Ajax What is it Good For?

An Introduction to Ajax: Part 1

Edward Traversa

http://www.dhtmlnirvana.com/
28 December 2005

Example Files

Note you will need to use either a local or remote server to view the example files. They can be downloaded from here:

Download Files

A Brief History

Many a year ago, Jeff Rouyer won the Netscape DHTML competition with his flying dude interface and put DHTML on the map. The winning entry, apart from being 3D in design which up until that point had not been seen on the Web, also had something that was quite uncharacteristic of Web pages.

Most of what you see today on the web in terms of DHTML Widgets can be attributed to a small number of people with Jeff leading the way. Granted there has been a progression over the years, but things like scrollers, tooltips, drag and drop, event handling, dynamic clipping etc were pioneered by Jeff many years before they became popular. For example, the sliding doors technique used in CSS today was used around ten years ago with JavaScript by Jeff. In short the guy is the grand daddy of DHTML and everyone else pales in comparison. I kid you not!

One of the unusual features of Jeff's Award winning site was than rather than have a user traverse across multiple Web pages which consumes resources by opening separate window instances, Jeff was loading content dynamically into an existing interface. In other words all the content was being loaded into an existing element and all other elements were not being changed or swapped. We are talking pre Internet Explorer 4 days here.

Internet Explorer 4 was released a little later to challenge Netscape's dominance on the web, but it did not have a similar method where content could be loaded directly into a layer element by using *layer.load();*

As the popularity of Internet Explorer 4 gained it became increasingly important to Jeff to maintain cross browser compatibility and initially created a series of *document.write()* in an external file to allow content to be imported into a pre-existing html file. Obviously this was resource intensive and he went on to improve this method for Internet Explorer 4 by created a little Java Applet to allow Internet Explorer 4 to pass external content into an existing element.

Not long after that, he thought of the idea of using an IFrame technique where content would be loaded into a hidden IFrame and then sucked back out and into a div element. This was the genesis of what today is known as Remote IFrame Scripting. Remember we are talking close to a decade ago which is a good indicator of how far ahead of the curve Jeff was at the time.

Around this time, Jeff and I struck up a friendship and began to work on refining that technique and over the years it progressed to what both Jeff and I currently use on our sites. We call it External HTML Loading.

As our friendship grew we started to think about writing a book. The whole problem with Jeff and I in this regard is that when we work together, we are too busy developing new things and rarely put pen to paper. In short we are too alike and enjoy the idea of creating new techniques as opposed to writing about existing techniques.

The interesting thing about working together in that manner was that we heavily were looking into using XML based techniques combined with existing HTML techniques. While we covered quite a bit of ground, the fact of the matter was that the only browser capable of using XML in any reasonable way was IE5+, so we tossed the idea out the window for the time being.

Enter Ajax

Recently Garret Smith of Adaptive Path made a good case for using Asynchronous JavaScript and XML and coined the acronym "AJAX" to represent that notion. It is similar to the Remote IFrame technique in that things are handled on the client side, but as we shall see differs in significant ways. That article reminded me of our earlier work on this front and sparked my interest in Ajax. Part of that interest is in writing this article and sharing my efforts with the development community.

To my way of thinking, Ajax is more a natural progression from an existing set of techniques as opposed to being something completely brand new. Yet within that progression, things are markedly different than in previous years in a couple of key areas:

- It provides a mechanism to mix and match xml with xhtml.
- It significantly reduces having to continually fetch things from a server (remote IFrame scripting is reasonably comparable here).
- It overcomes some speed bottlenecks that traditional Web development has fallen prey too. In most instances an Ajax based site will load quicker than a comparable traditional Web site.

- When done well, it significantly reduces initial load times.

Before proceeding, it should be noted that Ajax isn't a panacea. It has some problems that it needs to overcome as it continues to mature. But it is a promising field to be involved within and as the years roll on, it probably will greatly affect the way we think about building Web pages and Web applications.

It also needs to be clear that Ajax isn't a technology as such but rather is a technique that combines well with other technologies and techniques. For example, xml, dhtml, css, xhtml. In fact, Ajax really is DHTML with the xmlhttprequest object thrown in. It is amazing to think how one object can change the whole playing field.

