limits

Release 2.6.3

unknown

Jun 05, 2022

CONTENTS

limits is a python library to perform rate limiting with commonly used storage backends (Redis, Memcached & MongoDB).

Get started by taking a look at *Installation* and *Quickstart*.

To learn more about the different strategies refer to the *Rate limiting strategies* section.

For an overview of supported backends refer to *Storage Backends*.

CHAPTER

ONE

INSTALLATION

Install the package with pip:

```
$ pip install limits
```

Redis

```
$ pip install limits[redis]
```

Includes

```
redis>3,<5.0.0
```

RedisCluster

```
$ pip install limits[rediscluster]
```

Includes

```
redis>=4.2.0
```

Memcached

```
$ pip install limits[memcached]
```

Includes

```
pymemcache>3,<4.0.0
```

MongoDB

```
$ pip install limits[mongodb]
```

Includes:

```
pymongo>3,<5
```

More details around the specifics of each storage backend can be found in *Storage Backends*

1.1 Async Storage

If you are using an async code base you can install the storage dependencies along with the package using the following extras:

Redis

```
$ pip install limits[async-redis]
```

Includes:

```
coredis[hiredis]>=3.4.0,<4;python_version>"3.7"
```

Memcached

```
$ pip install limits[async-memcached]
```

Includes:

```
emcache>=0.6.1;python_version<"3.11" # not yet supported
```

MongoDB

```
$ pip install limits[async-mongodb]
```

Includes:

```
motor>=2.5,<4
```

CHAPTER

TWO

QUICKSTART

2.1 Initialize the strategy & storage

2.1.1 Initialize the storage backend

In Memory

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

Memcached

```
from limits import storage
memory_storage = storage.MemcachedStorage(
    "memcached://localhost:11211"
)
```

Redis

```
from limits import storage
memory_storage = storage.RedisStorage("redis://localhost:6379/1")
```

2.1.2 Initialize a rate limiter with the Moving Window Strategy

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

2.2 Describe the rate limit

2.2.1 Initialize a rate limit using the string notation

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

2.2.2 Initialize a rate limit explicitly using a subclass of `RateLimitItem`

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

2.3 Test the limits

2.3.1 Consume 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")
```

2.3.2 Check without consuming

```
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")
```

2.4 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")
moving_window.clear(one_per_minute, "test_namespace", "foo")
assert True == moving_window.hit(one_per_minute, "test_namespace", "foo")
```

CHAPTER

THREE

RATE LIMIT STRING NOTATION

Instead of manually constructing instances of *RateLimitItem* you can instead use the following *Parsing functions*.

- *parse()*
- *parse_many()*

These functions accept rate limits specified as strings following the format:

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

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

3.1 Examples

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

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 not implemented for the `memcached` storage backend.

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

STORAGE BACKENDS

5.1 Supported versions

limits is tested and known to work with the following versions of the dependency libraries and the associated storage versions.

The CI tests against these versions on a nightly basis and you can see the results in github.

Redis

Dependency versions:

```
redis>3,<5.0.0
```

Dependency versions (async):

```
coredis[hiredis]>=3.4.0,<4;python_version>"3.7"
```

Redis

```
7.0
6.2.6
6.0.16
5.0.14
```

Redis with SSL

```
6.2.6
6.0.16
```

Redis Sentinel

```
6.2.6
6.0.16
5.0.14
```

Redis Cluster

Dependency versions:

```
redis>=4.2.0
```

Dependency versions (async):

```
coredis[hiredis]>=3.4.0,<4;python_version>"3.7"
```

Redis cluster

```
7.0
6.2.6
6.0.16
5.0.14
```

Memcached

Dependency versions:

```
pymemcache>3,<4.0.0
```

Dependency versions (async):

```
emcache>=0.6.1;python_version<"3.11" # not yet supported
```

Memcached

```
1.6.15
1.6.6
1.5.16
1.4.34
```

MongoDB

Dependency versions:

```
pymongo>3,<5
```

Dependency versions (async):

```
motor>=2.5,<4
```

MongoDB

```
5.0.3
4.4.9
4.2.17
```

5.2 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 API documentation in the *Storage* section for details.

5.3 Examples

5.3.1 In-Memory

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

5.3.2 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

5.3.3 Memcached on Google App Engine

Deprecated since version 2.0.

Requires that you are working in the GAE SDK and have those API libraries available.

`gaememcached://`

5.3.4 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..

For scenarios where a redis connection pool is already available and can be reused, it can be provided in *options*, for example:

```
pool = redis.connections.BlockingConnectionPool.from_url("redis://.....")
storage_from_string("redis://", connection_pool=pool)
```

Depends on: redis

5.3.5 Redis over SSL

The official Redis client redis 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

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

When authentication details are provided in the url they will be used for both the sentinel and as connection arguments for the underlying redis nodes managed by the sentinel.

If you need fine grained control it is recommended to use the additional *options* arguments. More details can be found in the API documentation for *RedisSentinelStorage* (or the aysnc version: *RedisSentinelStorage*).

Depends on: redis

5.3.7 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:7001`

Depends on: redis

5.3.8 MongoDB

Requires the location(s) of a mongodb installation using the uri schema described by the Mongodb URI Specification

Examples:

- Local instance: `mongodb://localhost:27017/`
- Instance with SSL: `mongodb://mymongo.com/?tls=true`
- Local instance with SSL & self signed/invalid certificate: `mongodb://localhost:27017/?tls=true&tlsAllowInvalidCertificates=true`

Depends on: pymongo

5.4 Async Storage

New in version 2.1.

When using limits in an async code base the same uri schema can be used to query for an async implementation of the storage by prefixing the scheme with `async+`.

For example:

- `async+redis://localhost:6379/0`
- `async+rediss://localhost:6379/0`
- `async+redis+cluster://localhost:7000,localhost:7001`
- `async+redis+sentinel://:sekret@localhost:26379/my-redis-service`
- `async+memcached://localhost:11211`
- `async+memory://`

For implementation details of currently supported async backends refer to *Async Storage*

CHAPTER

SIX

ASYNC SUPPORT

New in version 2.1.

A new namespace `limits.aio` is available which mirrors the original `limits.storage` and `limits.strategies` packages.

The following async storage backends are implemented:

- In-Memory
- Redis (via coredis)
- Memcached (via emcache)
- MongoDB (via motor)

6.1 Quick start

This example demonstrates the subtle differences in the `limits.aio` namespace:

```
from limits import parse
from limits.storage import storage_from_string
from limits.aio.strategies import MovingWindowRateLimiter

redis = storage_from_string("async+redis://localhost:6379")

moving_window = MovingWindowRateLimiter(redis)
one_per_minute = parse("1/minute")

async def hit():
   return await moving_window.hit(one_per_minute, "test_namespace", "foo")
```

Refer to *Async Storage* for more implementation details of the async storage backends, and *Strategies* (Async) for the async rate limit strategies API.

