Falcon-Limiter

Release 1.0.1

Mar 31, 2022

Contents

Version: 1.0.1

Falcon-Limiter provides advanced rate limiting support to the Falcon web framework.

Rate limiting strategies are provided with the help of the popular Limits library.

The library aims to be compatible with CPython 3.6+ and PyPy 3.5+.

CHAPTER 1

Quickstart

1.1 WSGI

Quick example - using *fixed-window* strategy and storing the hits against limits in the memory:

```
import falcon
from falcon_limiter import Limiter
from falcon_limiter.utils import get_remote_addr

limiter = Limiter(
    key_func=get_remote_addr,
    default_limits="5 per minute,2 per second"
)

# use the default limit for all methods of this class
@limiter.limit()
class ThingsResource:
    def on_get(self, req, resp):
        resp.body = 'Hello world!'

# add the limiter middleware to the Falcon app
app = falcon.API(middleware=limiter.middleware)

things = ThingsResource()
app.add_route('/things', things)
```

1.2 ASGI (Async)

Quick example - using *fixed-window* strategy and storing the hits against limits in the memory:

```
import falcon.asgi
from falcon_limiter import AsyncLimiter
from falcon_limiter.utils import get_remote_addr

limiter = AsyncLimiter(
    key_func=get_remote_addr,
    default_limits="5 per minute,2 per second"
)

# use the default limit for all methods of this class
@limiter.limit()
class ThingsResource:
    async def on_get(self, req, resp):
        resp.body = 'Hello world!'

# add the limiter middleware to the Falcon app
app = falcon.asgi.App(middleware=limiter.middleware)

things = ThingsResource()
app.add_route('/things', things)
```

See *Async (experimental)* for more about Async.

1.3 A more complicated example

When making calls against this app, above >5 calls per minute or >2 per seconds you will receive an HTTP 429 error response with message: *"Reached allowed limit 5 hits per 1 minute!"*

A second, more complicated example - using the *moving-window* strategy with a shared Redis backend and running the application behind a reverse proxy:

```
import falcon
from falcon_limiter import Limiter

# a custom key function
def get_access_route_addr(req, resp, resource, params) -> str:
    """ Get the requestor's IP by discounting 1 reverse proxy
    """
    return req.access_route[-2]

limiter = Limiter(
    key_func=get_access_route_addr,
    default_limits="5 per minute,2 per second",
    # only count HTTP 200 responses against the limit:
    default_deduct_when=lambda req, resp, resource, req_succeeded:
        resp.status == falcon.HTTP_200,
    config={
        'RATELIMIT_KEY_PREFIX': 'myapp',  # to allow multiple apps in the same Redis␣
↪db
        'RATELIMIT_STORAGE_URL': f'redis://:{REDIS_PSW}@{REDIS_HOST}:{REDIS_PORT}',
        'RATELIMIT_STRATEGY': 'moving-window'
    }
)

class ThingsResource:
```

(continues on next page)

(continued from previous page)

```
    # no rate limit on this method
    def on_get(self, req, resp):
        resp.body = 'Hello world!'

    # a more strict rate limit applied to this method
    # with a custom key function serving up the user_id
    # from the request context as key
    @limiter.limit(limits="3 per minute,1 per second",
        key_func=lambda req, resp, resource, params: req.context.user_id)
    def on_post(self, req, resp):
        resp.body = 'Hello world!'

class SpecialResource:
    # dynamic_limits allowing the 'admin' user a higher limit than others
    @limiter.limit(dynamic_limits=lambda req, resp, resource, params:
        '999/minute,9999/second' if req.context.user == 'admin'
        else '5 per minute,2/second')
    def on_get(self, req, resp):
        resp.body = 'Hello world!'

# add the limiter middleware to the Falcon app
app = falcon.API(middleware=limiter.middleware)

things = ThingsResource()
special = SpecialResource()
app.add_route('/things', things)
app.add_route('/special', special)
```

CHAPTER 2

Installation

Install the extension with pip:

```
$ pip install Falcon-Limiter
```

CHAPTER 3

Set Up

Rate limiting is managed through a `Limiter` instance:

```
import falcon
from falcon_limiter import Limiter
from falcon_limiter.utils import get_remote_addr

limiter = Limiter(
    key_func=get_remote_addr,
    default_limits="5 per minute,2 per second"
)
```

