

Using CORS



By Monsur Hossain

Published: October 26th, 2011

Updated: October 29th, 2013

Comments: 12

Introduction

APIs are the threads that let you stitch together a rich web experience. But this experience has a hard time translating to the browser, where the options for cross-domain requests are limited to techniques like [JSON-P](#) (which has limited use due to security concerns) or setting up a custom proxy (which can be a pain to set up and maintain).

[Cross-Origin Resource Sharing](#) (CORS) is a W3C spec that allows cross-domain communication from the browser. By building on top of the XMLHttpRequest object, CORS allows developers to work with the same idioms as same-domain requests.

The use-case for CORS is simple. Imagine the site [alice.com](#) has some data that the site [bob.com](#) wants to access. This type of request traditionally wouldn't be allowed under the browser's [same origin policy](#). However, by supporting CORS requests, [alice.com](#) can add a few special response headers that allows [bob.com](#) to access the data.

As you can see from this example, CORS support requires coordination between both the server and client. Luckily, if you are a client-side developer you are shielded from most of these details. The rest of this article shows how clients can make cross-origin requests, and how servers can configure themselves to support CORS.

Making a CORS Request

This section shows how to make a cross-domain request in JavaScript.

Creating the XMLHttpRequest object

CORS is supported in the following browsers:

- Chrome 3+
- Firefox 3.5+
- Opera 12+
- Safari 4+
- Internet Explorer 8+

(see the complete list of supported browsers at <http://caniuse.com/#search=cors>)

Chrome, Firefox, Opera and Safari all use the [XMLHttpRequest2 object](#). Internet Explorer uses the similar XDomainRequest object, which works in much the same way as its XMLHttpRequest counterpart, but adds additional [security precautions](#).

To get started, you will first need to create the appropriate request object. Nicholas Zakas wrote [a simple helper method](#) to help sort out the browser differences:

```
function createCORSRequest(method, url) {
    var xhr = new XMLHttpRequest();
    if ("withCredentials" in xhr) {

        // Check if the XMLHttpRequest object has a "withCredentials"
        // property.
        // "withCredentials" only exists on XMLHttpRequest2 objects.
        xhr.open(method, url, true);

    } else if (typeof XDomainRequest != "undefined") {

        // Otherwise, check if XDomainRequest.
        // XDomainRequest only exists in IE, and is IE's way of making
        // CORS requests.
        xhr = new XDomainRequest();
        xhr.open(method, url);

    } else {

        // Otherwise, CORS is not supported by the browser.
        xhr = null;

    }
    return xhr;
}

var xhr = createCORSRequest('GET', url);
if (!xhr) {
    throw new Error('CORS not supported');
}
```

Event handlers

The original XMLHttpRequest object had only one event handler, onreadystatechange, which handled all responses. Although onreadystatechange is still available, XMLHttpRequest2 introduces a bunch of new event handlers. Here is a complete list:

Event Handler	Description
onloadstart*	When the request starts.
onprogress	While loading and sending data.
onabort*	When the request has been aborted. For instance, by invoking the abort() method.
onerror	When the request has failed.
onload	When the request has successfully completed.
ontimeout	When the author specified timeout has passed before the request could complete.
onloadend*	When the request has completed (either in success or failure).

* starred items are not supported by IE's XDomainRequest

source: <http://www.w3.org/TR/XMLHttpRequest2/#events>

For most cases, you will at the very least want to handle the onload and onerror events:

```
xhr.onload = function() {
    var responseText = xhr.responseText;
```

```
console.log(responseText);  
// process the response.  
};  
  
xhr.onerror = function() {  
  console.log('There was an error!');  
};
```

Browsers don't do a good job of reporting what went wrong when there is an error. For example, Firefox reports a status of 0 and an empty statusText for all errors. Browsers also report an error message to the console log, but this message cannot be accessed from JavaScript. When handling onerror, you will know that an error occurred, but not much else.

withCredentials

Standard CORS requests do not send or set any cookies by default. In order to include cookies as part of the request, you need to set the XMLHttpRequest's `.withCredentials` property to true:

```
xhr.withCredentials = true;
```

In order for this to work, the server must also enable credentials by setting the Access-Control-Allow-Credentials response header to "true". See the [server section](#) for details.

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

The `.withCredentials` property will include any cookies from the remote domain in the request, and it will also set any cookies from the remote domain. Note that these cookies still honor same-origin policies, so your JavaScript code can't access the cookies from document.cookie or the response headers. They can only be controlled by the remote domain.

