Cross-Domain AJAX with Pure JavaScript

Making use of web services using nothing but JavaScript (and a clever hack)

Frank W. Zammetti fzammetti@omnytex.com http://www.zammetti.com

Mashups are all the rage these days; mixing together various web services using JavaScript as the glue to create a whole new application is a very seductive siren song. AJAX would seem to be the perfect viable for reaching this nirvana… and it likely would be were it not for that annoying same-domain restriction slapping you in the face every time! In this article you'll learn how to get around that limitation using nothing but JavaScript and some fancy script-foo!

A long time ago in a galaxy far, far away... an author was sued for copyright infringement!

Wait, let's try that again.

Once upon a time, there was the Web. And it was good. You clicked a link, a new page appeared, and you did this time and again to get around a site and get some useful porninformation.

Then, a guy by the name of Jesse James Garrett decided we could do better and that laundry detergent was the natural way to do it!

From that insight, AJAX was born. AJAX, or Asynchronous JavaScript and XML, is a technique made possible by a creation of Microsoft in the form of the `XMLHttpRequest` object (so take that all you people that say they never innovate!) that allows for out-of-band partial page refreshes. In simpler terms this means that you can refresh some portion of a web page without having to reload the whole thing again.

On the surface this sounds like a pretty simple and pedestrian development, nothing Earth-shattering. When we stop and think about it, however, it represents a fundamental paradigm shift in the way webapps are developed. It allows you to create webapps that are more like "proper" fat-clients, the Visual Basic, Swing and C++ apps that are the bread and butter of operating systems like Windows, Mac and Linux.

Even though the acronym suggests and the original deployments used JavaScript and XML, the fact is that neither of those are requirements. It's really the concept underlying AJAX that matters. Although the JavaScript part has stuck around (you'll rarely see anyone doing anything else, although in IE, for example, you can still use VBScript instead), the XML part has all but gone the way of the dodo. JSON, or JavaScript Object Notation, is where it's at today. JSON is of course just a simple way to represent an object, like so:

```
{ firstName : 'Frank', lastName : 'Zammetti', age : 36 }
```

That represents a person object with three fields: `firstName`, `lastName` and `age`. If you `eval()` this string of text (ignoring the fact that `eval()` is considered evil by most!) you'll get a real, honest-to-goodness JavaScript object that you can use like so (assuming `p` is the resultant object): `alert(p.firstName);`

Passing strings of JSON back and forth via AJAX has proven to be much better in most respects than XML; it's less verbose, easier to create and easier to marshal back and forth between the string representation that you need going across the wire and objects in JavaScript. So, AJAX could probably rightly be called AJAJ today, where XML is replaced by JSON, by and large.

The problem with AJAX: Domain limitations

Now, AJAX is an extremely important technique to master in modern web development, but it comes burdened by one extremely significant limitation: an AJAX request can only be made to the domain that served the script making the request. In other words, if page1.html is served from www.example.com, then JavaScript in this page can only make an AJAX request to example.com. This limitation is very much on purpose, for security reasons, and is called the same domain/origin restriction policy (it's referred to in a couple of different ways, but it all means the same thing).

The reason this is a significant limitation is because it means you cannot have an application running in a browser that accesses data from other domains. For instance, if you wanted to retrieve a list of stock prices from a provider of such information and display it in a ticker on the page served by your application's (separate) do-

main, you can't pull it off with AJAX using the `XMLHttpRequest` object (not without playing some games anyway). This means that creating so-called "sovereign" webapps, that is, an application that runs completely in a browser without the need for a server (apart from the initial load of the application possibly), isn't possible with AJAX.

Or is it?

Hacks 'R' Us: Ways around the limitation

There are in fact a couple of ways around this limitation that allow you to write those sovereign webapps. One of the most common is the server-side proxy approach. In this approach, the JavaScript code running in the users' browser makes requests not directly to the domain it wishes to get data from but instead makes them to a proxy running on the server the page was served from. This proxy then makes the request to the target domain on behalf of the client, and returns the data it gets from the target domain to the client. No same domain restriction issues here since the browser only communicates with the domain from which it was served.

The down-side to this is that you need a server component to make things work. The other subtle problem is that the client must be served from that server. The alternative, running the client directly from a local file system, won't work. This means the application can't even be launched without the server, which isn't necessarily as important a consideration, but still is something you'd like to be able to do in some situations. Indeed, a true sovereign webapp should be runnable without the need for a server even to serve it.

