

# Common no-cors misconceptions

Between tasks I spend a fair time on Stack Overflow, which is (truth be told) a love/hate relationship.

There's a few recurring questions that I feel deserve a longer explanation. Last time, I wrote about [how to use MySQL in Node.js](#), but today I wanted to write about the `fetch` `no-cors` setting.

## What is CORS?

Browsers have a 'sandbox'. This sandbox prevents a script on some domain to do things on other domains.

For example, if a script runs on `domain1.example.com`, it is not allowed to do a `PUT` request on `domain2.example.org`. It also can't embed an `<iframe>` from `domain2` and read its contents.

A script making a request to a different domain is very useful though, so workarounds were invented. Examples are server-side proxy scripts, using Flash and passing messages via `iframe` urls (and later on `postMessage`).

But then [CORS](#) came along, which offered a better solution. CORS allows `domain2` to say: "`domain1` is allowed to make requests".

The browser sandbox is not specific to exactly the domain. It's actually the combination of the scheme (`http` or `https`), the domain and the port. Those 3 parts combined are called the "origin".

So, this is the first popular misconception: CORS does not add security, it selectively removes it. CORS is strictly opt-in, so if an origin does not have support for CORS headers, the old behavior stays in effect.

## What is `no-cors`?

If you're trying to `fetch()` to another origin, and this origin does not opt into CORS, your request will return a network error and some messaging in the console.

If you don't control the API you are calling, you might be looking for alternative solutions, and you might run into `no-cors`:

```
const result = await fetch('https://domain2.example.org', {
  method: 'POST',
  mode: 'no-cors',
  headers: {
    'Content-Type': 'application/json',
  },
  body: JSON.stringify({ foo: 'bar' }),
});
```

If you see the `no-cors` setting, you might think that you are simply turning off the CORS check, and when trying `no-cors` it the first thing you might notice is that the request now actually shows up on your network tab!

However, you will quickly run into new problems. `no-cors` doesn't just turn off the check. It has greater consequences that make the request 'safe'.

These are the main effects:

- Any request header except: `Accept`, `Accept-Language`, `Content-Language`, `Content-Type` will be silently stripped from the request.
- If `Content-Type` is something other than `application/x-www-form-urlencoded`, `multipart/form-data` or `text/plain`, the header is ignored and set to `text/plain`.
- If the HTTP request is anything other than `GET`, `HEAD` or `POST`, you will get an error.
- You will not be able to read the response body, status or headers.

In other words, you can fire off a `GET`, `HEAD` or `POST` request, but won't know if it succeeded, failed, or what the response was, and if you `POST` you can only really encode the request body as a HTML form, or `text/plain`.

This makes the usefulness of `no-cors` pretty limited to a few cases. In fact, every case I've seen of people using `no-cors`, it was a mistake.

This is the second misconception: `no-cors` allows you to make certain cross-origin requests, but it severely limits what you can do. For almost every case, `no-cors` is not what you want.

## Why does `no-cors` limit to specifically those features?

Before `fetch` and `XMLHttpRequest`, it was already possible to make cross-domain requests with Javascript. The way to do this would be by creating a HTML form, and calling `.submit()`. With HTML forms it's also not possible for Javascript to see the response or use custom request bodies.

Because that feature existed before Javascript and before the need for sandboxing, there was no need for an extra security layer to add this feature to `fetch()`. The world was already prepared for browsers doing these requests.

The one exception to this is perhaps `text/plain` requests, which (as far as I know) was a newly lifted restriction that was deemed to not be risky.

## So how do I fix my problem?

There's 2 main solutions:





1. Add CORS support to the API you are using. This only works if you have control over the target.
2. Instead of making the request from your domain, something else needs to make the request for you.

If you can't add CORS headers, you will likely want to build a small server-side script that can make these requests on your behalf.

Instead of calling the target directly, your script can now call your script, which has to do the request for you server-side.

# Web mentions

Reposts: 

Likes:    



[La semana PHP](#) • March 31 2020

Las ideas erróneas más comunes sobre no-cors [evertpot.com/no-cors/](https://evertpot.com/no-cors/) vía [@evertp](#)

---



[Astrolabit](#) • April 7 2020

Common no-cors misconceptions [evertpot.com/no-cors/](https://evertpot.com/no-cors/)

---