**CORS Error: Cross-Origin Resource Sharing**

It is a security mechanism implemented by browsers to control which external domains (origins) can access resources (like APIs or assets).

An **asset** generally refers to any **static resource** or file that your application uses to **render content or provide functionality.**

**Examples**: Images, Audios, Videos, Stylesheets, JavaScript files, Fonts, Other static files

The CORS error is thrown by the browser (frontend environment), not the backend.

CORS Request Flow:

1. You click a button → Frontend JavaScript sends a request

2. The Browser **automatically adds an Origin header**

Ex: **Origin: https://frontend.com.**

Everything happens **automatically in the background by the browser**. You don't need to manually add the Origin header — the **browser does** it **whenever your frontend JS makes a cross-origin request.** This tells the backend "**This request is coming from this domain**".

3. The Se**rver receives the request** and **processes it normally** — it **does NOT give any CORS error** at this point. It just responds normally.

4. The Browser receives the response and checks: **Does the response include Access-Control-Allow-Origin: https://frontend.com**

**5. If yes -> Browser gives the response to frontend JS**

**If No -> Browser blocks the response and throws CORS error in console**

An origin is defined as**: protocol + domain + port**

If a web app tries to access a different origin, the browser blocks it UNLESS the server at that origin explicitly allows it via CORS headers.

When Does CORS Trigger?

Triggered when JavaScript in the browser makes requests like:

1. fetch("https://api.example.com")
2. axios.post(…)
3. XHR (AJAX)
4. Web Fonts, WebGL textures, Canvas images, Audio & Video sources

**Web Fonts**: Fonts loaded from external sources (like Google Fonts) are considered assets.

Example:

**@font-face {**

**font-family: 'CustomFont';**

**src: url('https://example.com/fonts/custom.woff2');**

**}**

**CORS relevance:**

Browsers enforce CORS for fonts because if JS can read font data from another domain, it could extract private content (security risk).

If the server hosting the font doesn't allow your domain via Access-Control-Allow-Origin, the font won't load.

Google Fonts are configured to allow cross-origin requests from any domain. That's why you can just include a font in your CSS from any website without running into CORS errors.

**Access-Control-Allow-Origin: ***

**Browser Flow**

**Simple Request

1.** Allowed without preflight if:

Methods: GET, POST, HEAD

Headers: Only Accept, Content-Type (form-types only), Origin, etc.

Content-Type: application/x-www-form-urlencoded, multipart/form-data, text/plain

Browser just sends request → If response contains Access-Control-Allow-Origin, browser allows → otherwise blocks.

2. A preflight request is an automatic check by the browser to see if the server allows a "non-simple" cross-origin request before actually sending the real request.

Think of it like asking permission first.

When Does a Preflight Happen?

The browser sends a preflight OPTIONS request only if the request is "non-simple."

Simple requests don't need preflight: **Method: GET, POST, HEAD**

Non-simple requests trigger preflight:

Methods: PUT, DELETE, PATCH

Custom headers: Authorization, X-Custom-Header

Content-Type like application/json

**Example of Preflight Flow:**

**1. Frontend JS sends a "non-simple" request**

```
fetch('https://api.example.com/data', {
```

method: 'PUT',

headers: {

'Content-Type': 'application/json',

'Authorization': 'Bearer token'

},

body: JSON.stringify({ name: 'John' })

});

**2. Browser sends a preflight OPTIONS request first:**

```
OPTIONS /data HTTP/1.1
```

Origin: https://frontend.com

Access-Control-Request-Method: PUT

Access-Control-Request-Headers: Content-Type, Authorization

This is just a permission check — no real data is sent yet.

**3. Server responds with CORS headers:**

```
HTTP/1.1 204 No Content
```

Access-Control-Allow-Origin: https://frontend.com

Access-Control-Allow-Methods: GET, POST, PUT, DELETE

Access-Control-Allow-Headers: Content-Type, Authorization

Access-Control-Max-Age: 86400

Access-Control-Allow-Origin: Which frontend domains are allowed

Access-Control-Allow-Methods: Which HTTP methods are allowed

Access-Control-Allow-Headers: Which custom headers are allowed

Access-Control-Max-Age: How long browser can cache this preflight response

**Browser sees preflight is allowed → sends the actual PUT request with the JSON body.**

**Why Preflight Exists?**

**Security**: prevents malicious sites from sending unsafe requests to your API.

Browser only allows the actual request if server explicitly approves method + headers + origin.

