Same-origin policy and CORS

Same-origin policy

- The same-origin policy has been implemented in browsers to protect against malicious websites
- Imagine you have your online banking in one tab in your browser, and a (second) webpage in another tab. What you don't know is that the second webpage is malicious and manage to read the content of a session-cookie from your bank. The malicious webpage can then contact the bank with your credentials (from the cookie) and transfer all your money to another bank account.
- To prevent this, the browser will deny fetch-calls (AJAX/XMLHttpRequest) to servers that **aren't in the same domain** as the origin of the webpage. For example, JavaScript in a webpage that originates from *mysite.com*, can only contact servers at *mysite.com*

Same-origin policy continued

- The browser will also block communication between origins using different **different protocols or ports**. E.g., if your webpage is «run» on the local filesystem (file://…) it can't access data on a site using http://.
- Usually, the webpages for the client will be on the same same domain as your webserver/webservice – using the same protocols (http://…) and ports. Therefore, the same-origin policy, will not be a problem.
- Also note that the same-origin policy only applies to **fetch (AJAX/XMLHttpRequest)** calls. I.e., when you are using JavaScript to access content on another domain.

Handle the same-origin policy

- Sometimes the same-origin policy can cause problems, e.g.:
  - You want to create a webservice/server that should be open to all (or some) domains – for example, a webservice for news or weather.
  - You want an app that runs locally (in the file-system) to access a server on the Internet.
- To solve this, we can handle the same-origin policy in two ways. (Note that it is the browser (client) that blocks access to a server):
  1. **Configuring the browser**: specify that the browser should allow requests to one, more, or any domain/origin. This requires that (as a developer) you have control over the browser.
  2. **Configuring the server**: specify that the server should send specific headers in the response that will make the browser allow the webpage to access the data from the server. This requires that (as a developer) you have control over the server.

1. Same-origin policy: configuring the client application

- It is possible to turn off the same-origin policy in the browser, but you can't expect a common user to do that.
- The solution is to publish the webapp as a native or hybrid app. Then *you* have control over the "browser" and can turn it off.

("Downgrading" the security like this will not represent a risk in the same way as in a browser, but of course, installing an app that can't be trusted, will always be a security risk for the user)

2. Same-origin policy: configuring the server (CORS)

- You can configure the server to send a response with specific headers so that the browser will allow the webpage to access the server. This is called *Cross-Origin Resource Sharing* (CORS).
  - Note: servers that requires a high level of security, e.g., your online bank – will typically **not** enable CORS.
- The headers that determines if the webpage can access the data (resource) is Access-Control-Allow-*property*. For example, if you want webpages on any domain/origin to access the data on the server, you could write (in node.js):

```
response.set("Access-Control-Allow-Origin", "*");
```

- **To handle CORS on the server it is possible (and easier) to use a dedicated package (e.g., the *cors*-package in node express).**

CORS: simple- and preflight requests (background reading)

- When the browser find that the webpage is sending a request to another domain (or other protocol or port), the browser will handle the request using one of two different methods: *simple* or *preflight*.
- Simple requests must meet the following rules:
  - The only methods that is allowed in the request are GET, POST and HEAD
  - Only certain headers and content types are allowed
  - No event handlers are used on any upload-object or any readable stream objects are used in the request.
- When a request doesn't meet these requirements, the browser will first send a "test"-request (*preflight*) to the server with the OPTION method. The browser will then send the original request if the response from the preflight contains headers that meets the requirements.
- Regardless if the request is handled as simple or preflight, the server must respond with an *Access-Control-Allow-Origin* header. If not, the browser will not allow access to data from the server.
- If the request is handled as preflight, additional Access-Control-Allow-*property* -headers must be added to the response. Again, if not, the browser will not allow access to data from the server.

CORS: simple- and preflight requests cont. (background reading)

- For example, if the webpage sends a request with the method DELETE, the browser will send a preflight request and the server must add a header in the response that allows a DELETE-method.
- Or, if the webpage sends a request with a header "Content-Type" : "application/json", the server must add a header in the response that allows Content-Type headers, e.g.:

```
response.set("Access-Control-Allow-Origin", "*");
response.set("Access-Control-Allow-Methods", "GET, PUT, POST, DELETE");
response.set("Access-Control-Allow-Headers", "x-access-token, content-type");
```

- Note 1: **`"Access-Control-Allow-Headers", "*"`** will allow all headers.
- Note 2: Be aware that the restrictions (certain methods and headers) applies only when you are using CORS, i.e. the webpage (client) and server are on different domains (or protocols/ports). If not, you don't have to add any Access-Control-Allow-*property* – headers in the response.
- Note 3: In node you can handle CORS manually by setting headers in the response (as shown above), or you can install a node package that handles CORS, e.g. *cors* (recommended).