Understanding Cross-Origin Resource Sharing (CORS)

CORS, which stands for Cross-Origin Resource Sharing, is an HTML5 feature that allows one site to access another site's resources despite being under different domain names. Let me explain that a little more. Prior to CORS, a web browser security restriction, known as the Same-Origin Policy, would prevent my web application from calling an external API. The browser would consider two resources to be of the same-origin only if they used the same protocol (http vs. https), the same port, and the same domain (even different subdomains would fail).

Before CORS, you could get around this security restriction by creating some sort of server-side component to shuttle API requests, which was often unduly complicated and unnecessary. You could also use JSONP (JSON with padding) in APIs that supported it but many did not, and, even if they did, JSONP is limited to GET requests only.

With CORS, my web app on one domain can freely communicate with your API on another domain, even using the methods POST, PUT, and DELETE, provided that your API's security restrictions specify that this is allowed and that you have established the communication through the CORS specification as well. This means that you can eliminate the need for a server-side component and do all the API communication on the client-side using JavaScript.

In this article, you'll learn about the CORS specification—how to enable it, use it with specific browsers, simplify it with JSON, and more—through a simple code example that uses the GET method to retrieve information from APIS on other domains.

CORS server-side settings

The focus of this article is on client-side communication using CORS, but let's take a quick look at resources you can use for understanding CORS server-side settings. These settings are all established through response headers that browsers interpret.

The W3C specification for CORS actually does a pretty good job of providing some simple examples of the response headers, such as the key header, `Access-Control-Allow-Origin`, and other headers that you must use to enable CORS on your web server. The specification understandably does not go into detail about any specific web server (such as IIS, Apache, and so forth). It also discusses the concept of the "preflight request" that you must use for requests such as PUT or DELETE that are not "simple methods" (simple methods are defined specifically as GET, HEAD, and POST).

In the specification, after offering some suggestions for security on cross-origin requests, the syntax section of the specification goes into detail on various types of headers you can specify on your server and what each header does. There are a variety of headers that enable you to fine-tune security such as the Access-Control-Allow-Credentials header, which, if enabled, allow you to share of things like cookies and HTTP authentication information. Another example is the Access-Control-Max-Age header, which specifies how long to cache a preflight request for non-simple method requests.

If you are looking for specific information on how set up CORS on a variety of common web servers, check out the Enable CORS sharing website. The site explains how to set up the Access-Control-Allow-Origin response header on a variety of web servers from Apache to IIS to ExpressJS and others. It does not go into detail on how to set up other response headers or how to handle preflight requests.

Finally, I recommend reading the good Using CORS tutorial by Monsur Hossain on the HTML5 Rocks website. While looking at the client-side communication, Monsur walks through the various response header types and how to handle complex requests like PUT or DELETE. The tutorial isn’t specific to any web server but, when combined with the instructions from enable-cors.org, should allow you to get up and running.

Browser support

To examine browser support, I built a simple example that performs a basic GET from an cross-origin API. While browser support for CORS is fairly widespread and a growing number of APIs do too, there are still a lot of APIs that don't support CORS yet. One API that has supported CORS for some time is the GitHub API. In my example below, I call the GitHub API using a GET request to retrieve a list of repositories based upon a "JavaScript" keyword and then populate the page with the results.

```
https://api.github.com/legacy/repos/search/javascript', true);      // Response
handlers.      xhr.onload = function () {          var repos =
JSON.parse(xhr.response), i, reposHTML = "";          for (i = 0; i <
repos.repositories.length; i++) {            reposHTML += "

" + repos.repositories[i].name + "
"  + repos.repositories[i].description + "

";         }         document.getElementById("allRepos").innerHTML = reposHTML;
    };          xhr.onerror = function () {         alert('error making the request.');
    };          xhr.send();    }
```

Note: This is a pretty trivial example I created for the purposes of illustration. I intentionally did not use of any frameworks.

The onLoadHandler() function creates an XMLHttpRequest and opens it for a GET request to the GitHub API URL. The third parameter for the open method, is set to true, and specifies that this request is asynchronous.

