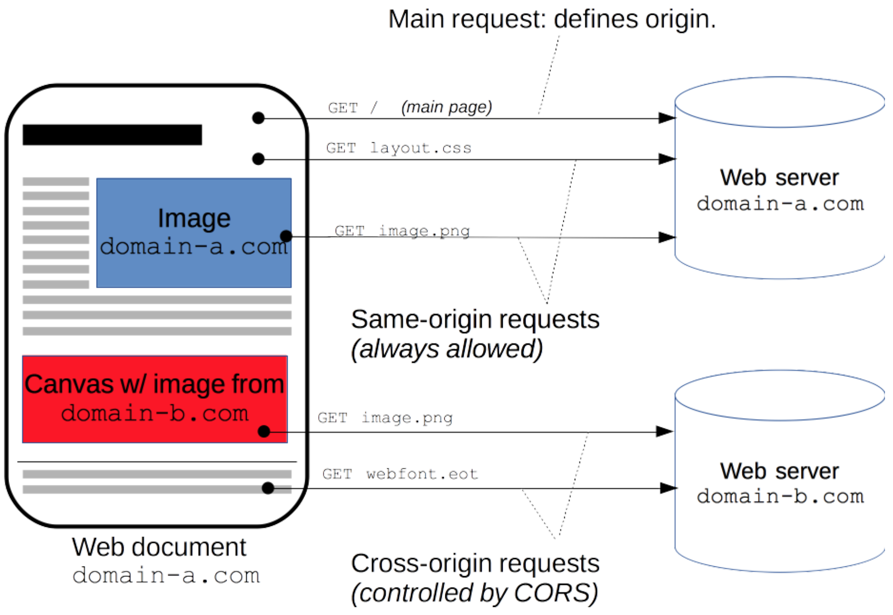
Cross-Origin Resource Sharing (CORS)

Cross-Origin Resource Sharing (CORS) is a mechanism that uses additional 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. A web application executes a **cross-origin HTTP request** when it requests a resource that has a different origin (domain, protocol, and port) than its own origin.

An example of a cross-origin request: The frontend JavaScript code for a web application served from `http://domain-a.com` uses `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 follow the same-origin policy. This means that a web application using those APIs can only request HTTP resources from the same origin the application was loaded from, unless the response from the other origin includes the right CORS headers.



The CORS mechanism supports secure cross-origin requests and data transfers between browsers and web servers. Modern browsers use CORS in an API container such as `XMLHttpRequest` or Fetch to help mitigate the risks of cross-origin HTTP requests.

Who should read this article?

Everyone, really.

More specifically, this article is for web administrators, server developers, and front-end developers. Modern browsers handle the client-side components of cross-origin sharing, including headers and policy enforcement. But this new standard means servers have to handle new request and response headers. Another article for server developers discussing cross-origin sharing from a server perspective (with PHP code snippets) is supplementary reading.

What requests use CORS?

This cross-origin sharing standard is used to enable cross-site HTTP requests for:

- Invocations of the `XMLHttpRequest` or Fetch APIs in a cross-site manner, as discussed above.
- Web Fonts (for cross-domain font usage in `@font-face` within CSS), so that servers can deploy TrueType fonts that can only be cross-site loaded and used by web sites that are permitted to do so.
- WebGL textures.
- Images/video frames drawn to a canvas using `drawImage()`.

This article is a general discussion of Cross-Origin Resource Sharing and includes a discussion of the necessary HTTP headers.

Functional overview

The Cross-Origin Resource Sharing standard works by adding new HTTP headers that allow servers to describe the set of origins that are permitted to read that information using a web browser. Additionally, for HTTP request methods that can cause side-effects on server's data (in particular, for HTTP methods other than `GET`, or for `POST` usage with certain MIME types), the specification mandates that browsers "preflight" the request, soliciting supported methods from the server with an HTTP `OPTIONS` request method, and then, upon "approval" from the server, sending the actual request with the actual HTTP request method. Servers can also notify clients whether "credentials" (including Cookies and HTTP Authentication data) should be sent with requests.

CORS failures result in errors, but for security reasons, specifics about what went wrong *are not available to JavaScript code*. All the code knows is that an error occurred. The only way to determine what specifically went wrong is to look at the browser's console for details.