Making the request

Now that your CORS request is configured, you are ready to make the request. This is done by calling the `send()` method:

```
xhr.send();
```

If the request has a body, it can be specified as an argument to `send()`.

And that's it! Assuming the server is properly configured to respond to CORS requests, your onreadystatechange handler will fire with the response, just like the standard same-domain XHR you are so familiar with.

End-to-End Example

Here is a full working sample of a CORS request. Run the sample and watch the network requests in the browser's debugger to see the actual request being made.

[Run Sample](#)

```

// Create the XHR object.
function createCORSRequest(method, url) {
    var xhr = new XMLHttpRequest();
    if ("withCredentials" in xhr) {
        // XHR for Chrome/Firefox/Opera/Safari.
        xhr.open(method, url, true);
    } else if (typeof XDomainRequest != "undefined") {
        // XDomainRequest for IE.
        xhr = new XDomainRequest();
        xhr.open(method, url);
    } else {
        // CORS not supported.
        xhr = null;
    }
    return xhr;
}

// Helper method to parse the title tag from the response.
function getTitle(text) {
    return text.match('<title>(.*?)</title>')[1];
}

// Make the actual CORS request.
function makeCorsRequest() {
    // This is a sample server that supports CORS.
    var url = 'http://html5rocks-cors.s3-website-us-east-1.amazonaws.com/index.html';

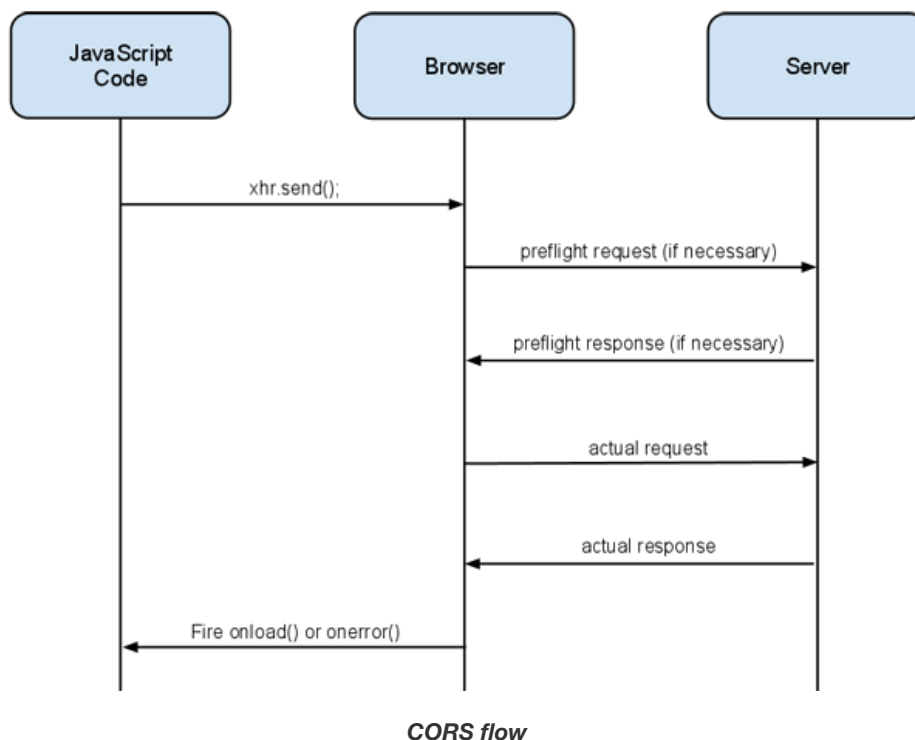
    var xhr = createCORSRequest('GET', url);
    if (!xhr) {
        alert('CORS not supported');
        return;
    }

    // Response handlers.
    xhr.onload = function() {
        var text = xhr.responseText;
        var title = getTitle(text);
        alert('Response from CORS request to ' + url + ': ' + title);
    };

    xhr.onerror = function() {
        alert('Woops, there was an error making the request.');
```

Adding CORS support to the server

Most of the heavy lifting for CORS is handled between the browser and the server. The browser adds some additional headers, and sometimes makes additional requests, during a CORS request on behalf of the client. These additions are hidden from the client (but can be discovered using a packet analyzer such as [Wireshark](#)).



Browser manufacturers are responsible for the browser-side implementation. This section explains how a server can configure its headers to support CORS.

