Cachy Documentation

Release 0.1

Sébastien Eustace

Apr 19, 2018

Contents

Cachy provides a simple yet effective caching library.

- A simple but powerful API
- Thread-safety
- Decorator syntax
- Support for **memcached**, **redis**, **database**, **file**, **dict** stores

Cachy supports python versions **2.7+** and **3.2+**

CHAPTER 1

Installation

You can install Cachy in 2 different ways:

- The easier and more straightforward is to use pip

```
pip install cachy
```

- Install from source using the official repository (https://github.com/sdispater/cachy)

Note: The available stores each requires specific packages that must be installed.

See the *Cache Prerequisites* section to get the list of needed packages.

CHAPTER 2

Configuration

Cachy provides a unified API for various caching systems.

All you need to get you started is the configuration describing the various cache stores and passing it to a `CacheManager` instance.

```
from cachy import CacheManager

config = {
    'stores': {
        'redis': {
            'driver': 'redis',
            'host': 'localhost',
            'port': 6379,
            'db': 0
        }
    }
}

cache = CacheManager(config)
```

If you have multiple stores configured you can specify which one is the default:

```
from cachy import CacheManager

config = {
    'stores': {
        'redis': {
            'driver': 'redis',
            'host': 'localhost',
            'port': 6379,
            'db': 0
        },
        'memcached': {
            'driver' 'memcached',
            'servers': [
```

```
                '127.0.0.1:11211'
            ]
        }
    }
}
```

An example cache configuration is located at examples/config.py.

The cache configuration file contains various options, which are documented within the file, so make sure to read over these options.

2.1 Cache Prerequisites

2.1.1 Database

When using the `database` cache driver, you will need the Orator ORM. You will also need to setup a table to contain the cache items. You will find an example SchemaBuilder declaration for the table below:

```
with schema.create('cache') as table:
    table.string('key').unique()
    table.text('value')
    table.integer('expiration')
```

2.1.2 Memcached

When using the `memcached` driver, you will need either the pure-python python-memcached (python3-memcached) or the **libmemcached** wrapper, pylibmc.

```
{
    'memcached': {
        'driver': 'memcached',
        'servers': [
            '127.0.0.1:11211'
        ]
    }
}
```

2.1.3 Redis

You will need the redis library in order to use the `redis` driver.

```
{
    'redis': {
        'driver': 'memcached',
        'host': 'localhost',
        'port': 6379,
        'db': 0
    }
}
```

2.1.4 File

You do not need any extra package to use the `file` driver.

```
{
    'file': {
        'driver': 'file',
        'path': '/my/cache/directory'
    }
}
```

2.1.5 Dict

You do not need any extra package to use the `dict` driver.

```
{
    'dict': {
        'driver': 'dict'
    }
}
```

2.2 Serialization

By default, Cachy will serialize objects using the `pickle` library. However, this can be changed in the configuration, either globally or at driver level.

The possible values are `pickle`, `json`, `msgpack`.

```
config = {
    'default': 'redis',
    'serializer': 'pickle',
    'stores': {
        'redis': {
            'driver': 'redis',
            'serializer': 'json',
            'host': 'localhost',
            'port': 6379,
            'db': 0
        },
        'memcached': {
            'driver' 'memcached',
            'servers': [
                '127.0.0.1:11211'
            ]
        }
    }
}
```

Warning: The serializer you choose will determine which types of objects you can serialize, the `pickle` serializer being the more permissive.

CHAPTER 3

Usage

As seen in the *Configuration* section, you first need to create a `CacheManager` instance.

3.1 Accessing Multiple Cache Stores

Using the `CacheManager` instance, you can access the configured cache stores via the `store` method. The key passed to the store method should correspond to one of the stores listed in the `stores` configuration dictionary:

```
value = cache.store('redis').get('foo')

cache.store('memcached').put('foo', 'bar', 10)
```

Note: If you do not specify a store the default store will be used.

```
value = cache.get('foo')
```

3.2 Retrieving Items From The Cache

The `get` method is used to retrieve items from the cache. If the item does not exist in the cache, `None` will be returned. If you wish, you can pass a second argument to the `get` method specifying the custom default value you wish to be returned if the item doesn't exist:

```
value = cache.get('foo')

value = cache.get('foo', 'default')
```

You may even pass a function as the default value. The result of the function will be returned if the specified item does not exist in the cache. Passing a function allows you to defer the retrieval of default values from a database or other external service:

```
value = cache.get('foo', lambda: db.table('users').get())
```

3.2.1 Checking For Item Existence

The `has` method may be used to determine if an item exists in the cache:

```
if cache.has('foo'):
    # ...
```