The `Limiter` instance is a Falcon Middleware and it has a `limit()` method which can be used as a decorator to decorate a whole class or individual methods:

```
import falcon
from falcon_limiter import Limiter
from falcon_limiter.utils import get_remote_addr

limiter = Limiter(
    key_func=get_remote_addr,
    default_limits="5 per minute,2 per second"
)

# use the default limit for all methods of this class
@limiter.limit()
class ThingsResource:
    def on_get(self, req, resp):
        resp.body = 'Hello world!'

# use the default limit for all methods of this class
@limiter.limit()
class ThingsResource2:
    # this will use the default limit from the class
    def on_get(self, req, resp):
```

(continues on next page)

(continued from previous page)

```
        resp.body = 'Hello world!'

    # this will use a custom limit overwriting the one set at class level
    @limiter.limit(limits="3 per minute,1 per second")
    def on_post(self, req, resp):
        resp.body = 'Hello world!'
```

You can provide a config dictionary to the `Limiter`, see *Configuring Falcon-Limiter*:

```
import falcon
from falcon_limiter import Limiter
from falcon_limiter.utils import get_remote_addr

limiter = Limiter(
    key_func=get_remote_addr,
    default_limits="5 per minute,2 per second",
    config={
        'RATELIMIT_KEY_PREFIX': 'myapp',  # to allow multiple apps in the same Redis␣
↪db
        'RATELIMIT_STORAGE_URL': f'redis://:{REDIS_PSW}@{REDIS_HOST}:{REDIS_PORT}',
        'RATELIMIT_STRATEGY': 'moving-window'
    }
)
```

The limiter instance needs to be specified as a middleware when creating the app by calling `falcon.API()`:

```
import falcon
from falcon_limiter import Limiter
from falcon_limiter.utils import get_remote_addr

limiter = Limiter(
    key_func=get_remote_addr,
    default_limits="5 per minute,2 per second"
)

@limiter.limit()
class ThingsResource:
    def on_get(self, req, resp):
        resp.body = 'Hello world!'

# add the limiter middleware to the Falcon app
app = falcon.API(middleware=limiter.middleware)
```

CHAPTER 4

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.

4.1 Examples

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

CHAPTER 5

Rate limiting strategies

Falcon-Limiter comes with three different rate limiting strategies built-in, provided by the Limits library.

Pick the one that works for your use-case by specifying it in your config as `RATELIMIT_STRATEGY` (one of `fixed-window`, `fixed-window-elastic-expiry`, or `moving-window`). The default configuration is `fixed-window`.

5.1 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.

5.2 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.

5.3 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.

5.4 Configuring Falcon-Limiter

Config values can be provided as a dictionary to the `Limiter()`. For example:

```python
from falcon_limiter import Limiter
from falcon_limiter.utils import get_remote_addr

limiter = Limiter(
    key_func=get_remote_addr,
    default_limits="5 per minute,2 per second",
    config={
        'RATELIMIT_KEY_PREFIX': 'myapp',
        'RATELIMIT_STORAGE_URL': 'redis://@redis:6379',
        'RATELIMIT_STRATEGY': 'moving-window'
    }
)
```

The following configuration values exist for Falcon-Limiter:

`RATELIMIT_STORAGE_URL`	A storage location conforming to the scheme in Supported versions. A basic in-memory storage can be used by specifying `memory://` though this should probably never be used in production. Some supported backends include: • Memcached: `memcached://host:port` • Memcached on Google App Engine: `gaememcached://host:port` • Redis listening on TCP: `redis://host:port` • Redis listening on a unix domain socket: `redis+unix:///path/to/socket?db=n` • Redis with password and db specified: `redis://:password@host:port?db=n` • Redis over SSL: `rediss://host:port` • Redis Sentinel: `redis+sentinel://host:26379/my-redis-service` or `redis+sentinel://host:26379,host:26380/my-redis-service` • Redis Cluster: `redis+cluster://localhost:7000` or `redis+cluster://localhost:7000,localhost:70001` • GAE Memcached: `gaememcached://host:port` For more examples and requirements of supported backends please refer to Supported versions.
`RATELIMIT_STORAGE_OPTIONS`	A dictionary to set extra options to be passed to the storage implementation upon initialization. (Useful if you're subclassing `limits.Storage` to create a subclassing `limits.storage.Storage` to create a custom Storage backend.)
`RATELIMIT_STRATEGY`	The rate limiting strategy to use. See *Rate limiting strategies* for details.
`RATELIMIT_KEY_PREFIX`	If you are using a shared backend - like a Redis instance shared by multiple apps, then to avoid a potential clash between the ratelimit records, you should provide a string in `RATELIMIT_KEY_PREFIX`, which will be added to the key.