As things currently stand in the Ajax field, we almost have two extremes going on. At one end of the continuum we have developers building all sorts of complicated JavaScript tags that in most instances are unnecessary and are overkill. At the other end of the scale we have these little examples floating around the Web that demonstrate how to load xml into a document or do some Ajax form processing.

While both ends of the spectrum have a part to play in the progression of the Web, I am not convinced that the principles of Ajax and how they can best be leveraged by a developer have been conveyed as well as they may have been.

This is particularly evident in the Ajax api's that are available today. Most are appalling poor and suffer from code bloat. It is interesting to note that DHTML Api's of the past suffer from the same problem. I was never an advocate for Api's of this sort and see nothing that changes my mind in this regard.

What is Ajax?

Ajax itself is a technique, but to use the technique effectively one must become familiar with the philosophy behind the technique. In other words, it is not just the use of the technique that is important, but rather developing a different mindset and approach to Web development that is central.

To date, I have mentioned the term Ajax a number of times but have not described the central object that makes Ajax, Ajax. At the heart of Ajax are the xmlhttprequest object and its Microsoft's activeX equivalent. It is this object that allows data to be transferred asynchronously. In case you are not clear what asynchronous means, it is the ability to handle processes independently from other processes. Synchronous which is the opposite of asynchronous, then means that processes are dependent upon other processes. To illustrate let us use a classical Web page scenario.

Assume we have page A and on this page we have a number of elements, including a couple of script and style tags. With synchronous data transfer the script tag needs to be parsed before the next element is parsed. In this way then the next element to be parsed by the browser is dependent upon the script tag being parsed first. Effectively we are creating a bottleneck of one connection between Web page and browser. Style tags and link elements in the head section of the document create the same bottleneck effect. They have to be parsed one at a time before other page elements can download.

Once they have been parsed then the elements in the body section can use concurrent connections to help speed up the download process. For example, most servers handle between 2 to 4 concurrent connections between web page and browser.

Consequently, this means that 2 to 4 images or other page elements can load concurrently. Yet before that process starts what is between the head tags have to be parsed first, which can considerably delay the loading of web pages. Most particularly if you are using multiple CSS and JavaScript tags. Nearly every Web page and blog on the internet uses this method. It isn't hard to see why this may be a problem in terms of increasing page load speeds.

Asynchronously loading methods differ from this scenario because the loading processes are handled independently of each other and to a significant degree overcomes the bottle of traditional Web page design. As will be demonstrated later, the whole gist of this technique is to use minimal JavaScript initially and then push all the head related tags, including the rest of the JavaScript, CSS etc tags through an xmlhttprequest object. This method uses multiple connections rather than the single one that most traditional Web pages use and consequently speeds things up considerably. More on that later as we go through this tutorial.

In this tutorial I will be using xmlhttprequest to represent both versions: the xmlhttprequest object itself, (supported by, Firefox, Opera 8, Safari and later Mozilla builds and the Microsoft equivalent ActiveX object. I do this for the sake of brevity rather than anything else.

What to Expect

We are going to walk through the basics of Ajax and culminate in the building of a little Ajax powered Fading Image Gallery application. In this way it is hoped that a deeper appreciation for what Ajax is capable of is conveyed. It also should be noted that in another upcoming tutorial, we will tweak that Image Gallery widget and add some form processing capabilities to it, add some more FX, get a few glitches out of that system etc.

In other words, this initial tutorial is meant to help you get up and running in building your first Ajax powered Web application. Later we will look at refinement and adding features so that you can gain a more comprehensive understanding of the techniques and mindset required for Ajax. In short this tutorial is not meant to be the be all and end all of Ajax, rather it is better thought of as development in progress. Though the development in progress in this case has some clear aims and techniques to impart that should hopefully serve you well in future Ajax endeavours.

Just before getting into the code, techniques etc, I would just like to advise people that there is no need to attempt to cut and paste code from this tutorial. All files, which include both the image gallery application and a sliding book like interface, plus all examples are provided in a downloadable zip file. You will need to note, that the examples will not run offline unless run a server as a local host. At any rate, providing the examples as a downloadable file gives you a chance to play around with the files and suggest further improvements etc.