**Key Points:**

1. Preflight is sent automatically by the browser; you don't write it manually.

2. Server must handle OPTIONS requests properly. If it doesn't → CORS error

3. Preflight happens before the actual request, so your API might not even see the real request if preflight fails.

4. Some servers cache preflight via Access-Control-Max-Age to reduce extra requests.

**Difference Between Simple Request and Preflight Request (CORS):**

**Feature Simple Request Preflight Request (Complex Request)**

1. Trigger Condition Safe and common HTTP requests Requests that may be risky or custom

2. Allowed Methods GET, POST, HEAD only Anything like PUT, DELETE, PATCH, etc

3. Allowed Headers Only basic headers (Accept, Content-Type with restricted values) Any custom header (Authorization, X-Token, etc.)

4. Content-Type Allowed Only these: Anything else like application/json

- application/x-www-form-urlencoded
- multipart/form-data
- text/plain

5. Preflight (OPTIONS) Sent? No Yes

6. Example fetch('/api/users') fetch('/api/users', { method: 'DELETE' }) or sending Authorization header

**Is POST and GET excluded from preflight?**

GET and POST can be simple requests → No preflight needed, but only under specific conditions

But GET or POST can also trigger preflight → If they use non-simple headers or content-types.

**When GET/POST trigger preflight (Complex Request → OPTIONS sent first)?**

Use headers like Authorization, X-Custom-Header, etc.

Use Content-Type: application/json

GET and POST are not automatically excluded from preflight.

They avoid preflight only when they use allowed headers + allowed content-types.

If they include JSON or custom headers, preflight (OPTIONS) will happen.

**How to Reduce Preflight Requests (for Faster APIs)?**

When your frontend makes a non-simple request (like sending JSON or Authorization headers), the browser first sends a CORS preflight (OPTIONS request). This adds extra delay.

To avoid or reduce preflights, make your requests look like simple requests.

**Strategy 1**: Use "Simple" Content-Type Instead of application/json

Instead of:

```
fetch('/api/user', {
method: 'POST',
headers: { 'Content-Type': 'application/json' },
body: JSON.stringify({ name: 'John' })
});
```

Use form data (which avoids preflight):

```
const formData = new URLSearchParams();
formData.append('name', 'John');
fetch('/api/user', {
method: 'POST',
headers: { 'Content-Type': 'application/x-www-form-urlencoded' }, // Simple
body: formData.toString()
});
```

No preflight, because this is a Simple Request.

**Strategy 2:**

Instead of:

```
fetch('/api/data', {
headers: {
"Authorization": "Bearer abc123"
}
});
```

if possible, send token in query param or cookie:

fetch(‘/api/data?token=abc123’);

**Strategy 3:** Enable Server-Side CORS Caching

In your backend response, include: Access-Control-Max-Age: 600

This tells the browser to cache preflight results for 10 minutes, reducing repeated preflights for the same request.

Example (Express.js):

app.use((req, res, next) => {

res.setHeader(“Access-Control-Allow-Origin”, “*”);

res.setHeader(“Access-Control-Allow-Methods”, “GET, POST, OPTIONS”);

res.setHeader(“Access-Control-Allow-Headers”, “Content-Type”);

res.setHeader(“Access-Control-Max-Age”, “600”); // Cache preflight for 10 mins

next();

});

**Strategy 4:** Use GET When Possible

If you’re not modifying data, prefer GET over POST, because GET with no extra headers never causes preflight.