CHAPTER

SEVEN

API REFERENCE

`limits`	Rate limiting with commonly used storage backends
`limits.strategies`	Rate limiting strategies
`limits.storage`	Implementations of storage backends to be used with *`limits.strategies.RateLimiter`* strategies
`limits.aio.strategies`	Asynchronous rate limiting strategies
`limits.aio.storage`	Implementations of storage backends to be used with *`limits.aio.strategies.RateLimiter`* strategies

7.1 Strategies

Default

The available built in rate limiting strategies which expect a single parameter: a subclass of *`Storage`*.

Provided by `limits.strategies`

class FixedWindowRateLimiter(*storage: Union[*Storage, limits.aio.storage.Storage*]*)

Reference: *Fixed Window*

hit(*item:* RateLimitItem, **identifiers: str*, *cost: int = 1*) → bool

Consume the rate limit

Parameters

- **item** – The rate limit item
- **identifiers** – variable list of strings to uniquely identify this instance of the limit
- **cost** – The cost of this hit, default 1

test(*item:* RateLimitItem, **identifiers: str*) → bool

Check if the rate limit can be consumed

Parameters

- **item** – The rate limit item
- **identifiers** – variable list of strings to uniquely identify this instance of the limit

get_window_stats(*item:* RateLimitItem, **identifiers: str*) → Tuple[int, int]

Query the reset time and remaining amount for the limit

Parameters

- **item** – The rate limit item
- **identifiers** – variable list of strings to uniquely identify this instance of the limit

Returns (reset time, remaining)

class FixedWindowElasticExpiryRateLimiter(*storage: Union[*Storage*,* limits.aio.storage.Storage*]*)

Reference: *Fixed Window with Elastic Expiry*

hit(*item:* RateLimitItem, **identifiers: str*, *cost: int = 1*) → bool

Consume the rate limit

Parameters

- **item** – The rate limit item
- **identifiers** – variable list of strings to uniquely identify this instance of the limit
- **cost** – The cost of this hit, default 1

get_window_stats(*item:* RateLimitItem, **identifiers: str*) → Tuple[int, int]

Query the reset time and remaining amount for the limit

Parameters

- **item** – The rate limit item
- **identifiers** – variable list of strings to uniquely identify this instance of the limit

Returns (reset time, remaining)

test(*item:* RateLimitItem, **identifiers: str*) → bool

Check if the rate limit can be consumed

Parameters

- **item** – The rate limit item
- **identifiers** – variable list of strings to uniquely identify this instance of the limit

class MovingWindowRateLimiter(*storage: Union[*Storage*,* limits.aio.storage.Storage*]*)

Reference: *Moving Window*

hit(*item:* RateLimitItem, **identifiers: str*, *cost: int = 1*) → bool

Consume the rate limit

Parameters

- **item** – The rate limit item
- **identifiers** – variable list of strings to uniquely identify this instance of the limit
- **cost** – The cost of this hit, default 1

Returns (reset time, remaining)

test(*item:* RateLimitItem, **identifiers: str*) → bool

Check if the rate limit can be consumed

Parameters

- **item** – The rate limit item
- **identifiers** – variable list of strings to uniquely identify this instance of the limit

get_window_stats(*item:* RateLimitItem, **identifiers: str*) → Tuple[int, int]

returns the number of requests remaining within this limit.

Parameters

- **item** – The rate limit item
- **identifiers** – variable list of strings to uniquely identify this instance of the limit

Returns tuple (reset time, remaining)

All strategies implement the same abstract base class:

class RateLimiter(*storage: Union[*Storage*,* limits.aio.storage.Storage*]*)

abstract hit(*item:* RateLimitItem, **identifiers: str*, *cost: int = 1*) → bool

Consume the rate limit

Parameters

- **item** – The rate limit item
- **identifiers** – variable list of strings to uniquely identify this instance of the limit
- **cost** – The cost of this hit, default 1

abstract test(*item:* RateLimitItem, **identifiers: str*) → bool

Check the rate limit without consuming from it.

Parameters

- **item** – The rate limit item
- **identifiers** – variable list of strings to uniquely identify this instance of the limit

abstract get_window_stats(*item:* RateLimitItem, **identifiers: str*) → Tuple[int, int]

Query the reset time and remaining amount for the limit

Parameters

- **item** – The rate limit item
- **identifiers** – variable list of strings to uniquely identify this instance of the limit

Returns (reset time, remaining)

Async

These variants should be used in for asyncio support. These strategies expose async variants and expect a subclass of *limits.aio.storage.Storage*

Provided by `limits.aio.strategies`

class FixedWindowRateLimiter(*storage: Union[*Storage*,* limits.aio.storage.Storage*]*)

Reference: *Fixed Window*

async hit(*item:* RateLimitItem, **identifiers: str*, *cost: int = 1*) → bool

Consume the rate limit

Parameters

- **item** – the rate limit item
- **identifiers** – variable list of strings to uniquely identify the limit

- **cost** – The cost of this hit, default 1

async test(*item:* RateLimitItem, **identifiers: str*) → bool

Check if the rate limit can be consumed

Parameters

- **item** – the rate limit item
- **identifiers** – variable list of strings to uniquely identify the limit

async get_window_stats(*item:* RateLimitItem, **identifiers: str*) → Tuple[int, int]

Query the reset time and remaining amount for the limit

Parameters

- **item** – the rate limit item
- **identifiers** – variable list of strings to uniquely identify the limit

Returns reset time, remaining

class FixedWindowElasticExpiryRateLimiter(*storage: Union[*Storage, limits.aio.storage.Storage*]*)

Reference: *Fixed Window with Elastic Expiry*

async hit(*item:* RateLimitItem, **identifiers: str*, *cost: int = 1*) → bool

Consume the rate limit

Parameters

- **item** – a *limits.limits.RateLimitItem* instance
- **identifiers** – variable list of strings to uniquely identify the limit
- **cost** – The cost of this hit, default 1

async get_window_stats(*item:* RateLimitItem, **identifiers: str*) → Tuple[int, int]

Query the reset time and remaining amount for the limit

Parameters

- **item** – the rate limit item
- **identifiers** – variable list of strings to uniquely identify the limit

Returns reset time, remaining

async test(*item:* RateLimitItem, **identifiers: str*) → bool

Check if the rate limit can be consumed

Parameters

- **item** – the rate limit item
- **identifiers** – variable list of strings to uniquely identify the limit

class MovingWindowRateLimiter(*storage: Union[*Storage, limits.aio.storage.Storage*]*)

Reference: *Moving Window*

async hit(*item:* RateLimitItem, **identifiers: str*, *cost: int = 1*) → bool

Consume the rate limit

Parameters

- **item** – the rate limit item
- **identifiers** – variable list of strings to uniquely identify the limit

- **cost** – The cost of this hit, default 1

async test(*item:* RateLimitItem, **identifiers: str*) → bool

Check if the rate limit can be consumed

Parameters

- **item** – the rate limit item
- **identifiers** – variable list of strings to uniquely identify the limit

async get_window_stats(*item:* RateLimitItem, **identifiers: str*) → Tuple[int, int]

returns the number of requests remaining within this limit.

