EcomLabs

GUÍA PASO A PASO

Rate limiting con ventana deslizante en 15 líneas

La función de 15 líneas para Redis, atómica y con un id único por petición; la versión con Postgres; el servidor que responde 429 con Retry-After, y valores de partida por ruta. Todo el código, probado.

PARA	APIs y tiendas en Node.js que necesitan limitar login, búsqueda, códigos o formularios
PROBADO CON	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
ACTUALIZADA	11 de noviembre de 2026
DOCUMENTACIÓN	redis.io/docs/latest/develop/programmability/eval-intro
ARTÍCULO	ecomlabs.dev/g/ratelimit
AUTOR	Luis Morales · @luismoralesdev

Índice

Antes de empezar	Qué vas a montar
Paso 01	La función de 15 líneas
Paso 02	Responder 429 con Retry-After
Paso 03	La versión con Postgres
Paso 04	Valores sugeridos por ruta
Paso 05	Lo que probamos
Al terminar	Checklist

ANTES DE EMPEZAR

Qué vas a montar

Un limitador de ventana deslizante en Redis (un script Lua que borra lo viejo, cuenta y anota la petición solo si hay cupo, en un paso atómico), la misma idea en Postgres y un servidor que responde `429` con `Retry-After`. Lo probamos el 11 de noviembre de 2026 con Node 22.22.3 y node-redis 6.3.1 (`npm install redis`) contra fakeredis 2.40.0, un emulador de Redis en Python, y con Postgres.js 3.4.9 (`npm install postgres`) contra PGlite 0.5.10 (PostgreSQL 18.6).

PASO 01

La función de 15 líneas

Guárdala como `limiter.mjs`. Son 15 líneas sin contar la línea en blanco:

```
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 };
}
```

- **Atómico:** Redis ejecuta el script entero sin intercalar otros comandos, así que nadie cuenta entre `ZCARD` y `ZADD`.
- **Id único:** `ZADD` con un miembro repetido solo actualiza su score. Con la hora sola como nombre, dos peticiones del mismo milisegundo contarían como una.
- **Devuelve** `{ allowed, retryAfterMs }`: 0 si pasa; si no, lo que falta para que salga la petición más antigua. La clave va en `KEYS`, nunca dentro del script.

PASO 02

Responder 429 con Retry-After

Guarda `server.mjs`. Cambia `LIMITS` por tus rutas y corre `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` va en segundos enteros (RFC 9110): por eso `Math.ceil`.
- `disableOfflineQueue: true` hace que, con Redis caído, el comando falle de inmediato y el `catch` deje pasar la petición (fallar abierto). Para el login quizá prefieras responder 503.
- Detrás de un proxy, `req.socket.remoteAddress` es el proxy. Usa solo la IP que agregó tu proxy de confianza en `X-Forwarded-For`, nunca la primera de la lista.

PASO 03

La versión con Postgres

Si no tienes Redis, crea la tabla con `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);
```

Y guarda `limiter-pg.mjs`, que recibe el cliente de Postgres.js y devuelve lo mismo que la versión de Redis:

```
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 };
  });
}
```

El candado por clave evita que dos transacciones cuenten a la vez. Programa también este borrado, con la ventana más larga que uses:

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

> **CUIDADO**
> `hashtext` es una función interna de PostgreSQL, sin documentación oficial, y su resultado puede cambiar entre versiones mayores. Aquí solo nombra un candado que dura una transacción, así que no importa. Cada petición limitada es una transacción de cuatro sentencias en tu base: úsalo para el login o formularios, no para cada `GET` de una API con mucho tráfico.

PASO 04

Valores sugeridos por ruta

Son puntos de partida, no mediciones. Ajústalos con los registros de tu tráfico real.

RUTA	CLAVE	LÍMITE	VENTANA
`POST /login`	IP, y aparte la cuenta	5	15 min
`POST /password-reset`	Cuenta	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`	Ninguna: verifica el HMAC y encola	—	—

En el login, el límite por cuenta lo puede agotar cualquiera a propósito: usa una espera corta, no un bloqueo largo.

PASO 05

Lo que probamos

- Límite 10 y 100 peticiones simultáneas en cinco conexiones: pasaron 10 (91 y 97 sin el script). Postgres en PGlite: 10 (64 sin transacción ni candado).
- 20 peticiones del mismo milisegundo: 20 miembros; con la hora sola como nombre, 1, y pasaron las 20.
- En el borde del minuto, con 10 por minuto: la ventana fija dejó pasar 20; la deslizante, 10.
- El servidor: cinco `200` en `/login` y luego `429` con `Retry-After: 900`; con Redis apagado, la petición pasó.

> **CUIDADO**
>
> fakeredis no es Redis: reproduce la atomicidad del script, pero no mide velocidad, memoria ni cluster. PGlite nunca solapa dos transacciones, así que lo que hace el candado en un PostgreSQL normal sale de su documentación. Repite las pruebas en staging, nunca en producción.

AL TERMINAR

Checklist

- [ ] Cada ruta sensible tiene clave, límite y ventana en un solo lugar.
- [ ] Contar y anotar ocurre en un paso atómico (script Lua o `pg_advisory_xact_lock`), con un id único por petición.
- [ ] La clave de Redis caduca con `PEXPIRE`; la tabla de Postgres tiene su borrado periódico.
- [ ] La respuesta es `429` con `Retry-After` en segundos enteros y un cuerpo que lo explica.
- [ ] Está decidido y probado qué pasa si Redis no responde.
- [ ] La IP sale de tu proxy de confianza, y los webhooks no se limitan por IP.