Django Ratelimit Documentation

Release 2.0.0

James Socol

Feb 21, 2019

Contents

CHAPTER 1

Project

Django Ratelimit is a ratelimiting decorator for Django views. build passing

Code https://github.com/jsocol/django-ratelimit

License Apache Software License

Issues https://github.com/jsocol/django-ratelimit/issues

Documentation http://django-ratelimit.readthedocs.org/

CHAPTER 2

Quickstart

Install:

```
pip install django-ratelimit
```

Use as a decorator in `views.py`:

```
from ratelimit.decorators import ratelimit

@ratelimit(key='ip')
def myview(request):
    # ...

@ratelimit(key='ip', rate='100/h')
def secondview(request):
    # ...
```

CHAPTER 3

Contents

3.1 Settings

3.1.1 RATELIMIT_CACHE_PREFIX

An optional cache prefix for ratelimit keys (in addition to the `PREFIX` value defined on the cache backend). Defaults to `'rl:'`.

3.1.2 RATELIMIT_ENABLE

Set to `False` to disable rate-limiting across the board. Defaults to `True`.

May be useful during tests with Django's `override_settings()` testing tool, for example:

```
from django.test import override_settings

with override_settings(RATELIMIT_ENABLE=False):
    result = call_the_view()
```

3.1.3 RATELIMIT_USE_CACHE

The name of the cache (from the `CACHES` dict) to use. Defaults to `'default'`.

3.1.4 RATELIMIT_VIEW

The string import path to a view to use when a request is ratelimited, in conjunction with `RatelimitMiddleware`, e.g. `'myapp.views.ratelimited'`. Has no default - you must set this to use `RatelimitMiddleware`.

3.1.5 `RATELIMIT_FAIL_OPEN`

Whether to allow requests when the cache backend fails. Defaults to `False`.

3.2 Using Django Ratelimit

3.2.1 Use as a decorator

Import:

```
from ratelimit.decorators import ratelimit
```

`@`**`ratelimit`**`(`*group=None*, *key=*, *rate=None*, *method=ALL*, *block=False*`)`

Parameters

- **`group`** – *None* A group of rate limits to count together. Defaults to the dotted name of the view.
- **`key`** – What key to use, see *Keys*.
- **`rate`** – *'5/m'* The number of requests per unit time allowed. Valid units are:
  - `s` - seconds
  - `m` - minutes
  - `h` - hours
  - `d` - days

  Also accepts callables. See *Rates*.
- **`method`** – *ALL* Which HTTP method(s) to rate-limit. May be a string, a list/tuple of strings, or the special values for `ALL` or `UNSAFE` (which includes `POST`, `PUT`, `DELETE` and `PATCH`).
- **`block`** – *False* Whether to block the request instead of annotating.

HTTP Methods

Each decorator can be limited to one or more HTTP methods. The `method=` argument accepts a method name (e.g. `'GET'`) or a list or tuple of strings (e.g. `('GET', 'OPTIONS')`).

There are two special shortcuts values, both accessible from the `ratelimit` decorator, the `RatelimitMixin` class, or the `is_ratelimited` helper, as well as on the root `ratelimit` module:

```
from ratelimit.decorators import ratelimit

@ratelimit(key='ip', method=ratelimit.ALL)
@ratelimit(key='ip', method=ratelimit.UNSAFE)
def myview(request):
    pass
```

`ratelimit.ALL` applies to all HTTP methods. `ratelimit.UNSAFE` is a shortcut for `('POST', 'PUT', 'PATCH', 'DELETE')`.

Examples

```
@ratelimit(key='ip', rate='5/m')
def myview(request):
    # Will be true if the same IP makes more than 5 POST
    # requests/minute.
    was_limited = getattr(request, 'limited', False)
    return HttpResponse()

@ratelimit(key='ip', rate='5/m', block=True)
def myview(request):
    # If the same IP makes >5 reqs/min, will raise Ratelimited
    return HttpResponse()

@ratelimit(key='post:username', rate='5/m', method=['GET', 'POST'])
def login(request):
    # If the same username is used >5 times/min, this will be True.
    # The `username` value will come from GET or POST, determined by the
    # request method.
    was_limited = getattr(request, 'limited', False)
    return HttpResponse()

@ratelimit(key='post:username', rate='5/m')
@ratelimit(key='post:tenant', rate='5/m')
def login(request):
    # Use multiple keys by stacking decorators.
    return HttpResponse()

@ratelimit(key='get:q', rate='5/m')
@ratelimit(key='post:q', rate='5/m')
def search(request):
    # These two decorators combine to form one rate limit: the same search
    # query can only be tried 5 times a minute, regardless of the request
    # method (GET or POST)
    return HttpResponse()

@ratelimit(key='ip', rate='4/h')
def slow(request):
    # Allow 4 reqs/hour.
    return HttpResponse()

rate = lambda r: None if request.user.is_authenticated else '100/h'
@ratelimit(key='ip', rate=rate)
def skipif1(request):
    # Only rate limit anonymous requests
    return HttpResponse()

@ratelimit(key='user_or_ip', rate='10/s')
@ratelimit(key='user_or_ip', rate='100/m')
def burst_limit(request):
    # Implement a separate burst limit.
    return HttpResponse()

@ratelimit(group='expensive', key='user_or_ip', rate='10/h')
def expensive_view_a(request):
    return something_expensive()
```