The two completed online examples that are provided are the Sliding Book Interface and the Ajax Fading Image Gallery.

Just in case you weren't aware, the site you are now on also uses Ajax where lots of little components are being dynamically loaded as you interact with the web page. Bear that in mind as you read through this tutorial.

Having said all that, there is no better way than to dive into some code and walk you through it.

Onwards we go...

Basic XML Formatting

Let us begin by construct a basic xml file that will be loaded into the main document. Our xml example file (basic.xml) looks like this:

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

<menu>

        <item>

                <description>
```

```
                    Possible Menu Item Here

                    </description>

          </item>

</menu>
```

Remember that XML doesn't really do anything other than describe the content it contains. For example an xml tag such as the following:

```
<mynameis>
   Eddie
</mynameis>
```

Allows the tags to describe what the content is. That is all the xml tags actually do.

In our basic XML example, we are eventually going to build a little navigation system therefore I used the tags, menu, items and description respectively to accurately represent what the tags contain.

Let us now turn our attention to the JavaScript that will allow us to load an xml file into an existing document. In case you are not clear, we are importing an XML Document into an existing HTML document. That is our immediate goal and forms much of the basis of what follows later on. It is worth paying attention to and going into some detail.

Ajax Manager Basic Script

```
Line 1: function ajaxManager()
{
Line 2: var args = ajaxManager.arguments;
Line 3: switch (args[0])
{
Line 4: case "load_page":
Line 5: if (document.getElementById) {
Line 6: var x = (window.ActiveXObject) ? new
ActiveXObject("Microsoft.XMLHTTP") : new XMLHttpRequest();
}
Line 7: if (x)
{
Line 8: x.onreadystatechange = function()
{
```

Line 9: if (x.readyState == 4 && x.status == 200)
{
Line 10: el = document.getElementById(args[2]);

Line 11: el.innerHTML = x.responseText;
}
}
Line 12: x.open("GET", args[1], true);
Line 13: x.send(null);
}
Line 14: break;
Line 15: case "start_up":
Line 16: ajaxManager('load_page', 'basic.xml', 'contentLYR');
Line 17: break;
}
}

As you may have noticed, I have numbered the JavaScript on a line by line basis. The best way to learn JavaScript is line by line, consequently it will be employed through this tutorial when apt.

Let's Dig in to this script.

Line 1: This creates a new function which we call ajaxManager.

Line 2: we create a variable called args that points to all the arguments that the function will contain. We do this to cut down on file size rather that typing out ajaxManager.arguments on each occasion we want to make an argument. In case you do not know what a JavaScript argument is used for consider the following:

```
function alertMe(message){
    alert(message);
}
```

In the alertMe function the argument employed is "message" and is contained within the parentheses. The argument itself is passed to a stock standard alert() method.

To generate an argument to represent the message we can do this on the body onload event.

```
<body onload="alertMe('my name is Eddie')">
```

What this does is then pass 'my name is eddie' to the argument message and when the alert is triggered then my name is eddie is displayed in the alert dialogue.

An argument is a system where you can pass information from an event to the function itself, which is particularly useful when working with programming languages.

Line 3: The function uses a case / switch methodology. Often rather than having a series of if / else statements we can use a case switch method with the same result with much less code and consequently more efficient processing of the script. A basic format for case switching is as follows.

```
switch (expression) {
    case label1:
// Code to be executed
    break;
    case label2:
// Code to be executed
   break;
}
```

In our code the expression is signified by an argument so that later we can use the labels of each case and call them from an event handler. By way of example,