Parameters

- **item** – the rate limit item
- **identifiers** – variable list of strings to uniquely identify the limit

Returns (reset time, remaining)

All strategies implement the same abstract base class:

class RateLimiter(*storage: Union[*Storage, limits.aio.storage.Storage*]*)

abstract async hit(*item:* RateLimitItem, **identifiers: str*, *cost: int = 1*) → bool

Consume the rate limit

Parameters

- **item** – the rate limit item
- **identifiers** – variable list of strings to uniquely identify the limit
- **cost** – The cost of this hit, default 1

abstract async test(*item:* RateLimitItem, **identifiers: str*) → bool

Check if the rate limit can be consumed

Parameters

- **item** – the rate limit item
- **identifiers** – variable list of strings to uniquely identify the limit

abstract async get_window_stats(*item:* RateLimitItem, **identifiers: str*) → Tuple[int, int]

Query the reset time and remaining amount for the limit

Parameters

- **item** – the rate limit item
- **identifiers** – variable list of strings to uniquely identify the limit

Returns (reset time, remaining))

7.2 Storage

7.2.1 Storage Factory function

Provided by `limits.storage`

storage_from_string(*storage_string: str*, ***options: Union[float, str, bool]*) → Union[*Storage*, *Storage*]

Factory function to get an instance of the storage class based on the uri of the storage. In most cases using it should be sufficient instead of directly instantiating the storage classes. for example:

```
from limits.storage import storage_from_string

memory = from_string("memory://")
memcached = from_string("memcached://localhost:11211")
redis = from_string("redis://localhost:6379")
```

The same function can be used to construct the *Async Storage* variants, for example:

```
from limits.storage import storage_from_string

memory = storage_from_string("async+memory://")
memcached = storage_from_string("async+memcached://localhost:11211")
redis = storage_from_string("asycn+redis://localhost:6379")
```

Parameters

- **storage_string** – a string of the form `scheme://host:port`. More details about supported storage schemes can be found at *Storage scheme*
- **options** – all remaining keyword arguments are passed to the constructor matched by *storage_string*.

Raises ***ConfigurationError*** – when the `storage_string` cannot be mapped to a registered *limits.storage.Storage* or *limits.aio.storage.Storage* instance.

7.2.2 Synchronous Storage

Provided by `limits.storage`

In-Memory

class MemoryStorage(*uri: Optional[str] = None*, ***_: str*)

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.

incr(*key: str*, *expiry: int*, *elastic_expiry: bool = False*, *amount: int = 1*) → int

increments the counter for a given rate limit key

Parameters

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

- **amount** – the number to increment by

get(*key: str*) → int

Parameters **key** – the key to get the counter value for

clear(*key: str*) → None

Parameters **key** – the key to clear rate limits for

acquire_entry(*key: str*, *limit: int*, *expiry: int*, *amount: int = 1*) → bool

Parameters

- **key** – rate limit key to acquire an entry in
- **limit** – amount of entries allowed
- **expiry** – expiry of the entry
- **amount** – the number of entries to acquire

get_expiry(*key: str*) → int

Parameters **key** – the key to get the expiry for

get_num_acquired(*key: str*, *expiry: int*) → int

returns the number of entries already acquired

Parameters

- **key** – rate limit key to acquire an entry in
- **expiry** – expiry of the entry

get_moving_window(*key: str*, *limit: int*, *expiry: int*) → Tuple[int, int]

returns the starting point and the number of entries in the moving window

Parameters

- **key** – rate limit key
- **expiry** – expiry of entry

Returns (start of window, number of acquired entries)

check() → bool

check if storage is healthy

reset() → Optional[int]

reset storage to clear limits

Redis

class RedisStorage(*uri: str*, *connection_pool: Optional[redis.connection.ConnectionPool] = None*, ***options: Union[float, str, bool]*)

Rate limit storage with redis as backend.

Depends on redis.

Parameters

- **uri** – 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://`.
- **connection_pool** – if provided, the redis client is initialized with the connection pool and any other params passed as *options*
- **options** – all remaining keyword arguments are passed directly to the constructor of `redis.Redis`

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

STORAGE_SCHEME: Optional[List[str]] = ['redis', 'rediss', 'redis+unix']

The storage scheme for redis

incr(*key: str*, *expiry: int*, *elastic_expiry: bool = False*, *amount: int = 1*) → int

increments the counter for a given rate limit key

Parameters

- **key** – the key to increment
- **expiry** – amount in seconds for the key to expire in
- **amount** – the number to increment by

get(*key: str*) → int

Parameters **key** – the key to get the counter value for

clear(*key: str*) → None

Parameters **key** – the key to clear rate limits for

acquire_entry(*key: str*, *limit: int*, *expiry: int*, *amount: int = 1*) → bool

Parameters

- **key** – rate limit key to acquire an entry in
- **limit** – amount of entries allowed
- **expiry** – expiry of the entry
- **amount** – the number to increment by

get_expiry(*key: str*) → int

Parameters **key** – the key to get the expiry for

check() → bool

check if storage is healthy

get_moving_window(*key: str*, *limit: int*, *expiry: int*) → Tuple[int, int]

returns the starting point and the number of entries in the moving window

Parameters

- **key** – rate limit key
- **expiry** – expiry of entry

Returns (start of window, number of acquired entries)

reset() → Optional[int]

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 RedisClusterStorage(*uri: str*, ***options: Union[float, str, bool]*)

Rate limit storage with redis cluster as backend

Depends on redis.

Changed in version 2.5.0: Cluster support was provided by the redis-py-cluster library which has been absorbed into the official redis client. By default the `redis.cluster.RedisCluster` client will be used however if the version of the package is lower than `4.2.0` the implementation will fallback to trying to use `rediscluster.RedisCluster`.

Parameters

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

Raises ***ConfigurationError*** – when the redis library is not available or if the redis cluster cannot be reached.

