EcomLabs

STEP-BY-STEP GUIDE

CORS does not protect your API

The checklist we use to review an endpoint: authentication, per-user authorization, rate limiting and a correct CORS setup. Each item comes with the Node code we tested with curl and Chromium.

FOR	Node.js APIs or any HTTP backend
TESTED WITH	Node 22 · curl 8.19 · Chromium 151 · Windows 11
UPDATED	October 30, 2026
DOCS	developer.mozilla.org/en-US/docs/Web/HTTP/Guides/CORS
ARTICLE	ecomlabs.dev/g/cors-en
AUTHOR	Luis Morales · @luismoralesdev

Contents

Before you start	What CORS protects, and what it does not
Step 01	Test your API the way an attacker would
Step 02	Require authentication on the server
Step 03	Check that the resource belongs to the user
Step 04	Rate-limit requests
Step 05	Configure CORS with an exact list
Step 06	If you authenticate with cookies
When you finish	Endpoint security checklist

BEFORE YOU START

What CORS protects, and what it does not

CORS is a browser rule: it decides whether JavaScript on a page from another origin may read your API's response. It does not stop curl, a script or another server, and for simple requests (a `GET` or a form `POST`) it does not even stop your server from running them. This guide is the checklist we use to review an endpoint, with the Node code we tested for each item.

All the code comes from a `node:http` server with no dependencies, tested on October 30, 2026 with Node 22, curl 8.19 and Chromium 151 (Playwright). The examples use an API on `http://127.0.0.1:8787` and a store on `https://tienda.example`.

STEP 01

Test your API the way an attacker would

Before changing anything, request a private resource with curl, first without an `Origin` header and then with an origin you do not allow. If either one returns data without a valid token, CORS is protecting you from nothing:

```
$ 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}
```

With a valid token, our API answered 200 both times, with no `Access-Control-Allow-Origin` the second time. curl ignores that header. Run it again without the `Authorization` header: you should get a 401.

STEP 02

Require authentication on the server

```
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;
}
```

The demo's `Map` stands in for your session system: in your app, this is where you verify the session cookie or the signed token. It returns the user id or `null`; never trust a user id that arrives in the body or the URL.

STEP 03

Check that the resource belongs to the user

```
  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" });
```

The line that matters is `order.userId !== userId`. Without it, any logged-in client can read any order by changing the number in the URL. We answer 404 so the API does not confirm which ids exist; a 403 works too.

STEP 04

Rate-limit requests

```
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();
```

Call it at the top of the handler, keyed by user, or by IP when there is no token, and answer 429 with `Retry-After` when it returns `true`. In our test, the sixth request within a minute got the 429.

> **CAUTION**
>
> This limiter lives in one process's memory. With several instances each one counts on its own, so use a shared store or your platform's rate limiter. Behind a proxy, `req.socket.remoteAddress` is the proxy's IP.

STEP 05

Configure CORS with an exact list

```
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");
  }
}
```

Call it before anything else, so 401, 404 and 429 responses carry the headers too and your frontend sees the real error:

```
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();
```

Start the API with the origin list in an environment variable, for example `ALLOWED_ORIGINS=https://tienda.example node api.mjs`. The headers it should send:

HEADER	VALUE	WHEN
`Access-Control-Allow-Origin`	One exact origin from the list	Whenever the origin is allowed
`Vary`	`Origin`	On every response
`Access-Control-Allow-Methods`	Only the ones you use	Answer to `OPTIONS`
`Access-Control-Allow-Headers`	Only the ones you send	Answer to `OPTIONS`
`Access-Control-Max-Age`	Seconds, for example `600`	Answer to `OPTIONS`
`Access-Control-Allow-Credentials`	`true`	Only if you use cookies

STEP 06

If you authenticate with cookies

The frontend calls `fetch(url, { credentials: "include" })` and the API answers with a specific origin and `Access-Control-Allow-Credentials`. This setup worked in Chromium 151:

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

The same call against an API that answered `Access-Control-Allow-Origin: *` failed with `TypeError: Failed to fetch`: with credentials, the wildcard is refused.

> **CAUTION**
> Never copy the request's `Origin` without checking it against the list, and never allow `null`. With credentials, either one lets another page read data with your user's session. A form `POST` with cookies skips the preflight too: protect it with a CSRF token.

WHEN YOU FINISH

Endpoint security checklist

- [ ] Every private endpoint requires authentication on the server.
- [ ] The user id comes from the session or token, never from the body or the URL.
- [ ] Every resource is checked against the user asking for it; other users' resources answer 404 or 403, with no data.
- [ ] There is a rate limit per user and per IP that answers 429 with `Retry-After`.
- [ ] Endpoints that change data do not rely on CORS or the preflight.
- [ ] With cookies, forms that change data carry a CSRF token.
- [ ] `Access-Control-Allow-Origin` comes from an exact list: no `includes`, no `startsWith`, no `null`.
- [ ] `Vary: Origin` and the CORS headers are on every response, 401, 404 and 429 included.
- [ ] The preflight only declares the methods and headers your frontend uses.
- [ ] curl with no token, no `Origin` or a fake `Origin` returns no data.