```
onmousedown="functionNameHere ('caselabelhere')"
```

or

```
ajaxManager('start_up')
```

would signify to the browser to use the function ajaxManager and to only execute the start_up case. In using the case / switch method in this manner we are effectively creating sub functions within the main function that often results in code brevity.

Line 4: Creates our first case. In this instance we are creating a case labelled load_page.

Line 5: We don't want older browser to use the script so we create an if condition to allow only browsers that support document.getElementById

Line 6: Now we are getting into the guts of the function. To use XMLHttpRequest with Internet Explorer 5 we need to use an Active X object where as Mozilla based browsers, Opera and Safari support XMLHttpRequest directly. So we have to create a little conditional switch to ensure that the right method is used for the right browsers. We do this by fist creating a variable named x and checking for the existence of an ActiveX object using a conditional operator. If an activeX object is found then the statement uses the method for IE if not it skips over to using XMLHttpRequest that is used by the other modern browsers.

For Internet Explorer, if we wanted to, we could use try and catch JavaScript methods to create some conditional branches to implement version dependent instances of XMLHTTP. For example Msxml2.XMLHTTP.3.0. In the next tutorial, we will implement these methods because they also are useful for error catching.

Using independent version conditions makes sense if you want to use a specific feature you know is supported by a particular version of XMLHTTP. In this instance however that isn't applicable so we can use the version independent method which is Microsoft.XMLHTTP. That will work on IE5 through to IE7. It also has the added benefit of keeping our code nice and lean. And we like lean code don't we!

Line 7: We check that either the Microsoft.XMLHTTP or XMLHttpRequest objects have been created through the usage of the previously named variable x.

Line 8: The onreadystatechange event is used to keep track of when the readystate of a document changes. In this statement we attach the event to Microsoft.XMLHTTP or XMLHttpRequest respectively by using the variable x.

We then create a function to handle all the processing that will follow. This is particularly useful to monitor the parsing and loading process of documents. The following readystates are available to developers.

0	uninitialized	Object is not initialized with data.
1	Loading	Object is loading its data.
2	loaded	Object has finished loading its data.
3	interactive	User can interact with the object even though it is not fully loaded.
4	complete	Object is completely initialized.

Line 9: We have another conditional statement to check if the readystate = 4 has been triggered; which means that the document loading is complete and ready to be used and also check that the status of the document is completed by asking whether status = 200. If this check is ok then the following statements are triggered.

Line 10: We want to create a JavaScript object to load the xml file into. We could just load into the document.body, but using a container is far more flexible as you can then position imported xml data anywhere you like. To this end we create a variable named el and point it to the document.getElementById(args[2]) statement. The argument in the statement will allow us to later specify which element we want to load the data into. We could have just used document.getElementById('contentLYR') where contentLYR would be a block element in the document body (usually a div tag) with an id value of contentLYR. And that would be perfectly acceptable in a lot of instances. However later on we are going to do some multiple data loading into different containers in the document so we want as much flexibility as possible.

Line 11: The next thing we want to do is actually place the data into an element. I find using innerHTML efficient in handling this task so let us use that method. Basically this line tells the browser to place the data into an element and display it.

Line 12: We want to open a request and obtain the data so this method allows for that. It uses three arguments Get, args[1] and true. Get as the term implies retrieves the data, the second argument args[1] will allow us to specify which data file to obtain and the third tells the browser to load the data asynchronously. If this third argument is set to true the data loads asynchronously and if set to false it does not.

The following table lists common XMLHttpRequest object methods:

Method	Description
abort()	Stops the current request
getAllResponseHeaders()	Returns complete set of headers (labels and values) as a string
getResponseHeader("headerLabel")	Returns the string value of a single header label
open("method", "URL"[, asyncFlag[, "userName"[, "password"]]])	Assigns destination URL, method, and other optional attributes of a pending request
send(content)	Transmits the request, optionally with postable string or DOM object data
setRequestHeader("label", "value")	**Assigns a label/value pair to the header to be sent with a request**

Line 13: If we wanted to transmit some data to and from a server we would specify the data to be transmitted. But since we don't want to send data we set the send argument to null

Line 14: A break statement closes the case.

That is the core of an xml asynchronous loading technique. And it is well worth the effort to get to know it intimately. If you don't understand it, try reading it again until you do. The next case is used to signify what is going to happen when the document loads.

Line 15: Creates a new case called start_up.

Line 16: This statement tells the browser what to do when the original document (not the document being loaded into the interface what to do. Here we specify that the ajaxManager function should be called, then to use the case load page, then to retrieve the xml file named basic.xml and to put it into a div tag called contentLYR.

Line 17: A break statement to close the case.

That's it for the JavaScript section. It's a pretty lightweight script that is capable of quite a few things that will be demonstrated later on.

Let's now turn our attention to the original document where the xml data will be loaded into. It's a pretty simple set up. All we need to do is insert a div tag with an id attribute and property of contentLYR like so;