STORAGE_SCHEME: Optional[List[str]] = ['redis+cluster']

The storage scheme for redis cluster

DEFAULT_OPTIONS: Dict[str, Union[float, str, bool]] = {'max_connections': 1000}

Default options passed to the `RedisCluster`

acquire_entry(*key: str*, *limit: int*, *expiry: int*, *amount: int = 1*) → bool

Parameters

- **key** – rate limit key to acquire an entry in
- **limit** – amount of entries allowed
- **expiry** – expiry of the entry
- **amount** – the number to increment by

check() → bool

check if storage is healthy

clear(*key: str*) → None

Parameters **key** – the key to clear rate limits for

get(*key: str*) → int

Parameters **key** – the key to get the counter value for

get_expiry(*key: str*) → int

Parameters **key** – the key to get the expiry for

get_moving_window(*key: str*, *limit: int*, *expiry: int*) → Tuple[int, int]

returns the starting point and the number of entries in the moving window

Parameters

- **key** – rate limit key
- **expiry** – expiry of entry

Returns (start of window, number of acquired entries)

incr(*key: str*, *expiry: int*, *elastic_expiry: bool = False*, *amount: int = 1*) → int

increments the counter for a given rate limit key

Parameters

- **key** – the key to increment
- **expiry** – amount in seconds for the key to expire in
- **amount** – the number to increment by

reset() → Optional[int]

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 RedisSentinelStorage(*uri: str*, *service_name: Optional[str] = None*, *use_replicas: bool = True*, *sentinel_kwargs: Optional[Dict[str, Union[float, str, bool]]] = None*, ***options: Union[float, str, bool]*)

Rate limit storage with redis sentinel as backend

Depends on redis package

Parameters

- **uri** – url of the form `redis+sentinel://host:port,host:port/service_name`
- **service_name** – sentinel service name (if not provided in `uri`)
- **use_replicas** – Whether to use replicas for read only operations
- **sentinel_kwargs** – kwargs to pass as `sentinel_kwargs` to `redis.sentinel.Sentinel`
- **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.

STORAGE_SCHEME: Optional[List[str]] = ['redis+sentinel']

The storage scheme for redis accessed via a redis sentinel installation

DEFAULT_OPTIONS: Dict[str, Union[float, str, bool]] = {'socket_timeout': 0.2}

Default options passed to `Sentinel`

get(*key: str*) → int

Parameters **key** – the key to get the counter value for

acquire_entry(*key: str*, *limit: int*, *expiry: int*, *amount: int = 1*) → bool

Parameters

- **key** – rate limit key to acquire an entry in
- **limit** – amount of entries allowed
- **expiry** – expiry of the entry
- **amount** – the number to increment by

clear(*key: str*) → None

Parameters **key** – the key to clear rate limits for

get_expiry(*key: str*) → int

Parameters **key** – the key to get the expiry for

get_moving_window(*key: str*, *limit: int*, *expiry: int*) → Tuple[int, int]

returns the starting point and the number of entries in the moving window

Parameters

- **key** – rate limit key
- **expiry** – expiry of entry

Returns (start of window, number of acquired entries)

incr(*key: str*, *expiry: int*, *elastic_expiry: bool = False*, *amount: int = 1*) → int

increments the counter for a given rate limit key

Parameters

- **key** – the key to increment
- **expiry** – amount in seconds for the key to expire in
- **amount** – the number to increment by

reset() → Optional[int]

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.

check() → bool

Check if storage is healthy by calling `aredis.StrictRedis.ping` on the slave.

Memcached

class MemcachedStorage(*uri: str*, ***options: Union[str, Callable[[], MemcachedClientP]]*)

Rate limit storage with memcached as backend.

Depends on pymemcache.

Parameters

- **uri** – 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.PooledClient` or `pymemcache.client.hash.HashClient` (if there are more than one hosts specified)

Raises *ConfigurationError* – when pymemcache is not available

STORAGE_SCHEME: Optional[List[str]] = ['memcached']

The storage scheme for memcached

get_client(*module: module*, *hosts: List[Tuple[str, int]]*, ***kwargs: str*) → MemcachedClientP

returns a memcached client.

Parameters

- **module** – the memcached module
- **hosts** – list of memcached hosts

property storage: limits.typing.MemcachedClientP

lazily creates a memcached client instance using a thread local

get(*key: str*) → int

Parameters **key** – the key to get the counter value for

clear(*key: str*) → None

Parameters **key** – the key to clear rate limits for

incr(*key: str*, *expiry: int*, *elastic_expiry: bool = False*, *amount: int = 1*) → int

increments the counter for a given rate limit key

Parameters

- **key** – the key to increment
- **expiry** – amount in seconds for the key to expire in
- **elastic_expiry** – whether to keep extending the rate limit window every hit.
- **amount** – the number to increment by

get_expiry(*key: str*) → int

Parameters **key** – the key to get the expiry for

check() → bool

Check if storage is healthy by calling the `get` command on the key `limiter-check`

reset() → Optional[int]

reset storage to clear limits

MongoDB

class MongoDBStorage(*uri: str*, *database_name: str = 'limits'*, ***options: Union[int, str, bool]*)

Rate limit storage with MongoDB as backend.

Depends on pymongo.

New in version 2.1.

Parameters

- **uri** – uri of the form `mongodb://[user:password]@host:port?...`, This uri is passed directly to `MongoClient`
- **database_name** – The database to use for storing the rate limit collections.
- **options** – all remaining keyword arguments are merged with *DEFAULT_OPTIONS* and passed to the constructor of `MongoClient`

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

DEFAULT_OPTIONS: Dict[str, Union[int, str, bool]] = {'connectTimeoutMS': 1000, 'serverSelectionTimeoutMS': 1000, 'socketTimeoutMS': 1000}

Default options passed to `MongoClient`

reset() → Optional[int]

Delete all rate limit keys in the rate limit collections (counters, windows)

clear(*key: str*) → None

Parameters **key** – the key to clear rate limits for

get_expiry(*key: str*) → int

Parameters **key** – the key to get the expiry for

get(*key: str*) → int

Parameters **key** – the key to get the counter value for

incr(*key: str*, *expiry: int*, *elastic_expiry: bool = False*, *amount: int = 1*) → int

increments the counter for a given rate limit key

Parameters

- **key** – the key to increment
- **expiry** – amount in seconds for the key to expire in
- **amount** – the number to increment by

check() → bool

Check if storage is healthy by calling `pymongo.mongo_client.MongoClient.server_info()`

get_moving_window(*key: str*, *limit: int*, *expiry: int*) → Tuple[int, int]

returns the starting point and the number of entries in the moving window

Parameters

- **key** – rate limit key
- **expiry** – expiry of entry

Returns (start of window, number of acquired entries)

acquire_entry(*key: str*, *limit: int*, *expiry: int*, *amount: int = 1*) → bool

Parameters

- **key** – rate limit key to acquire an entry in
- **limit** – amount of entries allowed
- **expiry** – expiry of the entry
- **amount** – the number of entries to acquire

7.2.3 Async Storage

Provided by `limits.aio.storage`

In-Memory

class MemoryStorage(*uri: Optional[str] = None*, ***_: str*)

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.

