Django-Limiter Documentation

Release 0.2-dirty

Ali-Akber Saifee

December 20, 2015

Contents

Rate limiting middleware for Django applications

CHAPTER 1

Configuration

The following django settings are honored by the middleware.

RATELIMIT_GLOBAL A comma (or some other delimiter) separated string that will be used to apply a global limit on all routes. *Rate limit string notation* for details.

RATELIMIT_CALLBACK The callback which will be called when a rate limit is hit. Defaults to a `429` status code being returned.

RATELIMIT_KEY_FUNCTION The default callable to use to derive the context of a request. This can either be the callable itself or a fully qualified path to a callable. The callable to accept a single request object as a parameter and return a string. Defaults to the `ip address` of the requesting user.

RATELIMIT_STORAGE_URL One of `memory://`, `redis://host:port` or `memcached://host:port`. Using the redis storage requires the installation of the redis package while memcached relies on pymemcache.

RATELIMIT_STRATEGY The rate limiting strategy to use. *Rate limiting strategies* for details.

RATELIMIT_HEADERS_ENABLED Enables returning *Rate-limiting Headers*. Defaults to `False`

RATELIMIT_ENABLED Overall kill switch for rate limits. Defaults to `True`

RATELIMIT_HEADER_LIMIT Header for the current rate limit. Defaults to `X-RateLimit-Limit`

RATELIMIT_HEADER_RESET Header for the reset time of the current rate limit. Defaults to `X-RateLimit-Reset`

RATELIMIT_HEADER_REMAINING Header for the number of requests remaining in the current rate limit. Defaults to `X-RateLimit-Remaining`

CHAPTER 2

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.

2.1 Examples

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

CHAPTER 3

Decorators

The decorators made available as instance methods of the `Limiter` instance are

limit() There are a few ways of using this decorator depending on your preference and use-case.

Single decorator The limit string can be a single limit or a delimiter separated string

```
@limit("100/day;10/hour;1/minute")
def my_view(request)
  ...
```

Multiple decorators The limit string can be a single limit or a delimiter separated string or a combination of both.

```
@limit("100/day")
@limit("10/hour")
@limit("1/minute")
def my_view(request):
  ...
```

Custom keying function By default rate limits are applied on per remote address basis. You can implement your own function to retrieve the key to rate limit by.

```
def my_key_func(request):
  ...

@limit("100/day", my_key_func)
def my_view(request):
  ...
```

Note: The key function must accept one argument which is a `django.http.HttpRequest` object

---

Dynamically loaded limit string(s) There may be situations where the rate limits need to be retrieved from sources external to the code (database, remote api, etc...). This can be achieved by providing a callable (which takes a single parameter - the `django.http.HttpRequest` object) to the decorator. The callable should return a rate limit string in the *Rate limit string notation*.

```
from django.conf import settings

def rate_limit_from_config(request):
    return settings.CUSTOM_LIMIT
```

```
@limit(rate_limit_from_config)
def my_view(request):
    ...
```

> **Danger:** The provided callable will be called for every request on the decorated route. For expensive retrievals, consider caching the response.

shared_limit() For scenarios where a rate limit should be shared by multiple views (For example when you want to protect views using the same resource with an umbrella rate limit).

Named shared limit

```
mysql_limit = shared_limit("100/hour", scope="mysql")

@mysql_limit
def my_view_1(request):
    ...

@mysql_limit
def my_view_2(request):
    ...
```

Dynamic shared limit: when a callable is passed as scope, the return value of the function will be used as the scope.

```
def host_scope(request):
    return request.META['HTTP_HOST']

host_limit = limiter.shared_limit("100/hour", scope=host_scope)

@host_limit
def my_view_1():
    ...

@host_limit
def my_view_2():
    ...
```

Note: Shared rate limits provide the same conveniences as individual rate limits

- Can be chained with other shared limits or individual limits
- Accept keying functions
- Accept callables to determine the rate limit value

---

exempt() This decorator simply marks a view as being exempt from any rate limits.

CHAPTER 4

Rate limiting strategies

djlimiter comes with three different rate limiting strategies built-in. Pick the one that works for your use-case by specifying it in your django settings as `RATELIMIT_STRATEGY` (one of `fixed-window`, `fixed-window-elastic-expiry`, or `moving-window`). The default configuration is `fixed-window`.

4.1 Fixed Window

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

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

4.2 Fixed Window with Elastic Expiry

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

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

4.3 Moving Window

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

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

CHAPTER 5

Rate-limiting Headers

If the configuration is enabled, information about the rate limit with respect to the route being requested will be added to the response headers. Since multiple rate limits can be active for a given route - the rate limit with the lowest time granularity will be used in the scenario when the request does not breach any rate limits.

`X-RateLimit-Limit`	The total number of requests allowed for the active window
`X-RateLimit-Remaining`	The number of requests remaining in the active window.
`X-RateLimit-Reset`	UTC seconds since epoch when the window will be reset.

