# Django Ratelimit Documentation

***Release 4.0.0***

**James Socol**

**Dec 04, 2022**

# Contents

CHAPTER 1

# Project

**Django Ratelimit** is a ratelimiting decorator for Django views, storing rate data in the configured Django cache backend.


**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

**Warning:** *django_ratelimit* requires a Django cache backend that supports atomic increment operations. The Memcached and Redis backends do, but the database backend does not. More information can be found in *Installation*

Install:

```
pip install django-ratelimit
```

Use as a decorator in `views.py`:

```
from django_ratelimit.decorators import ratelimit

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

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

Before activating django-ratelimit, you should ensure that your cache backend is setup to be both persistent and work across multiple deployment worker instances (for instance UWSGI workers). Read more in the Django docs on caching.

CHAPTER 3

# Contents

## 3.1 Installation

### 3.1.1 Create or use a compatible cache

`django_ratelimit` requires a cache backend that

1. Is shared across any worker threads, processes, and application servers. Cache backends that use sharding can be used to help scale this.
2. Implements *atomic increment*.

Redis and Memcached backends have these features and are officially supported. Backends like local memory and filesystem are not shared across processes or servers. Notably, the database backend does **not** support atomic increments.

If you do not have a compatible cache backend, you'll need to set one up, which is out of scope of this document, and then add it to the `CACHES` dictionary in settings.

> **Warning:** Without atomic increment operations, `django_ratelimit` will appear to work, but there is a race condition between reading and writing usage count data that can result in undercounting usage and permitting more traffic than intended.

### 3.1.2 Configuration

`django_ratelimit` has reasonable defaults, and if your `default` cache is compatible, and your application is not behind a reverse proxy, you can skip this section.

For a complete list of configuration options, see *Settings*.

#### Cache Settings

If you have added an additional `CACHES` entry for ratelimiting, you'll need to tell `django_ratelimit` to use this via the `RATELIMIT_USE_CACHE` setting:

```
# your_apps_settings.py
CACHES = {
    'default': {},
    'cache-for-ratelimiting': {},
}

RATELIMIT_USE_CACHE = 'cache-for-ratelimiting'
```

#### Reverse Proxies and Client IP Address

`django_ratelimit` reads client IP address from `request.META['REMOTE_ADDR']`. If your application is running behind a reverse proxy such as nginx or HAProxy, you will need to take steps to ensure you have access to the correct client IP address, rather than the address of the proxy.

There are security risks for libraries to *assume* how your network is set up, and so `django_ratelimit` does not provide any built-in tools to address this. However, the *Security chapter* does provide suggestions on how to approach this.

### 3.1.3 Enforcing Ratelimits

The most common way to enforce ratelimits is via the `ratelimit` *decorator*:

```
from django_ratelimit.decorators import ratelimit

@ratelimit(key='user_or_ip', rate='10/m')
def myview(request):
    # limited to 10 req/minute for a given user or client IP

# or on class methods
class MyView(View):
    @method_decorator(ratelimit(key='user_or_ip', rate='1/s'))
    def get(self, request):
        # limited to 1 req/second
```

## 3.2 Settings

### 3.2.1 RATELIMIT_CACHE_PREFIX

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

### 3.2.2 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()
```

### 3.2.3 RATELIMIT_USE_CACHE

**Warning:** *django_ratelimit* requires a Django cache backend that supports atomic increment operations. The Memcached and Redis backends do, but the database backend does not.

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

### 3.2.4 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`.

### 3.2.5 RATELIMIT_FAIL_OPEN

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

### 3.2.6 RATELIMIT_IP_META_KEY

Set the source of the client IP address in the request.META object. Defaults to `None`.

There are several potential values:

**None** Use `request.META['REMOTE_ADDR']` as the source of the client IP address.

**A callable object** If set to a callable, the callable will be passed the full `request` object. The callable must return the client IP address. For example: `RATELIMIT_IP_META_KEY = lambda r: r.META['HTTP_X_CLIENT_IP']`

**A dotted path to a callable** Any string containing a `.` will be treated as a dotted path to a callable, which will be imported and called on the `request` object, as above.