```
<div id="contentLYR">
</div>
```

Then in a style sheet give it some properties and values. This is what I have used for this example:

```
<style type="text/css">
#contentLYR {
position:absolute;
width:200px;
height:115px;
z-index:1;
left: 200px;
top: 200px;
}
</style>
```

Nothing out of the ordinary there. You will just need some familiarity with CSS to arrange layout and presentation. There are a bunch of good CSS tutorials on the Web if you find yourself in unfamiliar territory.

Finally we call the ajaxManager and specify which case to employ from the body tag using an onload event like so:

```
<body onload="ajaxManager('start_up')">
```

Let's take a look at how this handles by viewing the example:

View Basic Example

As you can see the function does what it intends to do, but it doesn't really highlight the versatility of XMLHttpRequest. Let's go with a few different examples to demonstrate.

XML Basic List Menu

In this next example, we change the start_up case to the following which indicates to the browser that we want navigation.xml loaded through the XMLHttpRequest object.

```
case "start_up":
    ajaxManager('load_page', 'navigation.xml', 'contentLYR');
break;
```

Everything else script wise remains the same, therefore no need to change anything other than that line from the previous script we just built. The navigation.xml file looks like this:

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

  <menu>

    <item id="about">

      <ul xmlns="http://www.w3.org/1999/xhtml">

       <li><a href="http://dhtmlnirvana.com/" tabindex="10">DHTML
Nirvana</a></li>

      <li><a href="http://www.truthrealization.com/" tabindex="10"
class="menu">Truth Realization</a></li>
```

```
    <li><a href="http://dhtmlnirvana.com/michiebaby/" tabindex="10" class="menu">Michie Baby</a></li>

   </ul>

  </item>

</menu>
```

You may notice that within the xml file I have used some html markup to define a list. To an xml parser the file is pure xml and the relevance of html or xhtml tags is of no consequence to the parser. In other words there is no difference between a div tag or an eddie tag in xml in terms of what the tag does.

To illustrate this concept let us look at two examples. This is a pretty important concept to grasp so bear with me. The first example takes a look at what happens when we point the browser to an xml file:

View Example

In this example, the browser parses the document and doesn't recognize the html tags other than as typical xml description tags. So none of the html markup gets applied and the document is not formatted as a list.

But watch what happens when we add an xhtml namespace to the xml file on the <ul> tag like so:

```
<ul xmlns="http://www.w3.org/1999/xhtml">
```

To really appreciate what is happening, view the following example, in either a Mozilla based browser or Opera (probably Safari will work here too?). Internet Explorer will not parse the document properly without the use of its proprietary parser. Bad IE!

View Example

As you can see the xml document is now formatted and works like an xhtml document would. This is because of the namespace which allows the browser to interpret the tags and then render them accordingly. Love those namespaces!

Unfortunately that still leaves us with the Internet Explorer formatting problem so to circumvent that, we import the xml document into an existing interface using XMLHttpRequest and its IE equivalent and we end with this:

View Example

Once you have an xml document into the interface using this method it is pretty simple to style using CSS as you would a normal html document. Then you can make people say "oh ahh, that's wicked". In other words pretty the elements up and make them presentable to a user.

Interactive XML

There is an added benefit that is not readily apparent in the examples thus far. We can add interactivity into the XML file.

If you have seen xml examples before, you may have noticed that many developers walk the DOM tree to get an event to trigger or provide interaction from the XML file. Sometimes you may indeed have to do this, so this is not an argument against walking the DOM tree, but rather is an argument against walking the DOM tree unnecessarily.

In many instances walking the DOM tree bloats code with the added disadvantage of having to use more processing power to work through the code. By using namespaces and existing xhtml technologies we can leverage the best out of both worlds and get highly functional Web applications, while avoiding code bloat and unnecessary performance lags.

