Cheatography

API Rate Limit Headers and 429 Handling Cheat Sheet

by SuperLede via cheatography.com/219377/cs/48597/

Pick the status code

`400` Bad Request	Malformed request: bad JSON, a missing field. Resending it unchanged gets the same answer
`401` Unauthorized	No key, unknown key, revoked key. RFC 9110 requires a `WWW-Au the nticate` field on a 401, e.g. `WWW-Au the nti cate: Bearer`
`402` Payment Required	Prepaid credits ran out. The spec still reserves 402 "for future use", so say why in the body
`403` Forbidden	The key is valid but this customer may not do this: account suspended, endpoint not on their plan
`429` Too Many Requests	Over a rate limit or a quota (RFC 6585). Send `Retry- After` when you know when a retry can work
`503` Service Unavailable	Your side can't serve the request: overloaded, or a dependency is down. `Retry- After` works here too

401, 402, 403: the customer fixes something before retrying. 429: check the error code and any `Retry- After`. 503: the problem is on your side.

What a 429 should carry

`Retry- After`	If present, wait this long before retrying. Seconds or an HTTP date (see Retry-After: two formats)
`RateLi mit -Poli cy`	Each limit: `q` = quota, `w` = window in seconds
`RateLimit`	Each limit now: `r` = available quota, `t` (optional) = how many seconds that quota applies for
Body	A stable error code, e.g. `rate_l imited`, plus `retry_ after` (see Example 429). Human-readable text is optional
Content-Type	`applic ati on/json` as in Example 429, or an RFC 9457 Problem Details body: `applic ation/ proble m+json`

What a 429 should carry (cont)

2xx responses	You can send the `RateLimit` fields on success too, so clients slow down before they run out

`RateLi mit -Policy` and `RateLimit` come from the IETF draft draft-ietf-httpapi-ratelimit-headers. The examples use draft-11. Draft-12 (November 2026) uses `a` for available quota and `w` for the window in `RateLimit`; `RateLi mit -Policy` keeps `q` and `w`. Check which draft your clients support.

Draft-06 and earlier used `RateLi mit -Limit`, `RateLi mit -Re maining` and `RateLi mit -Reset`. Non-standard `X-Rate Lim it-*` headers vary by API: `Reset` can be seconds left or a Unix timestamp.

Example 429

```
HTTP/1.1 429 Too Many Requests
Conten t-Type: applic ati on/json
Retry- After: 1
RateLi mit -Po licy: " per _se con d";q =10;w=1
RateLi mit -Po licy: " per _mi nut e";q =30 0;w=60
RateLimit: " per _se con d";r =0;t=1
RateLimit: " per _mi nut e";r =21 2;t=18
{"er ror ": " rat e_l imi ted ", " ret ry_ aft er": 1}
```

UsageRail's example, draft-11 syntax. Each field can also be sent as one line with the two limits comma-separated. The per-second bucket is empty. The per-minute bucket has 212 of 300 tokens; here `t=18` is how long it takes to refill if no more requests arrive.

Retry-After: two formats

Delay	`Retry- After: 120` Whole seconds, a non-negative integer
HTTP date	`Retry- After: Sat, 21 Nov 2026 07:00:00 GMT` IMF-fixdate, always GMT
Servers	Send seconds. A date depends on the client's clock agreeing with yours
Round up	`Math.ceil`. Rounding down sends the client back a moment early, into another 429
Clients	Parse both. For a date, wait until that time; if it's already past, wait 0
Where it's used	503 and 3xx (RFC 9110, section 10.2.3), and 429 (RFC 6585)

Sponsored by **Readable.com**
Measure your website readability!
https://readable.com

API Rate Limit Headers and 429 Handling Cheat Sheet

by SuperLede via cheatography.com/219377/cs/48597/

Two kinds of 429

	Rate limit	Quota used up
Window	e.g. a second or a minute	e.g. a billing period
`Retry- After`	e.g. 1–60 s	If supplied, seconds until reset, e.g. `86400`
Example code	`rate_l imited`	`quota_ exc eeded`
Client should	Back off and retry	Stop, tell the user, upgrade or wait

A 429 alone can mean "wait a second" or "come back next month". I work on UsageRail; its Node SDK puts the reason in an `error` field: `rate_l imited` or `quota_ exc eeded` on a 429, `insuff ici - ent _cr edits` on a 402.