New in version 2.1.

STORAGE_SCHEME: Optional[List[str]] = ['async+memory']

The storage scheme for in process memory storage for use in an async context

async incr(*key: str*, *expiry: int*, *elastic_expiry: bool = False*, *amount: int = 1*) → int

increments the counter for a given rate limit key

Parameters

- **key** – the key to increment
- **expiry** – amount in seconds for the key to expire in
- **elastic_expiry** – whether to keep extending the rate limit window every hit.
- **amount** – the number to increment by

async get(*key: str*) → int

Parameters **key** – the key to get the counter value for

async clear(*key: str*) → None

Parameters **key** – the key to clear rate limits for

async acquire_entry(*key: str*, *limit: int*, *expiry: int*, *amount: int = 1*) → bool

Parameters

- **key** – rate limit key to acquire an entry in
- **limit** – amount of entries allowed
- **expiry** – expiry of the entry
- **amount** – the number of entries to acquire

async get_expiry(*key: str*) → int

Parameters **key** – the key to get the expiry for

async get_num_acquired(*key: str*, *expiry: int*) → int

returns the number of entries already acquired

Parameters

- **key** – rate limit key to acquire an entry in
- **expiry** – expiry of the entry

async get_moving_window(*key: str*, *limit: int*, *expiry: int*) → Tuple[int, int]

returns the starting point and the number of entries in the moving window

Parameters

- **key** – rate limit key
- **expiry** – expiry of entry

Returns (start of window, number of acquired entries)

async check() → bool

check if storage is healthy

async reset() → Optional[int]

reset storage to clear limits

Redis

class RedisStorage(*uri: str*, *connection_pool: Optional[coredis.ConnectionPool] = None*, ***options: Union[float, str, bool]*)

Rate limit storage with redis as backend.

Depends on coredis

New in version 2.1.

Parameters

- **uri** – uri of the form:
  - `async+redis://[:password]@host:port`
  - `async+redis://[:password]@host:port/db`
  - `async+rediss://[:password]@host:port`
  - `async+unix:///path/to/sock` etc…

  This uri is passed directly to `coredis.Redis.from_url()` with the initial `async` removed, except for the case of `async+redis+unix` where it is replaced with `unix`.
- **connection_pool** – if provided, the redis client is initialized with the connection pool and any other params passed as *options*
- **options** – all remaining keyword arguments are passed directly to the constructor of `coredis.Redis`

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

STORAGE_SCHEME: Optional[List[str]] = ['async+redis', 'async+rediss', 'async+redis+unix']

The storage schemes for redis to be used in an async context

async incr(*key: str*, *expiry: int*, *elastic_expiry: bool = False*, *amount: int = 1*) → int

increments the counter for a given rate limit key

Parameters

- **key** – the key to increment
- **expiry** – amount in seconds for the key to expire in
- **amount** – the number to increment by

async get(*key: str*) → int

Parameters **key** – the key to get the counter value for

async clear(*key: str*) → None

Parameters **key** – the key to clear rate limits for

async acquire_entry(*key: str*, *limit: int*, *expiry: int*, *amount: int = 1*) → bool

Parameters

- **key** – rate limit key to acquire an entry in
- **limit** – amount of entries allowed
- **expiry** – expiry of the entry
- **amount** – the number of entries to acquire

async get_expiry(*key: str*) → int

Parameters **key** – the key to get the expiry for

async check() → bool

Check if storage is healthy by calling `coredis.Redis.ping()`

async reset() → Optional[int]

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.

async get_moving_window(*key: str*, *limit: int*, *expiry: int*) → Tuple[int, int]

returns the starting point and the number of entries in the moving window

Parameters

- **key** – rate limit key
- **expiry** – expiry of entry

Returns (start of window, number of acquired entries)

Redis Cluster

class RedisClusterStorage(*uri: str*, ***options: Union[float, str, bool]*)

Rate limit storage with redis cluster as backend

Depends on coredis

New in version 2.1.

Parameters

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

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

STORAGE_SCHEME: Optional[List[str]] = ['async+redis+cluster']

The storage schemes for redis cluster to be used in an async context

DEFAULT_OPTIONS: Dict[str, Union[float, str, bool]] = {'max_connections': 1000}

Default options passed to `coredis.RedisCluster`

async reset() → Optional[int]

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

async acquire_entry(*key: str*, *limit: int*, *expiry: int*, *amount: int = 1*) → bool

Parameters

- **key** – rate limit key to acquire an entry in
- **limit** – amount of entries allowed
- **expiry** – expiry of the entry
- **amount** – the number of entries to acquire

async check() → bool

Check if storage is healthy by calling `coredis.Redis.ping()`

async clear(*key: str*) → None

Parameters **key** – the key to clear rate limits for

async get(*key: str*) → int

Parameters **key** – the key to get the counter value for

async get_expiry(*key: str*) → int

Parameters **key** – the key to get the expiry for

async get_moving_window(*key: str*, *limit: int*, *expiry: int*) → Tuple[int, int]

returns the starting point and the number of entries in the moving window

Parameters

- **key** – rate limit key
- **expiry** – expiry of entry

Returns (start of window, number of acquired entries)

async incr(*key: str*, *expiry: int*, *elastic_expiry: bool = False*, *amount: int = 1*) → int

increments the counter for a given rate limit key

Parameters

- **key** – the key to increment
- **expiry** – amount in seconds for the key to expire in
- **amount** – the number to increment by

Redis Sentinel

class RedisSentinelStorage(*uri: str*, *service_name: Optional[str] = None*, *use_replicas: bool = True*, *sentinel_kwargs: Optional[Dict[str, Union[float, str, bool]]] = None*, ***options: Union[float, str, bool]*)

Rate limit storage with redis sentinel as backend

Depends on coredis

New in version 2.1.