**Types of CORS Errors:**

**Scenario What Happens Example**

No CORS headers sent by server Browser blocks response Access from localhost:3000 to api.com

Allowed Origin mismatch Server replies with different Access-Control-Allow-Origin Frontend origin: http://abc.com, server allows http://xyz.com

Missing Allow-Headers You send custom header like Authorization, but server doesn’t allow it Request blocked

Credentials issue (Cookies/Auth) Using with Credentials: true but header missing Missing “Access-Control-Allow-Credentials: true”

**Hidden / Confusing CORS Scenarios Developers Face:**

**Common Confusion What’s Really Happening**

“But server responded fine in Postman!” Postman is not a browser, so CORS doesn’t apply there

“I see the response in Network tab, but code throws error” Browser fetches the response but blocks JS from accessing it.

"But I am calling the same IP or domain!" Even different ports (like localhost:3000 → localhost:5000) are different origins. "It works in development but fails in production" Your production origin/domain is not allowed in CORS settings.

"Why does it work when I disable cache?" Sometimes browser caches preflight results incorrectly. You must use Access-Control-Max-Age.

**If you want the browser to send cookies (or make the response include cookies) on cross-origin requests, both client and server must opt into credentials.**

**Browser blocks credentialed responses unless the server explicitly allows credentials**.

**Special Cases You Should Know:**

1. **CORS in File Downloads / Images /
   tags**
2. does not throw CORS error (unless you read pixel data in JS).

and can fetch any origin without CORS because browser trusts them.

BUT if you use fetch(image_url).then(res => res.blob()), CORS applies.

**2. WebSockets and CORS**

`WebSockets (WS / WSS) do not use CORS policy, but some servers enforce Origin manually for s`

**3. Reverse Proxy Can Bypass CORS**

`Instead of changing backend CORS settings, frontend developers often proxy requests:`

React (vite, create-react-app) with proxy setting

Nginx / Apache reverse proxy

This hides cross-origin from the browser, so no CORS needed.

4. CORS is Always Checked by Browser — Never by Mobile Apps or Servers

**Worst Misconception About CORS: "CORS is a backend issue."**

**Server-side CORS setup:**

**Core idea:** The server must explicitly tell browsers which origins, methods, headers, and credentials are allowed. It does this by returning the appropriate Access-Control-* headers on the actual response — and on the preflight OPTIONS response when the browser sends one.

**Essential response headers:**

**Access-Control-Allow-Origin: or ***

When using credentials (cookies), never use *. You must echo the exact origin.

**Access-Control-Allow-Methods: GET,POST,PUT,DELETE,OPTIONS**

**Access-Control-Allow-Headers:** (e.g. Content-Type, Authorization)

**Access-Control-Allow-Credentials:** **true** (when cookies/credentials needed)

**Access-Control-Max-Age:** (cache preflight result)

**Server must handle OPTIONS (preflight)**

For non-simple requests the browser first sends an OPTIONS to ask permission.

The server must respond to OPTIONS with 2xx (often 204) and include the CORS headers.

**Examples:**

**1) Minimal Express (recommended: CORS package)**

**Code:**

```
// server.js
const express = require('express');
const cors = require('cors');
const app = express();
app.use(express.json());
// Allow single origin + credentials
app.use(cors({
origin: 'https://frontend.example.com',
credentials: true,
methods: ['GET','POST','PUT','DELETE','OPTIONS'],
allowedHeaders: ['Content-Type','Authorization']
}));
// Example endpoints
app.options('*', cors()); // optional explicit preflight handler
app.post('/login', (req, res) => {
// set cookie
res.cookie('session', 'abc', {
httpOnly: true, secure: true, sameSite: 'none'
});
```

res.json({ok:true});

});

app.listen(4000);

**Part Meaning**

**app.options(...)** This tells Express to handle HTTP OPTIONS method requests. OPTIONS method is what browsers send for preflight CORS checks.

**'*'** This means match all routes / endpoints. So any preflight request to any URL (like /api/users, /delete, /login, etc.) will be handled.

