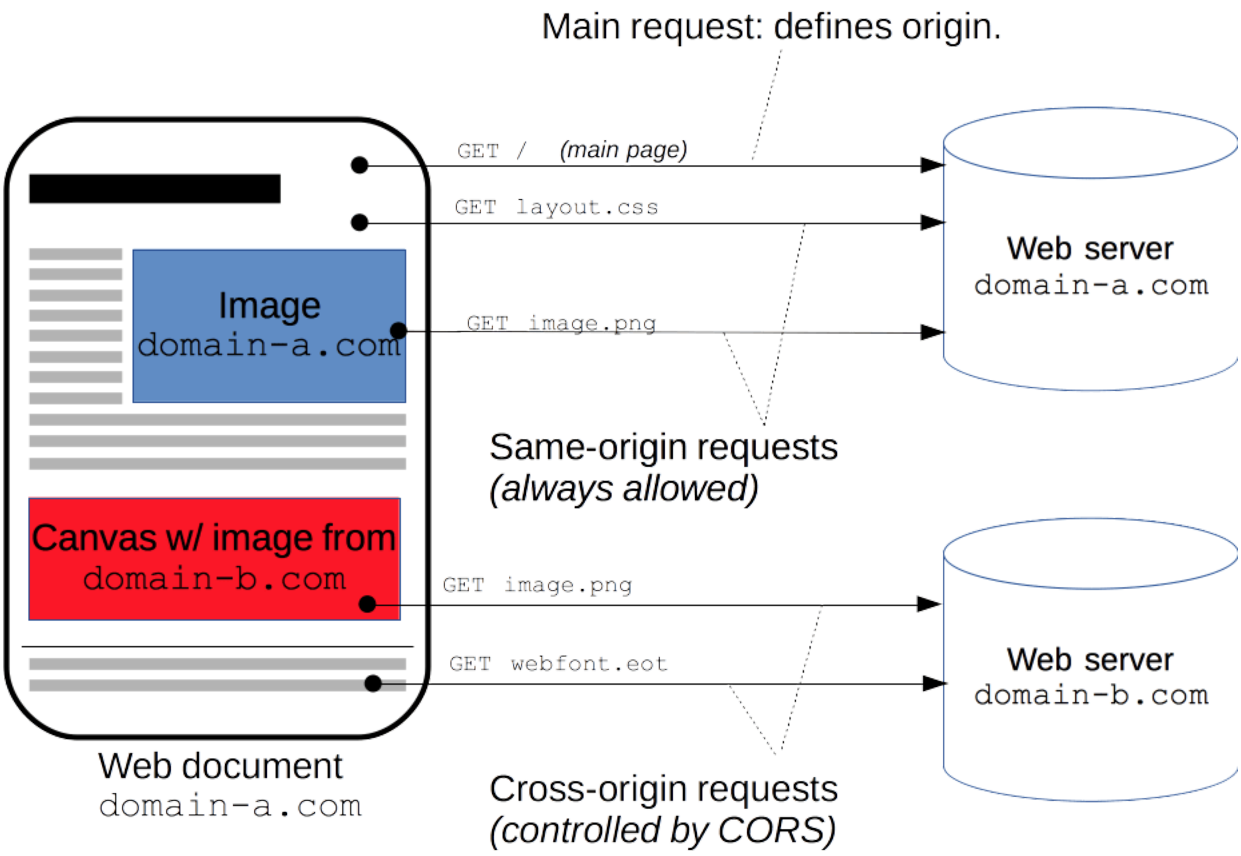
Cross-Origin Resource Sharing (CORS)

Cross-Origin Resource Sharing (CORS) is an HTTP-header based mechanism that allows a server to indicate any origins (domain, scheme, or port) other than its own from which a browser should permit loading resources. CORS also relies on a mechanism by which browsers make a "preflight" request to the server hosting the cross-origin resource, in order to check that the server will permit the actual request. In that preflight, the browser sends headers that indicate the HTTP method and headers that will be used in the actual request.

An example of a cross-origin request: the front-end JavaScript code served from `https://domain-a.com` uses `XMLHttpRequest` to make a request for `https://domain-b.com/data.json`.