Parameters

- **uri** – url of the form `async+redis+sentinel://host:port,host:port/service_name`
- **optional** (*sentinel_kwargs,*) – sentinel service name (if not provided in *uri*)
- **use_replicas** – Whether to use replicas for read only operations
- **optional** – kwargs to pass as `sentinel_kwargs` to `coredis.sentinel.Sentinel`
- **options** – all remaining keyword arguments are passed directly to the constructor of `coredis.sentinel.Sentinel`

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

STORAGE_SCHEME: Optional[List[str]] = ['async+redis+sentinel']

The storage scheme for redis accessed via a redis sentinel installation

DEFAULT_OPTIONS: Dict[str, Union[float, str, bool]] = {'stream_timeout': 0.2}

Default options passed to `Sentinel`

async acquire_entry(*key: str*, *limit: int*, *expiry: int*, *amount: int = 1*) → bool

Parameters

- **key** – rate limit key to acquire an entry in
- **limit** – amount of entries allowed

- **expiry** – expiry of the entry
- **amount** – the number of entries to acquire

async clear(*key: str*) → None

Parameters **key** – the key to clear rate limits for

async get_moving_window(*key: str*, *limit: int*, *expiry: int*) → Tuple[int, int]

returns the starting point and the number of entries in the moving window

Parameters

- **key** – rate limit key
- **expiry** – expiry of entry

Returns (start of window, number of acquired entries)

async incr(*key: str*, *expiry: int*, *elastic_expiry: bool = False*, *amount: int = 1*) → int

increments the counter for a given rate limit key

Parameters

- **key** – the key to increment
- **expiry** – amount in seconds for the key to expire in
- **amount** – the number to increment by

async reset() → Optional[int]

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.

async get(*key: str*) → int

Parameters **key** – the key to get the counter value for

async get_expiry(*key: str*) → int

Parameters **key** – the key to get the expiry for

async check() → bool

Check if storage is healthy by calling `coredis.StrictRedis.ping()` on the replica.

Memcached

class MemcachedStorage(*uri: str*, ***options: Union[float, str, bool]*)

Rate limit storage with memcached as backend.

Depends on emcache

New in version 2.1.

Parameters

- **uri** – memcached location of the form `async+memcached://host:port,host:port`
- **options** – all remaining keyword arguments are passed directly to the constructor of `emcache.Client`

Raises *ConfigurationError* – when emcache is not available

STORAGE_SCHEME: Optional[List[str]] = ['async+memcached']

The storage scheme for memcached to be used in an async context

async get(*key: str*) → int

Parameters **key** – the key to get the counter value for

async clear(*key: str*) → None

Parameters **key** – the key to clear rate limits for

async incr(*key: str*, *expiry: int*, *elastic_expiry: bool = False*, *amount: int = 1*) → int

increments the counter for a given rate limit key

Parameters

- **key** – the key to increment
- **expiry** – amount in seconds for the key to expire in
- **elastic_expiry** – whether to keep extending the rate limit window every hit.
- **amount** – the number to increment by

async get_expiry(*key: str*) → int

Parameters **key** – the key to get the expiry for

async check() → bool

Check if storage is healthy by calling the `get` command on the key `limiter-check`

async reset() → Optional[int]

reset storage to clear limits

MongoDB

class MongoDBStorage(*uri: str*, *database_name: str = 'limits'*, ***options: Union[float, str, bool]*)

Rate limit storage with MongoDB as backend.

Depends on motor

New in version 2.1.

Parameters

- **uri** – uri of the form `async+mongodb://[user:password]@host:port?...`, This uri is passed directly to `AsyncIOMotorClient`
- **database_name** – The database to use for storing the rate limit collections.
- **options** – all remaining keyword arguments are merged with *DEFAULT_OPTIONS* and passed to the constructor of `AsyncIOMotorClient`

Raises *ConfigurationError* – when the motor or pymongo are not available

STORAGE_SCHEME: Optional[List[str]] = ['async+mongodb', 'async+mongodb+srv']

The storage scheme for MongoDB for use in an async context

DEFAULT_OPTIONS: Dict[str, Union[float, str, bool]] = {'connectTimeoutMS': 1000, 'serverSelectionTimeoutMS': 1000, 'socketTimeoutMS': 1000}

Default options passed to `AsyncIOMotorClient`

async reset() → Optional[int]

Delete all rate limit keys in the rate limit collections (counters, windows)

async clear(*key: str*) → None

Parameters **key** – the key to clear rate limits for

async get_expiry(*key: str*) → int

Parameters **key** – the key to get the expiry for

async get(*key: str*) → int

Parameters **key** – the key to get the counter value for

async incr(*key: str*, *expiry: int*, *elastic_expiry: bool = False*, *amount: int = 1*) → int

increments the counter for a given rate limit key

Parameters

- **key** – the key to increment
- **expiry** – amount in seconds for the key to expire in
- **elastic_expiry** – whether to keep extending the rate limit window every hit.
- **amount** – the number to increment by

async check() → bool

Check if storage is healthy by calling `motor.motor_asyncio.AsyncIOMotorClient.server_info()`

async get_moving_window(*key: str*, *limit: int*, *expiry: int*) → Tuple[int, int]

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)

async acquire_entry(*key: str*, *limit: int*, *expiry: int*, *amount: int = 1*) → bool

Parameters

- **key** – rate limit key to acquire an entry in
- **limit** – amount of entries allowed
- **expiry** – expiry of the entry
- **amount** – the number of entries to acquire

7.2.4 Abstract storage classes

class Storage(*uri: Optional[str] = None*, ***options: Union[float, str, bool]*)

Base class to extend when implementing a storage backend.

STORAGE_SCHEME: Optional[List[str]]

The storage schemes to register against this implementation

abstract incr(*key: str*, *expiry: int*, *elastic_expiry: bool = False*, *amount: int = 1*) → int

increments the counter for a given rate limit key

Parameters

- **key** – the key to increment
- **expiry** – amount in seconds for the key to expire in
- **elastic_expiry** – whether to keep extending the rate limit window every hit.
- **amount** – the number to increment by

abstract get(*key: str*) → int

Parameters **key** – the key to get the counter value for

abstract get_expiry(*key: str*) → int

Parameters **key** – the key to get the expiry for

abstract check() → bool

check if storage is healthy

abstract reset() → Optional[int]

reset storage to clear limits

abstract clear(*key: str*) → None

resets the rate limit key

Parameters **key** – the key to clear rate limits for

class MovingWindowSupport

Abstract base for storages that intend to support the moving window strategy

acquire_entry(*key: str*, *limit: int*, *expiry: int*, *amount: int = 1*) → bool

Parameters

- **key** – rate limit key to acquire an entry in
- **limit** – amount of entries allowed
- **expiry** – expiry of the entry
- **amount** – the number of entries to acquire

get_moving_window(*key: str*, *limit: int*, *expiry: int*) → Tuple[int, int]

returns the starting point and the number of entries in the moving window

Parameters

- **key** – rate limit key
- **expiry** – expiry of entry

Returns (start of window, number of acquired entries)

Async variants

class Storage(*uri: Optional[str] = None*, ***options: Union[float, str, bool]*)

Base class to extend when implementing an async storage backend.

New in version 2.1.