Another more recent technique that allows you to achieve the sovereign webapp goal and avoid the server-side of things is the iframe "hack", as I like to call it. This technique hinges on the fact that iframes can modify each others' URL fragment identifiers (the portion of a URL after the # mark, like 123 in the URL http://www.mydomain.com/page1.html#123) without security restrictions and without causing the iframe's contents to reload. Since iframes can access any domain, there is no same-domain issue in play with them.

The technique is somewhat trickier to make use of than the server-side proxy approach. To wit, the parent page changes the fragment identifier on an iframe contained within it. The iframe, which has a poller running via a JavaScript interval, notices the changed identifier. The identifier instructs the iframe to load contents from a given URL. The iframe dutifully does so, and the contents are returned. The result comes back, which includes a bit of JavaScript that updates the fragment identifier on the parent page. The parent page, which also has a poller running, sees the change and extracts the fragment identifier, which is typically a bit of JSON that is the real data returned from the remote service, and voila, you have a cross-domain AJAX call (not a true AJAX call mind you, but here I'm referring to the underlying concept, not the `XMLHttpRequest` object itself).

While this gets around the same-domain restriction, and avoids the server-side component, and also allows you to run an application off the local file system rather than having to get it off a server, it requires cooperation from the target domain. More importantly though, it's a fairly complex string of events that has to occur for it to work and it also has implications in terms of CPU usage on the client machine due to the poller processes running. Perhaps the most significant problem with this technique though is that you are limited to how much data you can send and receive because you're manipulating a URL, which means the request is a GET, which are typically limited to around 2k. Internet Explorer is the bottleneck here, as most other browsers allow much larger URLs. Even given that though, that fact is there is any limitation at all is probably a deal-breaker (and by the way, servers have limitations as well, so to be safe you pretty much have to stick with under 2k, which isn't much).

Another technique is to use browser plug-ins as a bridge to a remote service. Flash is probably the most-used in this case, but others like Silverlight and Java applets are also sometimes seen. Here, you have an embedded object that acts as the proxy for your JavaScript (Flash, for example, does not have domain limitations either, so it's a good choice).

As the title states, we're here to talk about pure JavaScript solutions though, so requiring a plug-in isn't what we're interested in, and likewise for the server-side proxy solution. The iframe trick is neat and fits the bill in terms of pure JavaScript, but it has other drawbacks, most notably payload size and general complexity. Is there another option perhaps?

I sure hope so, otherwise this article is about to end abruptly!

Enter the better hack: JSON-P

There's no question that the iframe approach is one big hack, albeit a pretty clever one. Unfortunately, until newer versions of HTML and JavaScript currently in development are fully available (they have concessions with regard to the domain origin policy) hacks are the best we can do. However, there is one hack that is arguably a bit cleverer, and is in many ways better: JSON-P.

JSON-P, or JSON with Padding, is a neat trick based on the fact that `<script>` tags in a document get loaded and immediately evaluated. More importantly, even `<script>` tags dynamically inserted into the DOM after the document has loaded are similarly loaded and evaluated immediately.

With that fact in mind, it's not much of a leap to realize that the function call in Listing 1 would work.

```
myFunction( { firstName : 'Frank', lastName :
'Zammetti', age : 36 } );
```

Listing 1: passing an object as a parameter to a function

If a server were to serve that content up as the content of a `<script>` tag, the browser would immediately evaluate it upon loading. What would happen is that `myFunction()` would be called, and passed to it would be a person object, just like we saw earlier. Since `<script>` tags have no domain restrictions placed on them, we have ourselves a viable hack!

So, how do you actually put this into practice? A properly setup JSON-P webapp will involve the following sequence of events:

Dynamically insert a `<script>` tag into the current document via JavaScript whose `src` attribute is the URL of a remote JSON-P service we wish to get data from.

The browser requests the resource and inserts the returned JavaScript into the document.

The browser then immediately evaluates the content, which results in a call to some local JavaScript function, passing the data payload returned by the service as a JavaScript object.

Normally, the `<script>` tag is then removed from the DOM, but that is optional.

That's all there is to it! No more domain restrictions, no server component on your end, no large CPU utilization! The only real limitation here is that the remote service you're interacting with must be a JSON-P service, but they are becoming more and more prevalent.

JSON-P in the flesh… err, code!

