

Using AJAX To Update Data On A Web Page

Contributed by David Hollingworth
 Thursday, 19 July 2007
 Last Updated Thursday, 19 July 2007

Introduction

Regular visitors to tullig.net will know that one of my passions is the weather and that I run a weather station at home. The data is updated on the TulligWeather section of this site with the current conditions being updated every two minutes by regenerating the entire page on the server at home and ftping it up to the tullig.net site every two minutes.

The disadvantages of this approach are:

- The entire page must be ftped up to the server each time
- The page doesn't refresh unless the user manually clicks the refresh button on the browser
- I could get round the previous point by using a refresh directive in the web page head section; but this would still refresh the entire page, which isn't very elegant and is what I want to avoid.

Instead of these techniques I decided to use Asynchronous Javascript And XML (AJAX) to automatically perform the data refreshes on the page without the need to either upload or refresh the entire page. The version of the current conditions you'll see on the site now is using these techniques and will update the data in the table every minute without having to reload the entire web page. Don't forget that in calm conditions the data doesn't change much in which case you might not see the page change much. However keep looking at the time in the top left hand corner of the table to see when updates occur.

In order to implement these techniques on your site you'll need 4 components:

-

A web page template laid out ready to take the data.

-

An XML file, generated by your weather station software and transferred to your web site.

-

A small script to serve up the XML file on request.

-

Some Javascript that requests the XML and updates the web page.

Whilst the examples I've used here all revolve around the display of weather data the same techniques can be used to display any sort of data once you can format it into an XML file.

If you're already displaying weather data on your web site then you're a good way down the road already. So let's make a start by looking at the web page template. [Web Page Template](#)

Most Automated Weather Station (AWS) software allows you to create web page templates into which you insert tags that the AWS replaces with actual weather data. For example I use Virtual Weather Station (VWS) from Ambient Weather. If I want VWS to insert the current temperature into a table cell then I'd put this into my HTML template: `<td>^vxv007^</td>` To prepare the HTML template to receive data we need to replace the AWS tag (`^vxv007^`) with an HTML span tag that has a unique ID. So my current temperature cell becomes: `<td></td>` Note that:

- The span tag is empty for now, it'll be populated by our Javascript when the page is loaded.
- The ID I've used is "curtemp"; but it could have been anything at all. For example I could have used `id="currenttemperature"` which is more readable; but takes longer to type! Eventually I've a whole HTML table full of span tags, each of which has a unique ID. If you don't use uniqueIDs then the HTML will be invalid and this may yield unpredictable results. Finally we need some means of triggering the page to load in the data and this is done by using the `onLoad` event on the web page's body tag: `<body onload="getData();">` Don't worry about where the `getData()`; comes from for now, we'll cover that in the Javascript section. As a further example here's the HTML for the first row of my current conditions table: `<tr><td>Temperature °C</td><td>°C</td><td>°C at </td><td>°C at </td><td>°C</td></tr>`

All the other rows are similar, each table cell having span tags with unique IDs.

Next we'll look at the XML file that drives the transfer of data to the web browser.

The XML File

Instead of our AWS software generating a new HTML document and ftping it to our web server we're going to get it to generate an XML file and send that instead. The XML file is of the format:

```
<?xml version="1.0" encoding="utf-8"?>
<currentconditions>

... Some data tags go in here

</currentconditions>
```