STORAGE_SCHEME: Optional[List[str]]

The storage schemes to register against this implementation

abstract async incr(*key: str*, *expiry: int*, *elastic_expiry: bool = False*, *amount: int = 1*) → int

increments the counter for a given rate limit key

Parameters

- **key** – the key to increment
- **expiry** – amount in seconds for the key to expire in
- **elastic_expiry** – whether to keep extending the rate limit window every hit.
- **amount** – the number to increment by

abstract async get(*key: str*) → int

Parameters **key** – the key to get the counter value for

abstract async get_expiry(*key: str*) → int

Parameters **key** – the key to get the expiry for

abstract async check() → bool

check if storage is healthy

abstract async reset() → Optional[int]

reset storage to clear limits

abstract async clear(*key: str*) → None

resets the rate limit key

Parameters **key** – the key to clear rate limits for

class MovingWindowSupport

Abstract base for storages that intend to support the moving window strategy

async acquire_entry(*key: str*, *limit: int*, *expiry: int*, *amount: int = 1*) → bool

Parameters

- **key** – rate limit key to acquire an entry in
- **limit** – amount of entries allowed
- **expiry** – expiry of the entry
- **amount** – the number of entries to acquire

async get_moving_window(*key: str*, *limit: int*, *expiry: int*) → Tuple[int, int]

returns the starting point and the number of entries in the moving window

Parameters

- **key** – rate limit key
- **expiry** – expiry of entry

Returns (start of window, number of acquired entries)

7.3 Rate Limits

Provided by `limits`

7.3.1 Parsing functions

parse(*limit_string: str*) → *RateLimitItem*

parses a single rate limit in string notation (e.g. `1/second` or `1 per second`)

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

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

parse_many(*limit_string: str*) → List[*RateLimitItem*]

parses rate limits in string notation containing multiple rate limits (e.g. `1/second; 5/minute`)

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

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

7.3.2 Rate limit granularities

All rate limit items implement `RateLimitItem` by declaring a `GRANULARITY`

class RateLimitItem(*amount: int*, *multiples: Optional[int] = 1*, *namespace: str = 'LIMITER'*)

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

Parameters

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

GRANULARITY: ClassVar[limits.limits.Granularity]

A tuple describing the granularity of this limit as (number of seconds, name)

classmethod check_granularity_string(*granularity_string: str*) → bool

Checks if this instance matches a *granularity_string* of type `n per hour`, `n per minute` etc, by comparing with *GRANULARITY*

get_expiry() → int

Returns the duration the limit is enforced for in seconds.

key_for(**identifiers: str*) → str

Constructs a key for the current limit and any additional identifiers provided.

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 RateLimitItemPerSecond(*amount: int*, *multiples: Optional[int] = 1*, *namespace: str = 'LIMITER'*)

per second rate limited resource.

classmethod check_granularity_string(*granularity_string: str*) → bool

Checks if this instance matches a *granularity_string* of type `n per hour`, `n per minute` etc, by comparing with *GRANULARITY*

get_expiry() → int

Returns the duration the limit is enforced for in seconds.

key_for(**identifiers: str*) → str

Constructs a key for the current limit and any additional identifiers provided.

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.

GRANULARITY: ClassVar[limits.limits.Granularity] = Granularity(seconds=1, name='second')

A second

class RateLimitItemPerMinute(*amount: int*, *multiples: Optional[int] = 1*, *namespace: str = 'LIMITER'*)

per minute rate limited resource.

classmethod check_granularity_string(*granularity_string: str*) → bool

Checks if this instance matches a *granularity_string* of type `n per hour`, `n per minute` etc, by comparing with *GRANULARITY*

get_expiry() → int

Returns the duration the limit is enforced for in seconds.

key_for(**identifiers: str*) → str

Constructs a key for the current limit and any additional identifiers provided.

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.

GRANULARITY: ClassVar[limits.limits.Granularity] = Granularity(seconds=60, name='minute')

A minute

class RateLimitItemPerHour(*amount: int*, *multiples: Optional[int] = 1*, *namespace: str = 'LIMITER'*)

per hour rate limited resource.

GRANULARITY: ClassVar[limits.limits.Granularity] = Granularity(seconds=3600, name='hour')

An hour

classmethod check_granularity_string(*granularity_string: str*) → bool

Checks if this instance matches a *granularity_string* of type `n per hour`, `n per minute` etc, by comparing with *GRANULARITY*

get_expiry() → int

Returns the duration the limit is enforced for in seconds.

key_for(**identifiers: str*) → str

Constructs a key for the current limit and any additional identifiers provided.

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 RateLimitItemPerDay(*amount: int*, *multiples: Optional[int] = 1*, *namespace: str = 'LIMITER'*)

per day rate limited resource.

GRANULARITY: ClassVar[limits.limits.Granularity] = Granularity(seconds=86400, name='day')

A day

classmethod check_granularity_string(*granularity_string: str*) → bool

Checks if this instance matches a *granularity_string* of type `n per hour`, `n per minute` etc, by comparing with *GRANULARITY*

get_expiry() → int

Returns the duration the limit is enforced for in seconds.

key_for(**identifiers: str*) → str

Constructs a key for the current limit and any additional identifiers provided.

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 RateLimitItemPerMonth(*amount: int*, *multiples: Optional[int] = 1*, *namespace: str = 'LIMITER'*)

per month rate limited resource.

GRANULARITY: ClassVar[limits.limits.Granularity] = Granularity(seconds=2592000, name='month')

A month

classmethod check_granularity_string(*granularity_string: str*) → bool

Checks if this instance matches a *granularity_string* of type `n per hour`, `n per minute` etc, by comparing with *GRANULARITY*

get_expiry() → int

Returns the duration the limit is enforced for in seconds.

key_for(**identifiers: str*) → str

Constructs a key for the current limit and any additional identifiers provided.

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 RateLimitItemPerYear(*amount: int*, *multiples: Optional[int] = 1*, *namespace: str = 'LIMITER'*)

per year rate limited resource.

GRANULARITY: ClassVar[limits.limits.Granularity] = Granularity(seconds=31104000, name='year')

A year

classmethod check_granularity_string(*granularity_string: str*) → bool

Checks if this instance matches a *granularity_string* of type `n per hour`, `n per minute` etc, by comparing with *GRANULARITY*

`get_expiry`() → int

Returns the duration the limit is enforced for in seconds.

`key_for`(**identifiers: str*) → str

Constructs a key for the current limit and any additional identifiers provided.

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.

