EcomLabs

STEP-BY-STEP GUIDE

Sliding window rate limiting in 15 lines

The 15-line Redis function, atomic and with a unique id per request; the Postgres version; a server that answers 429 with Retry-After, and starting values per route. All the code, tested.

FOR	Node.js APIs and stores that need to limit login, search, discount codes or forms
TESTED WITH	Node 22.22.3 · node-redis 6.3.1 · fakeredis 2.40.0 (Lua 5.1) · PGlite 0.5.10 (PostgreSQL 18.6) · Postgres.js 3.4.9 · Windows 11
UPDATED	November 11, 2026
DOCS	redis.io/docs/latest/develop/programmability/eval-intro
ARTICLE	ecomlabs.dev/g/ratelimit-en
AUTHOR	Luis Morales · @luismoralesdev

Contents

Before you start	What you will build
Step 01	The 15-line function
Step 02	Answer 429 with Retry-After
Step 03	The Postgres version
Step 04	Suggested values per route
Step 05	What we tested
When you are done	Checklist

BEFORE YOU START

What you will build

A sliding window rate limiter in Redis (a Lua script that removes old entries, counts, and records the request only if there is room, in one atomic step), the same idea in Postgres, and a server that answers `429` with `Retry-After`. We tested it on November 11, 2026 with Node 22.22.3 and node-redis 6.3.1 (`npm install redis`) against fakeredis 2.40.0, a Python Redis emulator, and with Postgres.js 3.4.9 (`npm install postgres`) against PGlite 0.5.10 (PostgreSQL 18.6).

STEP 01

The 15-line function

Save it as `limiter.mjs`. It is 15 lines, not counting the blank one:

```
const SCRIPT = `
redis.call('ZREMRANGEBYSCORE', KEYS[1], '-inf', ARGV[1] - ARGV[2])
if redis.call('ZCARD', KEYS[1]) < tonumber(ARGV[3]) then
  redis.call('ZADD', KEYS[1], ARGV[1], ARGV[4])
  redis.call('PEXPIRE', KEYS[1], ARGV[2])
  return 0
end
local oldest = redis.call('ZRANGE', KEYS[1], 0, 0, 'WITHSCORES')
return oldest[2] + ARGV[2] - ARGV[1]`;

export async function slidingWindow(redis, key, limit, windowMs, now = Date.now()) {
  const member = `${now}-${crypto.randomUUID()}`;
  const args = [now, windowMs, limit, member].map(String);
  const retryAfterMs = await redis.eval(SCRIPT, { keys: [key], arguments: args });
  return { allowed: retryAfterMs === 0, retryAfterMs };
}
```

- **Atomic:** Redis runs the whole script without interleaving other commands, so nobody counts between `ZCARD` and `ZADD`.
- **Unique id:** `ZADD` with an existing member only updates its score. With the timestamp alone as the name, two requests in the same millisecond would count as one.
- **Returns** `{ allowed, retryAfterMs }`: 0 when it passes; otherwise, the time until the oldest request leaves the window. The key goes in `KEYS`, never inside the script.

STEP 02

Answer 429 with Retry-After

Save `server.mjs`. Replace `LIMITS` with your routes and run `REDIS_URL=redis://… node server.mjs`:

```
import http from "node:http";
import { createClient } from "redis";
import { slidingWindow } from "./limiter.mjs";

const MINUTE = 60_000;
const LIMITS = {
  "POST /login": { limit: 5, windowMs: 15 * MINUTE },
  "POST /api/discount-code": { limit: 10, windowMs: MINUTE },
  "GET /api/search": { limit: 60, windowMs: MINUTE },
};

const redis = await createClient({ url: process.env.REDIS_URL, disableOfflineQueue: true })
  .on("error", (err) => console.error("redis:", err.message))
  .connect();

http.createServer(async (req, res) => {
  const route = `${req.method} ${new URL(req.url, "http://localhost").pathname}`;
  const rule = LIMITS[route];
  if (rule) {
    const key = `rl:${route}:${req.socket.remoteAddress}`;
    const verdict = await slidingWindow(redis, key, rule.limit, rule.windowMs).catch((err) => {
      console.error("rate limiter unavailable, letting the request through:", err.message);
      return { allowed: true };
    });
    if (!verdict.allowed) {
      const seconds = Math.ceil(verdict.retryAfterMs / 1000);
      res.writeHead(429, { "Content-Type": "application/json", "Retry-After": String(seconds) });
      res.end(JSON.stringify({ error: "too_many_requests", retryAfterSeconds: seconds }));
      return;
    }
  }
  res.writeHead(200, { "Content-Type": "application/json" });
  res.end(JSON.stringify({ ok: true, route }));
}).listen(Number(process.env.PORT ?? 3000));
```