(continues on next page)

(continued from previous page)

```
@ratelimit(group='expensive', key='user_or_ip', rate='10/h')
def expensive_view_b(request):
    # Shares a counter with expensive_view_a
    return something_else_expensive()

@ratelimit(key='header:x-cluster-client-ip')
def post(request):
    # Uses the X-Cluster-Client-IP header value.
    return HttpResponse()

@ratelimit(key=lambda r: r.META.get('HTTP_X_CLUSTER_CLIENT_IP',
                                     r.META['REMOTE_ADDR'])
def myview(request):
    # Use `X-Cluster-Client-IP` but fall back to REMOTE_ADDR.
    return HttpResponse()
```

Class-Based Views

New in version 0.5.

The `@ratelimit` decorator also works on class-based view methods, though *make sure the ‘‘method‘‘ argument matches the decorator*:

```
class MyView(View):
    @ratelimit(key='ip', method='POST')
    def post(self, request, *args):
        # Something expensive...
```

Note: Unless given an explicit `group` argument, different methods of a class-based view will be limited separate.

3.2.2 Class-Based View Mixin

class `ratelimit.mixins.`**`RatelimitMixin`**

New in version 0.4.

Ratelimits can also be applied to class-based views with the `ratelimit.mixins.RatelimitMixin` mixin. They are configured via class attributes that are the same as the *decorator*, prefixed with `ratelimit_`, e.g.:

```
class MyView(RatelimitMixin, View):
    ratelimit_key = 'ip'
    ratelimit_rate = '10/m'
    ratelimit_block = False
    ratelimit_method = 'GET'

    def get(self, request, *args, **kwargs):
        # Calculate expensive report...
```

Changed in version 0.5: The name of the mixin changed from `RateLimitMixin` to `RatelimitMixin` for consistency.

3.2.3 Helper Function

In some cases the decorator is not flexible enough. If this is an issue you use the `is_ratelimited` helper function. It's similar to the decorator.

Import:

```
from ratelimit.utils import is_ratelimited
```

`is_ratelimited`(*request*, *group=None*, *key=*, *rate=None*, *method=ALL*, *increment=False*)

Parameters

- **`request`** – *None* The HTTPRequest object.
- **`group`** – *None* A group of rate limits to count together. Defaults to the dotted name of the view.
- **`key`** – What key to use, see *Keys*.
- **`rate`** – *'5/m'* The number of requests per unit time allowed. Valid units are:
  - `s` - seconds
  - `m` - minutes
  - `h` - hours
  - `d` - days

  Also accepts callables. See *Rates*.
- **`method`** – *ALL* Which HTTP method(s) to rate-limit. May be a string, a list/tuple, or `None` for all methods.
- **`increment`** – *False* Whether to increment the count or just check.

3.2.4 Exceptions

class `ratelimit.exceptions.`**`Ratelimited`**

If a request is ratelimited and `block` is set to `True`, Ratelimit will raise `ratelimit.exceptions.Ratelimited`.

This is a subclass of Django's `PermissionDenied` exception, so if you don't need any special handling beyond the built-in 403 processing, you don't have to do anything.

If you are setting `handler403` in your root URLconf, you can catch this exception in your custom view to return a different response, for example:

```
def handler403(request, exception=None):
    if isinstance(exception, Ratelimited):
        return HttpResponse('Sorry you are blocked', status=429)
    return HttpResponseForbidden('Forbidden')
```

3.2.5 Middleware

There is optional middleware to use a custom view to handle `Ratelimited` exceptions.

To use it, add `ratelimit.middleware.RatelimitMiddleware` to your `MIDDLEWARE_CLASSES` (toward the bottom of the list) and set `RATELIMIT_VIEW` to the full path of a view you want to use.

The view specified in `RATELIMIT_VIEW` will get two arguments, the `request` object (after ratelimit processing) and the exception.

3.3 Ratelimit Keys

The `key=` argument to the decorator takes either a string or a callable.

3.3.1 Common keys

The following string values for `key=` provide shortcuts to commonly used ratelimit keys:

- `'ip'` - Use the request IP address (i.e. `request.META['REMOTE_ADDR']`)

  Note: If you are using a reverse proxy, make sure this value is correct or use an appropriate `header:` value. See the *security* notes.

- `'get:X'` - Use the value of `request.GET.get('X', '')`.
- `'post:X'` - Use the value of `request.POST.get('X', '')`.
- `'header:x-x'` - Use the value of `request.META.get('HTTP_X_X', '')`.

  Note: The value right of the colon will be translated to all-caps and any dashes will be replaced with underscores, e.g.: x-client-ip => X_CLIENT_IP.

- `'user'` - Use an appropriate value from `request.user`. Do not use with unauthenticated users.
- `'user_or_ip'` - Use an appropriate value from `request.user` if the user is authenticated, otherwise use `request.META['REMOTE_ADDR']` (see the note above about reverse proxies).

Note: Missing headers, GET, and POST values will all be treated as empty strings, and ratelimited in the same bucket.

Warning: Using user-supplied data, like data from GET and POST or headers directly from the User-Agent can allow users to trivially opt out of ratelimiting. See the note in *the security chapter*.

3.3.2 String values

Other string values not from the list above will be treated as the dotted Python path to a callable. See *below* for more on callables.

3.3.3 Callable values

New in version 0.3.

Changed in version 0.5: Added support for python path to callables.

Changed in version 0.6: Callable was mistakenly only passed the `request`, now also gets `group` as documented.

If the value of `key=` is a callable, or the path to a callable, that callable will be called with two arguments, the *group* and the `request` object. It should return a bytestring or unicode object, e.g.:

```
def my_key(group, request):
    return request.META['REMOTE_ADDR'] + request.user.username
```

3.4 Rates

3.4.1 Simple rates

Simple rates are of the form `X/u` where `X` is a number of requests and `u` is a unit from this list:

- `s` - second
- `m` - minute
- `h` - hour
- `d` - day

(For example, you can read `5/s` as "five per second.")

You may also specify a number of units, i.e.: `X/Yu` where `Y` is a number of units. If `u` is omitted, it is presumed to be seconds. So, the following are equivalent, and all mean "one hundred requests per five minutes":

- `100/5m`
- `100/300s`
- `100/300`

3.4.2 Callables

New in version 0.5.

Rates can also be callables (or dotted paths to callables, which are assumed if there is no `/` in the value).

Callables receive two values, the *group* and the `request` object. They should return a simple rate string, or a tuple of integers `(count, seconds)`. For example:

```
def my_rate(group, request):
    if request.user.is_authenticated:
        return '1000/m'
    return '100/m'
```

Or equivalently:

```
def my_rate_tuples(group, request):
    if request.user.is_authenticated:
        return (1000, 60)
    return (100, 60)
```

Callables can return `0` in the first place to disallow any requests (e.g.: `0/s`, `(0, 60)`). They can return `None` for "no ratelimit".

3.5 Security considerations

3.5.1 Client IP address

IP address is an extremely common rate limit *key*, so it is important to configure correctly, especially in the equally-common case where Django is behind a load balancer or other reverse proxy.

Django-Ratelimit is **not** the correct place to handle reverse proxies and adjust the IP address, and patches dealing with it will not be accepted. There is too much variation in the wild to handle it safely.

This is the same reason Django dropped `SetRemoteAddrFromForwardedFor` middleware in 1.1: no such "mechanism can be made reliable enough for general-purpose use" and it "may lead developers to assume that the value of `REMOTE_ADDR` is 'safe'."

Risks

Mishandling client IP data creates an IP spoofing vector that allows attackers to circumvent IP ratelimiting entirely. Consider an attacker with the real IP address 3.3.3.3 that adds the following to a request:

```
X-Forwarded-For: 1.2.3.4
```

A misconfigured web server may pass the header value along, e.g.:

```
X-Forwarded-For: 3.3.3.3, 1.2.3.4
```

Alternatively, if the web server sends a different header, like `X-Cluster-Client-IP` or `X-Real-IP`, and passes along the spoofed `X-Forwarded-For` header unchanged, a mistake in ratelimit or a misconfiguration in Django could read the spoofed header instead of the intended one.

Remediation

There are two options, configuring django-ratelimit or adding global middleware. Which makes sense depends on your setup.

Middleware

Writing a small middleware class to set `REMOTE_ADDR` to the actual client IP address is generally simple:

```
class ReverseProxy(object):
    def process_request(self, request):
        request.META['REMOTE_ADDR'] = # [...]
```

where `# [...]` depends on your environment. This middleware should be close to the top of the list:

```
MIDDLEWARE_CLASSES = (
    'path.to.ReverseProxy',
    # ...
)
```

Then the `@ratelimit` decorator can be used with the `ip` key:

```
@ratelimit(key='ip', rate='10/s')
```

Ratelimit keys

Alternatively, if the client IP address is in a simple header (i.e. a header like `X-Real-IP` that *only* contains the client IP, unlike `X-Forwarded-For` which may contain intermediate proxies) you can use a `header:` key:

```
@ratelimit(key='header:x-real-ip', rate='10/s')
```

3.5.2 Brute force attacks

One of the key uses of ratelimiting is preventing brute force or dictionary attacks against login forms. These attacks generally take one of a few forms:

- One IP address trying one username with many passwords.
- Many IP addresses trying one username with many passwords.
- One IP address trying many usernames with a few common passwords.
- Many IP addresses trying many usernames with one or a few common passwords.

Note: Unfortunately, the fourth case of many IPs trying many usernames can be difficult to distinguish from regular user behavior and requires additional signals, such as a consistent user agent or a common network prefix.

Protecting against the single IP address cases is easy:

```
@ratelimit(key='ip')
def login_view(request):
    pass
```

Also limiting by username provides better protection:

```
@ratelimit(key='ip')
@ratelimit(key='post:username')
def login_view(request):
    pass
```

Using passwords as key values is not recommended. Key values are never stored in a raw form, even as cache keys, but they are constructed with a fast hash function.

Denial of Service

However, limiting based on field values may open a denial of service vector against your users, preventing them from logging in.

For pages like login forms, consider implenting a soft blocking mechanism, such as requiring a captcha, rather than a hard block with a `PermissionDenied` error.

Network Address Translation

Depending on your profile of your users, you may have many users behind NAT (e.g. users in schools or in corporate networks). It is reasonable to set a higher limit on a per-IP limit than on a username or password limit.

3.5.3 User-supplied Data

Using data from GET (`key='get:X'`) POST (`key='post:X'`) or headers (`key='header:x-x'`) that are provided directly by the browser or other client presents a risk. Unless there is some requirement of the attack that requires the client *not* change the value (for example, attempting to brute force a password requires that the username be consistent) clients can trivially change these values on every request.

Headers that are provided by web servers or reverse proxies should be independently audited to ensure they cannot be affected by clients.

The `User-Agent` header is especially dangerous, since bad actors can change it on every request, and many good actors may share the same value.

3.6 Upgrade Notes

See also the *CHANGELOG <../CHANGELOG>*.

3.6.1 From <=0.4 to 0.5

Quickly:

- Rate limits are now counted against fixed, instead of sliding, windows.
- Rate limits are no longer shared between methods by default.
- Change `ip=True` to `key='ip'`.
- Drop `ip=False`.
- A key must always be specified. If using without an explicit key, add `key='ip'`.
- Change `fields='foo'` to `post:foo` or `get:foo`.
- Change `keys=callable` to `key=callable`.
- Change `skip_if` to a callable `rate=<callable>` method (see *Rates*.
- Change `RateLimitMixin` to `RatelimitMixin` (note the lowercase `l`).
- Change `ratelimit_ip=True` to `ratelimit_key='ip'`.
- Change `ratelimit_fields='foo'` to `post:foo` or `get:foo`.
- Change `ratelimit_keys=callable` to `ratelimit_key=callable`.

Fixed windows

Before 0.5, rates were counted against a *sliding* window, so if the rate limit was `1/m`, and three requests came in:

```
1.2.3.4 [09/Sep/2014:12:25:03] ...
1.2.3.4 [09/Sep/2014:12:25:53] ... <RATE LIMITED>
1.2.3.4 [09/Sep/2014:12:25:59] ... <RATE LIMITED>
```

Even though the third request came nearly two minutes after the first request, the second request moved the window. Good actors could easily get caught in this, even trying to implement reasonable back-offs.

Starting in 0.5, windows are *fixed*, and staggered throughout a given period based on the key value, so the third request, above would not be rate limited (it's possible neither would the second one).

