limits Documentation

Release 1.5.1+79.g2ed40c5.dirty

Ali-Akber Saifee

Dec 15, 2021

CONTENTS

limits provides utilities to implement rate limiting using various strategies and storage backends such as redis & memcached.

CHAPTER

ONE

RATE LIMIT STRING NOTATION

Rate limits are specified as strings following the format:

    [count] [per|/] [n (optional)] [second|minute|hour|day|month|year]

You can combine multiple rate limits by separating them with a delimiter of your choice.

1.1 Examples

- 10 per hour
- 10/hour
- 10/hour;100/day;2000 per year
- 100/day, 500/7days

CHAPTER

TWO

CUSTOM STORAGE BACKENDS

The **limits** package ships with a few storage implementations which allow you to get started with some common data stores (redis & memcached) used for rate limiting.

To accommodate customizations to either the default storage backends or different storage backends altogether, **limits** uses a registry pattern that makes it painless to add your own custom storage (without having to submit patches to the package itself).

Creating a custom backend requires:

1. Subclassing *limits.storage.Storage*
2. Providing implementations for the abstractmethods of *limits.storage.Storage*
3. If the storage can support the *Moving Window* strategy - additionally implementing the *acquire_entry* instance method.
4. Providing naming *schemes* that can be used to lookup the custom storage in the storage registry. (Refer to *Storage scheme* for more details)

2.1 Example

The following example shows two backend stores: one which doesn't implement the *Moving Window* strategy and one that does. Do note the `STORAGE_SCHEME` class variables which result in the classes getting registered with the **limits** storage registry:

```
import urlparse
from limits.storage import Storage
import time

class AwesomeStorage(Storage):
    STORAGE_SCHEME = ["awesomedb"]
    def __init__(self, uri, **options):
        self.awesomesness = options.get("awesomeness", None)
        self.host = urlparse.urlparse(uri).netloc
        self.port = urlparse.urlparse(uri).port

    def check(self):
        return True

    def get_expiry(self, key):
        return int(time.time())
```

(continues on next page)

(continued from previous page)

```
    def incr(self, key, expiry, elastic_expiry=False):
        return

    def get(self, key):
        return 0


class AwesomerStorage(Storage):
    STORAGE_SCHEME = ["awesomerdb"]
    def __init__(self, uri, **options):
        self.awesomesness = options.get("awesomeness", None)
        self.host = urlparse.urlparse(uri).netloc
        self.port = urlparse.urlparse(uri).port

    def check(self):
        return True

    def get_expiry(self, key):
        return int(time.time())

    def incr(self, key, expiry, elastic_expiry=False):
        return

    def get(self, key):
        return 0

    def acquire_entry(
            self, key, limit, expiry, no_add=False
    ):
        return True
```

Once the above implementations are declared you can look them up using the factory method described in *Storage scheme* in the following manner:

```
from limits.storage import storage_from_string

awesome = storage_from_string("awesomedb://localhoax:42", awesomeness=0)
awesomer = storage_from_string("awesomerdb://localhoax:42", awesomeness=1)
```

CHAPTER

THREE

STORAGE BACKENDS

3.1 Storage scheme

limits uses a url style storage scheme notation (similar to the JDBC driver connection string notation) for configuring and initializing storage backends. This notation additionally provides a simple mechanism to both identify and configure the backend implementation based on a single string argument.

The storage scheme follows the format `{scheme}://{parameters}`

limits.storage.storage_from_string() is provided to lookup and construct an instance of a storage based on the storage scheme. For example:

```
import limits.storage
uri = "redis://localhost:9999"
options = {}
redis_storage = limits.storage.storage_from_string(uri, **options)
```

The additional *options* key-word arguments are passed as is to the constructor of the storage and handled differently by each implementation. Please refer to the class documentation of *Backend Implementations* for details.

