**How to Debug Any CORS Error**

Your request is hitting an error due to CORS. Not all is lost! Most CORS errors are quick & easy to debug and fix, once you understand the basics. Let's sort it out.

You know you're hitting a CORS error when you see error messages like:

Access to fetch at '<https://example.com>' from origin '<http://localhost:8000>' has been blocked by CORS policy.

No 'Access-Control-Allow-Origin' header is present on the requested resource

Cross-Origin Request Blocked: The Same Origin Policy disallows reading the remote resource at <https://example.com/>

Response to preflight request doesn't pass access control check

The value of the 'Access-Control-Allow-Origin' header in the response must not be the wildcard '\*' when the request's credentials mode is 'include'

Method PUT is not allowed by Access-Control-Allow-Methods in preflight response.

Request header field custom is not allowed by Access-Control-Allow-Headers in preflight response.

In each of these cases, you've asked JavaScript running in your page to send a request to a different origin, and at some stage the browser is refusing to do what you want.

**What is CORS?**

When you include JavaScript in a web page, you're running code on your user's computer, inside their browsing session.

That's a lot of power, and browsers are designed to protect users from the risks of this. CORS is one of these protections, aiming to protect the user and the services they use from two main attacks:

* CORS stops you from using the user's existing login session (their cookies and other cached authentication details) when communicating with other servers. JavaScript on your web page shouldn't be able to send requests to the Facebook API using their existing Facebook session. Without CORS, any web page could talk to other servers as you.
* CORS stops you from talking to servers that might only be accessible from their machine, but which aren't accessible publicly. Your web page should not be able to send requests to my-intranet-server.local, which might be an internal company server or your home router, and it should not be able to talk to servers that are listening only for localhost requests. Servers like these are often unauthenticated and very trusting, because they aren't connected to the public internet. Without CORS, any web page you visit could access them.

This only applies to cross origin requests, e.g. requests from https://example.com to https://google.com. The protocol, domain, and port all count as part of a URL's origin, but the path does not, so https://example.com/abc and https://example.com/def have the same origin, but http://localhost:123 and http://localhost:456 do not.

CORS protects against the above attacks by requiring the target server to opt into receiving dangerous requests from the source server, and to opt in to allowing pages from other origins to read responses. The Facebook API and your local network servers can accept requests from web pages running on other origins if they want to, but only if they agree.

**Why doesn't my CORS work?**

Your CORS request is failing because you're sending a request that the target server hasn't agreed to allow.

There's two classes of CORS request:

* 'Simple' cross-origin requests. There are basic requests that use no unsafe headers, don't stream requests or responses, and only use HEAD, GET or POST methods (with limited safe content types). Any request that's possible here would also be possible by e.g. loading an image or posting a form to the cross-origin request (and we can't stop those, for huge backwards compatibility reasons).

You can always send simple requests, but you might not be allowed to read the response.

* 'Preflighted' cross-origin requests. These are more complex requests, that aren't easy to send in other ways. A 'preflight' request will be sent to ask the server for permission before sending any of these requests, and if it's rejected, you won't be able to send the request at all.

If the preflight request is successful, the real request is sent, and the final response to that still has to follow the same rules as a 'simple' response for you to be allowed to read it.

When a request is preflighted, before sending the real request the browser sends an OPTIONS request with headers explaining the real request that it wants to send. It expects a response including headers that explicitly allow the real request.

There's three ways that this might hit an error:

1. You're sending a simple request, which is sent immediately, but the headers on the response don't allow you to read it.
2. You're sending a preflighted request, and the headers on the preflight response don't allow you to send the real request.
3. You're sending a preflighted request, the preflight went OK and the request was sent, but the headers on the final response for the real request don't allow you to read it.

The browser error message should show you which is happening for you. You can know if your request is being preflighted by looking for an OPTIONS request that's sent immediately before it.

The rules for the final (after preflight, if applicable) response are:

* The response must include a Access-Control-Allow-Origin header, whose value either matches the page's origin or is \*. The page's origin is sent in the request in an Origin header.
* If the request included credentials (e.g. fetch(url, { credentials: 'include' })) then the response headers must include Access-Control-Allow-Credentials: true, and the Access-Control-Allow-Origin header must match *exactly* (i.e. \* is not allowed).

