Engineer

Rate limiting

a cap on how many requests a client may send per interval

What it is

Rate limiting is a cap on how many requests one client may send to a service within a given stretch of time, enforced by the service or by something standing in front of it. Every limit answers three questions: who counts as the client, how the requests are counted, and what happens to the request that goes over. The HTTP status code built for the last of these, 429 Too Many Requests, was defined in RFC 6585 in April 2012, and that RFC deliberately leaves the first two open. A server may count per resource, across the whole server or across a group of servers, and may recognise the client by its authentication credentials or by a stateful cookie.

In practice the key is one of a handful of things: the client's IP address, the logged-in account, an API key, or one of those combined with the path, so that a login form and a product page get different limits. The answer to a request over the limit comes in three kinds. The server can refuse it with a 429, preferably with a `Retry-After` header that says how long to wait, given either as a number of seconds or as an HTTP date. It can hold the request back and process it a moment later, which smooths a burst instead of rejecting it. Or, when the volume itself is the attack, it can drop the connection without answering. RFC 6585 suggests exactly that, because writing a polite 429 for every request in a flood costs resources too.

How the counting works decides how a limit behaves at its edges. A fixed window counts requests per calendar minute or hour and starts again from zero at the boundary. It is cheap and easy to explain, and it lets a client spend its whole allowance in the last second of one window and again in the first second of the next: with a limit of 100 per minute, that is 200 requests in two seconds. A sliding window always looks back over the same span from the present moment, so there is no boundary to exploit; HAProxy's stick tables keep counters of this kind, and one example in its manual counts the connection rate over a 30-second

sliding window. The leaky bucket, which nginx uses, lets requests through at a fixed average rate and keeps a small queue for those that arrive faster, refusing whatever would overflow it. A token bucket works the other way round: tokens accumulate at a steady rate up to a ceiling, each request spends one, and a client that has been quiet for a while can spend its savings in a burst.

Use the concept in a decision

Put “What is rate limiting?” to work in a real decision. We can help relate the concept to the system you are building or buying. Accounts, passwords and permissions arranged so people hold exactly the access their job needs, and leavers lose it that day.

Explore our services Get in touch →

Test the assumptions behind the term

Use the explanation and linked work to frame questions for your project. We can examine the assumptions with you before a tool is chosen. Wrote 1,000 RBAC rules to secure Linux servers — 1,000 access rules written to close a security gap

Wrote 1,000 RBAC rules to secure Linux servers Get in touch →

Why it matters

A public endpoint has no say in who calls it or how often. A scraper copying a catalogue, a script stuck in a retry loop, an attacker trying leaked passwords against the login form and a partner whose nightly job was written in a hurry all look the same to the server: more requests than it was sized for, from a few sources. Without a limit, the busiest caller decides how much capacity everybody else gets. With one, a single client can only use its share, and the rest of the service carries on at normal speed while that client waits.

Some requests are far more expensive than others. A product page might be served from a cache in a millisecond, while a search, an export or anything that calls a paid external API costs real CPU time or real money on every call. Those are the paths an abuser targets, precisely because they cost the caller nothing. A single global limit set low enough to protect the export would strangle ordinary browsing, so useful rate limiting is set per path, with each limit chosen for what that path costs and how it is legitimately used.

A limit is also part of an API’s contract. Clients that know the rules can pace themselves and never hit them, which is cheaper for both sides than discovering the limit through refusals. HTTP has no standard header for it. Many APIs send `X-RateLimit-Limit`, `X-RateLimit-Remaining` and `X-RateLimit-Reset`, but the same name carries different values from one implementation to the next, as an IETF draft that sets out to standardise the idea records. That draft defines two fields,

`RateLimit-Policy` for the quota and its window and `RateLimit` for what remains; its eleventh version is dated 23 May 2026 and, like every Internet-Draft, is a work in progress rather than a standard.

What it looks like in practice

In nginx, a limit takes two directives: one in the `http` block that sets up the counter, and one where it applies:

```
limit_req_zone $binary_remote_addr zone=perip:10m rate=1r/s;
limit_req_status 429;

server {
    location /search/ {
        limit_req zone=perip burst=5 nodelay;
        proxy_pass http://app;
    }
}
```

The key is the client's address in binary form, 4 bytes long for an IPv4 address and 16 for an IPv6 one, and the counters live in a 10-megabyte shared memory zone called `perip`. The documentation puts the state for one key at 128 bytes on a 64-bit platform, so a megabyte holds about 8 thousand addresses and this zone about 80 thousand; when it fills up, nginx evicts the least recently used one. `rate=1r/s` allows one request per second on average. `burst=5` lets up to five requests above that rate wait their turn instead of being refused, and `nodelay` serves them at once instead of spacing them out at the configured rate. Anything beyond the burst is rejected. nginx would answer that rejection with 503 by default, which is why `limit_req_status` sets 429. Rates below one per second are written per minute, so `30r/m` is one request every two seconds. Before a new limit goes live, `limit_req_dry_run on` keeps counting as usual and marks would-be refusals as `REJECTED_DRY_RUN` in the `$limit_req_status` variable, without refusing anything.