Warning: Enabling the headers has an additional cost with certain storage / strategy combinations.

- Memcached + Fixed Window: an extra key per rate limit is stored to calculate `X-RateLimit-Reset`
- Redis + Moving Window: an extra call to redis is involved during every request to calculate `X-RateLimit-Remaining` and `X-RateLimit-Reset`

The header names can be customised if required by using django settings (*Configuration*).

CHAPTER 6

Recipes

6.1 Custom Rate limit domains

By default, all rate limits are applied on a per `remote address` basis. However, you can easily customize your rate limits to be based on any other characteristic of the incoming request. On a django settings level this can be achieved by settings the `RATELIMIT_KEY_FUNCTION` to either point to a callable or a fully qualified path to a callable. This callable should:

- Expect a single `django.http.HttpRequest` object as a parameter.
- Return a string that classifies the request.

6.2 Using Django Class Views

If you are using a class based approach to defining view functions, the regular method of decorating a view function to apply a per route rate limit will not work. You can add rate limits to your views by using the following approach (also described in Decorating the class).

```
class MyView(django.views.generic):
    def get(self):
        return HttpResponse("get")

    def put(self):
        return HttpResponse("put")
```

```
urlpatterns = patterns('',
    (r'^myview/', limit("2/second")(MyView.as_view())),
)
```

Note: This approach is limited to either sharing the same rate limit for all http methods of a given `django.views.generic.View` or applying the declared rate limit independently for each http method (to accomplish this, pass in `True` to the `per_method` keyword argument to `limit()`).

6.3 Logging

djlimiter uses standard python logging. To enable logging, configure the `djlimiter` logger in your `settings.py`.:

```
LOGGING = {
    'version': 1,
    'disable_existing_loggers': False,
    'handlers': {
        'stream': {
            'level': 'DEBUG',
            'class': 'logging.StreamHandler',
        },
    },
    'loggers': {
        'djlimiter': {
            'handlers': ['stream'],
            'level': 'INFO',
            'propagate': True,
        },
    },
}
```

For more details about configuring django logging refer to Configuring logging

CHAPTER 7

API

7.1 Core

class djlimiter.**Limiter**
Bases: object

process_request(*request*)

Parameters **request** –

Returns

process_response(*request*, *response*)

Parameters

- **request** –
- **response** –

Returns

djlimiter.**limit**(*limit_value*, *key_function=None*, *per_method=False*)
decorator to be used for rate limiting individual views

Parameters

- **limit_value** – rate limit string or a callable that returns a string. *Rate limit string notation* for more details.
- **key_func** (*function*) – function/lambda to extract the unique identifier for the rate limit. defaults to remote address of the request.
- **per_method** (*bool*) – whether the limit is sub categorized into the http method of the request.

djlimiter.**exempt**(*fn*)
decorator to mark a view or all views in a blueprint as exempt from rate limits.

Parameters **fn** – the view to wrap.

Returns

djlimiter.**shared_limit**(*limit_value*, *scope*, *key_function=None*)
decorator to be applied to multiple views sharing the same rate limit.

Parameters

- **limit_value** – rate limit string or a callable that returns a string. *Rate limit string notation* for more details.
- **scope** – a string or callable that returns a string for defining the rate limiting scope.
- **key_func** (*function*) – function/lambda to extract the unique identifier for the rate limit. defaults to remote address of the request.

7.2 Exceptions

exception djlimiter.**RateLimitExceeded**(*limit*)

Bases: django.http.response.HttpResponse

exception raised when a rate limit is hit (status code: 429).

CHAPTER 8

Usage

8.1 Quick start

Add the rate limiter to your django projects' *settings.py* and enable a global rate limit for all views in your project:

```
MIDDLEWARE_CLASSES += ("djlimiter.Limiter",)
RATELIMIT_GLOBAL = "10/second; 50/hour"
```

In one of the apps' view:

```
@limit("5/second")
def index(request):
    ...

@exempt
def ping(request):
    ...
```

The above example will result in the following characteristics being applied to the django project:

- A global rate limit of 10 per second, and 50 per hour applied to all routes.
- The `index` route will have an explicit rate limit of 5/second
- The `ping` route will be exempt from any global rate limits.

Every time a request exceeds the rate limit, the view function will not get called and instead a 429 http error will be raised.

Refer to *Recipes* for more examples.

CHAPTER 9

References

- Redis rate limiting pattern #2
- DomainTools redis rate limiter
- limits: python rate limiting utilities

CHAPTER 10

Changelog

10.1 0.2 2015-12-20

- Django 1.8/1.9 compatibility

10.2 0.1.1 2015-01-09

- Bug Fix: remove duplicate hits when rate limits are stacked.
- Bug Fix: multiple rate limits returned by dynamic limits weren't respected.
- Documentation tweaks.

10.3 0.1.0 2015-01-09

- first release.

E

L

P

R

S