HTTP access control (CORS)

Cross-site HTTP requests are HTTP requests for resources from a **different domain** than the domain of the resource making the request. For instance, a resource loaded from Domain A (`http://domaina.example`) such as an HTML web page, makes a request for a resource on Domain B (`http://domainb.foo`), such as an image, using the `img` element (`http://domainb.foo/image.jpg`). This occurs very commonly on the web today — pages load a number of resources in a cross-site manner, including CSS stylesheets, images and scripts, and other resources.

Cross-site HTTP requests initiated from within scripts have been subject to well-known restrictions, for well-understood security reasons. For example HTTP Requests made using the `XMLHttpRequest` object were subject to the same-origin policy. In particular, this meant that a web application using `XMLHttpRequest` *could only* make HTTP requests to the domain it was loaded from, and not to other domains. Developers expressed the desire to safely evolve capabilities such as `XMLHttpRequest` to make cross-site requests, for better, safer mash-ups within web applications.

The Web Applications Working Group within the W3C has recommended the new Cross-Origin Resource Sharing (CORS) mechanism, which provides a way for web servers to support cross-site access controls, which enable secure cross-site data transfers. Of particular note is that this specification is used within an **API container** such as `XMLHttpRequest` as a mitigation mechanism, allowing the crossing of the same-domain restriction in modern browsers. The information in this article is of interest to web administrators, server developers and web developers. Another article for server programmers discussing cross-origin sharing from a server perspective (with PHP code snippets) is supplementary reading. On the client, the browser handles the components of cross-origin sharing, including headers and policy enforcement. The introduction of this new

capability, however, does mean that servers have to handle new headers, and send resources back with new headers.

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

- Invocations of the `XMLHttpRequest` API 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 drawn to a canvas using drawImage.

This article is a general discussion of Cross-Origin Resource Sharing, and includes a discussion of the HTTP headers as implemented in Firefox 3.5.

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 user 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.

Subsequent sections discuss scenarios, as well as 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" here, 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 here.

Simple requests

A simple cross-site request is one that:

- Only uses GET, HEAD or POST. If POST is used to send data to the server, the Content-Type of the data sent to the server with the HTTP POST request is one of `application/x-www-form-urlencoded`, `multipart/form-data`, or `text/plain`.
- Does not set custom headers with the HTTP Request (such as `X-Modified`, etc.)

> *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.*

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:

```
var invocation = new XMLHttpRequest();
var url = 'http://bar.other/resources/publ

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

Let us look at what the browser will send 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
Accept: text/html,application/xhtml+xml,appl
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0
Connection: keep-alive
Referer: http://foo.example/examples/access-
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 by Firefox 3.5. Note that 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 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 be 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:

- It uses methods **other** than `GET`, `HEAD` or `POST`. Also, if `POST` is used to send request data with a Content-Type **other** than `application/x-www-form-urlencoded`, `multipart/form-data`, or `text/plain`, e.g. if the `POST` request sends an XML payload to the server using `application/xml` or `text/xml`, then the request **is** preflighted.
- It sets custom headers in the request (e.g. the request uses a header such as `X-PINGOTHER`)

> *Note: Starting in Gecko 2.0, the `text/plain`, `application/x-www-form-urlencoded`, and `multipart/form-data` data encodings can all be sent cross-site without preflighting. Previously, only `text/plain` could be sent without preflighting.*

An example of this kind of invocation might be:

```
1  var invocation = new XMLHttpRequest();
2  var url = 'http://bar.other/resources/post
3  var body = '<?xml version="1.0"?><person><
4
5  function callOtherDomain(){
6    if(invocation)
7      {
8        invocation.open('POST', url, true);
9        invocation.setRequestHeader('X-PINGO
10       invocation.setRequestHeader('Content
11       invocation.onreadystatechange = hand
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 (`POST`) uses a Content-Type of `application/xml`, and since a custom header is set, this request is preflighted.

Let's take a look at the full exchange between client and server:

```
OPTIONS /resources/post-here/ HTTP/1.1
Host: bar.other
User-Agent: Mozilla/5.0 (Macintosh; U; Intel
Accept: text/html,application/xhtml+xml,appl
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0
Connection: keep-alive
Origin: http://foo.example
Access-Control-Request-Method: POST
Access-Control-Request-Headers: X-PINGOTHER


HTTP/1.1 200 OK
Date: Mon, 01 Dec 2008 01:15:39 GMT
Server: Apache/2.0.61 (Unix)
Access-Control-Allow-Origin: http://foo.exam
Access-Control-Allow-Methods: POST, GET, OPT
Access-Control-Allow-Headers: X-PINGOTHER
Access-Control-Max-Age: 1728000
Vary: Accept-Encoding, Origin
Content-Encoding: gzip
Content-Length: 0
Keep-Alive: timeout=2, max=100
Connection: Keep-Alive
Content-Type: text/plain

POST /resources/post-here/ HTTP/1.1
Host: bar.other
User-Agent: Mozilla/5.0 (Macintosh; U; Intel
Accept: text/html,application/xhtml+xml,appl
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0
Connection: keep-alive
X-PINGOTHER: pingpong
Content-Type: text/xml; charset=UTF-8
Referer: http://foo.example/examples/preflig
Content-Length: 55
Origin: http://foo.example
Pragma: no-cache
Cache-Control: no-cache

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


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.exam
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. Firefox 3.1 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 an **idempotent** 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 11 and 12 respectively):

