Lesson Monday

Intermediate JavaScript (/intermediate-javascript) / Asynchrony and APIs (/intermediate-javascript/asynchrony-and-apis) / SOP and CORS

Text

We don't talk about web application security much at Epicodus because it's beyond the scope of what we teach. However, there is one important topic we need to cover briefly now that we are working with API calls.

Same-Origin Policy

Browsers use **same-origin policy (SOP)** to prevent cross-site scripting attacks. A cross-site scripting attack (https://owasp.org/www-community/attacks/xss/) is when a malicious user attempts to access another site via the browser. This generally involves injecting malicious scripts into a web application in an attempt to gain access, get data, or sabotage a site.

Same-origin policy means that a request can only be made from one URL to another if the receiver and the sender have the same **protocol**, **host**, and **port**.

Here's a quick example:

- https://thisisthehost.com/somepage

In the URL above, `https` is the **protocol** and `thisisthehost.com` is the **host**. There is no **port**, and generally there won't be when we are navigating between pages, but a port would look something like this `:8080` and would come right after the host.

Let's say that the following URL wants to make a request of the URL above:

- https://thisisthehost.com/someotherpage

That would be entirely fine. The protocol and host are the same. The only thing different is the **path**, which is `someotherpage` instead of `somepage`, and SOP does not care about the path being different.

How about the following URL?

- http://somebodysuspicious.com

Well, the host name should give it away, but this is a different origin for two reasons. There's a different protocol (`http` instead of `https`) *and* there's a different host: `somebodysuspicious.com`. Because of SOP, a browser will not allow this URL to make a request to `https://thisisthehost.com/somepage`.

How about this URL?

- http://thisisthehost.com/someotherpage

This request will also fail. Even though the host name is the same, the `http` protocol is different.

So why is this important when it comes to making API calls?

Well, our applications are entirely client-side, which means that all the code we are running is running in the browser. This means that SOP applies to all of our applications.

But wait a minute... How have we been able to make API calls to the OpenWeather API then? How about other APIs we are working with like Giphy?

Well, these APIs have enabled a feature called **cross-origin resource sharing** or **CORS** for short. The name is pretty self-explanatory. It's a mechanism that allows resources to be shared across different origins.

Cross-Origin Resource Sharing

It's not really necessary to know the ins and outs of how cross-origin resource sharing works, but on the most basic level, it means attaching a few extra headers to requests and responses. That all happens behind the scenes so we don't need to worry about it.

What matters for us is that the APIs we work with must have CORS enabled if we want to work with them. If an API doesn't have CORS, we can't make API calls from a browser application.

For that reason, if you want to build a client-side application that makes API calls, you need to do some research first and make sure that CORS is allowed. It's not enough to make sure an API call works from Postman. In fact, that doesn't tell us anything about whether it works from the browser! **Postman isn't making calls from the browser so the rules of SOP don't apply.**

If you try to make a request from the browser to an API call that doesn't allow CORS, you'll get the following error in the console:

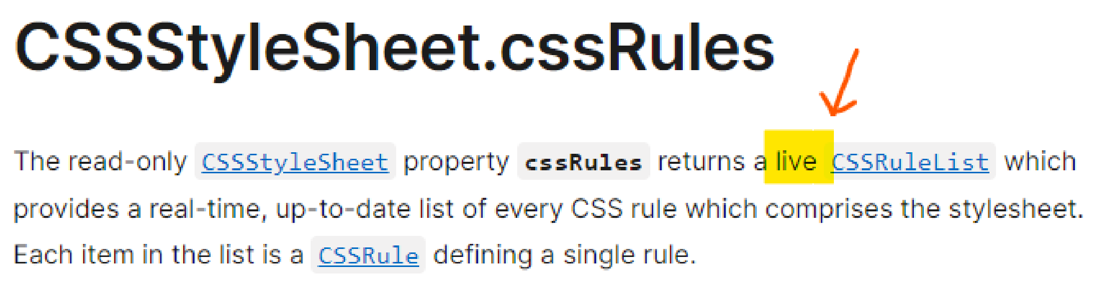
```
Access to fetch at 'https://othersite.com' from origin
'https://mysite.com' has been blocked by CORS policy: No
'Access-Control-Allow-Origin' header is present on the
requested resource.
```

Note that the names of the sites will be different depending on the URL you are trying to access and the one you are working with.