In the next example we are going to trigger a simple show / hide event from the XML file. There are two lines in the xml file (examples 5 folder) that are important to look at.

Line 4: `<ul xmlns="http://www.w3.org/1999/xhtml" id="eddies">`

Line 13: `<a href="#" onmousedown="ajaxManager('hide_menu')">Hide Menu</a>`

Let's see what each line does.

Line 4: On this line we add an id attribute with a value of "eddies". We do this so we can target that particular element with JavaScript later on.

Line 13: On this line we create a an anchor tag and denote the onmousedown event to fire a hide_menu case in the ajaxManager() function. You will note that there is no need to use a parent object as in parent.ajaxManager('hide_menu') which would be the case if we were to use frames. When the xml document is imported and fully loaded into the main interface it acts as it were part of the interface itself. Take note of that as it will save a lot of headaches later on.

As you may have noticed we are pointing to a case in a JavaScript function, so let us look at that.

```
case "hide_menu":
      document.getElementById("eddies").style.visibility = "hidden";
break;
```

All we are doing with the addition of this case to the ajaxManager function is first targeting the ul elements by obtaining the id value of eddie that we set earlier and then setting its visibility property to hidden.

You can see how this works in practice from the following link:

View Example

Pretty neat huh? And what is cool is that not a lot of additional JavaScript was needed to dynamically manipulate that file. The simpler you can keep things, the better off you will be.

Importing Different Documents

One of the limitations of the xmlhttprequest object is that it cannot handle the importing of media such as images, flash, svg etc directly. To conceptualize this, let us attempt to import an image directly into a document via xmlhttprequest. Again this is one of those scenarios where it is better to view the example with Firefox, Opera 8 rather than IE. IE will just throw an error, which is perfectly acceptable, but doesn't give us much in the way of conceptualising what is occurring.

View Example

As you can see the image doesn't render since it is a binary format which cannot be parsed correctly by an xmlhttprequest object. And since we want to build an image gallery widget powered by Ajax we definitely need images imported into a document. Let's fix that.

The way around this dilemma is to import an html file with an image tag in the document. Oh you didn't know you can import html as well as xml? Well now you know so bob's your uncle. Let us look at the example:

View Example

There is nothing extraordinary about the html file. It is just a simple basic html file with an xhtml DTD and contains an image tag. That's it.

Our start_up case in the JavaScript ajaxManager() function has simply changed to reflect to pointing to the correct file to load into the existing document.

```
case "start_up":
      ajaxManager('load_page', 'image1.html', 'contentLYR');
break;
```

It might be prudent to step back for a second and think about what is occurring with this technique. When we are importing the html file, the html file itself is read as a string and the image is not rendered. But once loaded into the document, the browser then interprets the html file as an html file and consequently the image does indeed render. The xmlhttprequest object retrieves the document; the html markup renders it, just like in the earlier xml example with a namespace. If you keep the head section of the imported document relatively free, then we don't hit the one connection snag and instead are using multiple connections constantly.

There is also a conceptual framework that needs to be explained in terms of using ajax effectively. Once the html file has been loaded it would be erroneous to think of that html file as a separate window instance or a separate frame or even a separate document.

A way to think of what is occurring is as a union of sorts between the existing interface and the file being imported so that the two become one. Practically it means that we don't have to navigate the documents via a frame based or window instance based approach.

At this point you should be starting to note that a genesis for an image gallery is starting to form. We can use the earlier technique of importing xml files to provide information about the image and we now just have learned how to import the image itself. So the basic functionality for an image gallery widget is in place, but will obviously need refinement to work effectively.

Before we move onto that though, I want to demonstrate a few other things about importing different media with xmlhttprequest.

We can import various file types as long as the browser supports the file type either natively or through a plugin. In this next example, let's bring in some flash using the same technique as the image example that was just presented. Please note that in this next example, I am going to load a 108k flash file.

View Example

Starting to get the picture? By using these methods we effectively are extending xhtml to not only use existing technology that is already supported but also bring in the new dimension of using xml with html. And best of all we aren't breaking W3C standards.