3.2 Examples

3.2.1 In-Memory

The in-memory storage (*limits.storage.MemoryStorage*) takes no parameters so the only relevant value is `memory://`

3.2.2 Memcached

Requires the location of the memcached server(s). As such the parameters is a comma separated list of `{host}:{port}` locations such as `memcached://localhost:11211` or `memcached://localhost:11211,localhost:11212,192.168.1.1:11211` etc... or a path to a unix domain socket such as `memcached:///var/tmp/path/to/sock`

Depends on: pymemcache

3.2.3 Memcached on Google App Engine

Requires that you are working in the GAE SDK and have those API libraries available. :code: *gaememcached://*

3.2.4 Redis

Requires the location of the redis server and optionally the database number. `redis://localhost:6379` or `redis://localhost:6379/n` (for database *n*).

If the redis server is listening over a unix domain socket you can use `redis+unix:///path/to/sock` or `redis+unix:///path/to/socket?db=n` (for database *n*).

If the database is password protected the password can be provided in the url, for example `redis://:foobared@localhost:6379` or `redis+unix//:foobered/path/to/socket` if using a UDS..

Depends on: redis-py

3.2.5 Redis over SSL

The official Redis client `redis-py` supports redis connections over SSL with the scheme You can add ssl related parameters in the url itself, for example: `rediss://localhost:6379/0?ssl_ca_certs=./tls/ca.crt&ssl_keyfile=./tls/client.key`.

Depends on: redis-py

3.2.6 Redis with Sentinel

Requires the location(s) of the redis sentinal instances and the *service-name* that is monitored by the sentinels. `redis+sentinel://localhost:26379/my-redis-service` or `redis+sentinel://localhost:26379,localhost:26380/my-redis-service`.

If the database is password protected the password can be provided in the url, for example `redis+sentinel://:sekret@localhost:26379/my-redis-service`

Depends on: redis-py

3.2.7 Redis Cluster

Requires the location(s) of the redis cluster startup nodes (One is enough). `redis+cluster://localhost:7000` or `redis+cluster://localhost:7000,localhost:70001`

Depends on: redis-py-cluster

CHAPTER

FOUR

RATE LIMITING STRATEGIES

4.1 Fixed Window

This is the most memory efficient strategy to use as it maintains one counter per resource and rate limit. It does however have its drawbacks as it allows bursts within each window - thus allowing an 'attacker' to by-pass the limits. The effects of these bursts can be partially circumvented by enforcing multiple granularities of windows per resource.

For example, if you specify a `100/minute` rate limit on a route, this strategy will allow 100 hits in the last second of one window and a 100 more in the first second of the next window. To ensure that such bursts are managed, you could add a second rate limit of `2/second` on the same route.

4.2 Fixed Window with Elastic Expiry

This strategy works almost identically to the Fixed Window strategy with the exception that each hit results in the extension of the window. This strategy works well for creating large penalties for breaching a rate limit.

For example, if you specify a `100/minute` rate limit on a route and it is being attacked at the rate of 5 hits per second for 2 minutes - the attacker will be locked out of the resource for an extra 60 seconds after the last hit. This strategy helps circumvent bursts.

4.3 Moving Window

> **Warning:** The moving window strategy is only implemented for the `redis` and `in-memory` storage backends. The strategy requires using a list with fast random access which is not very convenient to implement with a memcached storage.

This strategy is the most effective for preventing bursts from by-passing the rate limit as the window for each limit is not fixed at the start and end of each time unit (i.e. N/second for a moving window means N in the last 1000 milliseconds). There is however a higher memory cost associated with this strategy as it requires `N` items to be maintained in memory per resource and rate limit.