**Any other string** Any other string will be treated as a key for the `request.META` object, e.g. `RATELIMIT_IP_META_KEY = 'HTTP_X_REAL_IP'`

### 3.2.7 RATELIMIT_IPV4_MASK

IPv4 mask for IP-based rate limit. Defaults to `32` (which is no masking)

### 3.2.8 RATELIMIT_IPV6_MASK

IPv6 mask for IP-based rate limit. Defaults to `64` (which mask the last 64 bits). Typical end site IPv6 assignment are from /48 to /64.

### 3.2.9 RATELIMIT_EXCEPTION_CLASS

A custom exception class, or a dotted path to a custom exception class, that will be raised by ratelimit when a limit is exceeded and `block=True`.

## 3.3 Using Django Ratelimit

### 3.3.1 Use as a decorator

Changed in version 4.0.

Import:

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

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

**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*. A rate of `0/s` disallows all requests. A rate of `None` means "no limit" and will allow all requests.
- **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** – *True* 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 or the `is_ratelimited` helper, as well as on the root `ratelimit` module:

```
from django_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', block=False)
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'], block=False)
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()

get_rate = lambda g, r: None if r.user.is_authenticated else '100/h'
@ratelimit(key='ip', rate=get_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()
```

(continues on next page)

(continued from previous page)

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

@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 g, 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.

Changed in version 3.0.

To use the `@ratelimit` decorator with class-based views, use the Django `@method_decorator`:

```
from django.utils.decorators import method_decorator
from django.views.generic import View

class MyView(View):
    @method_decorator(ratelimit(key='ip', rate='1/m', method='GET'))
    def get(self, request):
        pass

@method_decorator(ratelimit(key='ip', rate='1/m', method='GET'), name='get')
class MyOtherView(View):
    def get(self, request):
        pass
```

It is also possible to wrap a whole view later, e.g.:

```
from django.urls import path

from myapp.views import MyView

from django_ratelimit.decorators import ratelimit

urlpatterns = [
    path('/', ratelimit(key='ip', method='GET', rate='1/m')(MyView.as_view())),
]
```

**Warning:** Make sure the `method` argument matches the method decorated.

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

### 3.3.2 Core Methods

New in version 3.0.

In some cases the decorator is not flexible enough to, e.g., conditionally apply rate limits. In these cases, you can access the core functionality in `ratelimit.core`. The two major methods are `get_usage` and `is_ratelimited`.

```
from django_ratelimit.core import get_usage, is_ratelimited
```

**get_usage**(*request*, *group=None*, *fn=None*, *key=None*, *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.
- **fn** – *None* A view function which can be used to calculate the group as if it was decorated by *@ratelimit*.
- **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.

**Returns dict or None** Either returns None, indicating that ratelimiting was not active for this request (for some reason) or returns a dict including the current count, limit, time left in the window, and whether this request should be limited.

**is_ratelimited**(*request*, *group=None*, *fn=None*, *key=None*, *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.
- **fn** – *None* A view function which can be used to calculate the group as if it was decorated by *@ratelimit*.

- **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.

**Returns bool** Whether this request should be limited or not.

`is_ratelimited` is a thin wrapper around `get_usage` that is maintained for compatibility. It provides strictly less information.

**Warning:** `get_usage` and `is_ratelimited` require either `group=` or `fn=` to be passed, or they cannot determine the rate limiting state and will throw.

### 3.3.3 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')
```

### 3.3.4 Middleware

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

To use it, add `ratelimit.middleware.RatelimitMiddleware` to your `MIDDLEWARE` (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.

## 3.4 Ratelimit Keys

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

### 3.4.1 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*.

### 3.4.2 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.

### 3.4.3 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
```

## 3.5 Rates

### 3.5.1 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.")

**Note:** Setting a rate of 0 per any unit of time will disallow requests, e.g. `0/s` will prevent any requests to the endpoint.

Rates may also be set to `None`, which indicates "there is no limit." Usage will not be tracked.

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`

### 3.5.2 Callables

New in version 0.5.

Rates can also be callables (or dotted paths to callables, which are assumed if there is a `.` 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".

