PerformantCode.com (http://performantcode.com/)

Grzegorz Mirek (http://performantcode.com/author/admin/)
Sep 16, 2018 · 10 min read

Do You Really Know CORS?

Cross-Origin Resource Sharing

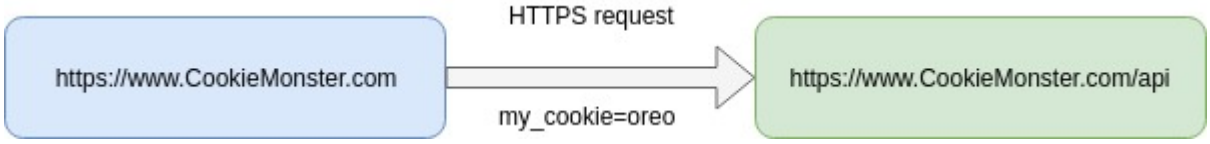
No 'Access-Control-Allow-Origin' header is present on the requested resource. Origin http://www.sesamestreet.com (http://xyz.com:8080) *is therefore not allowed access.*

If you work with a frontend sometimes, the chances are that you've seen the error above before. When it had happened to you for the first time, like any proper developer does, you googled it. As a result, you have probably seen some advice on StackOverflow to include `Access-Control-Allow-Origin` in your server's response and then, you can happily return to your code.

Surprisingly, this is the end of the experience with Cross-Origin Resource Sharing (CORS) for many developers. They know how to fix the problem, but they don't always understand why the problem exists in the first place. In this article, we will dive deeper into this topic, trying to understand what problem CORS really solves. However, we will start with the Same-Origin Policy (SOP) concept.

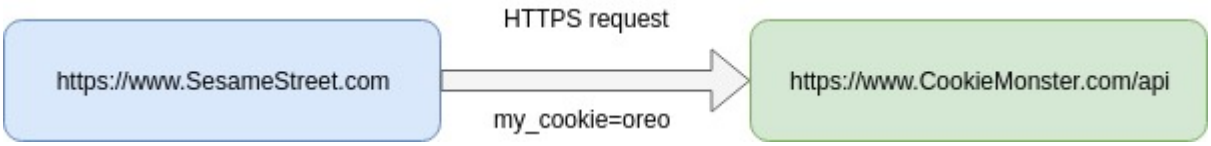
What Is the Same-Origin Policy?

SOP is a security mechanism implemented in almost all of the modern browsers. It does not allow documents or scripts loaded from one origin to access resources from other origins. To understand why it's so critical, it's important to realize that for any HTTP request to a particular domain, browsers automatically attach any cookies bounded to that domain. Let's imagine a cookie `my_cookie=oreo` which is stored for the domain CookieMonster.com.

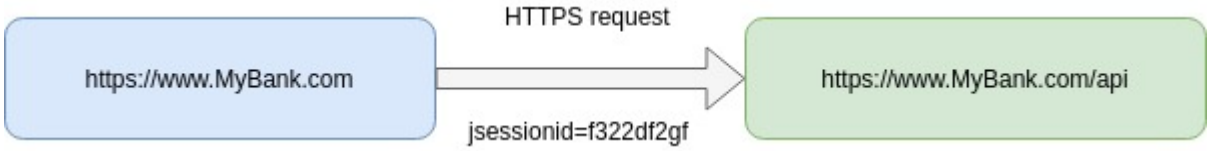


CookieMonster.com is a single-page application that uses REST API exposed at CookieMonster.com/api. Thus, for every HTTP call to the API, `my_cookie=oreo` will be attached to the request.

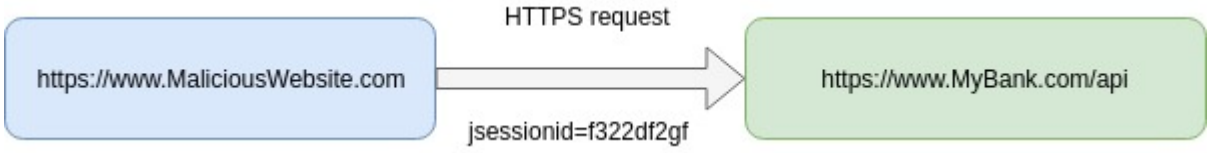
Without Same-Origin Policy, the following scenario would be possible:



SesameStreet.com could reuse the user's cookie which was previously stored for CookieMonster.com. Sounds scary? Not yet? Let's look at the more thought-provoking scenario:



and then:



Without SOP, MaliciousWebsite.com would be able to send a request to MyBank.com/api using the session cookie stored for MyBank.com. Why is that? Because, as previously mentioned, browsers automatically attach cookies bounded to a destination domain. It doesn't matter if a request originated from MyBank.com or MaliciousWebsite.com. As long as a request goes to MyBank.com, the cookies stored for MyBank.com would be used. As you can see, without Same-Origin Policy, Cross-Site Request Forgery (CSRF) attack can be relatively simple – assuming that an authentication is based solely on a session cookie, as opposed to a token-based authentication. That's one of the reasons the SOP was introduced.

