_until_complete(cache.delete("a"))
53     loop.run_until_complete(cache.delete("b"))
54     loop.run_until_complete(cache.delete("c"))
55     loop.run_until_complete(cache.delete("d"))
56
57
58 if __name__ == "__main__":
59     test_run()
```

### 3.3.1 BasePlugin

```
class aiocache.plugins.BasePlugin
```

```
    classmethod add_hook(func, hooks)
    do_nothing(*args, **kwargs)
    post_add(*args, **kwargs)
    post_clear(*args, **kwargs)
    post_delete(*args, **kwargs)
    post_exists(*args, **kwargs)
    post_expire(*args, **kwargs)
    post_get(*args, **kwargs)
    post_increment(*args, **kwargs)
    post_multi_get(*args, **kwargs)
    post_multi_set(*args, **kwargs)
    post_raw(*args, **kwargs)
    post_set(*args, **kwargs)
    pre_add(*args, **kwargs)
    pre_clear(*args, **kwargs)
    pre_delete(*args, **kwargs)
    pre_exists(*args, **kwargs)
    pre_expire(*args, **kwargs)
    pre_get(*args, **kwargs)
    pre_increment(*args, **kwargs)
```

```

pre_multi_get (*args, **kwargs)
pre_multi_set (*args, **kwargs)
pre_raw (*args, **kwargs)
pre_set (*args, **kwargs)

```

### 3.3.2 TimingPlugin

**class** aiocache.plugins.**TimingPlugin**

Calculates average, min and max times each command takes. The data is saved in the cache class as a dict attribute called `profiling`. For example, to access the average time of the operation get, you can do `cache.profiling['get_avg']`

```

post_add (client, *args, took=0, **kwargs)
post_clear (client, *args, took=0, **kwargs)
post_delete (client, *args, took=0, **kwargs)
post_exists (client, *args, took=0, **kwargs)
post_expire (client, *args, took=0, **kwargs)
post_get (client, *args, took=0, **kwargs)
post_increment (client, *args, took=0, **kwargs)
post_multi_get (client, *args, took=0, **kwargs)
post_multi_set (client, *args, took=0, **kwargs)
post_raw (client, *args, took=0, **kwargs)
post_set (client, *args, took=0, **kwargs)
classmethod save_time (method)

```

### 3.3.3 HitMissRatioPlugin

**class** aiocache.plugins.**HitMissRatioPlugin**

Calculates the ratio of hits the cache has. The data is saved in the cache class as a dict attribute called `hit_miss_ratio`. For example, to access the hit ratio of the cache, you can do `cache.hit_miss_ratio['hit_ratio']`. It also provides the “total” and “hits” keys.

```

post_get (client, key, took=0, ret=None)
post_multi_get (client, keys, took=0, ret=None)

```

## 3.4 Configuration

### 3.4.1 Cache aliases

The caches module allows to setup cache configurations and then use them either using an alias or retrieving the config explicitly. To set the config, call `caches.set_config`:

**classmethod** `caches.set_config (config)`

Set (override) the default config for cache aliases from a dict-like structure. The structure is the following:

```

{
  'default': {
    'cache': "aiocache.SimpleMemoryCache",
    'serializer': {
      'class': "aiocache.serializers.StringSerializer"
    }
  },
  'redis_alt': {
    'cache': "aiocache.RedisCache",
    'endpoint': "127.0.0.10",
    'port': 6378,
    'serializer': {
      'class': "aiocache.serializers.PickleSerializer"
    },
    'plugins': [
      {'class': "aiocache.plugins.HitMissRatioPlugin"},
      {'class': "aiocache.plugins.TimingPlugin"}
    ]
  }
}

```

‘default’ key must always exist when passing a new config. Default configuration is:

```

{
  'default': {
    'cache': "aiocache.SimpleMemoryCache",
    'serializer': {
      'class': "aiocache.serializers.StringSerializer"
    }
  }
}

```

You can set your own classes there. The class params accept both str and class types.

All keys in the config are optional, if they are not passed the defaults for the specified class will be used.

To retrieve a copy of the current config, you can use `caches.get_config` or `caches.get_alias_config` for an alias config.

Next snippet shows an example usage:

```

1 import asyncio
2
3 from aiocache import caches, SimpleMemoryCache, RedisCache
4 from aiocache.serializers import StringSerializer, PickleSerializer
5
6 caches.set_config({
7     'default': {
8         'cache': "aiocache.SimpleMemoryCache",
9         'serializer': {
10            'class': "aiocache.serializers.StringSerializer"
11        }
12    },
13    'redis_alt': {
14        'cache': "aiocache.RedisCache",
15        'endpoint': "127.0.0.1",
16        'port': 6379,
17        'timeout': 1,

```

(continues on next page)



(continued from previous page)

