rush Documentation

Release 2021.04.0

Ian Stapleton Cordasco

Mar 27, 2023

Contents

CHAPTER 1

Overview

rush is a library that provides a composable and extensible framework for implementing rate limiting algorithms and storage backends. By default, rush comes with two algorithms for rate limiting and two backends for storage. The backends should work with all of the limiters so there should be no need for compatibility checking.

It also ships with a complete set of typestubs in the code as rush requires Python 3.6 or newer.

1.1 Default Algorithms

By default, rush comes with two algorithms:

- Periodic rate-limiting based on the period of time specified.
- Generic Cell Rate Limiting which is based on the algorithm defined for Asynchronous Transfer Mode networks.

Both limiters are implemented in pure Python.

1.2 Default Storage Backends

By default, rush comes with two storage backends:

- Dictionary based - primarily used for integration testing within the library itself.
- Redis

More storage backends could be implemented as necessary.

CHAPTER 2

Installation

```
pip install rush
```

```
pipenv install rush
```

CHAPTER 3

Quickstart

Since rush aims to be composable, its preliminary API can be considered rough and experimental. These imports will not break, but porcelain *may* be added at a future date.

```
from rush import quota
from rush import throttle
from rush.limiters import periodic
from rush.stores import dictionary

t = throttle.Throttle(
   limiter=periodic.PeriodicLimiter(
      store=dictionary.DictionaryStore()
   ),
   rate=quota.Quota.per_hour(
      count=5000,
      burst=500,
   ),
)

limit_result = t.check('expensive-operation/user@example.com', 1)
print(limit_result.limited)  # => False
print(limit_result.remaining)  # => 5499
print(limit_result.reset_after)  # => 1:00:00
```

CHAPTER 4

Table of Contents

4.1 Using Rush's Throttle

The primary interface intended to be used by Rush's users is the *Throttle* class. It does the heavy lifting in ensuring that the limiter is used and works to abstract away the underlying moving pieces.

class rush.throttle.**Throttle**(*rate: rush.quota.Quota*, *limiter: rush.limiters.base.BaseLimiter*)
The class that acts as the primary interface for throttles.

This class requires the intantiated rate quota and limiter and handles passing the right arguments to the limiter.

limiter
The instance of the rate limiting algorithm that should be used by the throttle.

rate
The instantiated *Quota* that tells the throttle and limiter what the limits and periods are for rate limiting.

check(*key: str*, *quantity: int*) → rush.result.RateLimitResult
Check if the user should be rate limited.

Parameters

- **key** (*str*) – The key to use for rate limiting.
- **quantity** (*int*) – How many resources is being requested against the rate limit.

Returns The result of calculating whether the user should be rate-limited.

Return type *RateLimitResult*

clear(*key: str*) → rush.result.RateLimitResult
Clear any existing limits for the given key.

Parameters **key** (*str*) – The key to use for rate limiting that should be cleared.

Returns The result of resetting the rate-limit.

Return type *RateLimitResult*

peek(*key: str*) → rush.result.RateLimitResult
Peek at the user's current rate-limit usage.

Note: This is equivalent to calling `check()` with a quantity of 0.

Parameters **key** (*str*) – The key to use for rate limiting.

Returns The current rate-limit usage.

Return type *RateLimitResult*

class rush.quota.**Quota**(*period: datetime.timedelta*, *count: int*, *maximum_burst: int = 0*)
The definition of a user's quota of resources.

period
The time between equally spaced requests. This must be greater than 0 seconds.

count
The number of requests to a resource allowed in the period. This must be greater than 0.

maximum_burst
The number of requests that will be allowed to exceed the rate in a single burst. This must be greater than or equal to 0 and defaults to 0.

limit
Return the calculated limit including maximum burst.

classmethod per_day(*count: int*, *, *maximum_burst: int = 0*) → Q
Create a quota based on the number allowed per day.

Parameters **count** (*int*) – The number of requests allowed per day.

Returns A new quota.

Return type Quota

classmethod per_hour(*count: int*, *, *maximum_burst: int = 0*) → Q
Create a quota based on the number allowed per hour.

Parameters **count** (*int*) – The number of requests allowed per hour.

Returns A new quota.

Return type Quota

classmethod per_minute(*count: int*, *, *maximum_burst: int = 0*) → Q
Create a quota based on the number allowed per minute.

Parameters **count** (*int*) – The number of requests allowed per minute.

Returns A new quota.

Return type Quota

classmethod per_second(*count: int*, *, *maximum_burst: int = 0*) → Q
Create a quota based on the number allowed per second.

Parameters **count** (*int*) – The number of requests allowed per second.

Returns A new quota.