CHAPTER

FIVE

API

5.1 Storage

5.1.1 Abstract storage class

class limits.storage.**Storage**(*uri=None*, ***options*)

Bases: object

Base class to extend when implementing a storage backend.

abstract check()

check if storage is healthy

abstract clear(*key*)

resets the rate limit key :param str key: the key to clear rate limits for

abstract get(*key*)

Parameters **key** (*str*) – the key to get the counter value for

abstract get_expiry(*key*)

Parameters **key** (*str*) – the key to get the expiry for

abstract incr(*key*, *expiry*, *elastic_expiry=False*)

increments the counter for a given rate limit key

Parameters

- **key** (*str*) – the key to increment
- **expiry** (*int*) – amount in seconds for the key to expire in
- **elastic_expiry** (*bool*) – whether to keep extending the rate limit window every hit.

abstract reset()

reset storage to clear limits

5.1.2 Backend Implementations

In-Memory

class limits.storage.**MemoryStorage**(*uri=None*, **_)
Bases: *limits.storage.base.Storage*

rate limit storage using collections.Counter as an in memory storage for fixed and elastic window strategies, and a simple list to implement moving window strategy.

acquire_entry(*key*, *limit*, *expiry*, *no_add=False*)

Parameters

- **key** (*str*) – rate limit key to acquire an entry in
- **limit** (*int*) – amount of entries allowed
- **expiry** (*int*) – expiry of the entry
- **no_add** (*bool*) – if False an entry is not actually acquired but instead serves as a 'check'

Return type bool

check()
check if storage is healthy

clear(*key*)

Parameters **key** (*str*) – the key to clear rate limits for

get(*key*)

Parameters **key** (*str*) – the key to get the counter value for

get_expiry(*key*)

Parameters **key** (*str*) – the key to get the expiry for

get_moving_window(*key*, *limit*, *expiry*)
returns the starting point and the number of entries in the moving window

Parameters

- **key** (*str*) – rate limit key
- **expiry** (*int*) – expiry of entry

Returns (start of window, number of acquired entries)

get_num_acquired(*key*, *expiry*)
returns the number of entries already acquired

Parameters

- **key** (*str*) – rate limit key to acquire an entry in
- **expiry** (*int*) – expiry of the entry

incr(*key*, *expiry*, *elastic_expiry=False*)
increments the counter for a given rate limit key

Parameters

- **key** (*str*) – the key to increment
- **expiry** (*int*) – amount in seconds for the key to expire in
- **elastic_expiry** (*bool*) – whether to keep extending the rate limit window every hit.

reset()
reset storage to clear limits

Redis

class limits.storage.**RedisStorage**(*uri*, ***options*)
Bases: limits.storage.redis.RedisInteractor, *limits.storage.base.Storage*

Rate limit storage with redis as backend.

Depends on the *redis-py* library.

Parameters

- **uri** (*str*) – uri of the form *redis://[:password]@host:port*, *redis://[:password]@host:port/db*, *rediss://[:password]@host:port*, *redis+unix:///path/to/sock* etc. This uri is passed directly to `redis.from_url()` except for the case of *redis+unix* where it is replaced with *unix*.
- **options** – all remaining keyword arguments are passed directly to the constructor of `redis.Redis`

Raises *ConfigurationError* – when the redis library is not available

acquire_entry(*key*, *limit*, *expiry*, *no_add=False*)

Parameters

- **key** (*str*) – rate limit key to acquire an entry in
- **limit** (*int*) – amount of entries allowed
- **expiry** (*int*) – expiry of the entry
- **no_add** (*bool*) – if False an entry is not actually acquired but instead serves as a 'check'

Returns True/False

check()
check if storage is healthy

clear(*key*)

Parameters **key** (*str*) – the key to clear rate limits for

get(*key*)

Parameters **key** (*str*) – the key to get the counter value for

get_expiry(*key*)

Parameters **key** (*str*) – the key to get the expiry for

incr(*key*, *expiry*, *elastic_expiry=False*)
increments the counter for a given rate limit key

Parameters

- **key** (*str*) – the key to increment
- **expiry** (*int*) – amount in seconds for the key to expire in

`reset()`

This function calls a Lua Script to delete keys prefixed with 'LIMITER' in block of 5000.