```
1 Access-Control-Request-Method: POST
2 Access-Control-Request-Headers: X-PINGOTHE
```

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 an `X-PINGOTHER` custom header. The server now has an opportunity to determine whether it wishes to accept a request under these circumstances.

Lines 15 - 27 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 18-21:

```
1 Access-Control-Allow-Origin: http://foo.ex
2 Access-Control-Allow-Methods: POST, GET, O
3 Access-Control-Allow-Headers: X-PINGOTHER
4 Access-Control-Max-Age: 1728000
```

The server responds with `Access-Control-Allow-Methods` and says that `POST`, `GET`, and `OPTIONS` are viable methods to query the resource in question. Note that this header is similar to the HTTP/1.1 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`, confirming that this is a permitted header 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, 1728000 seconds is 20 days.

Requests with credentials

The most interesting capability exposed by both `XMLHttpRequest` and Access Control is the ability to make "credentialed" requests that are aware of HTTP Cookies and HTTP Authentication information. By default, in cross-site `XMLHttpRequest` invocations, browsers will **not** send credentials. A specific flag has to be set on the `XMLHttpRequest` object 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:

```
1  var invocation = new XMLHttpRequest();
2  var url = 'http://bar.other/resources/cred
3
4  function callOtherDomain(){
5    if(invocation) {
6      invocation.open('GET', url, true);
7      invocation.withCredentials = true;
8      invocation.onreadystatechange = handle
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.

Here is a sample exchange between client and server:

```
GET /resources/access-control-with-credentia
Host: bar.other
User-Agent: Mozilla/5.0 (Macintosh; U; Intel
Accept: text/html,application/xhtml+xml,appl
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0
Connection: keep-alive
Referer: http://foo.example/examples/credent
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_s
X-Powered-By: PHP/5.2.6
Access-Control-Allow-Origin: http://foo.exam
Access-Control-Allow-Credentials: true
Cache-Control: no-cache
Pragma: no-cache
Set-Cookie: pageAccess=3; expires=Wed, 31-De
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.
Important note: when responding to a credentialed request, server **must** specify a domain, and cannot use wild carding. The above example would fail if the header was wildcarded as: `Access-Control-Allow-Origin: *`. Since the `Access-Control-Allow-Origin`

explicitly mentions `http://foo.example`, the credential-cognizant content is returned to the invoking web content. Note that in line 22, a further cookie is set.

All of these examples can be seen working here. The next section deals with the actual HTTP headers.

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:

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

The `origin` parameter specifies a URI that may access the resource. The browser must enforce this. For requests **without** credentials, the server may specify "*" as a wildcard, thereby allowing any origin to access the resource.

For example, to allow http://mozilla.com to access the resource, you can specify:

```
1 Access-Control-Allow-Origin: http://mozill
```

If the server specifies an origin host rather than "*", then it must 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

(Firefox 4 / Thunderbird 3.3 / SeaMonkey 2.1)

This header lets a server whitelist headers that

browsers are allowed to access. For example:

```
1 Access-Control-Expose-Headers: X-My-Custom
```

This allows the `X-My-Custom-Header` and `X-Another-Custom-Header` headers to be exposed to the browser.

Access-Control-Max-Age

This header indicates how long the results of a preflight request can be cached. For an example of a preflight request, see the above examples.

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

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

Access-Control-Allow-Credentials

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.

```
1 Access-Control-Allow-Credentials: true | f
```

Credentialed requests are discussed above.

Access-Control-Allow-Methods

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.

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

An example of a preflight request is given above,

including an example which sends this header to the browser.

Access-Control-Allow-Headers

Used in response to a preflight request to indicate which HTTP headers can be used when making the actual request.

```
1 Access-Control-Allow-Headers: <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-origing sharing request headers programmatically.

Origin

Indicates the origin of the cross-site access request or preflight request.

```
1 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` can be the empty string; this is useful, for example, if the source is a `data` URL.*

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

Access-Control-Request-Method

Used when issuing a preflight request to let the server know what HTTP method will be used when the actual request is made.

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

Examples of this usage can be found above.

Access-Control-Request-Headers

Used when issuing a preflight request to let the server know what HTTP headers will be used when the actual request is made.

```
1 Access-Control-Request-Headers: <field-nam
```

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; aims to supplant CORS spec.
CORS	REC Recommendation	Initial definition.

Browser compatibility

Desktop Mobile

Feature	Chrome	Firefox (Gecko)	Internet Explorer	Opera	Saf
Basic support	4	3.5	8 (via XDomainRequest) 10	12	4

Note

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

- Code Samples Showing `XMLHttpRequest` and Cross-Origin Resource Sharing
- Cross-Origin Resource Sharing From a Server-Side Perspective (PHP, etc.)
- Cross-Origin Resource Sharing specification
- `XMLHttpRequest`
- Further Discussion of the Origin Header
- Using CORS with All (Modern) Browsers
- Using CORS - HTML5 Rocks