So, how do you actually write this code? In its simplest form, it's just the page shown in Listing 2.

```
<html>
  <head>
    <title>A simple example</title>

    <script>

      function makeRequest() {
        var scriptTag = document.
createElement('script');
        scriptTag.setAttribute('src',
          'http://local.yahooapis.com/
LocalSearchService/V3/localSearch?' +
          'appid=YahooDemo&&output=json&callback=myCa
llback&' +
          'query=pizza&zip=90210'
        );
        scriptTag.setAttribute('type', 'text/
javascript');
        var headTag = document.
getElementsByTagName('head').item(0);
        headTag.appendChild(scriptTag);
      }

      function myCallback(inResults) {
        var s = '';
        for (var i = 0; i < inResults.ResultSet.
Result.length; i++) {
          s += '<br>';
          for (var p in inResults.ResultSet.Result[i])
{
            s += p + ' = ' + inResults.ResultSet.
Result[i][p] + '<br>';
          }
        }
        document.getElementById('divResults').
innerHTML = s;
      }

    </script>

  </head>

  <body>
    <input type='button' value='Click to call remote
service'
      onClick='makeRequest();'>
    <br><br>
    <div id='divResults'
      style='border:1px solid #000000;width:300ps;heig
ht:300px;overflow:scroll;'></div>
  </body>

</html>
```

Listing 2: A simple example.

This is a complete, working example. If you save this to a file and then load it up in your browser and click the button you'll see some information returned from Yahoo!'s search service, which is a popular JSON-P service (Yahoo!, incidentally, is credited with having come up with JSON-P in the first place).

The Yahoo! services all accept some useful parameters as part of the request, which is typical of JSON-P web services (most people have taken to calling these types of services "web services", or "web APIs"). One parameter to specify is the type of output as JSON. These services can actually return data in a number of formats, but all we care about here is JSON-P of course. You also need to specify a key, which is a registered user allowed to access the service. The Yahoo! demo key is being used here, but you can get your own at http://developer.yahoo.com. This is necessary if you intend to write an application of any significant size in terms of the volume of requests it will be making.

The other key parameter you specify is callback. This is the name of the JavaScript function that the JSON payload returned should be wrapped in. This function must exist on your page in global scope for this all to work.

Something is rotten in the state of JSON-P: Error handling

Now, this JSON-P approach is by no means perfect either, just like the other alternatives. One of the significant problems it has is that of error handling. In short, there basically is none!

If the remote service returns an error code then you're in pretty good shape. So long as your callback can interpret and handle the error information provided in the response, then this is a situation that isn't really a problem.

Unfortunately, that's not the error handling I'm really talking about. The situation that is problematic is if the remote server doesn't furnish a suitable response, then you are basically hosed, to put it bluntly!

For example, if the URL you furnish for the remote service is incorrect and an HTTP 404 (page not found) error results, there's no way to know that. Your callback will simply never fire. This could wind up being not too big of a deal, if your app can gracefully continue to function anyway, or it could be catastrophic, such as a locked UI (imagine showing a "please wait" message that blocks the rest of the UI, but it never gets cleared unless the callback function executes).

Another problem is if the callback you specify for some reason isn't callable when the response comes back. This won't usually be a problem unless you yourself as the developer screws things up, but hey, we all make mistakes, so it could happen!

One fairly decent way around this is to set up a timer that will time out any response that hasn't come back in a specified amount of time. It's not perfect, but it does work. There's obviously a little more to making it work than that, but that is the basic idea.

Listing 3 shows a much more robust approach that incorporates this timeout concept.

```
/**
 * JSONP makes JSON-P requests and does so with a timeout mechanism so you'll
 * always know whether the request was successful or not.
 */
var JSONP = {

  /* Amount of time in seconds before a request is considered timed out. */
  timeoutSeconds : 5,

  /* Reference to the document's head tag. */
  headTag : document.getElementsByTagName('head').item(0),

  /* Collection of objects, one for each in-flight request. */
  requestObjects : { },

  /**
   * Call this method to fire off a JSON-P request.
   *
   * @param inConfig An object containing whatever parameters are needed to
   * make the remote call.  The attributes of this object are used to construct
   * a query string.
   */
  request : function(inConfig) {

    // Create unique ID for request.
    var requestID = 'req_' + new Date().getTime();

    // Create request object and populate with internal data.
    var requestObject = { };
    requestObject.requestID = requestID;
    requestObject.callback = function(inResponse) {
      JSONP.callback(requestID, inResponse);
    };
    requestObject.realCallback = inConfig.callback;
    requestObject.onTimeout = inConfig.onTimeout;
    inConfig.callback = 'JSONP.requestObjects.' + requestID + '.callback';

    // Add query string to URL.
    if (inConfig.url.charAt(inConfig.url.length - 1) != '?') {
      inConfig.url += '?';
    }
```