CHAPTER 6

Recipes

6.1 Application is served from behind a reverse proxy

Falcon applications are frequently served from behind loadbalancers and reverse proxies. In such a case care must be given to pick up the right IP address - the one representing the requestor and NOT the reverse proxy, otherwise you will be applying a shared rate limit to all your users coming through that reverse proxy.

Usually the reverse proxy appends the IP address to one of the header (like X-Forwarded-For) and in Falcon the list of IP addresses from those headers are amde available under the `access_route` Request attribute. See https://falcon.readthedocs.io/en/stable/api/request_and_response.html#falcon.Request.access_route

An example recipe to handle a single reverse proxy in front of our application is to provide a custom key function which derives the requestor's IP address from the `access_route`:

```
import falcon
from falcon_limiter import Limiter

# a custom key function
def get_access_route_addr(req, resp, resource, params) -> str:
    """ Get the requestor's IP by discounting 1 reverse proxy
    """
    return req.access_route[-2]

limiter = Limiter(
    key_func=get_access_route_addr,
    default_limits="5 per minute,2 per second",
)

@limiter.limit()
class ThingsResource:
    def on_get(self, req, resp):
        resp.body = 'Hello world!'

    # this endpoint is routed differently (through 2 proxies), so it
```

(continues on next page)

(continued from previous page)

```
    # requires a custom key function specific to this method
    @limiter.limit(key_func=lambda req, resp, resource, params: req.access_route[-3])
    def on_post(self, req, resp):
        resp.body = 'Hello world!'
```

Note: A custom key function must accept the 'usual' Falcon response attributes and return a string:

def custom_key_func(req, resp, resource, params) -> str:

6.2 Ratelimit by resource and method

A custom key function can be useful in other scenarios too, for example when you want to ratelimit by resource and method. This is the scenario where you would want the ratelimit counted SEPARATELY for each endpoint.

```
def get_key(req, resp, resource, params) -> str:
    """ Build a key from the IP + resource name + method name """
    user_key = get_remote_addr(req, resp, resource, params)
    return f"{user_key}:{resource.__class__.__name__}:{req.method}"

limiter = Limiter(
    key_func=get_key,
    default_limits=["10 per hour", "2 per minute"]
)
```

6.3 Ratelimit by user instead of IP

A custom key function can be also be used to implement rate limit by authenticated user instead of IP. This can be useful in scenarios when the users are coming from a proxied environment (like most corporate environment), as they will be sharing the same public IP.

First you will need to authenticate your user and place the user id onto the request context, so then your custom key function can pick it up from there.

```
def get_key_(req, resp, resource, params) -> str:
    """ Build a key from the user id stored on the request context
    or the IP when that is user id not available """
    if hasattr(req.context, 'user_id'):
        return req.context.user_id
    else:
        return get_remote_addr(req, resp, resource, params)

limiter = Limiter(
    key_func=get_key,
    default_limits=["10 per hour", "2 per minute"]
)
```

6.4 Dynamic limits

With the use of the `default_dynamic_limits` and `dynamic_limits` parameters you can define the limits dynamically, at the time of the processing of the request.

This allows you to define different limits by users - for example allowing an admin user higher limit than others, or differentiating the limits based on the 'subscription' the given requester belongs to.

```
from falcon_limiter.utils import get_remote_addr

limiter = Limiter(
    key_func=get_remote_addr,
    # the default limit is 9999/second for admin and
    # 20/minute,2/second for everybody else:
    default_dynamic_limits=lambda req, resp, resource, params:
        '9999/second' if req.context.user == 'admin'
        else '20/minute,2/second'
)

@limiter.limit()
class ThingsResource:
    # this endpoint gets a 5/second limit for those sending
    the APIUSER=admin header:
    @limiter.limit(dynamic_limits=lambda req, resp, resource, params:
        '5/second'if req.get_header('APIUSER') == 'admin'
        else '20/minute,2/second'
    def on_get(self, req, resp):
        resp.body = 'Hello world!'

    def on_post(self, req, resp):
        resp.body = 'Hello world!'
```

6.5 Customizing rate limits based on response

For scenarios where the decision to count the current request towards a rate limit can only be made after the request has completed, a callable can be provided.