Subsequent sections discuss scenarios, as well as provide a breakdown of the HTTP headers used.

Examples of access control scenarios

Here, we present three scenarios that illustrate how Cross-Origin Resource Sharing works. All of these examples use the `XMLHttpRequest` object, which can be used to make cross-site invocations in any supporting browser.

The JavaScript snippets included in these sections (and running instances of the server-code that correctly handles these cross-site requests) can be found "in action" at http://arunranga.com/examples/access-control/, and will work in browsers that support cross-site `XMLHttpRequest`.

A discussion of Cross-Origin Resource Sharing from a server perspective (including PHP code snippets) can be found in the Server-Side Access Control (CORS) article.

Simple requests

Some requests don't trigger a CORS preflight. Those are called "simple requests" in this article, though the Fetch spec (which defines CORS) doesn't use that term. A request that doesn't trigger a CORS preflight—a so-called "simple request" — is one that **meets all the following conditions**:

- The only allowed methods are:
  - `GET`
  - `HEAD`
  - `POST`
- Apart from the headers set automatically by the user agent (for example, `Connection`, `User-Agent`, or any of the other headers with names defined in the Fetch spec as a "forbidden header name"), the only headers which are allowed to be manually set are those which the Fetch spec defines as being a "CORS-safelisted request-header", which are:
  - `Accept`
  - `Accept-Language`
  - `Content-Language`
  - `Content-Type` (but note the additional requirements below)
  - `DPR`
  - `Downlink`
  - `Save-Data`

    - `Viewport-Width`
    - `Width`
- The only allowed values for the `Content-Type` header are:
    - `application/x-www-form-urlencoded`
    - `multipart/form-data`
    - `text/plain`
- No event listeners are registered on any `XMLHttpRequestUpload` object used in the request; these are accessed using the `XMLHttpRequest.upload` property.
- No `ReadableStream` object is used in the request.

> **Note:** These are the same kinds of cross-site requests that web content can already issue, and no response data is released to the requester unless the server sends an appropriate header. Therefore, sites that prevent cross-site request forgery have nothing new to fear from HTTP access control.

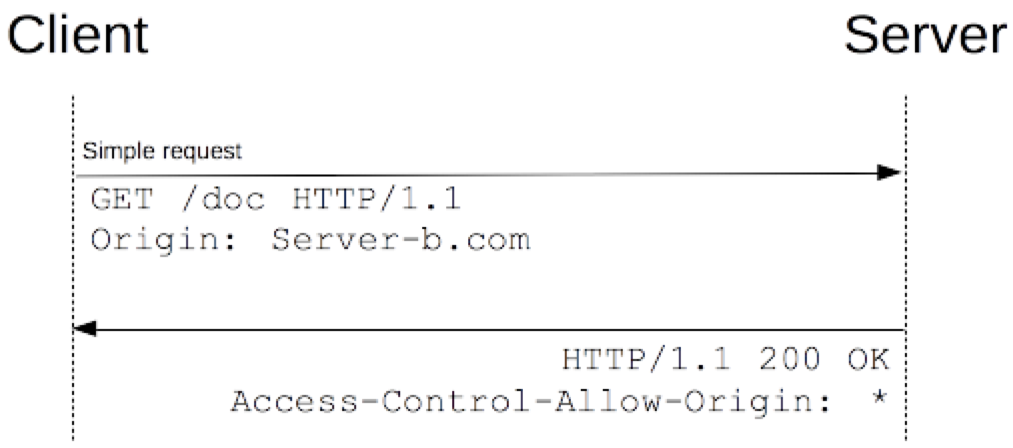
> **Note:** WebKit Nightly and Safari Technology Preview place additional restrictions on the values allowed in the `Accept`, `Accept-Language`, and `Content-Language` headers. If any of those headers have "non-standard" values, WebKit/Safari does not consider the request to meet the conditions for a "simple request". What WebKit/Safari considers "non-standard" values for those headers is not documented except in the following WebKit bugs: Require preflight for non-standard CORS-safelisted request headers Accept, Accept-Language, and Content-Language, Allow commas in Accept, Accept-Language, and Content-Language request headers for simple CORS, and Switch to a blacklist model for restricted Accept headers in simple CORS requests. No other browsers implement those extra restrictions, because they're not part of the spec.