For security reasons, browsers restrict cross-origin HTTP requests initiated from 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 resources from the same origin the application was loaded from unless the response from other origins includes the right CORS headers.



The CORS mechanism supports secure cross-origin requests and data transfers between browsers and servers. Modern browsers use CORS in APIs such as `XMLHttpRequest` or Fetch to 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 of cross-origin sharing, including headers and policy enforcement. But the CORS standard means servers have to handle new request and response headers.

What requests use CORS?

This cross-origin sharing standard can enable cross-origin HTTP requests for:

- Invocations of the `XMLHttpRequest` or Fetch APIs, 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 loaded cross-origin and used by web sites that are permitted to do so.
- WebGL textures.
- Images/video frames drawn to a canvas using `drawImage()` .
- CSS Shapes from images.

This is a general article about 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 let servers describe which origins are permitted to read that information from a web browser. Additionally, for HTTP request methods that can cause side-effects on server data (in particular, HTTP methods other than `GET` , or `POST` with certain MIME types), the specification mandates that browsers "preflight" the request, soliciting supported methods from the server with the HTTP `OPTIONS` request method, and then, upon "approval" from the server, sending the actual request. Servers can also inform clients whether "credentials" (such as Cookies and HTTP Authentication) should be sent with requests.

CORS failures result in errors but for security reasons, specifics about the error *are not available to JavaScript*. 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

We present three scenarios that demonstrate how Cross-Origin Resource Sharing works. All these examples use `XMLHttpRequest` , which can make cross-origin requests in any supporting browser.

Simple requests

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

- One of the allowed methods:
  - `GET`
  - `HEAD`
  - `POST`
- Apart from the headers automatically set by the user agent (for example, `Connection` , `User-Agent` , or the other headers 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 a CORS-safelisted request-header , which are:
  - `Accept`
  - `Accept-Language`
  - `Content-Language`
  - `Content-Type` (please note the additional requirements below)
  - `Range` (only with a simple range header value ; e.g., `bytes=256-` or `bytes=127-255`)

  Note: Firefox has not implemented `Range` as a safelisted request-header yet. See bug 1733981 .

- The only type/subtype combinations allowed for the media type specified in the `Content-Type` header are:
  - `application/x-www-form-urlencoded`
  - `multipart/form-data`
  - `text/plain`
- If the request is made using an `XMLHttpRequest` object, no event listeners are registered on the object returned by the `XMLHttpRequest.upload` property used in the request; that is, given an `XMLHttpRequest` instance `xhr` , no code has called `xhr.upload.addEventListener()` to add an event listener to monitor the upload.

- No `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 "nonstandard" values, WebKit/Safari does not consider the request to be a "simple request". What values WebKit/Safari consider "nonstandard" 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
> - Switch to a blacklist model for restricted Accept headers in simple CORS requests
>
> No other browsers implement these extra restrictions because they're not part of the spec.

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

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

xhr.open('GET', url);
xhr.onreadystatechange = someHandler;
xhr.send();
```

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

Let's look at what the browser will send to the server in this case:

```
GET /resources/public-data/ HTTP/1.1
Host: bar.other
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.14; rv:71.0) Gecko/20100101 Firefox/71.0
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
Connection: keep-alive
Origin: https://foo.example
```

The request header of note is `Origin`, which shows that the invocation is coming from `https://foo.example`.

Now let's see how the server responds:

```
HTTP/1.1 200 OK
Date: Mon, 01 Dec 2008 00:23:53 GMT
Server: Apache/2
Access-Control-Allow-Origin: *
Keep-Alive: timeout=2, max=100
Connection: Keep-Alive
Transfer-Encoding: chunked
Content-Type: application/xml

[…XML Data…]
```

In response, the server returns a `Access-Control-Allow-Origin` header with `Access-Control-Allow-Origin: *`, which means that the resource can be accessed by **any** origin.