Types of CORS requests

Cross-origin requests come in two flavors:

1. simple requests
2. "not-so-simple requests" (a term I just made up)

Simple requests are requests that meet the following criteria:

- HTTP Method matches (case-sensitive) one of:
 - HEAD
 - GET
 - POST
- HTTP Headers matches (case-insensitive):
 - Accept
 - Accept-Language
 - Content-Language
 - Last-Event-ID
 - Content-Type, but only if the value is one of:
 - application/x-www-form-urlencoded
 - multipart/form-data
 - text/plain

Simple requests are characterized as such because they can already be made from a browser without using CORS. For example, a JSON-P request can issue a cross-domain GET request. Or HTML could be used to do a form POST.

Any request that does not meet the criteria above is a not-so-simple request, and requires a little extra communication between the browser and the server (called a preflight request), which we'll get into below.

Handling a simple request

Lets start by examining a simple request from the client. The table below shows the JavaScript code for a simple GET request on the left, along with the actual HTTP request that the browser emits; CORS specific headers are in bold.

JavaScript:

```
var url = 'http://api.alice.com/cors';  
var xhr = createCORSRequest('GET', url);  
xhr.send();
```

HTTP Request:

```
GET /cors HTTP/1.1  
Origin: http://api.bob.com  
Host: api.alice.com  
Accept-Language: en-US  
Connection: keep-alive  
User-Agent: Mozilla/5.0...
```

The first thing to note is that a valid CORS request **always** contains an Origin header. This Origin header is added by the browser, and can not be controlled by the user. The value of this header is the scheme (e.g. http), domain (e.g. bob.com) and port (included only if it is not a default port, e.g. 81) from which the request originates; for example: http://api.alice.com.

The presence of the Origin header does not necessarily mean that the request is a cross-origin request. While all cross-origin requests will contain an Origin header, some same-origin requests might have one as well. For example, Firefox doesn't include an Origin header on same-origin requests. But Chrome and Safari include an Origin header on same-origin POST/PUT/DELETE requests (same-origin GET requests will not have an Origin header). Here is an example of a same-origin request with an Origin header:

HTTP Request:

```
POST /cors HTTP/1.1  
Origin: http://api.bob.com  
Host: api.bob.com
```

The good news is that browsers don't expect CORS response headers on same-origin requests. The response to a same-origin request is sent to user, regardless of whether it has CORS headers or not. However, if your server code returns an error if the Origin doesn't match a list of allowed domains, be sure to include the origin the request comes from.

Here's a valid server response; the CORS-specific headers are bolded

HTTP Response:

```
Access-Control-Allow-Origin: http://api.bob.com
Access-Control-Allow-Credentials: true
Access-Control-Expose-Headers: FooBar
Content-Type: text/html; charset=utf-8
```

All CORS related headers are prefixed with "Access-Control-". Here's some more details about each header.

Access-Control-Allow-Origin (required) - This header must be included in all valid CORS responses; omitting the header will cause the CORS request to fail. The value of the header can either echo the Origin request header (as in the example above), or be a '*' to allow requests from any origin. If you'd like any site to be able to access your data, using '*' is fine. But if you'd like finer control over who can access your data, use an actual value in the header.

Access-Control-Allow-Credentials (optional) - By default, cookies are not included in CORS requests. Use this header to indicate that cookies should be included in CORS requests. The only valid value for this header is `true` (all lowercase). If you don't need cookies, don't include this header (rather than setting its value to `false`).

The `Access-Control-Allow-Credentials` header works in conjunction with the [`withCredentials` property](#) on the `XMLHttpRequest` 2 object. Both these properties must be set to `true` in order for the CORS request to succeed. If `.withCredentials` is `true`, but there is no `Access-Control-Allow-Credentials` header, the request will fail (and vice versa).

It's recommended that you don't set this header unless you are sure you want cookies to be included in CORS requests.

Access-Control-Expose-Headers (optional) - The `XMLHttpRequest` 2 object has a `getResponseHeader()` method that returns the value of a particular response header. During a CORS request, the `getResponseHeader()` method can only access simple response headers. Simple response headers are defined as follows:

- Cache-Control
- Content-Language
- Content-Type
- Expires
- Last-Modified
- Pragma

If you want clients to be able to access other headers, you have to use the `Access-Control-Expose-Headers` header. The value of this header is a comma-delimited list of response headers you want to expose to the client.