If the response doesn't follow those rules, then the server hasn't opted in to your request, and you won't be allowed to read the response.

If you're in cases 1 or 3, you must be breaking one of these rules.

The rules for the preflight request are:

* The preflight response must include a Access-Control-Allow-Origin header, whose value either matches the page's origin or is \*. The page's origin is sent in the preflight request in an Origin header.
* If the page wants to send custom headers, then it will include Access-Control-Request-Headers listing the headers in the preflight OPTIONS request, and the server must include a Access-Control-Allow-Headers header that includes all those headers in the response. \* can also be used here, but it won't match an Authorization header - that must always be listed explicitly.
* If the page wants to use a non-simple HTTP method, it will include Access-Control-Request-Method in the preflight OPTIONS request, and the server must include a Access-Control-Allow-Methods header that includes that method in the response.
* If the page wants to send credentials (e.g. fetch(url, { credentials: 'include' })) the response must include a Access-Control-Allow-Credentials: true header, and the Access-Control-Allow-Origin header must match *exactly* (i.e. \* is not allowed).

If your preflight OPTIONS response doesn't follow these rules, then you won't be allowed to send the real request at all.

If you're in case 2, you must be breaking one of these rules.

It's also possible that you're in case 2, but you actually don't want to read the response - you just want to send it. To do that, you'll need to simplify your request such that it's a simple request. You can use { mode: 'no-cors' } on your fetch options to enforce this (but note that this doesn't change the rules, it just enforces that it's a simple request where you can't read the result).

**How can I fix my CORS error?**

To know exactly why your request is failing, you need to inspect the traffic itself, find where you're breaking the rules above, and then either:

* Change the request to make it a simple request
* Change the server's response to follow the rules above
* If all else fails, proxy the request through your own server on your own origin, so it's not a cross-origin request (proxying avoids the attacks above, because it doesn't let you use the cookies or authentication details from the user's browser, and it requires the target server to be accessible from your source server)

To inspect the traffic, you can use your browser built-in tools, but it's usually easier to use a dedicated HTTP debugger like [HTTP Toolkit](https://httptoolkit.tech/). Dedicated tools make it much easier to see the data, rather than (for example) Chrome's very cramped and fiddly network tab, and you can also breakpoint responses and edit the headers to test how the browser will handle changes without actually changing your server. Also, [some Chrome versions](https://httptoolkit.tech/blog/chrome-79-doesnt-show-cors-preflight) don't show all CORS requests.

Hopefully, once you examine your CORS requests & responses, it's clear where you're breaking the rules above.

If not, try walking through [Will It CORS](https://httptoolkit.tech/will-it-cors/). This is a self-explaining implementation of the CORS rules: you can input step by step what you're trying to do, and it'll tell you what will happen and why, and how you can change it.

There's also a few common mistakes that you should watch out for:

* Trying to request content from another origin that isn't explicitly available cross-origin. If its not your server, and it doesn't actively want CORS requests, you're not going to work around most issues: you need to proxy the request, ask the owner to allow it, or do something entirely different.
* Always returning \* for Access-Control-Allow-Origin, and then trying to send credentials.
* Adding CORS headers for preflight OPTIONS requests, but forgetting to also include CORS headers on the final request too.
* Unnecessarily sending custom request headers. This will trigger a preflight request. You can often get by just using the [CORS-safe request headers](https://developer.mozilla.org/en-US/docs/Glossary/CORS-safelisted_request_header) instead, or moving request data into the body of your request.
* Incorrectnyl caching CORS response headers independent of their origin, by not using Vary: Origin. If you do this then responses for requests from one origin may be cached and returned for later requests from a different origin. That mismatched data can quickly break things.
* Trying to access response headers without including an Access-Control-Expose-Headers header. In this case, all headers except the [CORS-safe response headers](https://developer.mozilla.org/en-US/docs/Glossary/CORS-safelisted_response_header) will be unexpectedly undefined, even though they were sent by the server.
* Sending cross-origin mixed-content requests (a request from https://... to http://...). These will always be blocked, regardless of the details, as insecure content like this is never allowed on HTTPS origins. There's not much you can do about this, other than changing to use HTTPS on both servers.

That covers the core of CORS, how it can go wrong, and how to fix it. Have more questions? Get in touch [on Twitter](https://twitter.com/pimterry).