For example, suppose web content on domain `http://foo.example` wishes to invoke content on domain `http://bar.other`. Code of this sort might be used within JavaScript deployed on foo.example:

```
const invocation = new XMLHttpRequest();
const url = 'http://bar.other/resources/public-data/';

function callOtherDomain() {
  if(invocation) {
    invocation.open('GET', url, true);
    invocation.onreadystatechange = handler;
    invocation.send();
  }
}
```

This will lead to a simple exchange between the client and the server, using CORS headers to handle the privileges:



Let us look at what the browser will send to the server in this case, and let's see how the server responds:

```
GET /resources/public-data/ HTTP/1.1
Host: bar.other
User-Agent: Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10.5; en-US; rv:1
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
Referer: http://foo.example/examples/access-control/simpleXSInvocation.
Origin: http://foo.example


HTTP/1.1 200 OK
Date: Mon, 01 Dec 2008 00:23:53 GMT
Server: Apache/2.0.61
Access-Control-Allow-Origin: *
```

```
Keep-Alive: timeout=2, max=100
Connection: Keep-Alive
Transfer-Encoding: chunked
Content-Type: application/xml

[XML Data]
```

Lines 1 - 10 are headers sent. The main HTTP request header of note here is the `Origin` header on line 10 above, which shows that the invocation is coming from content on the domain `http://foo.example`.

Lines 13 - 22 show the HTTP response from the server on domain `http://bar.other`. In response, the server sends back an `Access-Control-Allow-Origin` header, shown above in line 16. The use of the `Origin` header and of `Access-Control-Allow-Origin` show the access control protocol in its simplest use. In this case, the server responds with a `Access-Control-Allow-Origin: *` which means that the resource can be accessed by **any** domain in a cross-site manner. If the resource owners at `http://bar.other` wished to restrict access to the resource to requests only from `http://foo.example`, they would send back:

`Access-Control-Allow-Origin: http://foo.example`

Note that now, no domain other than `http://foo.example` (identified by the ORIGIN: header in the request, as in line 10 above) can access the resource in a cross-site manner. The `Access-Control-Allow-Origin` header should contain the value that was sent in the request's `Origin` header.

Preflighted requests

Unlike “simple requests” (discussed above), "preflighted" requests first send an HTTP request by the `OPTIONS` method to the resource on the other domain, in order to determine whether the actual request is safe to send. Cross-site requests are preflighted like this since they may have implications to user data.

In particular, a request is preflighted if **any of the following conditions** is true:

- **If** the request uses any of the following methods:

    - `PUT`
    - `DELETE`
    - `CONNECT`
    - `OPTIONS`
    - `TRACE`
    - `PATCH`
- **Or if**, apart from the headers set automatically by the user agent (for example, `Connection`, `User-Agent`, or any of the **OTHER** header with a name defined in the Fetch spec as a "forbidden header name"), the request includes any headers other than those which the Fetch spec defines as being a "CORS-safelisted request-header", which are the following:
    - `Accept`
    - `Accept-Language`
    - `Content-Language`
    - `Content-Type` (but note the additional requirements below)
    - `DPR`
    - `Downlink`
    - `Save-Data`
    - `Viewport-Width`
    - `Width`
- **Or if** the `Content-Type` header has a value **OTHER THAN** the following:
    - `application/x-www-form-urlencoded`
    - `multipart/form-data`
    - `text/plain`
- **Or if** one or more event listeners are registered on an `XMLHttpRequestUpload` object used in the request.
- **Or if** a `ReadableStream` object is used in the request.