> **Warning:** This operation was designed to be fast, but was not tested on a large production based system. Be careful with its usage as it could be slow on very large data sets.

Redis Cluster

`class limits.storage.RedisClusterStorage(uri, **options)`

Bases: *limits.storage.redis.RedisStorage*

Rate limit storage with redis cluster as backend

Depends on *redis-py-cluster* library

Parameters

- **uri** (*str*) – url of the form *redis+cluster://[:password]@host:port,host:port*
- **options** – all remaining keyword arguments are passed directly to the constructor of `rediscluster.RedisCluster`

Raises *ConfigurationError* – when the rediscluster library is not available or if the redis host cannot be pinged.

`reset()`

Redis Clusters are sharded and deleting across shards can't be done atomically. Because of this, this reset loops over all keys that are prefixed with 'LIMITER' and calls delete on them, one at a time.

> **Warning:** This operation was not tested with extremely large data sets. On a large production based system, care should be taken with its usage as it could be slow on very large data sets

Redis Sentinel

`class limits.storage.RedisSentinelStorage(uri, service_name=None, **options)`

Bases: *limits.storage.redis.RedisStorage*

Rate limit storage with redis sentinel as backend

Depends on *redis-py* library

Parameters

- **uri** (*str*) – url of the form *redis+sentinel://host:port,host:port/service_name*
- **optional** (*str service_name,*) – sentinel service name (if not provided in *uri*)
- **options** – all remaining keyword arguments are passed directly to the constructor of `redis.sentinel.Sentinel`

Raises *ConfigurationError* – when the redis library is not available or if the redis master host cannot be pinged.

check()
check if storage is healthy

get(*key*)

Parameters **key** (*str*) – the key to get the counter value for

get_expiry(*key*)

Parameters **key** (*str*) – the key to get the expiry for

Memcached

class limits.storage.**MemcachedStorage**(*uri*, ***options*)
Bases: *limits.storage.base.Storage*

Rate limit storage with memcached as backend.

Depends on the *pymemcache* library.

Parameters

- **uri** (*str*) – memcached location of the form *memcached://host:port,host:port*, *memcached:///var/tmp/path/to/sock*
- **options** – all remaining keyword arguments are passed directly to the constructor of `pymemcache.client.base.Client`

Raises *ConfigurationError* – when *pymemcache* is not available

check()
check if storage is healthy

clear(*key*)

Parameters **key** (*str*) – the key to clear rate limits for

get(*key*)

Parameters **key** (*str*) – the key to get the counter value for

get_client(*module*, *hosts*, ***kwargs*)
returns a memcached client. :param module: the memcached module :param hosts: list of memcached hosts :return:

get_expiry(*key*)

Parameters **key** (*str*) – the key to get the expiry for

incr(*key*, *expiry*, *elastic_expiry=False*)
increments the counter for a given rate limit key

Parameters

- **key** (*str*) – the key to increment
- **expiry** (*int*) – amount in seconds for the key to expire in
- **elastic_expiry** (*bool*) – whether to keep extending the rate limit window every hit.

property storage
lazily creates a memcached client instance using a thread local

Google App Engine Memcached

class limits.storage.GAEMemcachedStorage(*uri*, ***options*)
Bases: *limits.storage.memcached.MemcachedStorage*

rate limit storage with GAE memcache as backend

Parameters

- **uri** (*str*) – memcached location of the form *memcached://host:port,host:port*, *memcached:///var/tmp/path/to/sock*
- **options** – all remaining keyword arguments are passed directly to the constructor of `pymemcache.client.base.Client`

Raises *ConfigurationError* – when *pymemcache* is not available

check()
check if storage is healthy

incr(*key*, *expiry*, *elastic_expiry=False*)
increments the counter for a given rate limit key

Parameters

- **key** (*str*) – the key to increment
- **expiry** (*int*) – amount in seconds for the key to expire in
- **elastic_expiry** (*bool*) – whether to keep extending the rate limit window every hit.