Return type Quota

class rush.result.**RateLimitResult**(*limit: int*, *limited: bool*, *remaining: int*, *reset_after: datetime.timedelta*, *retry_after: datetime.timedelta*)

A result of checking a ratelimit.

The attributes on this object are:

limit
The integer limit that was checked against, e.g., if the user's ratelimit is 10,000 this value will be 10,000 regardless of how much they have consumed.

limited
Whether or not the user should be ratelimited (a.k.a., throttled).

remaining
The integer representing how much of the user's ratelimit is left. This should be the number of requests made during the time period, `N`, subtracted from the limit, `L`, or `L - N`.

reset_after
This will be a `timedelta` representing how much time is left until the ratelimit resets. For example if the ratelimit will reset in 800ms then this might look like:

```
datetime.timedelta(0, 0, 800000)
# == datetime.timedelta(milliseconds=800)
```

retry_after
This will be a `timedelta` representing the length of time after which a retry can be made.

resets_at(*from_when: Optional[datetime.datetime] = None*) → datetime.datetime
Calculate the reset time from UTC now.

Returns The UTC timezone-aware datetime representing when the limit resets.

retry_at(*from_when: Optional[datetime.datetime] = None*) → datetime.datetime
Calculate the retry time from UTC now.

Returns The UTC timezone-aware datetime representing when the user can retry.

4.2 Rush's Rate Limiting Algorithms

By default, rush includes the following algorithms:

- *Generic Cell Rate Limiting*
- *Redis Lua Generic Cell rate Limiting*
- *Periodic*

It also has a base class so you can create your own.

class rush.limiters.gcra.**GenericCellRatelimiter**
This class implements a very specific type of "leaky bucket" designed for Asynchronous Transfor Mode networks called Generic Cell Rate Algorithm. The algorithm itself can be challenging to understand, so let's first cover the benefits:

- It doesn't require users to sit idle for a potentially long period of time while they wait for their period to be done.
- It leaks the used amount of resources based off a clock and requires no extra threads, processes, or some other process to leak things.
- It is fast, even implemented purely in Python.

This can be thought of as having a sliding window where users have some number of requests they can make. This means that even as time moves, your users can still make requests instead of waiting terribly long.

Example instantiation:

```
from rush.limiters import gcra
from rush.stores import dictionary

gcralimiter = gcra.GenericCellRatelimiter(
   store=dictionary.DictionaryStore()
)
```

class rush.limiters.redis_gcra.**GenericCellRatelimiter**

This class implements a very specific type of "leaky bucket" designed for Asynchronous Transfor Mode networks called Generic Cell Rate Algorithm. The algorithm itself can be challenging to understand, so let's first cover the benefits:

- It doesn't require users to sit idle for a potentially long period of time while they wait for their period to be done.
- It leaks the used amount of resources based off a clock and requires no extra threads, processes, or some other process to leak things.
- It is fast, even implemented purely in Python.

This can be thought of as having a sliding window where users have some number of requests they can make. This means that even as time moves, your users can still make requests instead of waiting terribly long.

This relies on Lua scripts that are loaded into Redis (and only compatible with Redis) and called from Python. The Lua scripts are borrowed from https://github.com/rwz/redis-gcra

Since this is implemented *only* for Redis this requires you to use *RedisStore*.

Example instantiation:

```
from rush.limiters import redis_gcra
from rush.stores import redis

gcralimiter = redis_gcra.GenericCellRatelimiter(
   store=redis.RedisStore("redis://localhost:6379")
)
```

class rush.limiters.periodic.**PeriodicLimiter**

This class uses a naive way of allowing a certain number of requests for the specified period of time. If your quota has a period of 60 seconds and a limit (count and maximum burst) of 60, then effectively a user can make 60 requests every 60 seconds - or 1 request per second. For example, let's say a user makes a request at 12:31:50 until 12:32:50, they would only have 59 requests remaining. If by 12:32:10 the user has made 60 requests, then they still have to wait until 12:32:50 before they can make more.

Example instantiation:

```
from rush.limiters import periodic
from rush.stores import dictionary

periodiclimiter = periodic.PeriodicLimiter(
   store=dictionary.DictionaryStore()
)
```

4.2.1 Writing Your Own Algorithm

Rush specifies a very small set of methods that a Rate Limiter needs to implement in order to be usable in a throttle.

class rush.limiters.base.**BaseLimiter**(*store: rush.stores.base.BaseStore*)
Base object defining the interface for limiters.