The first line is essential information for the browser, from there on you can call your tags anything at all except that the data tags must be enclosed in 'root' tags. Here I've called the root tags 'currentconditions'; but you can call them anything. Just ensure the opening and closing tag names are exactly the same. The data tags that go between the root tags can also be called anything. However for the Javascript I've written to work properly my data tags must match my HTML span IDs. For example I've got a span for my current temperature whose ID is "curtemp". Therefore the XML tag that will take my current temperature must also be called curtemp. If the XML tag names don't match the HTML span IDs then nothing is going to work. I could have programmed it so you can have different XML tag names and span IDs; but that would have been a lot more work in the Javascript. In case you haven't guessed it the XML data tags will contain the AWS data tags. For example here's the data tag for the current temperature: <curtemp>^vxv007^</curtemp> And here's the complete XML file: <?xml version="1.0" encoding="utf-

```
8"?><currentconditions><curtime>^vst143^</curtime><curdate>^vst142^</curdate><curtemp>^vxv007^</curtemp><maxtemp>^vhi007^</maxtemp><maxtemp>^vht007^</maxtemp><mintemp>^vlo007^</mintemp><mintemp>^vlt007^</mintemp><avgtemp>^vda007^</avgtemp><curdp>^vxv022^</curdp><maxdp>^vhi022^</maxdp><maxdpt>^vht022^</maxdpt><mindp>^vlo022^</mindp><mindpt>^vlt022^</mindpt><avgdp>^vda022^</avgdp><currh>^vxv005^</currh><maxrh>^vhi005^</maxrh><maxrht>^vht005^</maxrht><minrh>^vlo005^</minrh><minrht>^vlt005^</minrht><avgrh>^vda005^</avgrh><curws>^vxv002^</curws><maxws>^vhi002^</maxws><maxwst>^vht002^</maxwst><minws>^vlo002^</minws><minwst>^vlt002^</minwst><avgws>^vda002^</avgws><winddir>^vxv001^</winddir><bft>^vst141^</bft><curwg>^vxv003^</curwg><maxwg>^vhi003^</maxwg><maxwgt>^vht003^</maxwgt><minwg>^vlo003^</minwg><minwgt>^vlt003^</minwgt><avgwg>^vda003^</avgwg><curwc>^vxv019^</curwc><maxwc>^vhi019^</maxwc><maxwct>^vht019^</maxwct><minwc>^vlo019^</minwc><minwct>^vlt019^</minwct><avgwc>^vda019^</avgwc><curmb>^vxv008^</curmb><maxmb>^vhi008^</maxmb><maxmbt>^vht008^</maxmbt><minmb>^vlo008^</minmb><minmbt>^vlt008^</minmbt><avgmb>^vda008^</avgmb><barotrend>^vst140^</barotrend><raintoday>^vxv121^</raintoday><rainhour>^vxv122^</rainhour><rain24hh>^vxv123^</rain24hh></currentconditions>
```

You'll notice that some of the XML tags are short names, like "avgwc". This is really to save me typing time and so long as I've a span with an id="avgwc" all will be well. When your AWS software processes the XML file it will replace all the AWS tags with real data so

that <curtemp>^vxv007^</curtemp> becomes <curtemp>16.9</curtemp> Finally you'll need to reconfigure your AWS software to ftp the XML file up to the web server instead of the HTML file you were transferring before. Having done this we'll need a small script to server up the XML file when it's requested. The Responder Script The job of the responder script is to send the XML file back to the web browser when the browser makes a request for it. I used the Ruby language for my script and here it is: #!/usr/bin/ruby

```
## Define any constants#XMLFile = 'The name and path of your XML file goes in here. E.g. /home/me/myfile.xml'
## Print the headers# puts "Content-type: text/xml" puts "Cache-Control: no-cache, must-revalidate" puts "Expires: Mon, 26 Jul 1997 05:00:00 GMT" puts "\n\n"
## Open the file and loop through its contents putting them to the output#open(XMLFile).each { |f| print f } exit This file has to go into the cgi-bin directory on your web server and must be made executable. Because not all web services provide Ruby here's a PHP version of the same thing: <?php
$handle = fopen("The name and path of your XML file goes in here. E.g. /home/me/myfile.xml", "r");
if ($handle){ // // Send the appropriate headers // header('Content-Type: text/xml'); header("Cache-Control: no-cache, must-revalidate"); //A date in the past header("Expires: Mon, 26 Jul 1997 05:00:00 GMT");
// // Read to the end of the file putting the contents on the // output while (!feof($handle)) { $buffer = fgets($handle);
echo $buffer; } fclose($handle);}
?>
```

NB. I've not tested the above PHP script; but it should work.

You can save the PHP script as a .php file (e.g. responder.php) anywhere you fancy on your web server. It doesn't have to be in the cgi-bin directory.