5.1.3 Utility Methods

limits.storage.storage_from_string(*storage_string*, ***options*)
factory function to get an instance of the storage class based on the uri of the storage

Parameters **storage_string** – a string of the form method://host:port

Returns an instance of `flask_limiter.storage.Storage`

5.2 Strategies

class limits.strategies.RateLimiter(*storage*)
Bases: `object`

abstract get_window_stats(*item*, **identifiers*)
returns the number of requests remaining and reset of this limit.

Parameters

- **item** – a `RateLimitItem` instance
- **identifiers** – variable list of strings to uniquely identify the limit

Returns tuple (reset time (int), remaining (int))

abstract hit(*item*, **identifiers*)
creates a hit on the rate limit and returns True if successful.

Parameters

- **item** – a RateLimitItem instance
- **identifiers** – variable list of strings to uniquely identify the limit

Returns True/False

abstract test(*item*, **identifiers*)
checks the rate limit and returns True if it is not currently exceeded.

Parameters

- **item** – a RateLimitItem instance
- **identifiers** – variable list of strings to uniquely identify the limit

Returns True/False

class limits.strategies.**FixedWindowRateLimiter**(*storage*)
Bases: *limits.strategies.RateLimiter*

Reference: *Fixed Window*

get_window_stats(*item*, **identifiers*)
returns the number of requests remaining and reset of this limit.

Parameters

- **item** – a RateLimitItem instance
- **identifiers** – variable list of strings to uniquely identify the limit

Returns tuple (reset time (int), remaining (int))

hit(*item*, **identifiers*)
creates a hit on the rate limit and returns True if successful.

Parameters

- **item** – a RateLimitItem instance
- **identifiers** – variable list of strings to uniquely identify the limit

Returns True/False

test(*item*, **identifiers*)
checks the rate limit and returns True if it is not currently exceeded.

Parameters

- **item** – a RateLimitItem instance
- **identifiers** – variable list of strings to uniquely identify the limit

Returns True/False

class limits.strategies.**FixedWindowElasticExpiryRateLimiter**(*storage*)
Bases: *limits.strategies.FixedWindowRateLimiter*

Reference: *Fixed Window with Elastic Expiry*

hit(*item*, **identifiers*)
creates a hit on the rate limit and returns True if successful.

Parameters

- **item** – a RateLimitItem instance
- **identifiers** – variable list of strings to uniquely identify the limit

Returns True/False

class limits.strategies.**MovingWindowRateLimiter**(*storage*)
Bases: *limits.strategies.RateLimiter*

Reference: *Moving Window*

get_window_stats(*item*, **identifiers*)
returns the number of requests remaining within this limit.

Parameters

- **item** – a RateLimitItem instance
- **identifiers** – variable list of strings to uniquely identify the limit

Returns tuple (reset time (int), remaining (int))

hit(*item*, **identifiers*)
creates a hit on the rate limit and returns True if successful.

Parameters

- **item** – a RateLimitItem instance
- **identifiers** – variable list of strings to uniquely identify the limit

Returns True/False

test(*item*, **identifiers*)
checks the rate limit and returns True if it is not currently exceeded.

Parameters

- **item** – a RateLimitItem instance
- **identifiers** – variable list of strings to uniquely identify the limit

Returns True/False

5.3 Rate Limits

5.3.1 Rate limit granularities

class limits.**RateLimitItem**(*amount*, *multiples=1*, *namespace='LIMITER'*)
Bases: object

defines a Rate limited resource which contains the characteristic namespace, amount and granularity multiples of the rate limiting window.

Parameters

- **amount** (*int*) – the rate limit amount
- **multiples** (*int*) – multiple of the 'per' granularity (e.g. 'n' per 'm' seconds)
- **namespace** (*string*) – category for the specific rate limit

classmethod check_granularity_string(*granularity_string*)
checks if this instance matches a granularity string of type 'n per hour' etc.