One more important thing to learn before we turn our attention back to the building of the image gallery widget and that is how to import JavaScript dynamically. This is a key concept so we will look at that next.

Speeding Load Times

Your heaviest load times are going to be either the JavaScript itself or CSS files in the head section. This is due to the fact that when JavaScript or CSS is being parsed by a browser it typically just uses one connection to do so where as other elements in the body section load through multiple connections. In short all attention is focused on parsing the JavaScript and the other elements have to wait until the JavaScript is parsed to render.

That is why doing this sort of thing in Web pages is notoriously poor:

```
<script src="blah1.js" type="text/javascript">
</script>
<script src="blah2.js" type="text/javascript">
</script>
<script src="blah3.js" type="text/javascript">
</script>
```

Each of those script tags will use one connection and considerably delay the loading of a page. Same thing goes for CSS and also links tags. You should take a long hard think about that when developing web pages. Multiple CSS files slow your pages down!

To circumvent this we can be clever and push JavaScript files through the xmlhttprequest object and consequently have them load asynchronously. We aren't going to get around using at least one script tag, but if the originating script is kept minimal then our load times are going to be reduced significantly.

In other words load only what is essential to get the web page of to a start and then load the rest of the JavaScript components dynamically or on demand. The same would apply to CSS styles, use minimal styles to provide backward compatibility and push the rest of the css through the xnlhttprequest object.

I like the term JavaScript components because it implies that there are all these components that eventually will come together in a functional way. In the past DHTML typically would load huge JavaScript files that tended to cover all the possibilities in one go, even if all those possibilities were not used by someone viewing the page: A kind of shotgun approach if you will.

But via the use of xmlhttprequest we can change that around and load only what is needed for minimal functionality and then load the rest when a user requests a particular site feature. Not only can this dramatically decrease page file sizes, but it also gets around the one connection per script tag problem.

Let us dive into it by first looking at an example:

View Example

In that example, we are loading in a separate JavaScript file named alert.js through the xmlhttprequest object. The alert.js file looks like this:

```
function showAlert() {
    alert('hi there, I was loaded externally');
}
```

It is just a simple alert contained within a function. The guts of the procedure is contained I the importjs.js file so let us look at that in more detail. In particular we are interested in the load_js case:

We will take a look at that function on the next page.

Line 1: `var getheadTag = document.getElementsByTagName('head')[0];`

Line 2: `setjs = document.createElement('script');`

Line 3: `setjs.setAttribute('type', 'text/javascript');`

Line 4: `getheadTag.appendChild(setjs);`

Line 5: `setjs.text = x.responseText;`

Most of function has been covered before and to avoid excessive repetition I just want to highlight the particular section we haven't covered as yet. You can look at the whole script in the examples 9 folder if you feel lost.

Line 1: The first thing we want to do is grab the head tag. We can do this by using document.getElementsByTagName() and specifying through an argument that the tag to obtain is the head tag.

Line 2: Next we want to create a script element and we can use the W3C DOM method of document.createElement. We use a variable named setjs, so that we can give it some attributes and then later slot it into the head section of the Web document.

Line 3: This line sets an attribute type with a value of text/javascript.

Line 4: We put the script tag and its attributes into the head section of the document using appendChild():

Line 5: We then want to fill the head section with the script itself, so we push the string of data received from the xmlhttprequest object into the head section.

I changed the load_page case to load_js so that you can use the widgets separately should the need arise. These are contained within the examples folder. I will demonstrate how to combine the two into one function when we get on to building the Image Gallery itself.

There is one glitch in this procedure where Safari doesn't recognise when a document is loaded into the interface so dynamically loading js or css fails with this browsers. That is a Safari bug which has nothing to do with the scripting methods used. Hopefully the next version of Safari will remedy this problem.

In the interim what you could do is redirect Safari to use the whole JavaScript file as opposed to breaking the JavaScript file into small components and loading them dynamically. Obviously you would lose some of the speed benefits, but functionality wise it would be equivalent to the other Web 2.0 capable browsers.

Building an Ajax Powered Image Gallery

Having gone through some of the fundamentals it is now time to turn our attention to putting what we have learned thus far into building an Image Gallery.