```

18     'serializer': {
19         'class': "aiocache.serializers.PickleSerializer"
20     },
21     'plugins': [
22         {'class': "aiocache.plugins.HitMissRatioPlugin"},
23         {'class': "aiocache.plugins.TimingPlugin"}
24     ]
25 }
26 })
27
28
29 async def default_cache():
30     cache = caches.get('default') # This always returns the same instance
31     await cache.set("key", "value")
32
33     assert await cache.get("key") == "value"
34     assert isinstance(cache, SimpleMemoryCache)
35     assert isinstance(cache.serializer, StringSerializer)
36
37
38 async def alt_cache():
39     # This generates a new instance every time! You can also use `caches.create('alt
40     ↪')`
41     # or even `caches.create('alt', namespace="test", etc...)` to override extra args
42     cache = caches.create(**caches.get_alias_config('redis_alt'))
43     await cache.set("key", "value")
44
45     assert await cache.get("key") == "value"
46     assert isinstance(cache, RedisCache)
47     assert isinstance(cache.serializer, PickleSerializer)
48     assert len(cache.plugins) == 2
49     assert cache.endpoint == "127.0.0.1"
50     assert cache.timeout == 1
51     assert cache.port == 6379
52     await cache.close()
53
54 def test_alias():
55     loop = asyncio.get_event_loop()
56     loop.run_until_complete(default_cache())
57     loop.run_until_complete(alt_cache())
58
59     cache = RedisCache()
60     loop.run_until_complete(cache.delete("key"))
61     loop.run_until_complete(cache.close())
62
63     loop.run_until_complete(caches.get('default').close())
64
65
66 if __name__ == "__main__":
67     test_alias()

```

When you do `caches.get('alias_name')`, the cache instance is built lazily the first time. Next accesses will return the **same** instance. If instead of reusing the same instance, you need a new one every time, use `caches.create('alias_name')`. One of the advantages of `caches.create` is that it accepts extra args that then are passed to the cache constructor. This way you can override args like `namespace`, `endpoint`, etc.

## 3.5 Decorators

aiocache comes with a couple of decorators for caching results from asynchronous functions. Do not use the decorator in synchronous functions, it may lead to unexpected behavior.

### 3.5.1 cached

```
class aiocache.cached (ttl=None, key=None, key_from_attr=None, key_builder=None, cache=<class
    'aiocache.backends.memory.SimpleMemoryCache'>, serializer=<class 'aiocache.serializers.JsonSerializer'>, plugins=None, alias=None, noself=False,
    **kwargs)
```

Caches the functions return value into a key generated with module\_name, function\_name and args.

In some cases you will need to send more args to configure the cache object. An example would be endpoint and port for the RedisCache. You can send those args as kwargs and they will be propagated accordingly.

Only one cache instance is created per decorated call. If you expect high concurrency of calls to the same function, you should adapt the pool size as needed.

#### Parameters

- **ttl** – int seconds to store the function call. Default is None which means no expiration.
- **key** – str value to set as key for the function return. Takes precedence over key\_from\_attr param. If key and key\_from\_attr are not passed, it will use module\_name + function\_name + args + kwargs
- **key\_builder** – Callable that allows to build the function dynamically. It receives same args and kwargs as the called function.
- **cache** – cache class to use when calling the set/get operations. Default is aiocache.SimpleMemoryCache.
- **serializer** – serializer instance to use when calling the dumps/loads. Default is JsonSerializer.
- **plugins** – list plugins to use when calling the cmd hooks Default is pulled from the cache class being used.
- **alias** – str specifying the alias to load the config from. If alias is passed, other config parameters are ignored. New cache is created every time.
- **noself** – bool if you are decorating a class function, by default self is also used to generate the key. This will result in same function calls done by different class instances to use different cache keys. Use noself=True if you want to ignore it.