On the receiving end, a client that hits the limit should read the answer rather than retry at once. A 429 from an API that follows the IETF draft might look like this:

```
HTTP/1.1 429 Too Many Requests
Content-Type: application/problem+json
Retry-After: 20
```

```
RateLimit-Policy: "default";q=100;w=60
RateLimit: "default";r=0;t=20
```

The policy allows 100 requests per 60-second window, none are left, and the quota comes back in 20 seconds. Where both are present, the draft says `Retry-After` wins. A well-behaved client sleeps for at least that long, adds a small random delay, and if there is no `Retry-After` at all, falls back to exponential backoff rather than a fixed interval.

The 22 rate limiters built to fail closed for a public directory of companies and professionals, from 2025, follow the per-path rule: each was built for the path it protects, because a login attempt, a search and a bulk export become abuse at very different rates. Their state lives in Redis, so a limit holds across every instance instead of being kept per process. The decisive choice was what happens when Redis is down: the limiters refuse traffic rather than wave it through. The page is frank about the price, that a Redis problem becomes one users can see, and about why it was accepted: the alternative was a Redis problem turning into a billing one. Cloudflare Turnstile, an origin lock and 351 lines of idempotency middleware, so that a retried write is not applied twice, sit around them.

Where it goes wrong

Each server keeps its own count. A limit held in process memory multiplies with every copy of the application: four instances behind a load balancer let a client through at four times the intended rate, and every restart resets the counters. nginx’s zone is shared memory on one machine, and keeping zones in step across servers is part of its commercial subscription; HAProxy’s peers protocol copies stick-table entries between instances but, its manual notes, without adding them up. Keep the counters in a store every instance consults, as the directory above does with Redis, or enforce the limit at the single point all traffic passes through.

Everyone arrives from the same address. Behind a reverse proxy, a CDN or a load balancer, the application sees one client address for all users unless the original one is passed along, and a per-IP limit then throttles everybody at once. A whole office or mobile network behind NAT has the same effect in a milder form. Key the limit on the real client address taken from a header that only your own proxy may set, and for logged-in traffic prefer the account or API key over the address.

The limiter reports a server fault. nginx's default answer to a rejected request is 503 Service Unavailable, which in HTTP means the server is overloaded or down for maintenance. Dashboards and alerts that treat 5xx responses as server errors will then wake someone up because a single client was greedy, and the client learns nothing about its own behaviour. Answer with 429, which puts the cause on the client's side, and log the rejections at a level of their own.

Every throttled client comes back in the same second. If a quota resets at a fixed moment, or every refused client is told the same `Retry-After`, they all return together; the IETF draft gives the example of a daily reset at 18:00:00, when all clients are likely to show up at 18:00:00. The burst that follows can be worse than the one that was limited. Add jitter on both sides: vary the wait the server announces, and let clients add a random delay of their own.

Nobody decided what happens when the store goes down. A limiter that keeps its counts in Redis or a database must do something when the store is unreachable. Code that catches the connection error and lets the request through has chosen to fail open, which turns the outage into an unlimited window for whoever is watching; code that refuses has chosen to fail closed, which turns it into an outage for everyone. Neither is wrong everywhere, but the choice belongs in a design decision per path, as the directory above made it, not in whatever a library does by default.

Counting refusals leaks information. If 401 and 403 responses count against a quota, the IETF draft warns, a malicious client could probe an endpoint to learn about another user's traffic. Decide which responses count, give failed authentication a limit of its own, and keep quota details from clients that have not authenticated.

Further reading

- RFC 6585: Additional HTTP Status Codes — defines 429 Too Many Requests and states what the code leaves to the server: who the user is and how requests are counted.
- RFC 9110: HTTP Semantics — the `Retry-After` field and its two forms, and what 503 means when the server sends it.
- RateLimit header fields for HTTP — the IETF HTTPAPI working group's draft of `RateLimit-Policy` and `RateLimit`, with a security section on synchronised returns and information leaks.

- Module ngx_http_limit_req_module — `limit_req`, `burst`, `nodelay`, `delay`, dry-run mode and the sizes of the shared memory zone.
- HAProxy 3.4 Configuration Manual — stick tables and their rate counters, and the peers protocol that copies them between instances.

Glossary

Choose for your actual constraints

Bring the deployment path, the operational risks and the people on call. We can scope a change with checks, a recovery route and clear ownership.

See what it costs Get in touch →

Make the terminology practical

Ask how “What is rate limiting?” applies to your organisation. A concrete example from your current system is enough to start the discussion.

Explore our services

welcome@engineer.company +45 50 10 61 84 Copenhagen, Denmark

Take this page with you

Contact card

Carefully Crafted With Love by D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Copyright © 2025 – present · Engineer ApS · All rights reserved.

Printed from https://engineer.company/glossary/rate-limiting/