## 3.6 Security considerations

### 3.6.1 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:

```
def reverse_proxy(get_response):
    def process_request(request):
        request.META['REMOTE_ADDR'] = # [...]
        return get_response(request)
    return process_request
```

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

```
MIDDLEWARE = (
    'path.to.reverse_proxy',
    # ...
)
```

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')
```

### 3.6.2 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 implementing 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.

### 3.6.3 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.

## 3.7 Upgrade Notes

See also the CHANGELOG.

### 3.7.1 From 3.x to 4.0

Quickly:

- Rename imports from `from ratelimit` to `from django_ratelimit`
- Check all uses of the `@ratelimit` decorator. If the `block` argument is not set, add `block=False` to retain the current behavior. `block=True` may optionally be removed.
- Django versions below 3.2 and Python versions below 3.7 are no longer supported.

#### Package name changed

To disambiguate with other ratelimit packages on PyPI and resolve distro packaging issues, the package name has been changed from `ratelimit` to `django_ratelimit`. See issue 214 for more information on this change.

When upgrading, import paths need to change to use the new package name.

Old:

```
from ratelimit.decorators import ratelimit
from ratelimit import ALL, UNSAFE
```

New:

```
from django_ratelimit.decorators import ratelimit
from django_ratelimit import ALL, UNSAFE
```

#### Default decorator behavior changed

In previous versions, the `@ratelimit` decorator did not block traffic that exceeded the rate limits by default. This has been reversed, and now the default behavior *is* to block requests once a rate limit has been exceeded. The old behavior of annotating the request object with a `.limited` property can be restored by explicitly setting `block=False` on the decorator.

Historically, the first use cases Django Ratelimit was built to support were HTML views like login and password-reset pages, rather than APIs. In these cases, rate limiting is often done based on user input like the username or email address. Instead of blocking requests, which could lead to a denial-of-service (DOS) attack against particular users, it is common to trigger some additional security measures to prevent brute-force attacks, like a CAPTCHA, temporary account lock, or even notify those users via email.

However, it has become obvious that the majority of views using the `@ratelimit` decorator tend to be either specific pages or API endpoints that do not present a DOS attack vector against other users, and that a more intuitive default behavior is to block requests that exceed the limits. Since there tend to only be a couple of pages or routes for uses like authentication, it makes more sense to opt those uses *out* of blocking, than opt all the others *in*.

### 3.7.2 From 2.0 to 3.0

Quickly:

- Ratelimit now supports Django >=1.11 and Python >=3.4.
- `@ratelimit` no longer works directly on class methods, add `@method_decorator`.
- `RatelimitMixin` is gone, migrate to `@method_decorator`.
- Moved `is_ratelimted` method from `ratelimit.utils` to `ratelimit.core`.

#### `@ratelimit` decorator on class methods

In 3.0, the decorator has been simplified and must now be used with Django's excellent `@method_decorator` utility. Migrating should be relatively straight-forward:

```
from django.views.generic import View
from ratelimit.decorators import ratelimit

class MyView(View):
    @ratelimit(key='ip', rate='1/m', method='GET')
    def get(self, request):
        pass
```

changes to

```
from django.utils.decorators import method_decorator
from django.views.generic import View
from ratelimit.decorators import ratelimit

class MyView(View):
    @method_decorator(ratelimit(key='ip', rate='1/m', method='GET'))
    def get(self, request):
        pass
```

#### RatelimitMixin

`RatelimitMixin` is a vestige of an older version of Ratelimit that did not support multiple rates per method. As such, it is significantly less powerful than the current `@ratelimit` decorator. To migrate to the decorator, use the `@method_decorator` from Django:

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

    def get(self, request):
        pass
```

becomes

```
class MyView(View):
    @method_decorator(ratelimit(key='ip', rate='10/m', method='GET'))
    def get(self, request):
        pass
```

The major benefit is that it is now possible to apply multiple limits to the same method, as with **:ref:‘the decorator <usage-decorator>‘_**.

### 3.7.3 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:

```
@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:

```
@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:

```
@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.

## 3.8 Contributing

### 3.8.1 Set Up

Create a virtualenv and install Django with pip:

```
$ pip install Django
```

### 3.8.2 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
  ```

### 3.8.3 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.

## 3.9 Cookbook

This section includes suggestions for common patterns that don't make sense to include in the main library because they depend on too many specific about consuming applications. These solutions may be close to copy-pastable, but in generally they are more directional and are provided under the same license as all other code in this repository.

### 3.9.1 Recipes

#### Sending `429 Too Many Requests`

RFC 6585 introduced a status code specific to rate-limiting situations: HTTP 429 Too Many Requests. Here's one way to send this status with Django Ratelimit.