7.4 Exceptions

`exception ConfigurationError`

exception raised when a configuration problem is encountered

`with_traceback`()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

CHAPTER

EIGHT

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* or *limits.aio.storage.Storage*
2. Providing implementations for the abstractmethods of *Storage*
3. If the storage can support the *Moving Window* strategy - additionally implementing the methods from *MovingWindowSupport*
4. Providing naming *schemes* that can be used to lookup the custom storage in the storage registry. (Refer to *Storage scheme* for more details)

8.1 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 MovingWindowSupport
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) -> bool:
        return True

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

(continues on next page)

(continued from previous page)

```
    def incr(self, key: str, expiry: int, elastic_expiry=False) -> int:
        return

    def get(self, key):
        return 0


class AwesomerStorage(Storage, MovingWindowSupport):
    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):
        return True

    def get_moving_window(
        self, key, limit, expiry
    ):
        return [0, 10]
```

Once the above implementations are declared you can look them up using the *Storage Factory function* 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

NINE

CHANGELOG

9.1 v2.6.3

Release Date: 2022-06-05

- Chores
  - Update development dependencies
  - Add CI for python 3.11
  - Increase test coverage for redis sentinel

9.2 v2.6.2

Release Date: 2022-05-12

- Compatibility Updates
  - Update motor requirements to include 3.x version
  - Update async redis sentinel implementation to remove use of deprecated methods.
  - Fix compatibility issue with asyncio redis `reset` method in cluster mode when used with coredis versions >= 3.5.0

9.3 v2.6.1

Release Date: 2022-04-25

- Bug Fix
  - Fix typing regression with strategy constructors Issue 88

9.4 v2.6.0

Release Date: 2022-04-25

- Deprecation
  - Removed tests for rediscluster using the redis-py-cluster library
- Bug Fix
  - Fix incorrect `__slots__` declaration in *limits.RateLimitItem* and it's subclasses (Issue #121)

9.5 v2.5.4

Release Date: 2022-04-25

- Bug Fix
  - Fix typing regression with strategy constructors Issue 88

9.6 v2.5.3

Release Date: 2022-04-22

- Chore
  - Automate Github releases

9.7 v2.5.2

Release Date: 2022-04-17

- Chore
  - Increase strictness of type checking and annotations
  - Ensure installations from source distributions are PEP-561 compliant

9.8 v2.5.1

Release Date: 2022-04-15

- Chore
  - Ensure storage reset methods have consistent signature

9.9 v2.5.0

Release Date: 2022-04-13

- Feature
  - Add support for using redis cluster via the official redis client
  - Update coredis dependency to use 3.x
- Deprecations
  - Deprecate using redis-py-cluster
- Chores
  - Remove beta tags for async support
  - Update code base to remove legacy syntax
  - Tighten up CI test dependencies

9.10 v2.4.0

Release Date: 2022-03-10

- Feature
  - Allow passing an explicit connection pool to redis storage. Addresses Issue 77

9.11 v2.3.3

Release Date: 2022-02-03

- Feature
  - Add support for dns seed list when using mongodb

9.12 v2.3.2

Release Date: 2022-01-30

- Chores
  - Improve authentication tests for redis
  - Update documentation theme
  - Pin pip version for CI

9.13 v2.3.1

Release Date: 2022-01-21

- Bug fix
  - Fix backward incompatible change that separated sentinel and connection args for redis sentinel (introduced in 2.1.0). Addresses Issue 97

9.14 v2.3.0

Release Date: 2022-01-15

- Feature
  - Add support for custom cost per hit
- Bug fix
  - Fix installation issues with missing setuptools

9.15 v2.2.0

Release Date: 2022-01-05

- Feature
  - Enable async redis for python 3.10 via coredis
- Chore
  - Fix typing issue with strategy constructors

9.16 v2.1.1

Release Date: 2022-01-02

- Feature
  - Enable async memcache for python 3.10
- Bug fix
  - Ensure window expiry is reported in local time for mongodb
  - Fix inconsistent expiry for fixed window with memcached
- Chore
  - Improve strategy tests

9.17 v2.1.0

Release Date: 2021-12-22

- Feature
  - Add beta asyncio support
  - Add beta mongodb support
  - Add option to install with extras for different storages
- Bug fix
  - Fix custom option for cluster client in memcached
  - Fix separation of sentinel & connection args in *limits.storage.RedisSentinelStorage*
- Deprecation
  - Deprecate GAEMemcached support
  - Remove use of unused *no_add* argument in *limits.storage.MovingWindowSupport.acquire_entry()*
- Chore
  - Documentation theme upgrades
  - Code linting
  - Add compatibility CI workflow

9.18 v2.0.3

Release Date: 2021-11-28

- Chore
  - Ensure package is marked PEP-561 compliant

9.19 v2.0.1

Release Date: 2021-11-28

- Chore
  - Added type annotations

9.20 v2.0.0

Release Date: 2021-11-27

- Chore
  - Drop support for python < 3.7

9.21 v1.6

Release Date: 2021-11-27

- Chore
  - Final release for python < 3.7

9.22 v1.5.1

Release Date: 2020-02-25

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

9.23 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

9.24 v1.4.1

Release Date: 2019-12-15

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

9.25 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

9.26 v1.3

Release Date: 2018-01-28

- Remove pinging redis on initialization

9.27 v1.2.1

Release Date: 2017-01-02

- Fix regression with csv as multiple limits

9.28 v1.2.0

Release Date: 2016-09-21

- Support reset for RedisStorage
- Improved rate limit string parsing

9.29 v1.1.1

Release Date: 2016-03-14

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

9.30 v1.1

Release Date: 2015-12-20

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

9.31 v1.0.9

Release Date: 2015-10-08

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

9.32 v1.0.7

Release Date: 2015-06-07

- No functional change

9.33 v1.0.6

Release Date: 2015-05-13

- Bug fixes for .test() logic

9.34 v1.0.5

Release Date: 2015-05-12

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

9.35 v1.0.3

Release Date: 2015-03-20

- Add support for passing options to storage backend

9.36 v1.0.2

Release Date: 2015-01-10

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

9.37 v1.0.1

Release Date: 2015-01-08

- Example usage in docs.

9.38 v1.0.0

Release Date: 2015-01-08

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

CHAPTER

TEN

DEVELOPMENT

The source is available on Github

To get started

```
$ git clone git://github.com/alisaifee/limits.git
$ cd limits
$ pip install -r requirements/dev.txt
```

Since *limits* integrates with various backend storages, local development and running tests requires a a working docker & docker-compose installation.

Running the tests will start the relevant containers automatically - but will leave them running so as to not incur the overhead of starting up on each test run. To run the tests:

```
$ pytest
```

Once you're done - you will probably want to clean up the docker containers:

```
$ docker-compose down
```

CHAPTER

ELEVEN

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

TWELVE

REFERENCES

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

CHAPTER

THIRTEEN

CONTRIBUTORS

- Timothee Groleau
- Zehua Liu
- David Czarnecki
- Laurent Savaete

INDEX

H

I

K

M

P

R

S

T

W