```
Access-Control-Allow-Origin: *
```

This pattern of the `Origin` and `Access-Control-Allow-Origin` headers is the simplest use of the access control protocol. If the resource owners at `https://bar.other` wished to restrict access to the resource to requests *only* from `https://foo.example`, (i.e no domain other than `https://foo.example` can access the resource in a cross-origin manner) they would send:

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

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

Preflighted requests

Unlike *simple requests*, for "preflighted" requests the browser first sends an HTTP request using the `OPTIONS` method to the resource on the other origin, in order to determine if the actual request is safe to send. Such cross-origin requests are preflighted since they may have implications for user data.

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

```
const xhr = new XMLHttpRequest();
xhr.open('POST', 'https://bar.other/resources/post-here/');
xhr.setRequestHeader('X-PINGOTHER', 'pingpong');
xhr.setRequestHeader('Content-Type', 'application/xml');
xhr.onreadystatechange = handler;
xhr.send('<person><name>Arun</name></person>');
```

The example above creates an XML body to send with the `POST` request. Also, a non-standard HTTP `X-PINGOTHER` request header is set. Such headers are not part of HTTP/1.1, 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.

> **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 look at the full exchange between client and server. The first exchange is the *preflight request/response*:

```
OPTIONS /doc HTTP/1.1
Host: bar.other
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.14; rv:71.0) Gecko/20100101 Firefox/71.0
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
Connection: keep-alive
Origin: https://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
Access-Control-Allow-Origin: https://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
```

Lines 1 - 10 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 9 and 10 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 do so with a `POST` request method. The `Access-Control-Request-Headers` header notifies the server that when the actual request is sent, it will do so with `X-PINGOTHER` and `Content-Type` custom headers. Now the server has an opportunity to determine whether it can accept a request under these conditions.

Lines 13 - 22 above are the response that the server returns, which indicate that the request method (`POST`) and request headers (`X-PINGOTHER`) are acceptable. Let's have a closer look at lines 16-19:

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

The server responds with `Access-Control-Allow-Origin: https://foo.example` , restricting access to the requesting origin domain only. It also responds with `Access-Control-Allow-Methods` , which says that `POST` and `GET` are valid methods to query the resource in question (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 without sending another preflight request. The default value is 5 seconds. In the present case, the max age is 86400 seconds (= 24 hours). Note that each browser has a maximum internal value that takes precedence when the `Access-Control-Max-Age` exceeds it.

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

```
POST /doc HTTP/1.1
Host: bar.other
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.14; rv:71.0) Gecko/20100101 Firefox/71.0
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
Connection: keep-alive
X-PINGOTHER: pingpong
Content-Type: text/xml; charset=UTF-8
Referer: https://foo.example/examples/preflightInvocation.html
Content-Length: 55
Origin: https://foo.example
Pragma: no-cache
Cache-Control: no-cache

<person><name>Arun</name></person>

HTTP/1.1 200 OK
Date: Mon, 01 Dec 2008 01:15:40 GMT
Server: Apache/2
Access-Control-Allow-Origin: https://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 XML payload]
```

Preflighted requests and redirects

Not all browsers currently support following redirects after a preflighted request. If a redirect occurs after such a 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 redirects.

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 thus still exhibit the originally required behavior.

Until 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
- Change the request such that it is a simple request that doesn't cause a preflight

If that's not possible, then another way is to:

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

> **Note:** When making credentialed requests to a different domain, third-party cookie policies will still apply. The policy is always enforced regardless of any setup on the server and the client as described in this chapter.

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-origin `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 `https://foo.example` makes a simple GET request to a resource on `https://bar.other` which sets Cookies. Content on foo.example might contain JavaScript like this:

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

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

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.

Here is a sample exchange between client and server:

```
GET /resources/credentialed-content/ HTTP/1.1
Host: bar.other
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.14; rv:71.0) Gecko/20100101 Firefox/71.0
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
Connection: keep-alive
Referer: https://foo.example/examples/credential.html
Origin: https://foo.example
Cookie: pageAccess=2

HTTP/1.1 200 OK
Date: Mon, 01 Dec 2008 01:34:52 GMT
Server: Apache/2
Access-Control-Allow-Origin: https://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 10 contains the Cookie destined for the content on `https://bar.other`, if bar.other did not respond with an `Access-Control-Allow-Credentials` `: true` (line 17), the response would be ignored and not made available to the web content.