- `Retry-After` is whole seconds (RFC 9110), hence `Math.ceil`.
- `disableOfflineQueue: true` makes a command fail right away while Redis is down, and the `catch` lets the request through (fail open). For login you may prefer to answer 503.
- Behind a proxy, `req.socket.remoteAddress` is the proxy. Only use the IP your trusted proxy added to `X-Forwarded-For`, never the leftmost one.

STEP 03

The Postgres version

Without Redis, create the table with `schema.sql`:

```
CREATE TABLE rate_limit_hits (
  key text        NOT NULL,
  at  timestamptz NOT NULL
);

CREATE INDEX rate_limit_hits_key_at ON rate_limit_hits (key, at);
```

And save `limiter-pg.mjs`, which takes a Postgres.js client and returns the same shape as the Redis version:

```
export async function slidingWindowPg(sql, key, limit, windowMs, now = new Date()) {
  const since = new Date(now.getTime() - windowMs);
  return sql.begin(async (tx) => {
    await tx`SELECT pg_advisory_xact_lock(hashtext(${key}))`;
    await tx`DELETE FROM rate_limit_hits WHERE key = ${key} AND at <= ${since}`;
    const [{ count, oldest }] = await tx`
      SELECT count(*)::int AS count, min(at) AS oldest FROM rate_limit_hits WHERE key = ${key}`;
    if (count >= limit) {
      return { allowed: false, retryAfterMs: oldest.getTime() + windowMs - now.getTime() };
    }
    await tx`INSERT INTO rate_limit_hits (key, at) VALUES (${key}, ${now})`;
    return { allowed: true, retryAfterMs: 0 };
  });
}
```

The per-key lock stops two transactions from counting at once. Also schedule this delete, using your longest window:

```
DELETE FROM rate_limit_hits WHERE at < now() - interval '1 hour';
```

> **CAUTION**
> `hashtext` is an internal PostgreSQL function with no official documentation, and its output can change between major versions. Here it only names a lock that lives for one transaction, so that does not matter. Every limited request is a four-statement transaction on your database: use it for login or forms, not for every `GET` on a busy API.

STEP 04

Suggested values per route

These are starting points, not measurements. Tune them against your real traffic logs.

ROUTE	KEY	LIMIT	WINDOW
`POST /login`	IP, and separately the account	5	15 min
`POST /password-reset`	Account	3	1 h
`POST /api/discount-code`	IP	10	1 min
`GET /api/search`	IP	60	1 min
`POST /contact`	IP	5	1 h
`POST /webhooks/shopify`	None: verify the HMAC and enqueue	—	—

On login, anyone can exhaust the per-account limit on purpose: use a short wait, not a long lockout.

STEP 05

What we tested

- Limit 10 and 100 concurrent requests over five connections: 10 passed (91 and 97 without the script). Postgres on PGlite: 10 (64 with no transaction and no lock).
- 20 requests in the same millisecond: 20 members; with the timestamp alone as the name, 1, and all 20 passed.
- At the minute boundary, at 10 per minute: the fixed window let 20 through; the sliding window, 10.
- The server: five `200`s on `/login`, then `429` with `Retry-After: 900`; with Redis down, the request went through.

> **CAUTION**
> fakeredis is not Redis: it reproduces the script's atomicity but measures no speed, memory or cluster behavior. PGlite never overlaps two transactions, so what the lock does on a regular PostgreSQL comes from its documentation. Repeat the tests in staging, never in production.

WHEN YOU ARE DONE

Checklist

- [ ] Every sensitive route has its key, limit and window in one place.
- [ ] Counting and recording happen in one atomic step (Lua script or `pg_advisory_xact_lock`), with a unique id per request.
- [ ] The Redis key expires with `PEXPIRE`; the Postgres table has its periodic delete.
- [ ] The response is a `429` with `Retry-After` in whole seconds and a body that explains it.
- [ ] What happens when Redis does not answer is decided and tested.
- [ ] The IP comes from your trusted proxy, and webhooks are not limited by IP.