Same Origin Policy

Same origin policy:In computing, the same-origin policy is an important concept in the web application security model. Under the policy, a web browser permits scripts contained in a first web page to access data in a second web page, but only if both web pages have the same origin.

An origin is defined as a combination of URI scheme, host name, and port number. This policy prevents a malicious script on one page from obtaining access to sensitive data on another web page. In order for two webpages or files to interact with each, they should be from the same origin.

When it is said same origin, that means they both should have the same port, protocol or scheme and host .

To illustrate that, the following table gives an overview of typical outcomes for checks against the URL "http://www.example.com/dir/page.html".

Compared URL	Outcome	Reason
http://www.example.com/dir/page2.html	Success	Same protocol,host and port
http://www.example.com/dir2/other.html	Success	Same protocol,host and port
http://username:password@www.example.com/dir2/other.html	Success	Same protocol,host and port
http://www.example.com:81/dir/other.html	Failure	Same protocol and host but different port
https://www.example.com/dir/other.html	Failure	Different protocol
http://example.com/dir/other.html	Failure	Different host (exact match required)
http://v2.www.example.com/dir/other.html	Failure	Different host (exact match required)
http://en.example.com/dir/other.html	Failure	Different host
http://www.example.com:80/dir/other.html	Depends	Port explicit. Depends on implementation in browser.

Same origin policy is essential aspect of security model. We can imagine without that policy, there would significant security issues. The following example illustrates a potential security risk that could arise without the same-origin policy. Assume that a user is visiting a banking website and doesn't log out. Then, the user goes to another site that has some malicious JavaScript code running in the background that requests data from the banking site. Because the user is still logged in on the banking site, the malicious code could do anything the user could do on the banking site. For example, it could get a list of the user's last transactions, create a new transaction, etc. This is because the browser can send and receive session cookies to the banking site based on the domain of the banking site.

The user visiting the malicious site would expect that the site he or she is visiting has no access to the banking session cookie. While it is true that the JavaScript has no direct access to the banking session cookie, it could still send and receive requests to the banking site with the banking site's session cookie. Because the script can essentially do the same as the user would do.

In some circumstances, the same-origin policy is too restrictive and some times we need exchange files from different domain and different servers (eg: using fonts from different domain). A web page may freely embed cross-origin images, stylesheets, scripts, iframes, and videos. Certain "cross-domain" requests, notably Ajax requests, are forbidden by default by the same-origin security policy.

There are several workarounds that could make the same origin policy allow interaction of files even though they dont meet the triple(protocol, host and port). These are as fellow:

1- If two webpages or files contain scripts that set domain to the same value, the same-origin policy would allow connection for these two files or webpages, and each can interact with the other. For example, cooperating scripts in documents loaded from orders.example.com and catalog.example.com might set their **document.domain** properties to **"example.com"**, thereby making the documents appear to have the same origin and enabling each document to read properties of the other. Setting this property implicitly sets the port to null, which most browsers will interpret differently from port 80 or even an unspecified port. To assure that access will be allowed by the browser, set the **document.domain** property of both pages.

EX:
file script on orders.example.com should have that code line set up
document.domain = "example.com";

Also file script on catalog.example.com should have the same code line set up
document.domain = "example.com";

2- The second technique that make **same-origin policy** permit connection or access to files from different domain is using **Cross-Origin Resource Sharing(CORS) standard.** CORS exists for security reasons and to limit which resources a browser can gain access to, from another website in secure way. In other words, it is a way in which a browser and server can interact to determine whether or not it is safe to allow the cross-origin request. Let's say our site exists at http://someexampledomain.com and we want the JavaScript files on that site to access http://anotherdomain.com, we can't do that unless the server at http://anotherdomain.com allows it.

CORS would extend HTTP with a new **Origin request header** and a **new Access-Control-Allow-Origin response header**. It allows servers to use a header to explicitly list origins that may request a file or to use a wildcard and allow a file to be requested by any site. Browsers such as Firefox 3.5, Safari 4 and Internet Explorer 10 use this header to allow the cross-origin HTTP requests with XMLHttpRequest that would otherwise have been forbidden by the same-origin policy.

If we try an AJAX/xmlhttprequest to a file like that without setting using header extension of CORS, we get an error message from the console in our browser.

```
XMLHttpRequest cannot load http://localhost:3000. No 'Access-Control-Allow-Origin' header is present on the requested resource. Origin 'null' is therefore not allowed access.
```

The CORS rules apply to the same hostname and are also bound to the same port. So if I had an HTML file with JavaScript served by http://localhost:3001 it would not be allowed to load things from http://localhost:3000.

To allow exactly that to happend and have our client side code separate from our server and just make it load data, we should extend the header response of http://http://localhost:3000. as fellow:

```
app.use(function(req, res, next) {
  res.header("Access-Control-Allow-Origin", "*");
  res.header("Access-Control-Allow-Headers", "Origin, X-Requested-With, Content-Type, Accept");
  next();
});
```

3-Another technique, cross-document messaging allows a script from one page to pass textual messages to a script on another page regardless of the script origins. Calling the postMessage() method on a Window object asynchronously fires an "onmessage" event in that window, triggering any user-defined event handlers. A script in one page still cannot directly access methods or variables in the other page, but they can communicate safely through this message-passing technique.

4-JSONP(JSON Padding) allows a page to receive JSON data from a different domain by adding a <script> element to the page which loads a JSON response with a callback from different domain