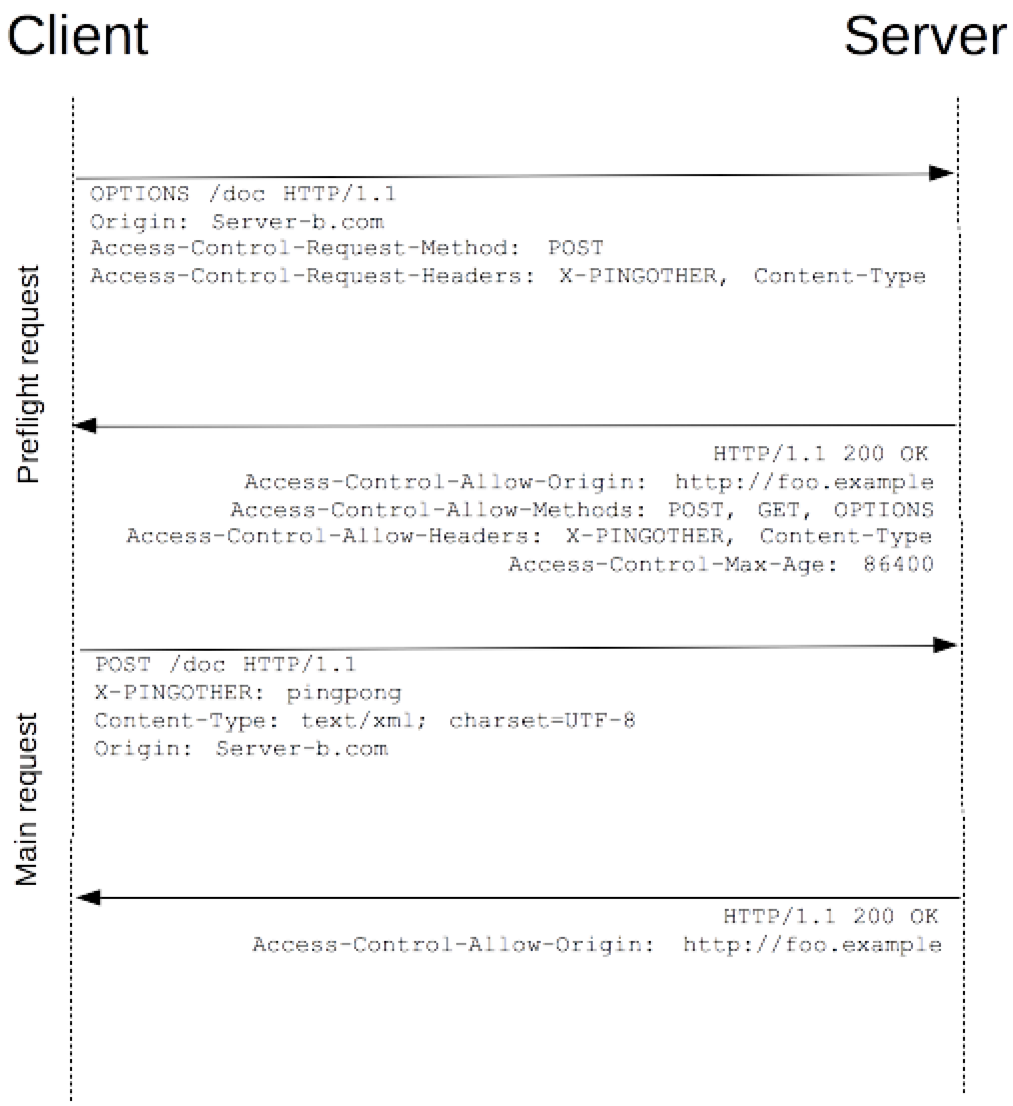
> **Note:** WebKit Nightly and Safari Technology Preview place additional restrictions on the values allowed in the `Accept`, `Accept-Language`, and `Content-Language` headers. If any of those headers have "non-standard" values, WebKit/Safari preflights the request. What WebKit/Safari considers "non-standard" values for those headers is not documented except

> in the following WebKit bugs: Require preflight for non-standard CORS-safelisted request headers Accept, Accept-Language, and Content-Language, Allow commas in Accept, Accept-Language, and Content-Language request headers for simple CORS, and Switch to a blacklist model for restricted Accept headers in simple CORS requests. No other browsers implement those extra restrictions, because they're not part of the spec.