Next, the code snippets create the event handlers for the request. We are only handling the onload and onerror events but there are a number of other events available using CORS including onloadstart, onprogress, onabort, ontimeout, and onloadend events. Within the onload method in this code snippet, I parse the JSON response and populate some very simple HTML inside a div tag. Once you have enabled the GitHub API for CORS requests, and used code to create these event handlers and send the request(s), you will not encounter a security error.

Chrome, FireFox, Opera, Safari browsers

Chrome supported CORS through the XMLHttpRequest level 2 as of version 3 (which seems like ages ago). Read more about this in the tutorial, New Tricks for XMLHttpRequest2, by Eric Bidelman The above example works well in Chrome. Firefox version 3.5 and up have support for CORS and the example runs well in the current version. Opera support was added somewhat later, with support for CORS only arriving as of version 12 (the current release is 12.1), however, this example runs well in the current release. I was unable to test the example code on Safari (Safari for Windows, which is the platform I am on, is deprecated) but seeing as support was added in version 4, I suspect this simple example would run well there as well.

Internet Explorer

Sadly, Internet Explorer is the only browser that merits its own section here. Technically, Internet Explorer 9 (the current version) supports CORS, but not through the XMLHttpRequest object. Rather, IE uses the XDomainRequest object, which, for purposes of our simple example, works mostly the same other than the call to open() doesn't accept the asynchronous argument. Here is the code using XDomainRequest for Internet Explorer.

```
function onloadHandler() {    var xhr = new XDomainRequest();    xhr.open('GET',
'https://api.github.com/legacy/repos/search/javascript');    // Response handlers.
   xhr.onload = function () {      var repos = JSON.parse(xhr.response), i,
reposHTML = "";      for (i = 0; i < repos.repositories.length; i++) {
         reposHTML += "

" + repos.repositories[i].name + "
"  + repos.repositories[i].description + "

";      }      document.getElementById("allRepos").innerHTML = reposHTML;    };
       xhr.onerror = function () {      alert('error making the request.');    };
   xhr.send(); }
```

If you were to test this code example, however, you'd see that it still doesn't work. Why? Well it turns out that XDomainRequest has a number of additional security restrictions, as described by Eric Law's blog: XDomainRequest - Restrictions, Limitations and Workarounds. One the security restrictions takes an "overly broad" view of restricting calls between different security protocols (for HTTP only, not HTTPS). While sending an unsecure HTTP call from a secure page would be undesirable and one could sympathize with blocking this, Internet Explorer also blocks calls to a secure request from unsecure pages. Thus, since GitHub's API calls all occur over HTTPS, you cannot call them from a page using an insecure HTTP. There is a convoluted workaround but it seems that in the end these issues will be resolved in Internet Explorer 10, which supports CORS via XMLHttpRequest; read more about this on the Internet Explorer blog, CORS for XHR in IE10.

Using jQuery

You can further simplify using CORS by relying on jQuery as it supports CORS requests through the jquery.ajax() method. Obviously, the same restrictions apply, as in it won't start magically working in Internet Explorer 9, but it can simplify your code dramatically. For example, the following jQuery code creates the functional equivalent as the code example above, but with far fewer lines of code.

```
   https://api.github.com/legacy/repos/search/javascript").done(function(data) {
       var i, repo;        $.each(data.repositories, function (i, repo) {
         $("#allRepos").append("

" + repo.name + "
"+ repo.description + "

");        });      });   });
```

As you can see, the CORS request to the GitHub API and handling the result are significantly simpler.

Where to go from here

My code examples only dig into doing a simple GET request through CORS. Obviously, this is no different than what you can achieve today through JSONP, except that we don't need to rely on the JSONP format. However, the real power of CORS lies in the ability to do POST, PUT, and other types of requests. For example, Google's YouTube API announced CORS support back in May in the blog, Unlocking JavaScript's Potential with CORS, which included a YouTube Uploads Using CORS example that allows you to upload a video to YouTube. All of the authentication and posting of the video occurs on the client-side through JavaScript (you can view the source on the example as well).

For more information on how to create and handle these types of requests, I highly recommend the Using CORS tutorial on the HTML5Rocks website.

+