EcomLabs

GUÍA PASO A PASO

CORS no protege tu API

La lista que usamos para revisar un endpoint: autenticación, autorización por usuario, límite de peticiones y una configuración de CORS correcta. Cada punto con el código de Node que probamos con curl y con Chromium.

PARA	APIs en Node.js o cualquier backend HTTP
PROBADO CON	Node 22 · curl 8.19 · Chromium 151 · Windows 11
ACTUALIZADA	30 de octubre de 2026
DOCUMENTACIÓN	developer.mozilla.org/en-US/docs/Web/HTTP/Guides/CORS
ARTÍCULO	ecomlabs.dev/g/cors
AUTOR	Luis Morales · @luismoralesdev

Índice

Antes de empezar	Qué protege CORS y qué no
Paso 01	Prueba tu API como lo haría un atacante
Paso 02	Exige autenticación en el servidor
Paso 03	Comprueba que el recurso es del usuario
Paso 04	Limita las peticiones
Paso 05	Configura CORS con una lista exacta
Paso 06	Si te autenticas con cookies
Al terminar	Checklist de seguridad para endpoints

ANTES DE EMPEZAR

Qué protege CORS y qué no

CORS es una regla del navegador: decide si el JavaScript de una página de otro origen puede leer la respuesta de tu API. No frena a curl, a un script ni a otro servidor, y en las peticiones simples (un `GET` o un `POST` de formulario) ni siquiera impide que tu servidor las ejecute. Esta guía es la lista que usamos para revisar un endpoint, con el código de Node que probamos para cada punto.

Todo el código es de un servidor `node:http` sin dependencias, probado el 30 de octubre de 2026 con Node 22, curl 8.19 y Chromium 151 (Playwright). Los ejemplos usan una API en `http://127.0.0.1:8787` y una tienda en `https://tienda.example`.

PASO 01

Prueba tu API como lo haría un atacante

Antes de cambiar nada, pide un recurso privado con curl sin cabecera `Origin` y luego con un origen que no permites. Si alguna de las dos responde con datos sin un token válido, CORS no te está protegiendo de nada:

```
$ curl -si http://127.0.0.1:8787/api/orders/1001 -H "Authorization: Bearer tok_ana" -H "Origin: https://otra-tienda.example"
HTTP/1.1 200 OK
Vary: Origin
Content-Type: application/json

{"id":"1001","userId":"ana","total":1250}
```

Con token válido, nuestra API respondió 200 a los dos, sin `Access-Control-Allow-Origin` en el segundo caso. curl no mira esa cabecera. Repite la prueba sin la cabecera `Authorization`: debe salir un 401.

PASO 02

Exige autenticación en el servidor

```
const TOKENS = new Map([
  ["tok_ana", "ana"],
  ["tok_beto", "beto"],
]);

function authenticate(req) {
  const [scheme, token] = (req.headers.authorization ?? "").split(" ");
  return scheme === "Bearer" ? (TOKENS.get(token) ?? null) : null;
}
```

El `Map` de la demo representa tu sistema de sesiones: en tu app, aquí verificas la cookie de sesión o el token firmado. Devuelve el id del usuario o `null`; nunca confíes en un id que venga en el cuerpo o en la URL.

PASO 03

Comprueba que el recurso es del usuario

```
  if (!userId) return send(res, 401, { error: "unauthenticated" });

  const match = req.url.match(/^\/api\/orders\/(\w+)$/);
  if (req.method === "GET" && match) {
    const order = ORDERS.get(match[1]);
    if (!order || order.userId !== userId) return send(res, 404, { error: "not_found" });
    return send(res, 200, order);
  }
  send(res, 404, { error: "not_found" });
```

La línea que importa es `order.userId !== userId`. Sin ella, cualquier cliente con sesión puede leer cualquier pedido cambiando el número en la URL. Respondemos 404 para no confirmar qué ids existen; un 403 también vale.

PASO 04

Limita las peticiones

```
const WINDOW_MS = 60_000;
const MAX_REQUESTS = 5;
const hits = new Map();

function rateLimited(key) {
  const now = Date.now();
  const entry = hits.get(key);
  if (!entry || now - entry.start >= WINDOW_MS) {
    hits.set(key, { start: now, count: 1 });
    return false;
  }
  entry.count += 1;
  return entry.count > MAX_REQUESTS;
}

setInterval(() => {
  const now = Date.now();
  for (const [key, entry] of hits) if (now - entry.start >= WINDOW_MS) hits.delete(key);
}, WINDOW_MS).unref();
```

Llámala al principio del manejador con el usuario como clave, o con la IP si no hay token, y responde 429 con `Retry-After` cuando devuelva `true`. En nuestra prueba, la sexta petición en un minuto recibió el 429.