SOP Applies to Certain Web APIs

We've talked about the Same Origin Policy (SOP) with regards to making API calls, but this policy is in place for any request for resources from a browser. This includes requests that happen implicitly by certain Web APIs that provide live-updated data. When a Web API provides live updated data, it means that the Web API is making regular calls (or "requests") to our application to have the most up-to-date data.

We saw an example of a Web API that provides live-updated data in the optional lesson on how to access our project's stylesheets via the CSS Object Model (CSSOM). The Web API that provides live-updated data is the `CSSStyleSheet.cssRules` property (https://developer.mozilla.org/en-US/docs/Web/API/CSSStyleSheet/cssRules), which provides a "live" list of all CSS rules in a given stylesheet.



CSSStyleSheet.cssRules

The read-only `CSSStyleSheet` property `cssRules` returns a live `CSSRuleList` which provides a real-time, up-to-date list of every CSS rule which comprises the stylesheet. Each item in the list is a `CSSRule` defining a single rule.

If you did not read the lesson Optional: Accessing Stylesheets in the CSSOM (https://www.learnhowtoprogram.com/lessons/optional-accessing-stylesheets-in-the-cssom), don't worry. We don't need to know how the CSSOM works (or about its object types) in order to understand the implication of live data. So, let's continue.

Anytime a Web API provides live data, the Web API does so by making regular requests to get updated information. This means if we don't serve our project from localhost, we'll run into errors.

That's because SOP will fail a request to a project that's opened in the browser, but not served. When we open a project in the browser, like by dragging and dropping our `index.html` into the browser, the URL will look something like this: `file:///C:/Users/staff/Desktop/oop-address-book-v2/index.html`. A URL that starts with `file:///` or `C:/Users/...` indicates that we've simply opened a local file in the browser from a Windows computer, and we're not serving the project with a web server. Notably, there's no domain, protocol, or port in the URL, which makes any request to this location automatically fail under SOP.

Remember, according to SOP a request from a browser needs to have a protocol, domain, and port, and whatever resource returned from the request also needs to have that same and matching information. If not, there needs to be a CORS policy in place. However, setting up a CORS policy is not the solution here, since the issue is caused by there being no protocol, domain, or port at all! Adding CORS would do nothing.

Instead, for any Web API that returns live data (and is therefore making regular requests to ensure that data is up-to-date), we need to make sure we're serving our project for the Web API to be able to function. It's as easy as that! Again, there's o need for CORS.

Working with APIs That Don't Allow CORS

Let's say you *really* want to work with an API that doesn't allow CORS. Is there any way around this restriction? Well, there are several options:

- **Make the API call server-side instead of client-side.** If you are continuing on to Ruby/Rails or C#/.NET, you'll learn how to make API calls server-side. Wait until then to work with an API that doesn't allow CORS.
- **Build a proxy.** This is well beyond the scope of the course and we don't recommend it unless you want a challenge. But you can use Node or another option to build a proxy server to

allow requests to servers that don't allow CORS. This option is only listed to demonstrate that it's hard to do a workaround the right way! To learn more about proxy servers, visit this MDN documentation (https://developer.mozilla.org/en-US/docs/Web/HTTP/Proxy_servers_and_tunneling).

- **Use a Chrome extension to alter the header.** Finally, there are several Chrome extensions you can use to work around CORS including the Moesif Origin & CORS Changer extension (https://chrome.google.com/webstore/detail/moesif-origin-cors-change/digfbfaphojjndkpccljibejjbppifbc?hl=en-US) or Allow CORS: Access-Control-Allow-Origin extension (https://chrome.google.com/webstore/detail/allow-cors-access-control/lhobafahddgcelffkeicbaginigeejlf?hl=en). They work by changing the header on your requests. They are easy to install and work well, but *only* on the local machine where the Chrome extension is installed. So while they will work fine for a learning project, they aren't good solutions for anything you might eventually want to deploy — or for a portfolio project you'd want to share or send to a potential employer. While you could technically add instructions for installing the extension in your README, that's just not going to look very professional.

So while you are welcome to use one of these extensions, just keep that fact in mind. We are still early in the program so you may not plan to turn learning projects into portfolio projects. That being said, we still recommend you pick APIs that allow CORS.

Previous (/intermediate-javascript/asynchrony-and-apis/separating-promise-logic)
Next (/intermediate-javascript/asynchrony-and-apis/speaking-up)
Lesson 21 of 33
Last updated more than 3 months ago.

disable dark mode

Epicodus (http://www.epicodus.com)