The following is an example of a request that will be preflighted.

```
1  const invocation = new XMLHttpRequest();
2  const url = 'http://bar.other/resources/post-here/';
3  const body = '<?xml version="1.0"?><person><name>Arun</name></person>';
4
5  function callOtherDomain(){
6    if(invocation)
7      {
8        invocation.open('POST', url, true);
9        invocation.setRequestHeader('X-PINGOTHER', 'pingpong');
10       invocation.setRequestHeader('Content-Type', 'application/xml');
11       invocation.onreadystatechange = handler;
12       invocation.send(body);
13     }
14 }
15
16 ......
```

In the example above, line 3 creates an XML body to send with the `POST` request in line 8. Also, on line 9, a "customized" (non-standard) HTTP request header is set (`X-PINGOTHER: pingpong`). Such headers are not part of the HTTP/1.1 protocol, but are generally useful to web applications. Since the request uses a Content-Type of `application/xml`, and since a custom header is set, this request is preflighted.



```
OPTIONS /doc HTTP/1.1
Origin: Server-b.com
Access-Control-Request-Method: POST
Access-Control-Request-Headers: X-PINGOTHER, Content-Type
```

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: http://foo.example
Access-Control-Allow-Methods: POST, GET, OPTIONS
Access-Control-Allow-Headers: X-PINGOTHER, Content-Type
Access-Control-Max-Age: 86400
```

```
POST /doc HTTP/1.1
X-PINGOTHER: pingpong
Content-Type: text/xml; charset=UTF-8
Origin: Server-b.com
```

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: http://foo.example
```

(Note: as described below, the actual POST request does not include the Access-Control-Request-* headers; they are needed only for the OPTIONS request.)

Let's take a look at the full exchange between client and server. The first exchange is the *preflight request/response*:

```
OPTIONS /resources/post-here/ HTTP/1.1
Host: bar.other
User-Agent: Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10.5; en-US; rv:1
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://foo.example
```

```
Access-Control-Request-Method: POST
Access-Control-Request-Headers: X-PINGOTHER, Content-Type


HTTP/1.1 204 No Content
Date: Mon, 01 Dec 2008 01:15:39 GMT
Server: Apache/2.0.61 (Unix)
Access-Control-Allow-Origin: http://foo.example
Access-Control-Allow-Methods: POST, GET, OPTIONS
Access-Control-Allow-Headers: X-PINGOTHER, Content-Type
Access-Control-Max-Age: 86400
Vary: Accept-Encoding, Origin
Keep-Alive: timeout=2, max=100
Connection: Keep-Alive
```

Once the preflight request is complete, the real request is sent:

```
POST /resources/post-here/ HTTP/1.1
Host: bar.other
User-Agent: Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10.5; en-US; rv:1
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
X-PINGOTHER: pingpong
Content-Type: text/xml; charset=UTF-8
Referer: http://foo.example/examples/preflightInvocation.html
Content-Length: 55
Origin: http://foo.example
Pragma: no-cache
Cache-Control: no-cache

<?xml version="1.0"?><person><name>Arun</name></person>


HTTP/1.1 200 OK
```

```
Date: Mon, 01 Dec 2008 01:15:40 GMT
Server: Apache/2.0.61 (Unix)
Access-Control-Allow-Origin: http://foo.example
Vary: Accept-Encoding, Origin
Content-Encoding: gzip
Content-Length: 235
Keep-Alive: timeout=2, max=99
Connection: Keep-Alive
Content-Type: text/plain

[Some GZIP'd payload]
```

Lines 1 - 12 above represent the preflight request with the `OPTIONS` method. The browser determines that it needs to send this based on the request parameters that the JavaScript code snippet above was using, so that the server can respond whether it is acceptable to send the request with the actual request parameters. OPTIONS is an HTTP/1.1 method that is used to determine further information from servers, and is a safe method, meaning that it can't be used to change the resource. Note that along with the OPTIONS request, two other request headers are sent (lines 10 and 11 respectively):

```
Access-Control-Request-Method: POST
Access-Control-Request-Headers: X-PINGOTHER, Content-Type
```

The `Access-Control-Request-Method` header notifies the server as part of a preflight request that when the actual request is sent, it will be sent with a `POST` request method. The `Access-Control-Request-Headers` header notifies the server that when the actual request is sent, it will be sent with a `X-PINGOTHER` and `Content-Type` custom headers. The server now has an opportunity to determine whether it wishes to accept a request under these circumstances.

Lines 14 - 26 above are the response that the server sends back indicating that the request method (`POST`) and request headers (`X-PINGOTHER`) are acceptable. In particular, let's look at lines 17-20:

```
Access-Control-Allow-Origin: http://foo.example
Access-Control-Allow-Methods: POST, GET
```

```
Access-Control-Allow-Headers: X-PINGOTHER, Content-Type
Access-Control-Max-Age: 86400
```

The server responds with `Access-Control-Allow-Methods` and says that `POST` and `GET` are viable methods to query the resource in question. Note that this header is similar to the `Allow` response header, but used strictly within the context of access control.