3.2.2 Incrementing / Decrementing Values

The `increment` and `decrement` methods can be used to adjust the value of integer items in the cache. Both of these methods optionally accept a second argument indicating the amount by which to increment or decrement the item's value:

```
cache.increment('key')

cache.increment('key', 3)

cache.decrement('key')

cache.decrement('key', 3)
```

3.2.3 Retrieve or Update

Sometimes you may wish to retrieve an item from the cache, but also store a default value if the requested item doesn't exist. For example, you may wish to retrieve all users from the cache or, if they don't exist, retrieve them from the database and add them to the cache. You may do this using the `remember` method:

```
value = cache.remember('users', 10, lambda: db.table('users').get())
```

If the item does not exist in the cache, the function passed to the remember method will be executed and its result will be placed in the cache.

You may also combine the `remember` and `forever` methods:

```
value = cache.remember_forever('users', 10, lambda: db.table('users').get())
```

Note: Using the `remember` method might not be the most practical in some cases, that's why you can use the `CacheManager` instance like a decorator.

See *Using Decorators*.

3.2.4 Retrieve and Delete

If you need to retrieve an item from the cache and then delete it, you can use the `pull` method. Like the `get` method, `None` will be returned if the item does not exist in the cache:

```
value = cache.pull('key')
```

3.3 Storing Items In The Cache

You can use the `put` method to store items in the cache. When you place an item in the cache, you will need to specify the number of minutes for which the value should be cached:

```
cache.put('key', 'value', 10)
```

Instead of passing the number of minutes until the item expires, you can also pass a `datetime` instance representing the expiration time of the cached item:

```
expires_at = datetime.now() + timedelta(minutes=10)

cache.put('key', 'value', expires_at)
```

The `add` method will only add the item to the cache if it does not already exist in the cache store. The method will return `True` if the item is actually added to the cache. Otherwise, the method will return `False`:

```
cache.add('key', 'value', 10)
```

The `forever` method can be used to store an item in the cache permanently. These values must be manually removed from the cache using the `forget` method:

```
cache.forever('key', 'value')
```

3.4 Removing Items From The Cache

You can remove items from the cache using the `forget`:

```
cache.forget('key')
```

3.5 Using Decorators

Instead of using the `remember` method, which might not be suitable for functions with complex logic, you can use the `CacheManager` instance as a decorator:

```
@cache
def get_users():
    return db.table('users').get()
```

This will store the result of the function for the default time of 60 minutes. The key will automatically be generated based on the function name, its arguments and keyword arguments.

You can also specify a key and the number of minutes the result will be stored in the cache:

```
@cache(key='key', minutes=30)
def get_users():
    return db.table('users').get()
```

Warning: The `key` keyword will only serve as a prefix for the automatically generated key. The final cache key will still depend on the arguments and keyword arguments.

You can also specify a store when using the cache manager as a decorator:

```
@cache('redis', key='key', minutes=30)
def get_users():
    return db.table('users').get()
```

CHAPTER 4

Custom Cache Drivers

To extend the `CacheManager` with a custom driver, you can use the `extend` method, which is used to bind a custom driver resolver to the manager.

For example, to register a new cache driver named "mongo":

```
cache.extend('mongo', MongoStore)
```

On initialization, the `MongoStore` class will be passed the driver configuration.

Note: Instead of the class you could also pass a function returning either a `Store` instance or a `Repository` instance.

CHAPTER 5

Cache Tags

Note: Cache tags are not supported when using the `file` or `database` cache drivers. Furthermore, when using multiple tags with caches that are stored "forever", performance will be best with a driver such as `memcached`, which automatically purges stale records.

5.1 Storing Tagged Cache Items

Cache tags allow you to tag related items in the cache and then flush all cached values that assigned a given tag. You may access a tagged cache by passing in an ordered array of tag names. For example, let's access a tagged cache and `put` value in the cache:

```
cache.tags('people', 'artists').put('John', john, minutes)
cache.tags('people', 'authors').put('Anne', anne, minutes)
```

However, you are not limited to the `put` method. You can use any cache storage method while working with tags.

5.2 Accessing Tagged Cache Items

To retrieve a tagged cache item, pass the same ordered list of tags to the `tags` method:

```
john = cache.tags('people', 'artists').get('John')
anne = cache.tags('people', 'authors').get('Anne')
```

You can flush all items that are assigned a tag or list of tags. For example, this statement would remove all caches tagged with either `people`, `authors`, or `both`. So, both ``Anne` and `John` would be removed from the cache:

```
cache.tags('people', 'authors').flush()
```

In contrast, this statement would remove only caches tagged with `authors`, so `Anne` would be removed, but not `John`.

```
cache.tags('authors').flush()
```