The `deduct_when` function can be either provided to the `Limiter` as `default_deduct_when` parameter or to the decorator as `deduct_when` parameter.

```
import falcon
from falcon_limiter import Limiter

limiter = Limiter(
    key_func=get_remote_addr,
    default_limits=["10 per hour", "2 per minute"],
    # this will apply to ALL limits:
    default_deduct_when=lambda req, resp, resource, req_succeeded:
        resp.status == falcon.HTTP_200
)

@limiter.limit()
class ThingsResource:
    # this deduct when only applies to this method
    @limiter.limit(deduct_when=lambda req, resp, resource, req_succeeded:
```

(continues on next page)

(continued from previous page)

```
        resp.status != falcon.HTTP_500)
    def on_get(self, req, resp):
        resp.body = 'Hello world!'

    def on_post(self, req, resp):
        resp.body = 'Hello world!'
```

6.6 Multiple decorators

For scenarios where there is a need for multiple decorators and the `@limiter.limit()` cannot be the topmost one, we need to register the decorators a special way.

This scenario is complicated because our `@limiter.limit()` just marks the fact that the given method is decorated with a limit, which later gets picked up by the middleware and triggers the rate limiting. If the `@limiter.limit()` is the topmost decorator then it is easy to pick that up, but if there are other decorators 'ahead' it, then those will 'hide' the `@limiter.limit()`. This is because decorators in Python are just syntactic sugar for nested function calls.

To be able to tell if the given endpoint was decorated by the `@limiter.limit()` decorator when that is NOT the topmost decorator, you need to decorate your method by registering your decorators using the `@register()` helper decorator.

See more about this issue at https://stackoverflow.com/questions/3232024/introspection-to-get-decorator-names-on-a-method

```
import falcon
from falcon_limiter import Limiter
from falcon_limiter.utils import register

limiter = Limiter(
    key_func=get_remote_addr,
    default_limits=["10 per hour", "2 per minute"]
)


class ThingsResource:
    # this is fine, as the @limiter.limit() is the topmost decorator:
    @limiter.limit()
    @another_decorator
    def on_get(self, req, resp):
        resp.body = 'Hello world!'

    # the @limiter.limit() is NOT the topmost decorator, so
    # this would NOT work - the limit would be ignored!!!!
    # DO NOT DO THIS:
    @another_decorator
    @limiter.limit()
    def on_post(self, req, resp):
        resp.body = 'WARNING: NO LIMITS ON THIS!'

    # instead register your decorators this way:
    @register(another_decorator, limiter.limit())
    def on_post(self, req, resp):
        resp.body = 'This is properly limited'

app = falcon.API(middleware=limiter.middleware)
```

Note: The deduct_when function must accept the 'usual' Falcon response attributes and return a boolean:

def my_deduct_when_func(req, resp, resource, params) -> bool:

6.7 async (experimental)

New in version 1.0.

Warning: Experimental

Async (ASGI) support has been added in experimental mode, thanks to the Limits library.

This was implemented by the addition of `falcon.limiter.AsyncLimiter` module, which copies the functionalities of `falcon.limiter.Limiter`.

Use `falcon.limiter.AsyncLimiter` for Async (ASGI) and `falcon.limiter.Limiter` for WSGI.

The following async storage backends are implemented:

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

6.7.1 Async examples

A few examples to demonstrate the use of the async module.

1. A basic example using the In-Memory storage option

```
import falcon.asgi
from falcon_limiter import AsyncLimiter
from falcon_limiter.utils import get_remote_addr

limiter = AsyncLimiter(
    key_func=get_remote_addr,
    default_limits="5 per minute,2 per second"
)

# use the default limit for all methods of this class
@limiter.limit()
class ThingsResource:
    async def on_get(self, req, resp):
        resp.body = 'Hello world!'

# add the limiter middleware to the Falcon app
app = falcon.asgi.App(middleware=limiter.middleware)

things = ThingsResource()
app.add_route('/things', things)
```

2. Redis storage

```python
import falcon.asgi
from falcon_limiter import AsyncLimiter
from falcon_limiter.utils import get_remote_addr
import logging

# include ERROR logs
logging.basicConfig()
logging.getLogger().setLevel(logging.ERROR)

limiter = AsyncLimiter(
    key_func=get_remote_addr,
    default_limits=["500 per hour", "20 per minute"],
    config={
        'RATELIMIT_KEY_PREFIX': 'myapp',
        'RATELIMIT_STORAGE_URL': 'async+redis://:MyRedisPassword@localhost:6379'
    }
)

# The deduct_when function is NOT async!
def deduct_when_func(req, resp, resource, req_succeeded):
    return resp.status == falcon.HTTP_200

class ThingsResource:
    @limiter.limit(limits="2 per hour;1 per minute",
                   deduct_when=deduct_when_func)
    async def on_get(self, req, resp):
        resp.body = 'Hello world!'

# add the limiter middleware to the Falcon app
app = falcon.asgi.App(middleware=limiter.middleware)

things = ThingsResource()
app.add_route('/things', things)
```

Please note, that when using the `AsyncLimiter`, then a class-level decorator will overwrite all method-level decorators of that class. This behaviour is different from the WSGI (eg sync) `Limiter`, where method-level decorators overwrite the class level one.