```
1 import asyncio
2
3 from collections import namedtuple
4
5 from aiocache import cached, RedisCache
6 from aiocache.serializers import PickleSerializer
7
8 Result = namedtuple('Result', "content, status")
9
10
11 @cached(
12     ttl=10, cache=RedisCache, key="key", serializer=PickleSerializer(), port=6379,
    ↪ namespace="main")
```

(continues on next page)

(continued from previous page)

```

13 async def cached_call():
14     return Result("content", 200)
15
16
17 def test_cached():
18     cache = RedisCache(endpoint="127.0.0.1", port=6379, namespace="main")
19     loop = asyncio.get_event_loop()
20     loop.run_until_complete(cached_call())
21     assert loop.run_until_complete(cache.exists("key")) is True
22     loop.run_until_complete(cache.delete("key"))
23     loop.run_until_complete(cache.close())
24
25
26 if __name__ == "__main__":
27     test_cached()

```

### 3.5.2 multi\_cached

```

class aiocache.multi_cached(keys_from_attr, key_builder=None, ttl=0, cache=<class 'aiocache.backends.memory.SimpleMemoryCache'>, serializer=<class 'aiocache.serializers.JsonSerializer'>, plugins=None, alias=None,
**kwargs)

```

Only supports functions that return dict-like structures. This decorator caches each key/value of the dict-like object returned by the function.

If `key_builder` is passed, before storing the key, it will be transformed according to the output of the function.

If the attribute specified to be the key is an empty list, the cache will be ignored and the function will be called as expected.

Only one cache instance is created per decorated function. If you expect high concurrency of calls to the same function, you should adapt the pool size as needed.

#### Parameters

- **keys\_from\_attr** – arg or kwarg name from the function containing an iterable to use as keys to index in the cache.
- **key\_builder** – Callable that allows to change the format of the keys before storing. Receives the key and same args and kwargs as the called function.
- **ttl** – int seconds to store the keys. Default is 0 which means no expiration.
- **cache** – cache class to use when calling the `multi_set/multi_get` operations. Default is `aiocache.SimpleMemoryCache`.
- **serializer** – serializer instance to use when calling the `dumps/loads`. Default is `JsonSerializer`.
- **plugins** – plugins to use when calling the cmd hooks Default is pulled from the cache class being used.
- **alias** – str specifying the alias to load the config from. If alias is passed, other config parameters are ignored. New cache is created every time.

```

1 import asyncio
2
3 from aiocache import multi_cached, RedisCache

```

(continues on next page)

(continued from previous page)

```

4
5  DICTIONARY = {
6      'a': "Z",
7      'b': "Y",
8      'c': "X",
9      'd': "W"
10 }
11
12
13 @multi_cached("ids", cache=RedisCache, namespace="main")
14 async def multi_cached_ids(ids=None):
15     return {id_: DICTIONARY[id_] for id_ in ids}
16
17
18 @multi_cached("keys", cache=RedisCache, namespace="main")
19 async def multi_cached_keys(keys=None):
20     return {id_: DICTIONARY[id_] for id_ in keys}
21
22
23 cache = RedisCache(endpoint="127.0.0.1", port=6379, namespace="main")
24
25
26 def test_multi_cached():
27     loop = asyncio.get_event_loop()
28     loop.run_until_complete(multi_cached_ids(ids=['a', 'b']))
29     loop.run_until_complete(multi_cached_ids(ids=['a', 'c']))
30     loop.run_until_complete(multi_cached_keys(keys=['d']))
31
32     assert loop.run_until_complete(cache.exists('a'))
33     assert loop.run_until_complete(cache.exists('b'))
34     assert loop.run_until_complete(cache.exists('c'))
35     assert loop.run_until_complete(cache.exists('d'))
36
37     loop.run_until_complete(cache.delete("a"))
38     loop.run_until_complete(cache.delete("b"))
39     loop.run_until_complete(cache.delete("c"))
40     loop.run_until_complete(cache.delete("d"))
41     loop.run_until_complete(cache.close())
42
43
44 if __name__ == "__main__":
45     test_multi_cached()

```

## 3.6 Testing

It's really easy to cut the dependency with aiocache functionality:

```

import asyncio

from unittest import MagicMock

from aiocache.base import BaseCache

```

(continues on next page)

(continued from previous page)

```
async def async_main():
    mocked_cache = MagicMock(spec=BaseCache)
    mocked_cache.get.return_value = "world"
    print(await mocked_cache.get("hello"))