Returns True/False

get_expiry()

Returns the size of the window in seconds.

key_for(**identifiers*)

Parameters **identifiers** – a list of strings to append to the key

Returns a string key identifying this resource with each identifier appended with a '/' delimiter.

class limits.**RateLimitItemPerYear**(*amount*, *multiples=1*, *namespace='LIMITER'*)
Bases: *limits.limits.RateLimitItem*

per year rate limited resource.

class limits.**RateLimitItemPerMonth**(*amount*, *multiples=1*, *namespace='LIMITER'*)
Bases: *limits.limits.RateLimitItem*

per month rate limited resource.

class limits.**RateLimitItemPerDay**(*amount*, *multiples=1*, *namespace='LIMITER'*)
Bases: *limits.limits.RateLimitItem*

per day rate limited resource.

class limits.**RateLimitItemPerHour**(*amount*, *multiples=1*, *namespace='LIMITER'*)
Bases: *limits.limits.RateLimitItem*

per hour rate limited resource.

class limits.**RateLimitItemPerMinute**(*amount*, *multiples=1*, *namespace='LIMITER'*)
Bases: *limits.limits.RateLimitItem*

per minute rate limited resource.

class limits.**RateLimitItemPerSecond**(*amount*, *multiples=1*, *namespace='LIMITER'*)
Bases: *limits.limits.RateLimitItem*

per second rate limited resource.

5.3.2 Utility Methods

limits.**parse**(*limit_string*)
parses a single rate limit in string notation (e.g. '1/second' or '1 per second'

Parameters **limit_string** (*string*) – rate limit string using *Rate limit string notation*

Raises **ValueError** – if the string notation is invalid.

Returns an instance of *RateLimitItem*

limits.**parse_many**(*limit_string*)
parses rate limits in string notation containing multiple rate limits (e.g. '1/second; 5/minute')

Parameters **limit_string** (*string*) – rate limit string using *Rate limit string notation*

Raises **ValueError** – if the string notation is invalid.

Returns a list of *RateLimitItem* instances.

5.4 Exceptions

exception limits.errors.**ConfigurationError**
Bases: Exception

exception raised when a configuration problem is encountered

CHAPTER

SIX

CHANGELOG

CHAPTER

SEVEN

V1.5.1

Release Date: 2020-02-25

- Bug fix
  - Remove duplicate call to ttl in RedisStorage
  - Initialize master/slave connections for RedisSentinel once

CHAPTER

EIGHT

V1.5

Release Date: 2020-01-23

- Bug fix for handling TTL response from Redis when key doesn't exist
- Support Memcache over unix domain socket
- Support Memcache cluster
- Pass through constructor keyword arguments to underlying storage constructor(s)
- CI & test improvements

CHAPTER

NINE

V1.4.1

Release Date: 2019-12-15

- Bug fix for implementation of clear in MemoryStorage not working with MovingWindow

CHAPTER

TEN

V1.4

Release Date: 2019-12-14

- Expose API for clearing individual limits
- Support for redis over unix domain socket
- Support extra arguments to redis storage

CHAPTER

ELEVEN

V1.3

Release Date: 2018-01-28

- Remove pinging redis on initialization

CHAPTER

TWELVE

V1.2.1

Release Date: 2017-01-02

- Fix regression with csv as multiple limits

CHAPTER

THIRTEEN

V1.2.0

Release Date: 2016-09-21

- Support reset for RedisStorage
- Improved rate limit string parsing

CHAPTER

FOURTEEN

V1.1.1

Release Date: 2016-03-14

- Support reset for MemoryStorage
- Support for *rediss://* storage scheme to connect to redis over ssl

CHAPTER

FIFTEEN

V1.1

Release Date: 2015-12-20

- Redis Cluster support
- Authentiation for Redis Sentinel
- Bug fix for locking failures with redis.