Users can inherit from this class to implement their own Rate Limiting Algorithm. Users must define the `rate_limit` and `reset` methods. The signatures for these methods are:

```
def rate_limit(
    self, key: str, quantity: int, rate: quota.Quota
) -> result.RateLimitResult:
   pass

def reset(self, key: str, rate: quota.Quota) -> result.RateLimitResult:
   pass
```

The `rate` parameter will always be an instance of *Quota*.

store
This is the passed in instance of a *Storage Backend*. The instance must be a subclass of *BaseStore*.

rate_limit(*key: str*, *quantity: int*, *rate: rush.quota.Quota*) → rush.result.RateLimitResult
Apply the rate-limit to a quantity of requests.

reset(*key: str*, *rate: rush.quota.Quota*) → rush.result.RateLimitResult
Reset the rate-limit for a given key.

4.3 Rush's Storage Backends

By default, rush includes the following storage backend:

- *In Memory Python Dictionary*
- *Redis*

It also has a base class so you can create your own.

class rush.stores.dictionary.**DictionaryStore**
This class implements a very simple, in-memory, non-permanent storage backend. It naively uses Python's in-built dictionaries to store rate limit data.

> **Warning:** This is not suggested for use outside of testing and initial proofs of concept.

class rush.stores.redis.**RedisStore**
This class requires a Redis URL in order to store rate limit data in Redis.

Note: This store requires installing rush with the "redis" extra, e.g.,

```
pip install -U rush[redis]
```

Example usage looks like:

```
from rush.stores import redis as redis_store

s = redis_store.RedisStore(
   url="redis://user:password@host:port",
)
```

Upon initialization, the store will create a Redis client and use that to store everything.

Further, advanced users can specify configuration parameters for the Redis client that correspond to the parameters in the redis-py documentation

4.3.1 Writing Your Own Storage Backend

Rush specifies a small set of methods that a backend needs to implement.

class rush.stores.base.**BaseStore**

Users must inherit from this class to implement their own Storage Backend. Users must define `compare_and_swap`, `set`, and `get` methods with the following signatures:

```
def get(self, key: str) -> typing.Optional[limit_data.LimitData]:
    pass

def set(
    self, *, key: str, data: limit_data.LimitData
) -> limit_data.LimitData:
    pass

def compare_and_swap(
    self,
    *,
    key: str,
    old: typing.Optional[limit_data.LimitData],
    new: limit_data.LimitData,
) -> limit_data.LimitData:
  pass
```

`compare_and_swap` must be atomic.

The way these methods communicate data back and forth between the backend and limiters is via the *LimitData* class.

class rush.limit_data.**LimitData**(*used*, *remaining*, *created_at: Union[str, datetime.datetime] = NOTHING*, ***, *time: Union[str, datetime.datetime, None] = None*)

Data class that organizes our limit data for storage.

This is a data class that represents the data stored about the user's current rate usage. It also has convenience methods for default storage backends.

created_at
A timezone-aware `datetime` object representing the first time we saw this user.

remaining
How much of the rate quota is left remaining.

time
An optional value that can be used for tracking the last time a request was checked by the limiter.

used
The amount of the rate quota that has already been consumed.

asdict() → Dict[str, str]
Return the data as a dictionary.

Returns A dictionary mapping the attributes to string representations of the values.

copy_with(*, *used: Optional[int] = None*, *remaining: Optional[int] = None*, *created_at: Optional[datetime.datetime] = None*, *time: Optional[datetime.datetime] = None*) → rush.limit_data.LimitData
Create a copy of this with updated values.

Parameters

- **used** (*int*) –
- **remaining** (*int*) –
- **created_at** (*datetime.datetime*) –
- **time** (*datetime.datetime*) –

Returns A new copy of this instance with the overridden values.

Return type LimitData

4.4 User Contributed Modules

4.4.1 Rush's Throttle Decorator

ThrottleDecorator is an inferace which allows Rush's users to limit calls to a function using a decorator. Both synchronous and asynchronous functions are supported.

class rush.contrib.decorator.**ThrottleDecorator**(*throttle: rush.throttle.Throttle*)
The class that acts as a decorator used to throttle function calls.

This class requires an intantiated throttle with which to limit function invocations.

throttle
The *Throttle* which should be used to limit decorated functions.

sleep_and_retry(*func: Callable*) → Callable
Wrap function with a naive sleep and retry strategy.

Parameters **func** (*Callable*) – The Callable to decorate.

Returns Decorated function.

Return type Callable

Example

```
from rush import quota
from rush import throttle
from rush.contrib import decorator
from rush.limiters import periodic
from rush.stores import dictionary
```

(continues on next page)

(continued from previous page)

```
t = throttle.Throttle(
    limiter=periodic.PeriodicLimiter(
        store=dictionary.DictionaryStore()
    ),
    rate=quota.Quota.per_second(
        count=1,
    ),
)

@decorator.ThrottleDecorator(throttle=t)
def ratelimited_func():
    return True

try:
    for _ in range(2):
        ratelimited_func()
except decorator.ThrottleExceeded as e:
    limit_result = e.result
    print(limit_result.limited)  # => True
    print(limit_result.remaining)  # => 0
    print(limit_result.reset_after)  # => ~0:00:01
```

4.5 Usage Examples with Rush

To make it clearer how rush can be used, we collect examples of how one *might* integrate Rush into their project.