Warning: That means that given a rate of X/u, you may see up to 2 * X requests in a short period of time. Make sure to set X accordingly if this is an issue.

This change still limits bad actors while being far kinder to good actors.

Staggering windows

To avoid a situation where all limits expire at the top of the hour, windows are automatically staggered throughout their period based on the key value. So if, for example, two IP addresses are hitting hourly limits, instead of both of those limits expiring at 06:00:00, one might expire at 06:13:41 (and subsequently at 07:13:41, etc) and the other might expire at 06:48:13 (and 07:48:13, etc).

Sharing rate limits

Before 0.5, rate limits were shared between methods based only on their keys. This was very confusing and unintuitive, and is far from the least-surprising thing. For example, given these three views:

```python
@ratelimit(ip=True, field='username')
def both(request):
    pass

@ratelimit(ip=False, field='username')
def field_only(request):
    pass

@ratelimit(ip=True)
def ip_only(request):
    pass
```

The pair `both` and `field_only` shares one rate limit key based on all requests to either (and any other views) containing the same `username` key (in `GET` or `POST`), regardless of IP address.

The pair `both` and `ip_only` shares one rate limit key based on the client IP address, along with all other views.

Thus, it's extremely difficult to determine exactly why a request is getting rate limited.

In 0.5, methods never share rate limits by default. Instead, limits are based on a combination of the *group*, rate, key value, and HTTP methods *to which the decorator applies* (i.e. **not** the method of the request). This better supports common use cases and stacking decorators, and still allows decorators to be shared.

For example, this implements an hourly rate limit with a per-minute burst rate limit:

```python
@ratelimit(key='ip', rate='100/m')
@ratelimit(key='ip', rate='1000/h')
def myview(request):
    pass
```

However, this view is limited *separately* from another view with the same keys and rates:

```python
@ratelimit(key='ip', rate='100/m')
@ratelimit(key='ip', rate='1000/h')
def anotherview(request):
    pass
```

To cause the views to share a limit, explicitly set the `group` argument:

```
@ratelimit(group='lists', key='user', rate='100/h')
def user_list(request):
    pass

@ratelimit(group='lists', key='user', rate='100/h')
def group_list(request):
    pass
```

You can also stack multiple decorators with different sets of applicable methods:

```
@ratelimit(key='ip', method='GET', rate='1000/h')
@ratelimit(key='ip', method='POST', rate='100/h')
def maybe_expensive(request):
    pass
```

This allows a total of 1,100 requests to this view in one hour, while this would only allow 1000, but still only 100 POSTs:

```
@ratelimit(key='ip', method=['GET', 'POST'], rate='1000/h')
@ratelimit(key='ip', method='POST', rate='100/h')
def maybe_expensive(request):
    pass
```

And these two decorators would not share a rate limit:

```
@ratelimit(key='ip', method=['GET', 'POST'], rate='100/h')
def foo(request):
    pass

@ratelimit(key='ip', method='GET', rate='100/h')
def bar(request):
    pass
```

But these two do share a rate limit:

```
@ratelimit(group='a', key='ip', method=['GET', 'POST'], rate='1/s')
def foo(request):
    pass

@ratelimit(group='a', key='ip', method=['POST', 'GET'], rate='1/s')
def bar(request):
    pass
```

Using multiple decorators

A single `@ratelimit` decorator used to be able to ratelimit against multiple keys, e.g., before 0.5:

```
@ratelimit(ip=True, field='username', keys=mykeysfunc)
def someview(request):
    # ...
```

To simplify both the internals and the question of what limits apply, each decorator now tracks exactly one rate, but decorators can be more reliably stacked (c.f. some examples in the section above).

The pre-0.5 example above would need to become four decorators:

```
@ratelimit(key='ip')
@ratelimit(key='post:username')
@ratelimit(key='get:username')
@ratelimit(key=mykeysfunc)
def someview(request):
    # ...
```

As documented above, however, this allows powerful new uses, like burst limits and distinct GET/POST limits.

3.7 Contributing

3.7.1 Set Up

Create a virtualenv and install Django with pip:

```
$ pip install Django
```

3.7.2 Running the Tests

Running the tests is as easy as:

```
$ ./run.sh test
```

You may also run the test on multiple versions of Django using tox.

- First install tox:

  ```
  $ pip install tox
  ```

- Then run the tests with tox:

  ```
  $ tox
  ```

3.7.3 Code Standards

I ask two things for pull requests.

- The flake8 tool must not report any violations.
- All tests, including new tests where appropriate, must pass.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Index