Handling a not-so-simple request

So that takes care of a simple GET request, but what if you want to do something more? Maybe you want to support other HTTP verbs like PUT or DELETE, or you want to support JSON using `Content-Type: application/json`. Then you need to handle what we're calling a not-so-simple request.

A not-so-simple request looks like a single request to the client, but it actually consists of two requests under the hood. The browser first issues a preflight request, which is like asking the server for permission to make the actual request. Once permissions have been granted, the browser makes the actual request. The browser handles the details of these

two requests transparently. The preflight response can also be cached so that it is not issued on every request.

Here's an example of a not-so-simple request:

JavaScript:

```
var url = 'http://api.alice.com/cors';  
var xhr = createCORSRequest('PUT', url);  
xhr.setRequestHeader(  
    'X-Custom-Header', 'value');  
xhr.send();
```

Preflight Request:

```
OPTIONS /cors HTTP/1.1  
Origin: http://api.bob.com  
Access-Control-Request-Method: PUT  
Access-Control-Request-Headers: X-Custom-Header  
Host: api.alice.com  
Accept-Language: en-US  
Connection: keep-alive  
User-Agent: Mozilla/5.0...
```

Like the simple request, the browser adds the Origin header to every request, including the preflight. The preflight request is made as an HTTP OPTIONS request (so be sure your server is able to respond to this method). It also contains a few additional headers:

Access-Control-Request-Method - The HTTP method of the actual request. This request header is always included, even if the HTTP method is a simple HTTP method as defined earlier (GET, POST, HEAD).

Access-Control-Request-Headers - A comma-delimited list of non-simple headers that are included in the request.

The preflight request is a way of asking permissions for the actual request, before making the actual request. The server should inspect the two headers above to verify that both the HTTP method and the requested headers are valid and accepted.

If the HTTP method and headers are valid, the server should respond with the following:

Preflight Request:

```
OPTIONS /cors HTTP/1.1  
Origin: http://api.bob.com  
Access-Control-Request-Method: PUT  
Access-Control-Request-Headers: X-Custom-Header  
Host: api.alice.com  
Accept-Language: en-US  
Connection: keep-alive  
User-Agent: Mozilla/5.0...
```

Preflight Response:

```
Access-Control-Allow-Origin: http://api.bob.com  
Access-Control-Allow-Methods: GET, POST, PUT
```


Access-Control-Allow-Headers: X-Custom-Header
Content-Type: text/html; charset=utf-8

Access-Control-Allow-Origin (required) - Like the simple response, the preflight response must include this header.

Access-Control-Allow-Methods (required) - Comma-delimited list of the supported HTTP methods. Note that although the preflight request only asks permissions for a single HTTP method, this response header can include the list of all supported HTTP methods. This is helpful because the preflight response may be cached, so a single preflight response can contain details about multiple request types.

Access-Control-Allow-Headers (required if the request has an **Access-Control-Request-Headers** header) - Comma-delimited list of the supported request headers. Like the **Access-Control-Allow-Methods** header above, this can list all the headers supported by the server (not only the headers requested in the preflight request).

Access-Control-Allow-Credentials (optional) - Same as simple request.

Access-Control-Max-Age (optional) - Making a preflight request on *every* request becomes expensive, since the browser is making two requests for every client request. The value of this header allows the preflight response to be cached for a specified number of seconds.

Once the preflight request gives permissions, the browser makes the actual request. The actual request looks like the simple request, and the response should be processed in the same way:

Actual Request:

```
PUT /cors HTTP/1.1
Origin: http://api.bob.com
Host: api.alice.com
X-Custom-Header: value
Accept-Language: en-US
Connection: keep-alive
User-Agent: Mozilla/5.0...
```

Actual Response:

```
Access-Control-Allow-Origin: http://api.bob.com
Content-Type: text/html; charset=utf-8
```

If the server wants to deny the CORS request, it can just return a generic response (like HTTP 200), without any CORS header. The server may want to deny the request if the HTTP method or headers requested in the preflight are not valid. Since there are no CORS-specific headers in the response, the browser assumes the request is invalid, and doesn't make the actual request:

Preflight Request:

```
OPTIONS /cors HTTP/1.1
Origin: http://api.bob.com
Access-Control-Request-Method: PUT
Access-Control-Request-Headers: X-Custom-Header
Host: api.alice.com
```

```
Accept-Language: en-US
Connection: keep-alive
User-Agent: Mozilla/5.0...
```

Preflight Response:

```
// ERROR - No CORS headers, this is an invalid request!
Content-Type: text/html; charset=utf-8
```

If there is an error in the CORS request, the browser will fire the client's `onerror` event handler. It will also print the following error to the console log:

```
XMLHttpRequest cannot load http://api.alice.com. Origin http://api.bob.com
is not allowed by Access-Control-Allow-Origin.
```

The browser doesn't give you a lot of details on why the error occurred, it only tells you that something went wrong.

