source: throttling.py

Throttling

> HTTP/1.1 420 Enhance Your Calm
>
> Twitter API rate limiting response

Throttling is similar to permissions, in that it determines if a request should be authorized. Throttles indicate a temporary state, and are used to control the rate of requests that clients can make to an API.

As with permissions, multiple throttles may be used. Your API might have a restrictive throttle for unauthenticated requests, and a less restrictive throttle for authenticated requests.

Another scenario where you might want to use multiple throttles would be if you need to impose different constraints on different parts of the API, due to some services being particularly resource-intensive.

Multiple throttles can also be used if you want to impose both burst throttling rates, and sustained throttling rates. For example, you might want to limit a user to a maximum of 60 requests per minute, and 1000 requests per day.

Throttles do not necessarily only refer to rate-limiting requests. For example a storage service might also need to throttle against bandwidth, and a paid data service might want to throttle against a certain number of a records being accessed.

How throttling is determined

As with permissions and authentication, throttling in REST framework is always defined as a list of classes.

Before running the main body of the view each throttle in the list is checked. If any throttle check fails an `exceptions.Throttled` exception will be raised, and the main body of the view will not run.

Setting the throttling policy

The default throttling policy may be set globally, using the `DEFAULT_THROTTLE_CLASSES` and `DEFAULT_THROTTLE_RATES` settings. For example.

```
REST_FRAMEWORK = {
    'DEFAULT_THROTTLE_CLASSES': (
        'rest_framework.throttling.AnonRateThrottle',
        'rest_framework.throttling.UserRateThrottle'
    ),
    'DEFAULT_THROTTLE_RATES': {
        'anon': '100/day',
        'user': '1000/day'
    }
}
```

The rate descriptions used in `DEFAULT_THROTTLE_RATES` may include `second`, `minute`, `hour` or `day` as the throttle period.

You can also set the throttling policy on a per-view or per-viewset basis, using the `APIView` class based views.

```
from rest_framework.response import Response
from rest_framework.throttling import UserRateThrottle
from rest_framework.views import APIView

class ExampleView(APIView):
    throttle_classes = (UserRateThrottle,)

    def get(self, request, format=None):
        content = {
            'status': 'request was permitted'
        }
        return Response(content)
```

Or, if you're using the `@api_view` decorator with function based views.

```
@api_view(['GET'])
@throttle_classes([UserRateThrottle])
def example_view(request, format=None):
    content = {
        'status': 'request was permitted'
    }
    return Response(content)
```

How clients are identified

The `X-Forwarded-For` and `Remote-Addr` HTTP headers are used to uniquely identify client IP addresses for throttling. If the `X-Forwarded-For` header is present then it will be used, otherwise the value of the `Remote-Addr` header will be used.

If you need to strictly identify unique client IP addresses, you'll need to first configure the number of application proxies that the API runs behind by setting the `NUM_PROXIES` setting. This setting should be an integer of zero or more. If set to non-zero then the client IP will be identified as being the last IP address in the `X-Forwarded-For` header, once any application proxy IP addresses have first been excluded. If set to zero, then the `Remote-Addr` header will always be used as the identifying IP address.

It is important to understand that if you configure the `NUM_PROXIES` setting, then all clients behind a unique NAT'd gateway will be treated as a single client.

Further context on how the `X-Forwarded-For` header works, and identifying a remote client IP can be found here.

Setting up the cache

The throttle classes provided by REST framework use Django's cache backend. You should make sure that you've set appropriate cache settings. The default value of `LocMemCache` backend should be okay for simple setups. See Django's cache documentation for more details.

If you need to use a cache other than `'default'`, you can do so by creating a custom throttle class and setting the `cache` attribute. For example:

```
class CustomAnonRateThrottle(AnonRateThrottle):
    cache = get_cache('alternate')
```

You'll need to remember to also set your custom throttle class in the `'DEFAULT_THROTTLE_CLASSES'` settings key, or using the `throttle_classes` view attribute.

API Reference

AnonRateThrottle

The `AnonRateThrottle` will only ever throttle unauthenticated users. The IP address of the incoming request is used to generate a unique key to throttle against.

The allowed request rate is determined from one of the following (in order of preference).

- The `rate` property on the class, which may be provided by overriding `AnonRateThrottle` and setting the property.
- The `DEFAULT_THROTTLE_RATES['anon']` setting.

`AnonRateThrottle` is suitable if you want to restrict the rate of requests from unknown sources.

UserRateThrottle

The `UserRateThrottle` will throttle users to a given rate of requests across the API. The user id is used to generate a unique key to throttle against. Unauthenticated requests will fall back to using the IP address of the incoming request to generate a unique key to throttle against.

The allowed request rate is determined from one of the following (in order of preference).

- The `rate` property on the class, which may be provided by overriding `UserRateThrottle` and setting the property.
- The `DEFAULT_THROTTLE_RATES['user']` setting.

An API may have multiple `UserRateThrottles` in place at the same time. To do so, override `UserRateThrottle` and set a unique "scope" for each class.

For example, multiple user throttle rates could be implemented by using the following classes...

```
class BurstRateThrottle(UserRateThrottle):
    scope = 'burst'

class SustainedRateThrottle(UserRateThrottle):
    scope = 'sustained'
```

...and the following settings.

```
REST_FRAMEWORK = {
    'DEFAULT_THROTTLE_CLASSES': (
        'example.throttles.BurstRateThrottle',
        'example.throttles.SustainedRateThrottle'
    ),
    'DEFAULT_THROTTLE_RATES': {
        'burst': '60/min',
        'sustained': '1000/day'
    }
}
```

`UserRateThrottle` is suitable if you want simple global rate restrictions per-user.

ScopedRateThrottle

The `ScopedRateThrottle` class can be used to restrict access to specific parts of the API. This throttle will only be applied if the view that is being accessed includes a `.throttle_scope` property. The unique throttle key will then be formed by concatenating the "scope" of the request with the unique user id or IP address.

The allowed request rate is determined by the `DEFAULT_THROTTLE_RATES` setting using a key from the request "scope".

For example, given the following views...

```
class ContactListView(APIView):
    throttle_scope = 'contacts'
    ...

class ContactDetailView(ApiView):
    throttle_scope = 'contacts'
    ...

class UploadView(APIView):
    throttle_scope = 'uploads'
    ...
```

...and the following settings.

```
REST_FRAMEWORK = {
    'DEFAULT_THROTTLE_CLASSES': (
        'rest_framework.throttling.ScopedRateThrottle',
    ),
    'DEFAULT_THROTTLE_RATES': {
        'contacts': '1000/day',
        'uploads': '20/day'
    }
}
```

User requests to either `ContactListView` or `ContactDetailView` would be restricted to a total of 1000 requests per-day. User requests to `UploadView` would be restricted to 20 requests per day.

Custom throttles

To create a custom throttle, override `BaseThrottle` and implement `.allow_request(self, request, view)`. The method should return `True` if the request should be allowed, and `False` otherwise.

Optionally you may also override the `.wait()` method. If implemented, `.wait()` should return a recommended number of seconds to wait before attempting the next request, or `None`. The `.wait()` method will only be called if `.allow_request()` has previously returned `False`.

If the `.wait()` method is implemented and the request is throttled, then a `Retry-After` header will be included in the response.

Example

The following is an example of a rate throttle, that will randomly throttle 1 in every 10 requests.

```
class RandomRateThrottle(throttles.BaseThrottle):
    def allow_request(self, request, view):
        return random.randint(1, 10) == 1
```