> **Warning:** Many of these are written by the maintainers as a immediate proof of concept rather than examples of best practices using those frameworks.

Other framework examples are *very* welcome. The maintainers may not have time, however, to keep them up-to-date so your continued contributions to keep them relevant is appreciated.

4.5.1 Flask

Flask is a popular micro-framework for writing web services. In our examples directory, we have a Flask application with a single route.

In the example, we use the requestor's IP address and optional credentials to throttle their traffic. We define both anonymous and authenticated rate limits.

We use the *RateLimitResult* object to determine how to respond and to generate the RateLimit headers on the response. Here are some relevant excerpts:

```
# examples/flask/src/limiterapp/__init__.py
REDIS_URL = os.environ.get("REDIS_URL")
if REDIS_URL:
    store = redis_store.RedisStore(url=REDIS_URL)
else:
    store = dict_store.DictionaryStore()
```

(continues on next page)

(continued from previous page)

```
anonymous_quota = quota.Quota.per_hour(50)
authenticated_quota = quota.Quota.per_hour(5000, maximum_burst=500)
limiter = gcra.GenericCellRatelimiter(store=store)
anonymous_throttle = throttle.Throttle(rate=anonymous_quota, limiter=limiter)
authenticated_throttle = throttle.Throttle(
    rate=authenticated_quota, limiter=limiter
)
```

Note: We only allow the dictionary store above because this is meant as an example and we want users to be able to not require Redis when playing around with this.

```
# examples/flask/src/limiterapp/views.py
auth = request.authorization
ip_address = request.headers.get("X-Forwarded-For", request.remote_addr)
username = "anonymous"
response = flask.Response()

if auth and auth.username and auth.password:
    throttle = limiterapp.authenticated_throttle
    username = auth.username
    log.info("sent credentials", username=auth.username)

userkey = f"{username}@{ip_address}"
result = throttle.check(key=userkey, quantity=1)
response.headers.extend(
    [
        ("X-RateLimit-Limit", result.limit),
        ("X-RateLimit-Remaining", result.remaining),
        ("X-RateLimit-Reset", result.resets_at().strftime(time_format)),
        ("X-RateLimit-Retry", result.retry_at().strftime(time_format)),
    ]
)
```

```
# examples/flask/src/limiterapp/views.py
if result.limited:
    log.info("ratelimited", username=username)
    response.status_code = 403
else:
    response.status_code = 200
    response.data = f"Hello from home, {username}"
```

Playing with this example

To set up this example you need pipenv. You can `cd` into the directory and run

```
pipenv install
```

To run the server you can run

```
pipenv run gunicorn -w4 limiterapp.views:app
```

If you want to try rush out with Redis, you should set up a `.env` file like so:

```
cp env.template .env
# edit .env to include your REDIS_URL
pipenv run gunicorn -w4 limiterapp.views:app
```

You can also run `black` against this project:

```
pipenv run black -l 78 --py36 --safe src/ test/
```

If you want to contribute better Flask practices, please do so. The maintainers of rush know that it's plausible to use `app.before_request` and middleware to handle this but wanted to keep the example small-ish and reasonably contained. If you think the existing example is hard to understand, we welcome any contributions to make it easier and clearer.

4.6 Rush's Release History

4.6.1 2021.04.0 - Released on 2021-04-01

Backwards Incompatibilities

- Add `compare_and_swap` method to Base store definition for atomic operations.

  This allows limiters to ensure there are no race-conditions by having the stores provide atomic interfaces. See also *BaseStore*.

Bugs Fixed

- Update built-in limiters to rely on `compare_and_swap` method from storage backends.

Features

- Add a decorator in `rush.contrib.decorator` written by Jude188 for potentially easier use of the library. See also *ThrottleDecorator*.

4.6.2 2018.12.1 - Released on 2018-12-25

I realized I missed one crucial thing for production usage.

Bugs Fixed

- Rely on stores to set the current time and provide the clock for limiters.

4.6.3 2018.12.0 - Released on 2018-12-22

This is the initial release of the rush library. It includes a rough API for using different rate limiting algorithms with storage backends. It aims to provide a composable set of algorithms and storage backends for use when rate-limiting (or throttling) activities. This release includes support for:

- Periodic Rate Limiting

- Generic Cell Rate Limiting
- Redis Storage
- In-memory Python Dictionary Storage

Index