To test that the XML file and responder script are working then you can call them directly in your web browser. For example: <http://your-host.domain/responder.php>. If all is well you'll get a rendition of your XML file in your browser. Possible problems are:

- If you get a page not found (404) error then you're addressing your responder script incorrectly.
- If the browser reports an error with the XML file then the most likely cause is a typo in one of the XML tags. For example: `<curtemp>^vxv007^</curtemp>` The start and end tags don't match. If the browser displays the XML OK then we're into the final stage, the Javascript. The Javascript

The javascript gets called when the HTML page is loaded using the onload event we saw earlier. The getData() function is a Javascript function that starts the chain of events to load and display the data. I'm not going to go into too much detail in the narrative about the script because it's fairly well commented, except to say that you'll need to put it into a script file, E.g. conditions.js, and save it somewhere sensible on your server. Then you include the script into your HTML page with a line in the head section of the document: `<script type="text/javascript" src="conditions.js"></script>`

Here the conditions.js file is in the same directory as our web page.

OK, here's the Javascript in full. There's a couple of bits you'd have to change for your implementation at the top of the file. You'll have to change the url to point to your domain and location of the responder script. You can also update the interval to a sensible number of seconds depending on how often your XML file is ftped to your server. I ftp my file every 60 seconds so I've got the update interval set to 30 seconds to catch each update as soon as possible.

```
var response;var request = false; // The object to handle the request for data
var reqType = "GET"; // Make the request type a GET as opposed to a POST
var url = "http://your-domain/your-responder-file.php";var asynch = true; // Make this an asynchronous request
var interval = 1000 * 30; // Update the page at 30 second intervals
function getData(){ // // Do the first request, then // httpRequest(); // // send the request at intervals //
setInterval(httpRequest, interval);}
/* Wrapper function for constructing a Request object.*/function httpRequest(){ // // Only create the request object. //
//Mozilla-based browsers if(window.XMLHttpRequest){ request = new XMLHttpRequest(); } else if
(window.ActiveXObject) { // IE browsers request=new ActiveXObject("Msxml2.XMLHTTP"); if (! request) {
ActiveXObject("Microsoft.XMLHTTP"); } }
//the request could still be null if neither ActiveXObject //initializations succeeded if(request) {
initReq(reqType,url,asynch); // Initialize the request } else { alert("Your browser does not permit the use of all "+
application's features!"); } }
/* Initialize a Request object that is already constructed */function initReq(reqType,url,bool){ /* Specify the function that
will handle the HTTP response */ request.onreadystatechange = handleResponse; // // In order to prevent IE browsers
from returning a cached version of // the XML file we append a unique value to the end of the URL. This is // ignored by
the responder script; but ensures the page gets updated. // var urlToSend = url + "?key=ms" + new Date().getTime();
request.open(reqType, urlToSend, bool); request.send();}
//event handler for XMLHttpRequestfunction handleResponse(){ // alert("readyState = " + request.readyState + "status = "
+ request.status); if(request.readyState == 4){ if(request.status == 200){ // // Get the XML document back
the request object // and display the results. // var doc = request.responseXML; displayDocInfo(doc);
alert("A problem occurred with communicating between the " + "XMLHttpRequest object and the server program.");
alert("request.status = " + request.status); } }//end outer if}
//// Loop through the XML document using the element names to// identify the ids of the span tags in the HTML
document.//function displayDocInfo(doc){ // Get the root of the XML document var root = doc.documentElement; var nds;
var thisTag; // If the root has children, i.e. if there's data elements // under the root: if(root.hasChildNodes()) {
nds=root.childNodes; // alert("Root node has " + nds.length + " children"); // For each of the child nodes for (var
i=0; i< nds.length; i++) { // Get the corresponding span tag from the HTML document, // e.g. span id="curtemp" /spa
thisTag = document.getElementById(nds[i].nodeName); // If there's some data contained in this node then i
(thisTag) { if(nds[i].hasChildNodes) { // set the tyag value to the XML node value
nds[i].firstChild.nodeValue; } else { // Set the tag value to an empty space thi
} // // We need to use the current time elsewhere so just get this nodes value // var localtime =
root.getElementsByTagName('curtime')[0].firstChild.nodeValue; thisTag = document.getElementById('localtime');
thisTag.innerHTML = localtime; } return;}

```

If all is well then:

- Your AWS software produces a new XML file every minute, having replaced the AWS tags with data, and ftps this to your web server.

- Every 30 seconds the Javascript makes a request of the responder script which sends up the latest XML file.
- The Javascript then uses the XML tag names to identify the span IDs and to set the span's innerHTML content to the XML data values.