The server also sends `Access-Control-Allow-Headers` with a value of "`X-PINGOTHER, Content-Type`", confirming that these are permitted headers to be used with the actual request. Like `Access-Control-Allow-Methods`, `Access-Control-Allow-Headers` is a comma separated list of acceptable headers.

Finally, `Access-Control-Max-Age` gives the value in seconds for how long the response to the preflight request can be cached for without sending another preflight request. In this case, 86400 seconds is 24 hours. Note that each browser has a maximum internal value that takes precedence when the `Access-Control-Max-Age` is greater.

Preflighted requests and redirects

Not all browsers currently support following redirects after a preflighted request. If a redirect occurs after a preflighted request, some browsers currently will report an error message such as the following.

> The request was redirected to 'https://example.com/foo', which is disallowed for cross-origin requests that require preflight

> Request requires preflight, which is disallowed to follow cross-origin redirect

The CORS protocol originally required that behavior but was subsequently changed to no longer require it. However, not all browsers have implemented the change, and so still exhibit the behavior that was originally required.

So until all browsers catch up with the spec, you may be able to work around this limitation by doing one or both of the following:

- change the server-side behavior to avoid the preflight and/or to avoid the redirect—if you have control over the server the request is being made to
- change the request such that it is a simple request that doesn't cause a preflight

But if it's not possible to make those changes, then another way that may be possible is to this:

1. Make a simple request (using `Response.url` for the Fetch API, or `XMLHttpRequest.responseURL`) to determine what URL the real preflighted request would end up at.
2. Make another request (the "real" request) using the URL you obtained from `Response.url` or `XMLHttpRequest.responseURL` in the first step.

However, if the request is one that triggers a preflight due to the presence of the `Authorization` header in the request, you won't be able to work around the limitation using the steps above. And you won't be able to work around it at all unless you have control over the server the request is being made to.

Requests with credentials

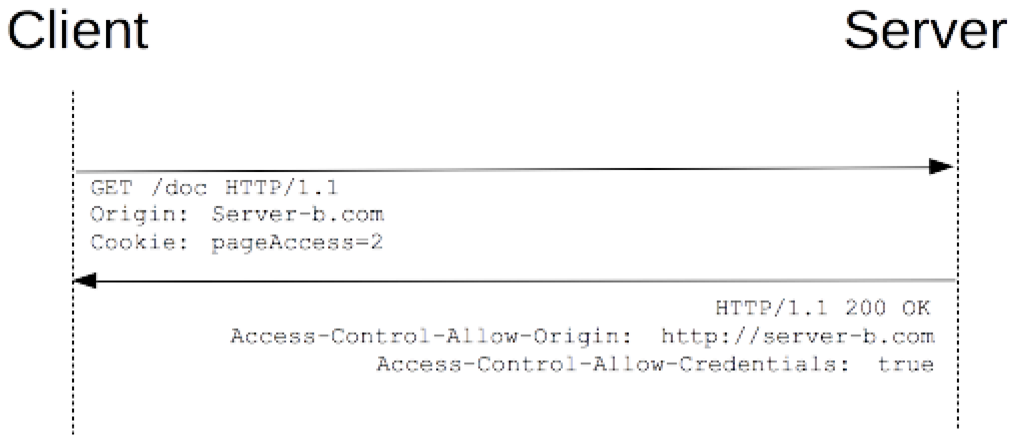
The most interesting capability exposed by both `XMLHttpRequest` or Fetch and CORS is the ability to make "credentialed" requests that are aware of HTTP cookies and HTTP Authentication information. By default, in cross-site `XMLHttpRequest` or Fetch invocations, browsers will **not** send credentials. A specific flag has to be set on the `XMLHttpRequest` object or the `Request` constructor when it is invoked.