if __name__ == "__main__":
    loop = asyncio.get_event_loop()
    loop.run_until_complete(async_main())
```

Note that we are passing the *BaseCache* as the spec for the Mock (you need to install `asynctest`).

Also, for debugging purposes you can use `AIOCACHE_DISABLE = 1 python myscript.py` to disable caching.



## CHAPTER 4

---

### Indices and tables

---

- `genindex`
- `modindex`
- `search`





**a**

`aiocache`, [23](#)



**A**

add() (aiocache.base.BaseCache method), 8  
 add\_hook() (aiocache.plugins.BasePlugin class method), 18  
 aiocache (module), 22, 23

**B**

BaseCache (class in aiocache.base), 8  
 BasePlugin (class in aiocache.plugins), 18

**C**

cached (class in aiocache), 22  
 clear() (aiocache.base.BaseCache method), 9  
 close() (aiocache.base.BaseCache method), 9

**D**

delete() (aiocache.base.BaseCache method), 9  
 do\_nothing() (aiocache.plugins.BasePlugin method), 18  
 dumps() (aiocache.serializers.JsonSerializer class method), 16  
 dumps() (aiocache.serializers.PickleSerializer class method), 16  
 dumps() (aiocache.serializers.StringSerializer class method), 16

**E**

encoding (aiocache.serializers.StringSerializer attribute), 16  
 exists() (aiocache.base.BaseCache method), 9  
 expire() (aiocache.base.BaseCache method), 9

**G**

get() (aiocache.base.BaseCache method), 10

**H**

HitMissRatioPlugin (class in aiocache.plugins), 19

**I**

increment() (aiocache.base.BaseCache method), 10

**J**

JsonSerializer (class in aiocache.serializers), 16

**L**

loads() (aiocache.serializers.JsonSerializer class method), 16  
 loads() (aiocache.serializers.PickleSerializer class method), 16  
 loads() (aiocache.serializers.StringSerializer class method), 16

**M**

MemcachedCache (class in aiocache), 12  
 multi\_cached (class in aiocache), 23  
 multi\_get() (aiocache.base.BaseCache method), 10  
 multi\_set() (aiocache.base.BaseCache method), 10

**P**

PickleSerializer (class in aiocache.serializers), 16  
 post\_add() (aiocache.plugins.BasePlugin method), 18  
 post\_add() (aiocache.plugins.TimingPlugin method), 19  
 post\_clear() (aiocache.plugins.BasePlugin method), 18  
 post\_clear() (aiocache.plugins.TimingPlugin method), 19  
 post\_delete() (aiocache.plugins.BasePlugin method), 18  
 post\_delete() (aiocache.plugins.TimingPlugin method), 19  
 post\_exists() (aiocache.plugins.BasePlugin method), 18  
 post\_exists() (aiocache.plugins.TimingPlugin method), 19  
 post\_expire() (aiocache.plugins.BasePlugin method), 18  
 post\_expire() (aiocache.plugins.TimingPlugin method), 19  
 post\_get() (aiocache.plugins.BasePlugin method), 18  
 post\_get() (aiocache.plugins.HitMissRatioPlugin method), 19  
 post\_get() (aiocache.plugins.TimingPlugin method), 19  
 post\_increment() (aiocache.plugins.BasePlugin method), 18

post\_increment() (aiocache.plugins.TimingPlugin method), 19  
post\_multi\_get() (aiocache.plugins.BasePlugin method), 18  
post\_multi\_get() (aiocache.plugins.HitMissRatioPlugin method), 19  
post\_multi\_get() (aiocache.plugins.TimingPlugin method), 19  
post\_multi\_set() (aiocache.plugins.BasePlugin method), 18  
post\_multi\_set() (aiocache.plugins.TimingPlugin method), 19  
post\_raw() (aiocache.plugins.BasePlugin method), 18  
post\_raw() (aiocache.plugins.TimingPlugin method), 19  
post\_set() (aiocache.plugins.BasePlugin method), 18  
post\_set() (aiocache.plugins.TimingPlugin method), 19  
pre\_add() (aiocache.plugins.BasePlugin method), 18  
pre\_clear() (aiocache.plugins.BasePlugin method), 18  
pre\_delete() (aiocache.plugins.BasePlugin method), 18  
pre\_exists() (aiocache.plugins.BasePlugin method), 18  
pre\_expire() (aiocache.plugins.BasePlugin method), 18  
pre\_get() (aiocache.plugins.BasePlugin method), 18  
pre\_increment() (aiocache.plugins.BasePlugin method), 18  
pre\_multi\_get() (aiocache.plugins.BasePlugin method), 18  
pre\_multi\_set() (aiocache.plugins.BasePlugin method), 19  
pre\_raw() (aiocache.plugins.BasePlugin method), 19  
pre\_set() (aiocache.plugins.BasePlugin method), 19

## R

raw() (aiocache.base.BaseCache method), 11  
RedisCache (class in aiocache), 11

## S

save\_time() (aiocache.plugins.TimingPlugin class method), 19  
set() (aiocache.base.BaseCache method), 11  
set\_config() (aiocache.caches class method), 19  
SimpleMemoryCache (class in aiocache), 12  
StringSerializer (class in aiocache.serializers), 16

## T

TimingPlugin (class in aiocache.plugins), 19