<https://www.youtube.com/watch?v=2PPSXonhIck>

**What is session ?**

Session means particular interval of time. http protocol is stateless , so to maintain the state of the user HttpSession object is introduced.

* User first login or enter into the system , server creates a session for the user.
* Server saves information related to the session in its memory or database and make it identifiable by a jsessionid.
* In response server sends the jsessionid to the user or browser.
* Next time if user enters again into the application it sends the jsessionid in request and server finds the information related to the particular session id.
* Session id can passed as a query parameter or inside the cookie.
* When the browser is closed or the session timeout happen, session object is deleted from the server memory, if that time browser request with the old session id it will not be recognized by the server.

**What is cookie ?**

Cookies are pieces of information stored on the client side, which are sent to the server with every request made by the client. Cookies are primarily used for authentication and maintaining sessions. Like Session cookie is also created by server side, and set into the response object. but stored at client side.

Whenever the user or browser first request the server the server set cookie with the response.

Next time onwards browser send the same cookie to the server. Server read the cookie and recognize the browser. Often cookies are used for authentication.

There are some attribute for cookie

* Secure : One of the simplest and most common ways to steal data, including cookies, is *sniffing*. Sniffing can be defined as passively reading data that is being transmitted. The 'Secure' attribute makes sure that the cookie will only be sent with requests made over an encrypted connection and an attacker won't be able to steal cookies by sniffing. However, we need to be very careful while setting this attribute. Just setting the attribute to 'Secure' does not necessarily mean that the cookie will always be transmitted over an encrypted connection.
* Domain :

The '**domain**' attribute signifies the domain for which the cookie is valid and can be submitted with every request for this domain or its subdomains. If this attribute is not specified, then the hostname of the originating server is used as the default value.

if the domain is [www.a.com](http://www.a.com), then [www.b.a.com](http://www.b.a.com) is allowed to send cookie to the server, but [www.b.com](http://www.b.com) is not allowd.

* Path :

If the domain has a subfolder a.com/ blog1 then path = “/blog1” means the server accepts only the cookies sent from this sub folder.

* HTTPOnly :

When this attribute is set, client-side scripts are not allowed to access the cookie. Now, the question that arises is, 'Why do I need to safeguard my cookies from client-side scripts? the answer is xss or Cross side scripting. xss steals the cookie information by client side scripting. If the httponly : true then xss cannot read any data from cookie, the creator of the domain can only extract data from the cookie.

* Expires : This attribute is used to set persistent cookies. It signifies how long the browser should use the persistent cookie and when the cookie should be deleted. If this attribute is not specified, then the lifetime of the cookie is the same as that of browser session, i.e. it will be a non-persistent cookie.

**Effect of CORS ?**

Cross-Origin Resource Sharing ([CORS](https://developer.mozilla.org/en-US/docs/Glossary/CORS)) is a mechanism that uses additional [HTTP](https://developer.mozilla.org/en-US/docs/Glossary/HTTP) headers to tell a browser to let a web application running at one origin (domain) have permission to access selected resources from a server at a different origin.

An example of a cross-origin request: The frontend JavaScript code for a web application served from http://domain-a.com uses [XMLHttpRequest](https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest) to make a request for http://api.domain-b.com/data.json.

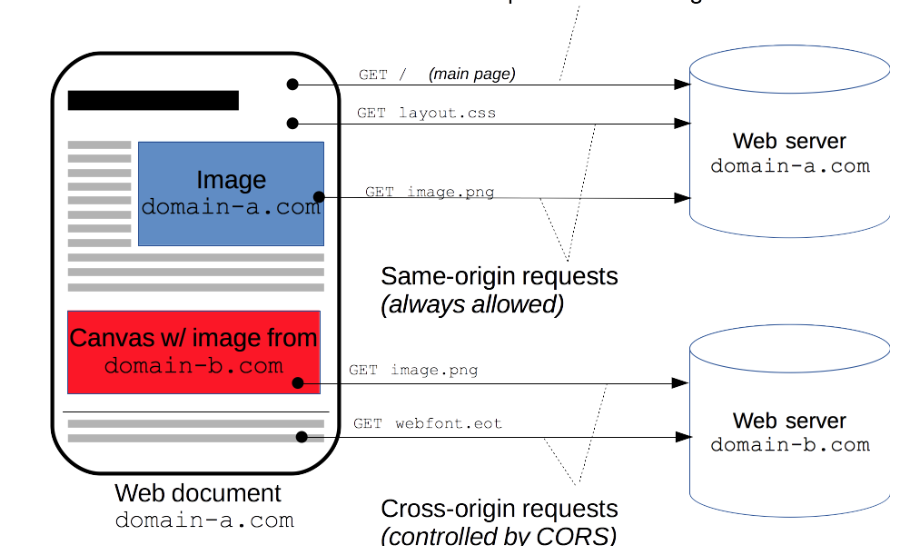
For security reasons, browsers restrict cross-origin HTTP requests initiated from within scripts. For example, XMLHttpRequest and the [Fetch API](https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API) follow the [same-origin policy](https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy)

the headers are like “access-control-allow origin “ we can define “\*” or the specific domain list.

We can also define http methods and tokens which are allowed in cross origin.

**Why same origin policy is applied ?**

A server by default does not allow cross origin request and response.



The main reason is CSRF or cross site request forgery attack.

lets define it by a scenario

1. The victim is already logged in to a site and get the cookie from from the server in response. Suppose the cookie is not readable by xss.
2. attacker sends a link to user’s email address and user clicks on the link. In onload event attacker forces the user to call the already logged in server, which is hidden from the user.
3. When user calls the server it automatically sends the cookie to the server again, but this time the hidden script is written in such a way that it can modify the server data, and server cannot recognize the user as it is ending the valid cookie.
4. To solve this problem we use same origin policy. As the request is coming from different origin the server does not allow the request itself.
5. But same origin policy has some draw back, It is not allowing attacker origin but with that it is blocking all other origins as well, that’s why CORS is introduced to relax the same origin policy for some specific origin.
6. But if we define access-control-allow origin as \*, then it will do the same thing.
7. Some of the headers for cors are

Access-Control-Allow-Methods: GET, POST

Access-Control-Allow-Credentials: true

Access-Control-Allow-Origin: <http://origin.foo>

Access-Control-Allow-Headers: Content-Type, \*

**Disadvantages of cookie and session in distributed system or microservices for authentication**

<https://medium.com/tech-tajawal/microservice-authentication-and-authorization-solutions-e0e5e74b248a>

**What is JWT token ?**

Json webtoken contains all the user information and not stored at server side. Every time client request to server it sends the jwt token to the server, the server extracts user information from the token. The server does not need to save anything in its memory. So in distributed system it is very useful.

* Tokens are not stored server side.(stateless)
* signed with a secret against tampering
* verified can be trusted by the server.
* Reduces database lookup but exposes xss vulnaribility.
* Typically sent to authorization header.
* Can be used there where cookies are not allowed.
* Contains header(meta), payload(actual data), signature delimeterd by “.” (dot)
* The jwt token is not encrypted, this is open source, we can easily read the data in jwt.io site.
* HMAC algorithm is used for digitally sign the token, and base64 urlencoding use for encoding.

**Disadvantages of jwt token**

jwt token is easily readable from the client storage as it is not stored in cookie. xss vulnaribility can be increased for this.

**Solution for xss vulnaribility for jwt token ?**

* Same origin policy with the help of CORS.
* Same origin policy will not be used in case of microservice, as there are several domain is used.
* Need to define a third party token generator, where jwt token is not sent to the client machine, client will only get access token, by passing the access token the requested server gets the jwt token. It can stop the xss vulnaribility.