Preflight requests and credentials

CORS-preflight requests must never include credentials. The *response* to a preflight request must specify `Access-Control-Allow-Credentials: true` to indicate that the actual request can be made with credentials.

> **Note:** Some enterprise authentication services require that TLS client certificates be sent in preflight requests, in contravention of the Fetch specification.
>
> Firefox 87 allows this non-compliant behavior to be enabled by setting the preference: `network.cors_preflight.allow_client_cert` to `true` (bug 1511151). Chromium-based browsers currently always send TLS client certificates in CORS preflight requests (Chrome bug 775438).

Credentialed requests and wildcards

When responding to a credentialed request:

- The server **must not** specify the "`*`" wildcard for the `Access-Control-Allow-Origin` response-header value, but must instead specify an explicit origin; for example: `Access-Control-Allow-Origin: https://example.com`
- The server **must not** specify the "`*`" wildcard for the `Access-Control-Allow-Headers` response-header value, but must instead specify an explicit list of header names; for example, `Access-Control-Allow-Headers: X-PINGOTHER, Content-Type`
- The server **must not** specify the "`*`" wildcard for the `Access-Control-Allow-Methods` response-header value, but must instead specify an explicit list of method names; for example, `Access-Control-Allow-Methods: POST, GET`

If a request includes a credential (most commonly a `Cookie` header) and the response includes an `Access-Control-Allow-Origin: *` header (that is, with the wildcard), the browser will block access to the response, and report a CORS error in the devtools console.

But if a request does include a credential (like the `Cookie` header) and the response includes an actual origin rather than the wildcard (like, for example, `Access-Control-Allow-Origin: https://example.com`), then the browser will allow access to the response from the specified origin.

Also note that any `Set-Cookie` response header in a response would not set a cookie if the `Access-Control-Allow-Origin` value in that response is the "`*`" wildcard rather an actual origin.

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 20 is sent by `bar.other`, and would thus not be saved if the user's browser is configured to reject all third-party cookies.

Cookie in the request (line 10) may also be suppressed in normal third-party cookie policies. The enforced cookie policy may therefore nullify the capability described in this chapter, effectively preventing you from making credentialed requests whatsoever.

Cookie policy around the SameSite attribute would apply.

The HTTP response headers

This section lists the HTTP response headers that servers return 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 tells browsers to allow any origin to access the resource.

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

```
Access-Control-Allow-Origin: https://mozilla.org
Vary: Origin
```

If the server specifies a single origin (that may dynamically change based on the requesting origin as part of an allowlist) 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 adds the specified headers to the allowlist that JavaScript (such as `getResponseHeader()`) in browsers is allowed to access.

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

For example, the following:

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

...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. This header is the server side response to the browser's `Access-Control-Request-Headers` header.

```
Access-Control-Allow-Headers: <header-name>[, <header-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-origin `XMLHttpRequest` capability do not have to set any cross-origin sharing request headers programmatically.

Origin

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

```
Origin: <origin>
```

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

> **Note:** The `origin` value can be `null`.

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 (such as with `setRequestHeader()`). This browser-side header will be answered by the complementary server-side header of `Access-Control-Allow-Headers`.

```
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.	Living Standard	New definition; supplants W3C CORS specification.

Browser compatibility

Report problems with this compatibility data on GitHub

	Chrome	Edge	Firefox	Internet Explorer	Opera	Safari	WebView Android	Chrome Android	Firefox for Android	Opera Android	Safari on iOS
`Access-Control-Allow-Origin`	Chrome4	Edge12	Firefox3.5	Internet Explorer 10	Opera12	Safari4	WebView Android 2	Chrome Android Yes	Firefox for Android 4	Opera Android 12	Safari on iOS 3

Full support

See also

- CORS errors
- Enable CORS: I want to add CORS support to my server
- `XMLHttpRequest`
- Fetch API
- Will it CORS? - an interactive CORS explainer & generator
- How to run Chrome browser without CORS
- Using CORS with All (Modern) Browsers
- 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"*

Last modified: Apr 27, 2022, by MDN contributors