6.8 Development

For development guidelines see https://github.com/zoltan-fedor/falcon-limiter#development

6.9 API Reference

If you are looking for information on a specific function, class or method of a service, then this part of the documentation is for you.

6.9.1 API Reference Guide

Limiter

Limiter

class `falcon_limiter.limiter.`**`Limiter`**(*key_func: Callable = <function get_remote_addr>, default_limits: str = ”, default_deduct_when: Callable = None, default_dynamic_limits: Callable = None, config: Optional[Dict[str, Any]] = None*)

This is the central class for the limiting

You need to initialize this object to setup the attributes of the Limiter and then supply the object's middleware to the Falcon app.

Parameters

- **`key_func`** (*`callable`*) – A function that will receive the usual Falcon request method arguments (req, resp, resource, params) and expected to return a string which will be used as a representation of the user for whom the rate limit will apply (eg the key).
- **`default_limits`** (*`str`*) – Optional string of limit(s) separated by ";", like '1/second;3 per hour'
- **`default_deduct_when`** (*`callable`*) – A function which determines at response time whether the given request should be counted against the limit or not. This allows the creation of strategies incorporating the response status code.
- **`default_dynamic_limits`** (*`callable`*) – A function which builds the 'limits' string dynamically based on the Falcon request method arguments (req, resp, resource, params). It is expected to return a 'limits' string like '1/second;3 per hour'.
- **`config`** (*`dict of str`*) – Optional config settings provided as a dictionary

`key_func`

A function that will receive the usual Falcon request method arguments (req, resp, resource, params) and expected to return a string which will be used as a representation of the user for whom the rate limit will apply (eg the key).

Type callable

`default_limits`

Optional string of limit(s) separated by ";", like '1/second;3 per hour'

Type str

`default_deduct_when`

A function which determines at response time whether the given request should be counted against the limit or not. This allows the creation of strategies incorporating the response status code.

Type callable

`default_dynamic_limits`

A function which builds the 'limits' string dynamically based on the Falcon request method arguments (req, resp, resource, params). It is expected to return a 'limits' string like '1/second;3 per hour'.

Type callable

`config`

Config settings stored as a dictionary

Type dict of str

`storage`

The storage backend that will be used to store the rate limits.

Type `Storage`

limiter

A *RateLimiter* object from the *limits* library, representing the rate limiting strategy and storage.

Type `RateLimiter`

limit(*limits: str = None*, *deduct_when: Callable = None*, *key_func: Callable = None*, *dynamic_limits: Callable = None*) → Callable

This is the decorator used to decorate a resource class or the requested method of the resource class with the default or with a custom limit

Parameters

- **limits** (*str*) – Optional string of limit(s) separated by ";", like '1/second;3 per hour' deduct_when (callable): A function that will receive the usual falcon response method arguments (req, resp, resource, params) and expected to return a boolean which is used to determine if the given response qualifies to count against the set limit.
- **deduct_when** (*callable*) – A function which determines at response time whether the given request should be counted against the limit or not. This allows the creation of strategies incorporating the response status code.
- **key_func** (*callable*) – A function that will receive the usual falcon response method arguments (req, resp, resource, params) and expected to return a string which will be used as a representation of the user for whom the rate limit will apply (eg the key).
- **dynamic_limits** (*callable*) – A function which builds the 'limits' string dynamically based on the Falcon request method arguments (req, resp, resource, params). It is expected to return a 'limits' string like '1/second;3 per hour'.

middleware

Falcon middleware integration

6.10 Additional Information

6.10.1 Changelog

Version 1.0.1

- Switching CI from Travis to GitHub Actions

Version 1.0.0

- Async support has been added in experimental mode

Version 0.1.2

- Fixing the issue with multiple decorators when *@limiter.limit()* is not the topmost one

Version 0.1.1

- Fixing document readability issues in Sphinx

Version 0.1.0

- Added the ability to supply a custom *key_func* on the class/methods decorators
- Added the dynamic_limits parameter allowing limits to be built at request time

Version 0.0.1

- Initial public release

6.10.2 License

```
MIT License

Copyright (c) 2020 Zoltan Fedor

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.
```

- search

Python Module Index

Index