A word about security

While CORS lays the groundwork for making cross-domain requests, the CORS headers are not a substitute for sound security practices. You shouldn't rely on the CORS header for securing resources on your site. Use the CORS headers to give the browser directions on cross-domain access, but use some other security mechanism, such as cookies or [OAuth2](#), if you need additional security restrictions on your content.

CORS from JQuery

JQuery's [\\$.ajax\(\)](#) method can be used to make both regular XHR and CORS requests. A few notes about JQuery's implementation:

- JQuery's CORS implementation doesn't support IE's `XDomainRequest` object. But there are JQuery plugins that enable this. See <http://bugs.jquery.com/ticket/8283> for details.
- The `$.support.cors` boolean will be set to `true` if the browser supports CORS (This returns `false` in IE, see bullet above). This can be a quick way to check for CORS support.

Here's sample code for making a CORS request with JQuery. The comments give more details on how certain properties interact with CORS.

```
$.ajax({
    // The 'type' property sets the HTTP method.
    // A value of 'PUT' or 'DELETE' will trigger a preflight request.
    type: 'GET',

    // The URL to make the request to.
    url: 'http://html5rocks-cors.s3-website-us-east-
1.amazonaws.com/index.html',

    // The 'contentType' property sets the 'Content-Type' header.
    // The JQuery default for this property is
    // 'application/x-www-form-urlencoded; charset=UTF-8', which does
    not trigger
    // a preflight. If you set this value to anything other than
```

```

// application/x-www-form-urlencoded, multipart/form-data, or
text/plain,
// you will trigger a preflight request.
contentType: 'text/plain',

xhrFields: {
  // The 'xhrFields' property sets additional fields on the
XMLHttpRequest.
  // This can be used to set the 'withCredentials' property.
  // Set the value to 'true' if you'd like to pass cookies to the
server.
  // If this is enabled, your server must respond with the header
  // 'Access-Control-Allow-Credentials: true'.
  withCredentials: false
},

headers: {
  // Set any custom headers here.
  // If you set any non-simple headers, your server must include
these
  // headers in the 'Access-Control-Allow-Headers' response header.
},

success: function() {
  // Here's where you handle a successful response.
},

error: function() {
  // Here's where you handle an error response.
  // Note that if the error was due to a CORS issue,
  // this function will still fire, but there won't be any
additional
  // information about the error.
}
});

```

Cross-Domain from Chrome Extensions

Chrome extensions support cross-domain requests in a two different ways:

1. Include domain in manifest.json - Chrome extensions can make cross-domain requests to any domain *if* the domain is included in the "permissions" section of the manifest.json file:

```
"permissions": [ "http://*.html5rocks.com"]
```

The server doesn't need to include any additional CORS headers or do any more work in order for the request to succeed.

2. CORS request - If the domain is not in the manifest.json file, then the Chrome extension makes a standard CORS request. The value of the Origin header is "chrome-extension://[CHROME EXTENSION ID]". This means requests from Chrome extensions are subject to the same CORS rules described in this article.