In this example, content originally loaded from `http://foo.example` makes a simple GET request to a resource on `http://bar.other` which sets Cookies. Content on foo.example might contain JavaScript like this:

```
const invocation = new XMLHttpRequest();
const url = 'http://bar.other/resources/credentialed-content/';

function callOtherDomain(){
  if(invocation) {
    invocation.open('GET', url, true);
7        invocation.withCredentials = true;
8        invocation.onreadystatechange = handler;
9        invocation.send();
10     }
11   }
```

Line 7 shows the flag on `XMLHttpRequest` that has to be set in order to make the invocation with Cookies, namely the `withCredentials` boolean value. By default, the invocation is made without Cookies. Since this is a simple `GET` request, it is not preflighted, but the browser will **reject** any response that does not have the `Access-Control-Allow-Credentials: true` header, and **not** make the response available to the invoking web content.



Client
Server
GET /doc HTTP/1.1
Origin: Server-b.com
Cookie: pageAccess=2
HTTP/1.1 200 OK
Access-Control-Allow-Origin: http://server-b.com
Access-Control-Allow-Credentials: true

Here is a sample exchange between client and server:

```
GET /resources/access-control-with-credentials/ HTTP/1.1
Host: bar.other
User-Agent: Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10.5; en-US; rv:1
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
Referer: http://foo.example/examples/credential.html
Origin: http://foo.example
Cookie: pageAccess=2


HTTP/1.1 200 OK
```

```
Date: Mon, 01 Dec 2008 01:34:52 GMT
Server: Apache/2.0.61 (Unix) PHP/4.4.7 mod_ssl/2.0.61 OpenSSL/0.9.7e mc
X-Powered-By: PHP/5.2.6
Access-Control-Allow-Origin: http://foo.example
Access-Control-Allow-Credentials: true
Cache-Control: no-cache
Pragma: no-cache
Set-Cookie: pageAccess=3; expires=Wed, 31-Dec-2008 01:34:53 GMT
Vary: Accept-Encoding, Origin
Content-Encoding: gzip
Content-Length: 106
Keep-Alive: timeout=2, max=100
Connection: Keep-Alive
Content-Type: text/plain


[text/plain payload]
```

Although line 11 contains the Cookie destined for the content on `http://bar.other`, if bar.other did not respond with an `Access-Control-Allow-Credentials: true` (line 19) the response would be ignored and not made available to web content.

Credentialed requests and wildcards

When responding to a credentialed request, the server **must** specify an origin in the value of the `Access-Control-Allow-Origin` header, instead of specifying the "`*`" wildcard.

Because the request headers in the above example include a `Cookie` header, the request would fail if the value of the `Access-Control-Allow-Origin` header were "*". But it does not fail: Because the value of the `Access-Control-Allow-Origin` header is `"http://foo.example"` (an actual origin) rather than the "`*`" wildcard, the credential-cognizant content is returned to the invoking web content.

Note that the `Set-Cookie` response header in the example above also sets a further cookie. In case of failure, an exception—depending on the API used—is raised.

Third-party cookies

Note that cookies set in CORS responses are subject to normal third-party cookie policies. In the example above, the page is loaded from `foo.example`, but the cookie on line 22 is sent by `bar.other`, and would thus not be saved if the user has configured their browser to reject all third-party cookies.

The HTTP response headers

This section lists the HTTP response headers that servers send back for access control requests as defined by the Cross-Origin Resource Sharing specification. The previous section gives an overview of these in action.

Access-Control-Allow-Origin

A returned resource may have one `Access-Control-Allow-Origin` header, with the following syntax:

```
Access-Control-Allow-Origin: <origin> | *
```

`Access-Control-Allow-Origin` specifies either a single origin, which tells browsers to allow that origin to access the resource; or else — for requests **without** credentials — the "`*`" wildcard, to tell browsers to allow any origin to access the resource.

For example, to allow code from the origin `http://mozilla.org` to access the resource, you can specify:

```
Access-Control-Allow-Origin: http://mozilla.org
```

If the server specifies a single origin rather than the "`*`" wildcard, then the server should also include `Origin` in the `Vary` response header — to indicate to clients that server responses will differ based on the value of the `Origin` request header.

Access-Control-Expose-Headers

The `Access-Control-Expose-Headers` header lets a server whitelist headers that browsers are allowed to access.

```
Access-Control-Expose-Headers: <field-name>[, <field-name>]*
```

For example, the following:

```
Access-Control-Expose-Headers: X-My-Custom-Header, X-Another-Custom-Hea
```

…would allow the `X-My-Custom-Header` and `X-Another-Custom-Header` headers to be exposed to the browser.