**cors()** This uses the CORS middleware to automatically reply with proper CORS headers such as:

- Access-Control-Allow-Origin
- Access-Control-Allow-Methods
- Access-Control-Allow-Headers

**2) Allow specific origin(s)**

**Code:**

```
app.use(cors({ origin: 'https://frontend.example.com' }));
```

**// or array:**

app.use(cors({ origin: ['https://a.com','https://b.com'] }));

**Security tips (server-side):**

1. Don't set Access-Control-Allow-Origin: * if you are sending credentials. Instead validate the Origin against a whitelist and echo it back.

2. Avoid reflecting arbitrary origin without whitelisting: that can open up security holes.

3. Keep Access-Control-Max-Age reasonable (minutes/hours) to reduce preflight but not allow stale policy.

4. For APIs that should be public and non-credentialed, * is fine.

**Bypassing / Proxying from the frontend (what devs do):**

**3 common bypass/proxy approaches:**

**1.Dev server proxy (local development):**

**Code:**

**// vite.config.js**

```
  export default {
  server: {
  proxy: {
  '/api': { target: 'https://api.example.com', changeOrigin: true, secure: false
}
  }
  }
  }
```

**Use: development convenience only. Production still needs proper CORS or a real reverse proxy.**

**2.Reverse proxy (production) — Nginx, Apache, CDN**

Host frontend.example.com and proxy /api to api.example.com at server level.

Browser sees same origin; CORS not involved.

This is the most robust production approach and allows same-origin cookies, simpler security.

**Example:**

```
const express = require("express");
const cors = require("cors");
const app = express();
const corsOptions = {
  origin: "http://localhost:3000", // Only allow this frontend
  methods: ["GET", "POST", "PUT", "DELETE"],
  allowedHeaders: ["Content-Type", "Authorization"],
  credentials: true, // Allow cookies/auth headers
};
app.use(cors(corsOptions));
app.options("*", cors(corsOptions)); // Preflight with same restrictions
app.get("/profile", (req, res) => {
  res.json({ user: "John Doe" });
});
app.listen(5000, () => console.log("Secure CORS server running"));
```

**3. Server-side forwarding (Your backend calls other backend)**

Your frontend talks to your own server (/api on same origin). Your server then calls the external API and returns data. Browser never calls the external origin directly.

This gives full control (and secures keys), but increases backend load and latency.

**Dev proxy → developer convenience only.**

**Reverse proxy → production, recommended when you want one origin for assets and API**

**Server-side forwarding → when you need to hide credentials or unify diverse services**

**Downsides of proxying:**

**Adds complexity to infra.**

**Might mask real CORS issues (test production).**

**Additional latency if not configured properly.**

**Debug checklist (practical step-by-step)**

Open DevTools → Network, make the request and inspect:

Is an OPTIONS request being sent? If yes, check its response.

Inspect Request Headers: is Origin present? are custom headers present?

Inspect Response Headers:

Access-Control-Allow-Origin: matches the origin? or * (and are you using credentials?)

Access-Control-Allow-Credentials: present when cookies are expected?

Access-Control-Allow-Headers / Allow-Methods: include what you need?

If response visible in Network but JS cannot read it -> browser blocked it (CORS).

Confirm server returns Set-Cookie with SameSite=None; Secure for cross-site cookies.

Test with curl to confirm server headers (remember curl won't simulate browser CORS behavior).

If using proxy, ensure production uses a similar proxy or proper CORS headers.

**Common mistakes & pitfalls:**

Using * with credentials -> cookies won't be sent.

Not handling OPTIONS at all -> preflight fails.

Testing with Postman (works) and assuming browser will behave same (it won't).

Allowing arbitrary Origin without validation -> security risk.

Forgetting SameSite=None and Secure on cookies (modern browsers block cross-site cookies otherwise).

Relying on dev proxy but failing to configure production proxy — works locally, fails production.

**When to prefer which approach:**

If you control both frontend & backend and want simplest UX: use reverse proxy/same-origin in production (Nginx).

If you want to keep API separate and use cookies: explicitly allow origin + credentials and set cookie attributes properly.

If stateless token-based API (mobile, third-party): use Authorization header (preflight cost is acceptable), or use cookies with CSRF mitigation depending on your threat model.

For public static assets (fonts, images) allow * where acceptable.