#### Create a custom error view

First, create a view that returns the correct type of response (e.g. content-type, shape, information, etc) for your application. For example, a JSON API may return something like `{"error": "ratelimited"}`, while other applications may return XML, HTML, etc, as needed. Or you may need to decide based on the type of request. Set the status code of the response to 429.

```
# myapp/views.py
def ratelimited_error(request, exception):
    # e.g. to return HTML
    return render(request, 'ratelimited.html', status=429)

def ratelimited_error(request, exception):
    # or other types:
    return JsonResponse({'error': 'ratelimited'}, status=429)
```

In your app's settings, install the `RatelimitMiddleware` *middleware* toward the bottom of the list. You must define `RATELIMIT_VIEW` as a dotted-path to your error view:

```
MIDDLEWARE = (
    # ... toward the bottom ...
    'ratelimit.middleware.RatelimitMiddleware',
    # ...
)

RATELIMIT_VIEW = 'myapp.views.ratelimited_error'
```

That's it! If you already have *the decorator* installed, you're good to go. Otherwise, you'll need to install it in order to trigger the error view.

#### Check the exception type in `handler403`

Alternatively, if you already have a `handler403` view defined, you can check the exception type and return a specific status code:

```
from django_ratelimit.exceptions import Ratelimited

def my_403_handler(request, exception):
    if isinstance(exception, Ratelimited):
```

(continues on next page)

(continued from previous page)

```
        return render(request, '429.html', status=429)
    return render(request, '403.html', status=403)
```

## Context

### Why doesn't Django Ratelimit handle this itself?

There are a couple of main reasons. The first is that Django has no built-in concept of a ratelimit exception, but it does have `PermissionDenied`. When a view throws a `PermissionDenied` exception, Django has built-in facilities for handling it as a client error (it returns an HTTP 403) instead of a server error (i.e. a 5xx status code).

The `Ratelimited` exception extends `PermissionDenied` so that, if nothing else, there should already be a way to make sure the application is sending a 4xx status code—even if it's not the most-correct status code available. `Ratelimited` should not be treated as a server error because the server is working correctly. (NB: That also means that the typical "error"-level logging is not invoked.) There is no way to convince the built-in handler to send any status besides 403.

Furthermore, it's impossible for Django Ratelimit to provide a default view that does a better job guessing at the appropriate response type than Django's built-in `PermissionDenied` view already does. We could include a default `429.html` template with as little information as Django's built-in `403.html`, but it would only be slightly more correct.

The correct response for your users will depend on your application. This means creating the right content-type (e.g. JSON, XML, HTML, etc) and content (whether it's an API error response or a human-readable one). Django Ratelimit can't guess that, so it's up to you to define.

Finally, a small historical note. Django Ratelimit actually predates RFC 6585 by about a year. At the time, 403 was as common as any status for ratelimit situations. Others were creating custom statuses, like Twitter's `420 Enhance Your Calm`.

## Per-User Ratelimits

One common business strategy includes adjusting rate limits for different types of users, or even different individual users for enterprise sales. With *callable rates* it is possible to implement per-user or per-group rate limits. Here is one example of how to implement per-user rates.

### A `Ratelimit` model

This example leverages the database to store per-user rate limits. Keep in mind the additional load this may place on your application's database—which may very well be the resource you intend to protect. Consider caching these types of queries.

```
# myapp/models.py
class Ratelimit(models.Model):
    group = models.CharField(db_index=True)
    user = models.ForeignKey(null=True)  # One option for "default"
    rate = models.CharField()

    @classmethod
    def get(cls, group, user=None):
        # use cache if possible
        try:
```

(continues on next page)

(continued from previous page)

```
            return cls.objects.get(group=group, user=user)
        except cls.DoesNotExist:
            return cls.objects.get(group=group, user=None)

# myapp/ratelimits.py
from myapp.models import Ratelimit
def per_user(group, request):
    if request.user.is_authenticated:
        return Ratelimit.get(group, request.user)
    return Ratelimit.get(group)

# myapp/views.py
@login_required
@ratelimit(group='search', key='user',
           rate='myapp.ratelimits.per_user')
def search_view(request):
    # ...
```

It would be important to consider how to handle defaults, cases where the rate is not defined in the database, or the group is new, etc. It would also be important to consider the performance impact of executing such a query as part of the rate limiting process and consider how to store this data.

CHAPTER 4

# Indices and tables

- genindex
- modindex
- search

# Index