Client retry rules

1. If you retry, wait at least as long as `Retry- After` says. It takes precedence over the `RateLimit` window.

2. Safe to retry and no `Retry- After`: exponential backoff with full jitter.

```
sleep = random(0,
  min(cap, base × 2^atte mpt))
```

3. e.g. base 0.5 s, cap 30 s, give up after 5 attempts.

4. Jitter matters. Clients that hit the limit together otherwise retry together and hit it again.

5. Retry 429 `rate_l imited`, 502, 503, 504 and dropped connections for idempotent requests (GET, PUT, DELETE). A POST only if the server honours an idempotency key, resent with the same key.

6. No automatic retries on 400, 401, 402, 403, 404, 422. Send it again once the cause is fixed: a refreshed token, topped-up credits, a resource that now exists.

7. On `quota_ exc eeded`, stop automatic retries. Show the reset time if the API provides one.

8. Every retry is a new request, and the API may count it against your limit or bill it.

Per-customer vs global limits

Per-customer	One bucket per customer. A noisy customer runs out of their own allowance, not everyone's
Global	One bucket for the whole API. Protects your servers, does nothing for fairness
Key on	The customer behind the API key, after authentication. Not the IP: customers behind one NAT share an IP

Per-customer vs global limits (cont)

Before auth	A coarse per-IP limit is fine for unauthenticated routes (sign-up, key creation)
Token bucket	Capacity sets the largest burst; the refill rate sets sustained traffic. For 10 requests/s, refill a token every 100 ms; a capacity of 10 allows a burst of 10
Several servers	In-memory buckets count per process. With 4 servers at 10/s each, one customer whose traffic is spread evenly gets up to 40/s. Split the limit across servers or use a shared counter
Restarts	In-memory buckets start full after a deploy: expect one extra burst

The per-customer API rate limiting docs explain the two-bucket 429 from Example 429 and why a 10/s limit split across four servers can still let about 12/s through.

When the limiter is down

Applies when the limit check is a network call (a shared counter store, a rate limit service).

Timeout	Short, e.g. 500 ms. Every checked request waits for the limiter
Fail open	Serve the request without a limit decision. It may go over the limit or go uncounted
Fail closed	Answer 503, with `Retry- After` if you can estimate recovery. Nothing unchecked is served, and the limiter's outage becomes your customers' outage
Choosing	Open for cheap reads your backend can absorb. Closed if one unchecked call can run up a bill
Don't retry the check	A check that timed out may already have been counted. If the retry is allowed too, both count
Limiter says 400 or 401	Your config is wrong (bad app key, bad parameter). Log it and answer 500; don't pass a 401 to your customer

Log every fail-open, so you know how much went through unchecked.

Sponsored by **Readable.com**
Measure your website readability!
https://readable.com

Minimal 429 in Express

```
// Per-customer token bucket. Runs after
authenticate
// has set req.cu sto merId. In memory, per
process.
const CAPACITY = 10; // burst size
const REFILL _PE R_SEC = 1; // steady rate
const buckets = new Map();
function rateLi mit (req, res, next) {
    const now = perfor man ce.n ow() / 1000;
    const bucket = bucket s.g et( req.cu sto merId)
       ?? { tokens: CAPACITY, updatedAt: now };
    buc ket.tokens = Math.m in( CAP ACITY,
bucket.tokens
       + (now - bucket.up dat edAt) * REFILL _PE -
R_SEC);
    buc ket.up datedAt = now;
    buc ket s.s et( req.cu sto merId, bucket);
   if (bucke t.t okens < 1) {
        const wait = Math.ceil(
           (1 - bucket.to kens) / REFILL _PE -
R_SEC);
        res.se t(' Ret ry- After', String (wa it));
        return res.st atu s(4 29).json(
           { error: 'rate_ lim ited', retry_ -
after: wait });
   }
    buc ket.tokens -= 1;
    next();
}
app.us e(a uth ent icate, rateLi mit);
```

Behind a test route that returns 200: send 11 requests from one customer within a second and the 11th gets 429 with `Retry- After : 1`, while a second customer still gets 200. `perfor man ce.n ow ()` is monotonic, so a wall-clock jump can't refill every bucket at once.

Sponsored by **Readable.com**
Measure your website readability!
https://readable.com