Listing 3: The JSONP class (JSONP.js) (continued on next page)

```
    var queryString = '';
    for (var attribute in inConfig) {
      var alc = attribute.toLowerCase();
      if (alc != 'url' && alc != 'ontimeout') {
        if (queryString != '') {
          queryString += '&';
        }
        queryString += attribute + '=' + inConfig[attribute];
      }
    }
    inConfig.url += queryString;
    requestObject.inConfig = inConfig;

    // Now create the script tag for this request.
    var scriptTag = document.createElement('script');
    requestObject.scriptTag = scriptTag;
    scriptTag.setAttribute('src', inConfig.url);
    scriptTag.setAttribute('type', 'text/javascript');

    // Now create the timeout.
    requestObject.timeout = setTimeout(function() {
      JSONP.timeoutElapsed(requestID);
    }, this.timeoutSeconds * 1000);

    // Kick off the request.
    this.headTag.appendChild(scriptTag);

    // Finally, put the requestObject in the collection.
    this.requestObjects[requestID] = requestObject;

  },

  /**
   * Internal intermediary callback that the JSON-P request calls.
   *
   * @param inRequestID The ID associated with the requestObject.
   * @param inResponse  The response from the remote server.
   */
  callback : function(inRequestID, inResponse) {

    // Get the request object associated with this request.
    var requestObject = JSONP.requestObjects[inRequestID];

    // Might not have a request object, if the request comes back after the
    // timeout period.
    if (requestObject) {
      // Clear the timeout so the request doesn't time out.
      clearTimeout(requestObject.timeout);
      // Call the specified callback.
      requestObject.realCallback(inResponse);
      // Delete the request object.
      delete JSONP.requestObjects[inRequestID];
    }

  },

  /**
   * This is called when a request timeout occurs.
   *
   * @param inRequestID The ID associated with the requestObject.
   */
  timeoutElapsed : function(inRequestID) {

    // Get the request object associated with this request.
    var requestObject = JSONP.requestObjects[inRequestID];
```

Listing 3: The JSONP class (JSONP.js) (continued on next page)

```
    // There should never be a case where there is no requestObject here,
    // but we'll check for it anyway, just in case I missed something.
    if (requestObject) {
      // Copy pertinent attributes of requestObject to a new object.
      var newRequestObject = { };
      newRequestObject.requestID = requestObject.requestID;
      newRequestObject.inConfig = { };
      for (var i in requestObject.inConfig) {
        newRequestObject.inConfig[i] = requestObject.inConfig[i];
      }
      // Delete the request object, but get onTimeout first.
      var onTimeout = requestObject.onTimeout;
      delete JSONP.requestObjects[inRequestID];
      // Now call the real timeout handler, if any.
      if (onTimeout) {
        onTimeout(newRequestObject);
      }
    }

  },

}; // End JSONP.
```

Listing 3: The JSONP class (JSONP.js)

```
<body>

  <head>

    <script src='jsonp.js'></script>
    <script>

      function test1() {
        var config = {
          url : 'http://search.yahooapis.com/ImageSearchService/V1/imageSearch?',
          appid : 'YahooDemo', query : 'Amanda Tapping', output : 'json',
          callback : test1Callback, onTimeout : timeoutHandler
        };
        JSONP.request(config);
      }

      function badTest() {
        var config = {
          url : 'gibberish', appid : 'gibberish', query : 'gibberish',
          output : 'json', callback : null, onTimeout : timeoutHandler
        };
        JSONP.request(config);
      }

      function timeoutHandler(inRequestObject) {
        // Iterate its attributes
        var s = '';
        for (var i in inRequestObject) {
          s += i + ' = ' + inRequestObject[i] + '\n';
        }
        for (var i in inRequestObject.inConfig) {
          s += i + ' = ' + inRequestObject.inConfig[i] + '\n';
        }
        // Show the output.
        alert('REQUEST TIMED OUT - Request Object Dump: \n\n' + s);
      }

      function test1Callback(inResponse) {
        var outputString = 'Total results returned: ' +
```

Listing 4: A working example of the JSONP class (continued on next page)

```
          inResponse.ResultSet.totalResultsReturned + '<br>';
        for (var i = 0; i < inResponse.ResultSet.Result.length; i++) {
          outputString += inResponse.ResultSet.Result[i].Title + '<br>';
        }
        document.getElementById('divResponse').innerHTML = outputString;
      }

    </script>

  </head>

  <body>

    <input type='button' value='Make Request' onClick='test1();'>
    <br><br>
    <div id='divResponse'
      style='width:400px;height:250px;border:1px solid #000000;overflow:auto;'>
      Search results will appear here</div>
    <br><br>
    <input type='button' value='Make Bad Request' onClick='badTest();'>

  </body>

</html>
```

Listing 4: A working example of the JSONP class.