CHAPTER

SIXTEEN

V1.0.9

Release Date: 2015-10-08

- Redis Sentinel storage support
- Drop support for python 2.6
- Documentation improvements

CHAPTER

SEVENTEEN

V1.0.7

Release Date: 2015-06-07

- No functional change

CHAPTER

EIGHTEEN

V1.0.6

Release Date: 2015-05-13

- Bug fixes for .test() logic

CHAPTER

NINETEEN

V1.0.5

Release Date: 2015-05-12

- Add support for testing a rate limit before hitting it.

CHAPTER

TWENTY

V1.0.3

Release Date: 2015-03-20

- Add support for passing options to storage backend

CHAPTER

TWENTYONE

V1.0.2

Release Date: 2015-01-10

- Improved documentation
- Improved usability of API. Renamed RateLimitItem subclasses.

CHAPTER

TWENTYTWO

V1.0.1

Release Date: 2015-01-08

- Example usage in docs.

CHAPTER

TWENTYTHREE

V1.0.0

Release Date: 2015-01-08

- Initial import of common rate limiting code from Flask-Limiter

CHAPTER

TWENTYFOUR

QUICKSTART

Initialize the storage backend:

```
from limits import storage
memory_storage = storage.MemoryStorage()
```

Initialize a rate limiter with the *Moving Window* strategy:

```
from limits import strategies
moving_window = strategies.MovingWindowRateLimiter(memory_storage)
```

Initialize a rate limit using the *Rate limit string notation*:

```
from limits import parse
one_per_minute = parse("1/minute")
```

Initialize a rate limit explicitly using a subclass of *RateLimitItem*:

```
from limits import RateLimitItemPerSecond
one_per_second = RateLimitItemPerSecond(1, 1)
```

Test the limits:

```
assert True == moving_window.hit(one_per_minute, "test_namespace", "foo")
assert False == moving_window.hit(one_per_minute, "test_namespace", "foo")
assert True == moving_window.hit(one_per_minute, "test_namespace", "bar")

assert True == moving_window.hit(one_per_second, "test_namespace", "foo")
assert False == moving_window.hit(one_per_second, "test_namespace", "foo")
time.sleep(1)
assert True == moving_window.hit(one_per_second, "test_namespace", "foo")
```

Check specific limits without hitting them:

```
assert True == moving_window.hit(one_per_second, "test_namespace", "foo")
while not moving_window.test(one_per_second, "test_namespace", "foo"):
    time.sleep(0.01)
assert True == moving_window.hit(one_per_second, "test_namespace", "foo")
```

Clear a limit:

```
assert True == moving_window.hit(one_per_minute, "test_namespace", "foo")
assert False == moving_window.hit(one_per_minute, "test_namespace", "foo")
```

(continues on next page)

(continued from previous page)

```
moving_window.clear(one_per_minute", "test_namespace", "foo")
assert True == moving_window.hit(one_per_minute, "test_namespace", "foo")
```

CHAPTER

TWENTYFIVE

DEVELOPMENT

Since *limits* integrates with various backend storages, local development and running tests can require some setup.

You can use the provided Makefile to set up all the backends. This will require a working docker installation. Additionally on OSX you will require the `memcached` and `redis-server` executables to be on the path:

```
make setup-test-backends
# hack hack hack
# run tests
pytest
```

CHAPTER

TWENTYSIX

PROJECTS USING *LIMITS*

- Flask-Limiter : Rate limiting extension for Flask applications.
- djlimiter: Rate limiting middleware for Django applications.
- sanic-limiter: Rate limiting middleware for Sanic applications.
- Falcon-Limiter : Rate limiting extension for Falcon applications.

CHAPTER

TWENTYSEVEN

REFERENCES

- Redis rate limiting pattern #2
- DomainTools redis rate limiter

27.1 Contributions

- Timothee Groleau
- Zehua Liu
- David Czarnecki

INDEX