Having said that, it has always been possible for the browser to make cross-origin requests by specifying a resource from a foreign domain in the `<img>`, `<script>` or `<iframe>` tag – and if applicable, cookies might be attached. The crucial difference is that AJAX call is fired from the JavaScript code, which has total control and can be potentially dangerous. On the other hand, tags are in the control of the browser, and no JS code can intercept HTTP requests that they trigger.

What is Origin?

In the previous paragraph, I used domains as an example. But Same-Origin Policy applies to origins. How is an origin defined? Two origins are considered to be equal if they have the same protocol, host, and port – sometimes referred to as

"scheme/host/port tuple". It explains why we see this error – even if both our backend and frontend run locally – they use different ports, and thus, they have different origins.

What Is CORS?

Even though some people call CORS a security mechanism, it's actually the opposite. It's a way to relax security, make it less restrictive. SOP is implemented in almost all modern browsers and because of that, a website from one origin is not allowed to access resources from foreign origins. CORS is a mechanism to make it possible.

How Does CORS Work?

CORS defines two types of requests: simple requests and preflighted requests.

Simple Requests

To put it simply, a simple request is the one that doesn't trigger the preflight request. A request doesn't trigger preflight request if it meets all of the following conditions:

- uses GET, HEAD or POST method
- doesn't have headers other than the small subset defined in the specification (https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS#Simple_requests) (any custom or Authorization header breaks this condition)
- the only allowed values for Content-Type header are `application/x-www-form-urlencoded`, `multipart/form-data`, `text/plain` (`application/json` breaks this condition)

A typical scenario would be:

1. SesameStreet.com is opened in a browser tab. It initiates AJAX request (using XMLHttpRequest or Fetch API) to GET CookieMonster.com/api/monsters

2. The browser notices that this is a cross-origin request, and attaches Origin request header:

```
1 GET api/monsters HTTP/1.1
2 Host: cookiemonster.com
3 Origin: https://www.sesamestreet.com
4 ...
```

3. The CORS-configured server checks the Origin header and if the origin is allowed, then it sets the `Access-Control-Allow-Origin` header to the Origin value:

```
1 HTTP/1.1 200 OK
2 Access-Control-Allow-Origin: https://www.sesamestreet.com
3 ...
```

4. When the response reaches the browser, the browser verifies if the value under `Access-Control-Allow-Origin` header matches the origin of the tab the request originated from.

Preflighted Requests

As described earlier, even adding a header such as Authorization causes a request to be preflighted. A request is preflighted if a browser first sends an additional preliminary OPTIONS request ("preflight request") in order to determine whether the actual request ("preflighted request") is safe to send. Let's look at the typical flow where we want to create a new monster resource:

1. A browser tab with SesameStreet.com makes AJAX POST request to CookieMonster.com/api/monsters with a JSON payload using POST method. The browser knows that sending POST with Content-Type different from `application/x-www-form-urlencoded`, `multipart/form-data`, `text/plain` has to be preflighted, so it sends OPTIONS request with three additional parameters:

- Origin – this one we already know
- Access-Control-Request-Method – HTTP method of the main (preflighted) request

- Access-Control-Request-Headers – HTTP headers of the main (preflighted) request

```
1 OPTIONS /api/monsters HTTP/1.1
2 Host: cookiemonster.com
3 Origin: https://www.sesamestreet.com
4 Access-Control-Request-Method: POST
5 Access-Control-Request-Headers: Content-Type
6 ...
```

2. The server responds with the allowed origin, methods and headers.

```
1 HTTP/1.1 200 OK
2 Access-Control-Allow-Origin: https://www.sesamestreet.com
3 Access-Control-Allow-Methods: POST, GET, OPTIONS
4 Access-Control-Allow-Headers: Content-Type
```

3. If the origin is allowed and if the HTTP method and headers of the main request are on the list returned by the server, the main request can be sent. This will be a regular cross-origin request, so it will include Origin header and the response will contain `Access-Control-Allow-Origin` once again.

Performance note: sending a preflight request every time can be a performance overhead. This can be mitigated by caching preflight requests using Access-Control-Max-Age response header.

Common Misconception About CORS

At the first glance, CORS configuration on a server side looks like some sort of ACL (Access Control List) – a server returns the origin that it accepts requests from. The only way to access a resource is to send a request from the origin whitelisted by a server, right? Not really. Remember that HTTP isn't used only by browsers and you can send an HTTP request from any client like curl, Postman, and so on. If you prepare a custom HTTP request in those tools, you can put any Origin header you want. You can also skip it and a server usually returns a correct result anyway. Why is that? Because as I mentioned earlier, Same-Origin Policy is a concept implemented in browsers. Other tools or software components don't care about it that much.