And how would you actually make use of this? It's not very hard at all and Listing 4 shows such an example.

I also included here a button you can click to try a bad request, so you can see how the class handles failures.

To use this class you simply call the `JSONP.request()` method. You pass to this method an object that configures the call. Only three of the attributes, strictly-speaking, are required.

`url` - The URL of the JSON-P web service to call. This can end with a question mark, but it doesn't have to.

`callback` - This is the function that will be called when the response comes back.

`onTimeout` - This is the function that will be called if the request times out.

The others are completely dependent on the JSON-P services you're calling. Here I'm toying around with Yahoo!'s search API. You can also set the timeout attribute on the `JSONP` object if you want. This is the number of seconds to wait for a response. If no valid response comes back in that time, then the request is considered to have timed out and your `onTimeout` function will be called.

Breaking it down: Walkthrough of the code

Let's run through how this code works so you have an appreciation for the effort it's saving you from!

The call to `doRequest()` results in the creation of a request object that describes the remote call. This object includes a unique ID (`requestID`), the callback function that the caller wants to execute when the response comes back (`realCallback`), and a function to execute in the case of a timeout. This object also includes an attribute `callback` that is an automatically-created function who's only job is to call the `callback()` method of the `JSONP` class. Doing this allows us to form a closure around the `requestID` so that when the response comes back it will not only include the response from the server, but the ID of the request. This is needed so that the request object can be retrieved from the collection of request objects contained with the `JSONP` class so that the response can be processed. This allows for multiple simultaneous requests to occur since each has a request object describing it that can be retrieved when the response comes back.

The next step is to construct a query string to pass to the remote service using the object passed in to this method. Once that's done the `<script>` tag is created with the specified URL, and with the query string appended to it, and the tag is then inserted into the DOM. You'll note that since we're using a query string here you have a limit in the amount of data that can be passed to the remote service. As I said, this isn't a perfect technique either, it's just arguably better.

Right before that insertion though, a `timeout` is kicked off. This `timeout` is set to kick off some number of seconds later, as determined by the `timeoutSeconds` attribute of the `JSONP` class. The function specified here, `JSONP.timeoutElapsed()` is what will be

called if a timeout occurs. Not that it too forms a closure around the `requestID`, again so that the request object can be pulled from the collection to be able to process the timeout.

Now, when the response comes back, the `JSONP.callback()` method will fire. This takes the `requestID` passed into it and uses it to retrieve the appropriate request object from the collection. With that the timeout is cleared, so the request won't time out now, and the real callback function is called. Finally, the request object is deleted from the collection.

In the case of a timeout, the `JSONP.timeoutElapsed()` method fires. The request object is again retrieved, and we then copy the attributes of that object that are needed to a new object. The original request object is then deleted. This copy-then-delete approach is done so that if by chance the response comes back because it simply took too long, the request object won't be found in the collection and the callback won't be called. Finally, the function specified as the `onTimeout` function is called and the timeout is handled as appropriate for the application.

In conclusion

With AJAX in your bag of tricks you have a potent weapon allowing you to create dynamic webapps that rival the best desktop applications. Used properly, AJAX can fundamentally change the way you write webapps and can produce results you (and more importantly, your employer!) will be very happy with.

Adding JSON-P into the mix takes you to an even higher level. The world of mashups and sovereign webapps becomes your playground and you can take the standards-based development model to places never before possible. The ability to "mash up" remote web services with nothing but plain old JavaScript is a potent technique that opens up all sorts of possibilities. One day down the road, we'll have a more "proper" way to do cross-domain AJAX calls, but until that day I hope that the `JSONP` class presented here serves you well. I'm sure there is room for improvement with it, but even as-is it should allow you to pull off some neat tricks without having to do much thinking. And at the end of the day, isn't doing less thinking what we're all really striving for?!?

Learn More

For more information on the APIs and services available from Yahoo!, go to http://developer.yahoo.com/everything.html

---

Frank W. Zammetti is an architect/lead developer/numerous other things for one of the largest financial institutions in the United States. Frank lives in Pennsylvania with his wife Traci and two children Andrew and Ashley. Frank has authored five books on topics such as AJAX, JavaScript, DWR, Dojo and ExtJS as well as a number of articles on various topics. Frank is a contributor to a number of open-source projects, a couple of which he leads and a few he founded. Visit http://www.zammetti.com for details on any of this, and more!