> **CUIDADO**
>
> Este límite vive en la memoria de un solo proceso. Con varias instancias cada una cuenta por su lado: usa un almacén compartido o el límite de tu proveedor. Detrás de un proxy, `req.socket.remoteAddress` es la IP del proxy.

PASO 05

Configura CORS con una lista exacta

```
const ALLOWED_ORIGINS = new Set((process.env.ALLOWED_ORIGINS ?? "").split(",").filter(Boolean));

function cors(req, res) {
  const origin = req.headers.origin;
  res.setHeader("Vary", "Origin");
  if (!origin || !ALLOWED_ORIGINS.has(origin)) return;
  res.setHeader("Access-Control-Allow-Origin", origin);
  if (req.method === "OPTIONS") {
    res.setHeader("Access-Control-Allow-Methods", "GET, POST");
    res.setHeader("Access-Control-Allow-Headers", "Authorization, Content-Type");
    res.setHeader("Access-Control-Max-Age", "600");
  }
}
```

Llámala antes que todo lo demás, para que también los 401, 404 y 429 lleven las cabeceras y tu frontend vea el error real:

```
const server = createServer(async (req, res) => {
  console.log(`${req.method} ${req.url} origin=${req.headers.origin ?? "-"}`);
  cors(req, res);
  if (req.method === "OPTIONS") return res.writeHead(204).end();
```

Arranca la API con la lista de orígenes en una variable de entorno, por ejemplo `ALLOWED_ORIGINS=https://tienda.example node api.mjs`. Las cabeceras que debe devolver:

CABECERA	VALOR	CUÁNDO
`Access-Control-Allow-Origin`	Un origen exacto de la lista	Siempre que el origen esté permitido
`Vary`	`Origin`	En todas las respuestas
`Access-Control-Allow-Methods`	Solo los que usas	Respuesta al `OPTIONS`
`Access-Control-Allow-Headers`	Solo las que envías	Respuesta al `OPTIONS`
`Access-Control-Max-Age`	Segundos, por ejemplo `600`	Respuesta al `OPTIONS`
`Access-Control-Allow-Credentials`	`true`	Solo si usas cookies

PASO 06

Si te autenticas con cookies

El frontend llama con `fetch(url, { credentials: "include" })` y la API responde con un origen concreto y `Access-Control-Allow-Credentials`. Esta configuración funcionó en Chromium 151:

```
res.setHeader("Vary", "Origin");
res.setHeader("Access-Control-Allow-Origin", "http://localhost:5173");
res.setHeader("Access-Control-Allow-Credentials", "true");
```

La misma llamada contra una API que respondía `Access-Control-Allow-Origin: *` falló con `TypeError: Failed to fetch`: con credenciales, el comodín no se acepta.

> **CUIDADO**
> Nunca copies el `Origin` de la petición sin compararlo con la lista, y nunca permitas `null`. Con credenciales, cualquiera de las dos deja que otra página lea datos con la sesión de tu usuario. Un `POST` de formulario con cookies tampoco pasa por preflight: protégelo con un token CSRF.

AL TERMINAR

Checklist de seguridad para endpoints

- [ ] Cada endpoint privado exige autenticación en el servidor.
- [ ] El id del usuario sale de la sesión o del token, nunca del cuerpo ni de la URL.
- [ ] Cada recurso se compara con el usuario que lo pide; los ajenos responden 404 o 403, sin datos.
- [ ] Hay un límite de peticiones por usuario y por IP que responde 429 con `Retry-After`.
- [ ] Los endpoints que cambian datos no dependen de CORS ni del preflight.
- [ ] Con cookies, los formularios que cambian datos llevan un token CSRF.
- [ ] `Access-Control-Allow-Origin` sale de una lista exacta, sin `includes`, `startsWith` ni `null`.
- [ ] `Vary: Origin` y las cabeceras CORS van en todas las respuestas, también en los 401, 404 y 429.
- [ ] El preflight solo declara los métodos y las cabeceras que tu frontend usa.
- [ ] curl sin token, sin `Origin` y con un `Origin` falso no devuelve datos.