There is a simple implication based on what you've just read. If there is a third-party API which you want your webpage to consume, but the API returns Access-Control-Allow-Origin header set to an origin other than your own, then you can circumvent that problem by setting up a proxy server. Why? Because as described above, SOP doesn't apply to server-to-server communication, only to browser-to-server one.

Is CORS Safe?

The most important question – is the CSRF scenario from the beginning of this article possible using CORS? The answer is that it depends. By default, **CORS does not include cookies into cross-origin requests**. This is different from older cross-origin techniques, such as JSON-P (JSON with Padding). This behavior greatly reduces CSRF vulnerabilities. However, if you really want to send cookies in your request, you can explicitly permit that. This requires coordination between both the client and the server side. Your website must set `withCredentials` property on the XMLHttpRequest to `true` (or `credentials` property in Fetch API set to `include`). Additionally, the server must respond with `Access-Control-Allow-Credentials` header set to `true`. With this combination, both parties agree to use credentials when sending a cross-origin request. Credentials are cookies, authorization headers or TLS client certificates. Thankfully, there is one security measure that prevents an excessive exhibitionism – the following combination won't work:

```
1 Access-Control-Allow-Credentials: true
2 Access-Control-Allow-Origin: *
```

If you allow credentials to be sent, then `Access-Control-Allow-Origin` cannot be set to the wildcard. The server must return an explicit origin that is allowed to access the resource. The bad news is that many servers blindly generate the `Access-Control-Allow-Origin` header based on the Origin value from the user's request. If that's the case, using `Access-Control-Allow-Credentials` set to `true` can be a serious security hole.

browser (http://performantcode.com/tag/browser/) cors (http://performantcode.com/tag/cors/)

fullstack (http://performantcode.com/tag/fullstack/) security (http://performantcode.com/tag/security/)

web (http://performantcode.com/tag/web/)

8 responses to “Do You Really Know CORS?”

Sagar Pathak (http://sagarpathak.com.np) says:

October 4, 2018 at 7:54 am (http://performantcode.com/web/do-you-really-know-cors/#comment-17)

Well written. Great post!!

Reply (http://performantcode.com/web/do-you-really-know-cors?replytocom=17#respond)

Paul says:

October 4, 2018 at 9:33 am (http://performantcode.com/web/do-you-really-know-cors/#comment-18)

The best article I have read on this topic.

Reply (http://performantcode.com/web/do-you-really-know-cors?replytocom=18#respond)

TiWu says:

October 4, 2018 at 10:49 am (http://performantcode.com/web/do-you-really-know-cors/#comment-19)

Nice article!
You say: “This can be mitigated by caching preflight requests using Access-Control-Max-Age response header.”

Could you elaborate on that? That would be something that we do on the server side?

Reply (http://performantcode.com/web/do-you-really-know-cors?replytocom=19#respond)

Author Grzegorz Mirek says:

October 4, 2018 at 10:08 pm (http://performantcode.com/web/do-you-really-know-cors/#comment-25)

Thank you! We would need to configure the server to return this header in the response to a preflight request – so the result can be cached in the browser.

Reply (http://performantcode.com/web/do-you-really-know-cors?replytocom=25#respond)

Nelson says:

October 4, 2018 at 12:11 pm (http://performantcode.com/web/do-you-really-know-cors/#comment-21)

Superb explanation of CORS. I wish all tech articles were written this well.

Reply (http://performantcode.com/web/do-you-really-know-cors?replytocom=21#respond)

Noel says:

October 4, 2018 at 2:00 pm (http://performantcode.com/web/do-you-really-know-cors/#comment-22)

Great explanation! Thank you very much, I really identified myself as the type of developer that always fixed the CORS problem but never really cared about it that much. 😀

Reply (http://performantcode.com/web/do-you-really-know-cors?replytocom=22#respond)

Michal says:

October 4, 2018 at 3:57 pm (http://performantcode.com/web/do-you-really-know-cors/#comment-23)

Very nice, thank you 🙂

Reply (http://performantcode.com/web/do-you-really-know-cors?replytocom=23#respond)

Andrew (http://andrewfschorr.com) says:

October 5, 2018 at 12:14 am (http://performantcode.com/web/do-you-really-know-cors/#comment-26)

Here's one question that's always bugged me – What's stopping a malicious user from sending an HTTP request from any API client like Postman, or even Curl from the CL? Something like a post with: {transferTo: myAccountId, amount: 1000000000}?

Obviously in any nontrivial web app it would fail because of authentication issues, but if a server doesn't do ANY sort of security checking, that should work, no? Does that mean that the onus is on the server developer of mybank.com? And if so, what would stop the malicious request from working on any server developed before the existence of CORS?

Reply (http://performantcode.com/web/do-you-really-know-cors?replytocom=26#respond)

Leave a Reply

Your email address will not be published. Required fields are marked *

COMMENT

NAME *

EMAIL *

WEBSITE

POST COMMENT