The first thing we are going to do is set up our entry page to have accessibility features by providing an alternate navigation system when JavaScript is switched off either by a user preference or because the browser doesn't support JavaScript.

We use a list based menu format to achieve this. The code for this is as follows:

```
<div id="accessLYR">
  <ul>
    <li><a href="image1.html" tabindex="0">View Image 1</a></li>
    <li><a href="image2.html" tabindex="1">View Image 2</a></li>
    <li><a href="image3.html" tabindex="2">View Image 3</a></li>
    <li><a href="image4.html" tabindex="3">View Image 4</a></li>
    <li><a href="image5.html" tabindex="4">View Image 5</a></li>
    <li><a href="image6.html" tabindex="5">View Image 6</a></li>
  </ul>
</div>
```

In the Image Gallery application, I am setting the visibility property to hidden for the element accessLYR when the page loads. But that doesn't mean everyone has to.

One thing we haven't discussed yet is the use of arrays. In the interface you are currently using, I have a JavaScript file named setarrays in the script folder. That file contains all the array information for loading pages, displaying titles, page numbers and so on. The file itself is loaded into the interface via the importing JavaScript technique we discussed earlier.

In the gallery widget, more for the sake of conceptual ease than functionality, I have kept the arrays in the one function. Baby steps are good when learning this stuff. Later in part 2 of this tutorial, I will demonstrate how to load them dynamically using the methods we learned earlier.

For now I have added an extra case in the ajaxManager() function called array_setup. This case allows us to store information in arrays like so:

```
galleryArray = new Array();
galleryArray[0] = "image1.html";
galleryArray[1] = "image2.html";

descriptionArray = new Array();
descriptionArray[0] = "description1.xml";
descriptionArray[1] = "description2.xml";
```

As you may imagine from looking at that code snippet, the galleryArray holds all our image.html url information, while the descriptionArray holds our description.xml information. One array is going to load the images, the other some description of the image.

To change files all you would need to do is change the array pointer to reflect a new file. It keeps things nice and simple. Well at least for me it does. Then later we cycle through the arrays to obtain the correct files.

Our load_page function has had a number of things added to it. Do not get confused here, the basic method of importing documents is intact and exactly the same as before. It is only that in order to build the gallery that some extra functionality was thrown in. Let us look at the additions to the load page function so that we can understand what is going:

Line 1: `pageWidth = (dom) ? innerWidth : document.body.clientWidth;`

Line 2: `preload = document.getElementById("preloadLYR");`

Line 3: `gallery = document.getElementById("galleryLYR");`

Let’s go through the lines of code again.

Line 1: A variable named pageWidth is created. It then checks for whether the browser is Internet Explorer or another standards based browser through the use of the variable named dom.

```
var dom = (!document.all && document.getElementById);
```

Which is the very first line in our script. If the browser is anything other than Internet Explorer we snag the current page width dimensions using innerWidth. If it is Internet Explorer then we use document.body.clientWidth;

Just so you gain a bit of a concept as to why we are doing this, later on we are going to use the pageWidth variable to auto center the images for us as they are imported. Shiny!

Line 2: This is a simple JavaScript object created by using a variable and then pointing that variable to an element named preloadLYR. We do this so as to not have to repeatedly type out document.getElementById("preloadLYR");

Line 3: Same as line 2 except we point to an element named galleryLYR.

The next change in the load_page case is seen when looking at the readystates of the xmlhttprequest object. What we are doing here is creating a little preload routine based on readystates. To do this we simply attach innerHTML to the variable preload and assign it a message for the browser to display. For example,

The next change in the load_page case is seen when looking at the readystates of the xmlhttprequest object. What we are doing here is creating a little preload routine based on readystates. To do this we simply attach innerHTML to the variable preload and assign it a message for the browser to display. For example:

```
preload.innerHTML = "initializing";
```

We then execute a series of if else statements based on the readystate of the xmlhttprequest object and change the message accordingly.

Later in separate additions to this first tutorial, we will make an uber cool preload widget, much like the flash ones you see around the Web. To do that we will use some header information and the readystates of the xmlhttprequest object. But for now, this will suffice. