# Cross Origin Resource Sharing

In this lesson, we'll look at cross origin resource sharing.

**We'll cover the following**

## Same origin requests

On the browser HTTP requests can only be triggered across the same origin through JavaScript. Simply put, an AJAX request from `example.com` can only connect to `example.com`.

This is because your browser contains useful information for an attacker, cookies, which are generally used to keep track of the user's session. Imagine if an attacker would set up a malicious page at `win-a-hummer.com` that immediately triggers an AJAX request to `your-bank.com`. If you're logged in on the bank's website, the attacker would then be able to execute HTTP requests with your credentials, potentially stealing your information or, worse, wiping your bank account out.

## Cross origin resource sharing directives

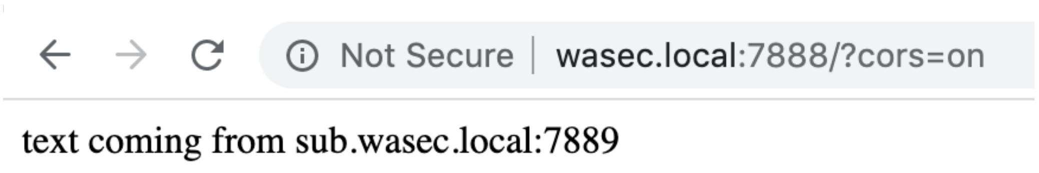
There might be some cases, though, that require you to execute cross-origin AJAX requests, and that is why browsers implement Cross Origin Resource Sharing (CORS), a set of directives that allow you to execute cross-domain requests.

The mechanics behind CORS are quite complex, and it wouldn't be practical for us to go over the whole specification, so I am going to focus on a stripped down version of CORS. All you need to know for now is that by using the `Access-Control-Allow-Origin` header, your application tells the browser that it's ok to receive requests from other origins.

The most relaxed form of this header is `Access-Control-Allow-Origin: *`, which allows any origin to access our application, but we can restrict it by simply adding the URL we want to whitelist with `Access-Control-Allow-Origin: https://example.com`.

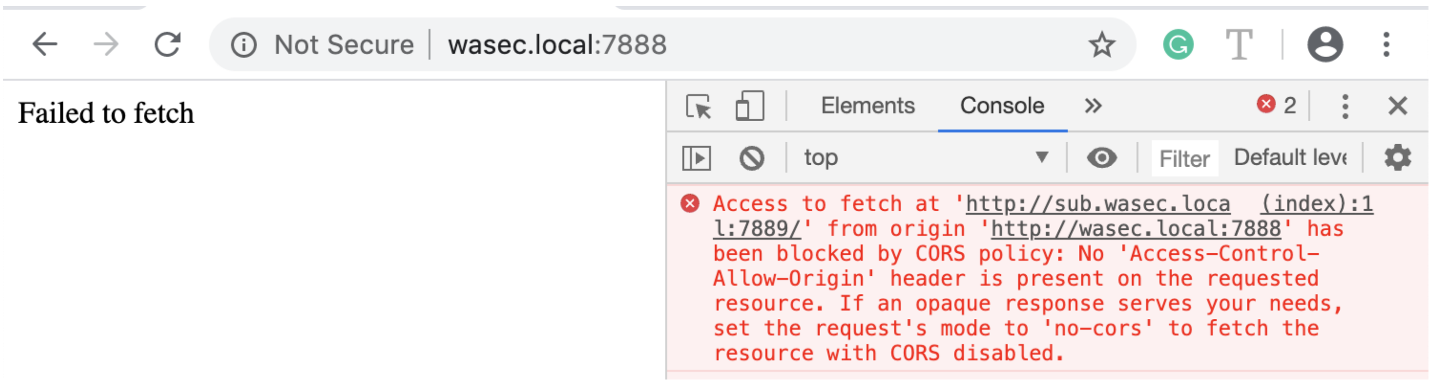
## Example #

If we take a look at the example at github.com/odino/wasec/tree/master/cors we can clearly see how the browser prevents access to a resource on a separate origin. I have set up the example to make an AJAX request from `wasec.local` to `sub.wasec.local`, and print the result of the operation to the browser. When the server behind `sub.wasec.local` is instructed to use CORS, the page works as you would expect. Try navigating to `http://wasec.local:7888/?cors=on`.



text coming from sub.wasec.local:7889

But when we remove the `cors` parameter from the URL, the browser intervenes and prevents us from accessing the content of the response.

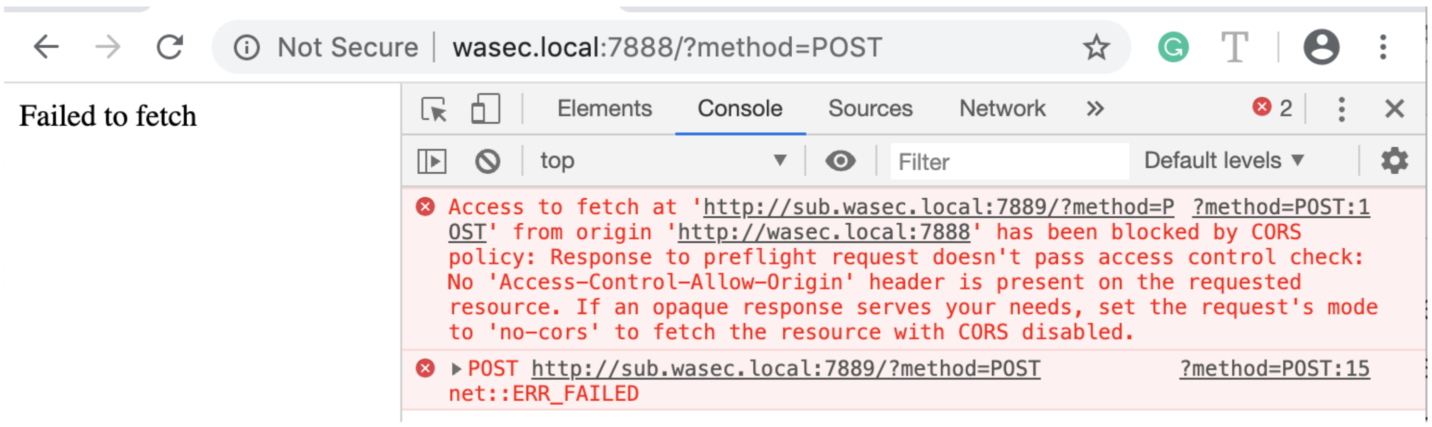


Failed to fetch

```
Access to fetch at 'http://sub.wasec.loca   (index):1
l:7889/' from origin 'http://wasec.local:7888' has
been blocked by CORS policy: No 'Access-Control-
Allow-Origin' header is present on the requested
resource. If an opaque response serves your needs,
set the request's mode to 'no-cors' to fetch the
resource with CORS disabled.
```

An important aspect we need to understand is that the browser executed the request but prevented the client from accessing it. This is extremely important, as it still leaves us vulnerable if our request would have triggered any side effect on the server. Imagine, for example, if our bank would allow us to transfer money by simply calling the url `my-bank.com/transfer?amount=1000&from=me&to=attacker`, that would be a disaster!

As we saw at the beginning of this chapter, `GET` requests are supposed to be idempotent, but what would happen if we tried triggering a `POST` request? Luckily, I’ve included this scenario in the example, so we can try it by navigating to `http://wasec.local:7888/?method=POST`:



```
Not Secure | wasec.local:7888/?method=POST

Failed to fetch

Access to fetch at 'http://sub.wasec.local:7889/?method=P  ?method=POST:1
OST' from origin 'http://wasec.local:7888' has been blocked by CORS
policy: Response to preflight request doesn't pass access control check:
No 'Access-Control-Allow-Origin' header is present on the requested
resource. If an opaque response serves your needs, set the request's mode
to 'no-cors' to fetch the resource with CORS disabled.
▶POST http://sub.wasec.local:7889/?method=POST          ?method=POST:15
net::ERR_FAILED
```

Response to a POST request

Instead of directly executing our `POST` request, which could potentially cause some serious trouble on the server, the browser sent a preflight request. This is nothing but an `OPTIONS` request to the server, asking it to validate whether our origin is allowed. In this case, the server did not respond positively, so the browser stops the process, and our `POST` request never reaches the target.

This tells us a couple things:

- CORS is not a simple specification. There are quite a few scenarios to keep in mind and you can easily get tangled in the nuances of features such as preflight requests.
- Never expose APIs that change state via `GET`. An attacker can trigger those requests without a preflight request, meaning there’s no protection at all.

I will conclude my overview of this feature here. If you’re interested in understanding CORS in-depth, MDN has a lengthy article that brilliantly covers the whole specification at developer.mozilla.org/en-US/docs/Web/HTTP/CORS.

> ### 🔍 CORS vs proxies
>
> Out of experience, I found myself more comfortable with setting up proxies that can forward the request to the correct server, on the backend rather than using CORS. This means that your application running at `example.com` can set up a proxy at `example.com/_proxy/other.com`, so that all requests falling under

> `_proxy/other.com/*` get proxied to `other.com`.

---

In the next lesson, we’ll study `X-Permitted-Cross-Domain-Policies` and `Referrer-Policy`.