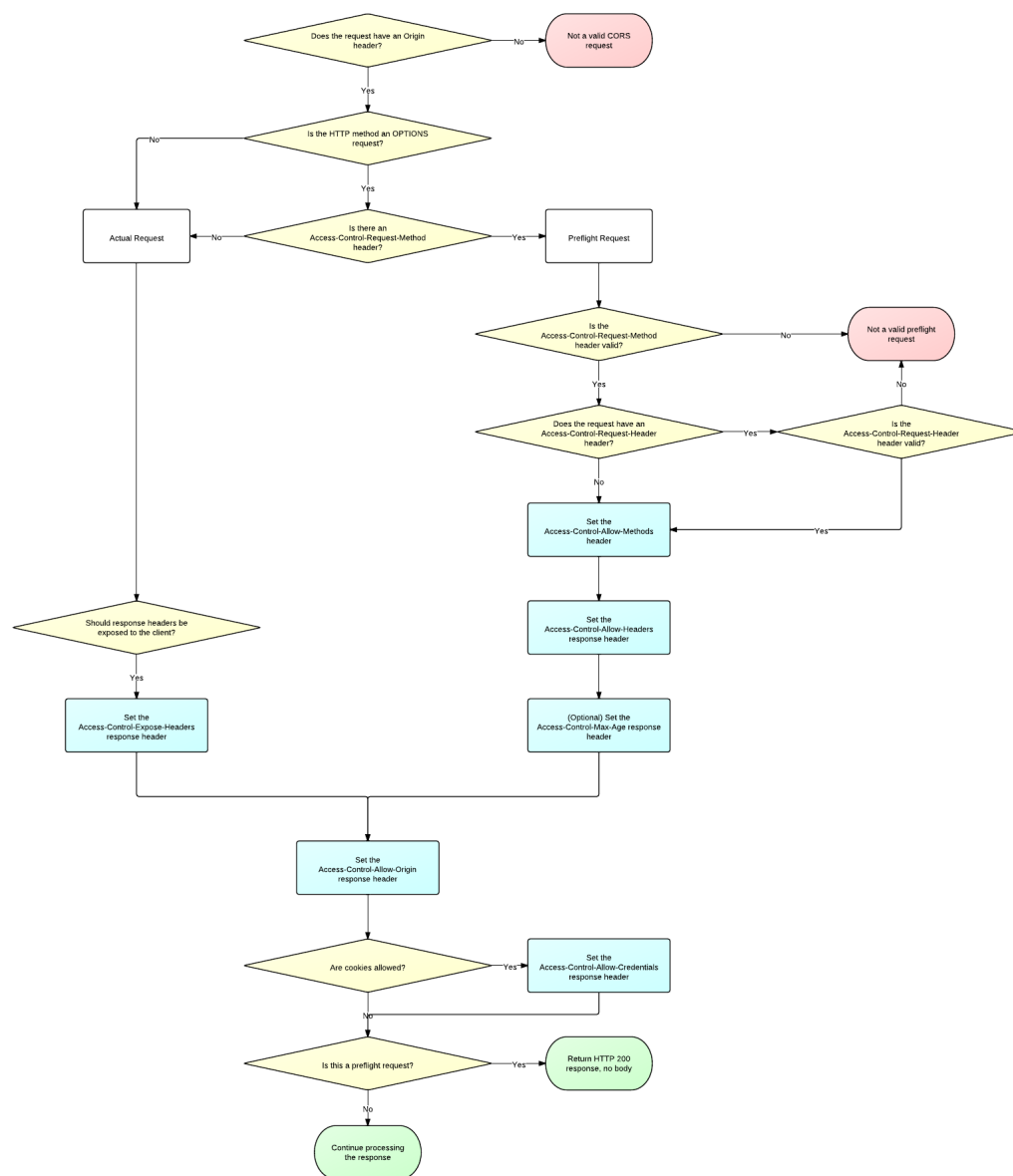
Known Issues

CORS support is still being actively developed in all browsers; here's a list of known issues (as of 10/2/2013):

1. ~~FIXED XMLHttpRequest's getAllResponseHeaders() doesn't honor Access-Control-Expose-Headers~~ Firefox doesn't return response headers when calling `getAllResponseHeaders()`. ([Firefox bug](#)). A similar [bug in WebKit](#) has been fixed.
2. No error information provided to onerror handler - When the onerror handler is fired, the status code is 0, and there is no statusText. This may be by design, but it can be confusing when trying to debug why CORS requests are failing.

CORS Server Flowchart

The flowchart below shows how a server should make the decisions as to which headers to add to a CORS response. Click the image to see a larger version.



CORS Server Flowchart

CORS w/ Images

In Canvas and WebGL contexts, cross origin images can pose big problems. You can use the `crossOrigin` attribute on the image element to address much of them. Read [Chromium](#)

Blog: [Using Cross-domain images in WebGL](#) and [Mozilla Hacks: Using CORS to load WebGL textures from cross-domain images](#) for the details. Implementation specifics can be found at the MDN page for [CORS-enabled Image](#).

Resources

Here are some resources if you'd like to learn more about CORS:

- [The CORS Spec](#)
- A [good intro](#) to CORS from Nicholas Zakas
- [enable-cors.org](#) - More details on how to enable CORS on your server.