Access-Control-Max-Age

The `Access-Control-Max-Age` header indicates how long the results of a preflight request can be cached. For an example of a preflight request, see the above examples.

```
Access-Control-Max-Age: <delta-seconds>
```

The `delta-seconds` parameter indicates the number of seconds the results can be cached.

Access-Control-Allow-Credentials

The `Access-Control-Allow-Credentials` header Indicates whether or not the response to the request can be exposed when the `credentials` flag is true. When used as part of a response to a preflight request, this indicates whether or not the actual request can be made using credentials. Note that simple `GET` requests are not preflighted, and so if a request is made for a resource with credentials, if this header is not returned with the resource, the response is ignored by the browser and not returned to web content.

```
Access-Control-Allow-Credentials: true
```

Credentialed requests are discussed above.

Access-Control-Allow-Methods

The `Access-Control-Allow-Methods` header specifies the method or methods allowed when accessing the resource. This is used in response to a preflight request. The conditions under which a request is preflighted are discussed above.

```
Access-Control-Allow-Methods: <method>[, <method>]*
```

An example of a preflight request is given above, including an example which sends this header to the browser.

Access-Control-Allow-Headers

The `Access-Control-Allow-Headers` header is used in response to a preflight request to indicate which HTTP headers can be used when making the actual request.

```
Access-Control-Allow-Headers: <field-name>[, <field-name>]*
```

The HTTP request headers

This section lists headers that clients may use when issuing HTTP requests in order to make use of the cross-origin sharing feature. Note that these headers are set for you when making invocations to servers. Developers using cross-site `XMLHttpRequest` capability do not have to set any cross-origin sharing request headers programmatically.

Origin

The `Origin` header indicates the origin of the cross-site access request or preflight request.

```
Origin: <origin>
```

The origin is a URI indicating the server from which the request initiated. It does not include any path information, but only the server name.

> **Note:** The `origin` value can be `null`, or a URI.

Note that in any access control request, the `Origin` header is **always** sent.

Access-Control-Request-Method

The `Access-Control-Request-Method` is used when issuing a preflight request to let the server know what HTTP method will be used when the actual request is made.

```
Access-Control-Request-Method: <method>
```

Examples of this usage can be found above.

Access-Control-Request-Headers

The `Access-Control-Request-Headers` header is used when issuing a preflight request to let the server know what HTTP headers will be used when the actual request is made.

```
Access-Control-Request-Headers: <field-name>[, <field-name>]*
```

Examples of this usage can be found above.

Specifications

Specification	Status	Comment
Fetch The definition of 'CORS' in that specification.	LS Living Standard	New definition; supplants W3C CORS specification.

Browser compatibility

Update compatibility data on GitHub

`Access-Control-Allow-Origin`	
Chrome	4
Edge	12
Firefox	3.5
IE	10
Opera	12
Safari	4
WebView Android	2
Chrome Android	Yes
Edge Mobile	Yes
Firefox Android	4
Opera Android	12
Safari iOS	3.2
Samsung Internet Android	Yes

▢	Full support

Compatibility notes

- Internet Explorer 8 and 9 expose CORS via the `XDomainRequest` object, but have a full implementation in IE 10.
- While Firefox 3.5 introduced support for cross-site `XMLHttpRequests` and Web Fonts, certain requests were limited until later versions. Specifically, Firefox 7 introduced the ability for cross-site HTTP requests for WebGL Textures, and Firefox 9 added support for Images drawn on a canvas using `drawImage()`.

See also

- CORS errors
- Enable CORS: I want to add CORS support to my server
- `XMLHttpRequest`
- Fetch API
- Using CORS with All (Modern) Browsers
- Using CORS - HTML5 Rocks
- Code Samples Showing `XMLHttpRequest` and Cross-Origin Resource Sharing
- Client-Side & Server-Side (Java) sample for Cross-Origin Resource Sharing (CORS)
- Cross-Origin Resource Sharing From a Server-Side Perspective (PHP, etc.)
- Stack Overflow answer with "how to" info for dealing with common problems:
  - How to avoid the CORS preflight
  - How to use a CORS proxy to get around *"No Access-Control-Allow-Origin header"*
  - How to fix *"Access-Control-Allow-Origin header must not be the wildcard"*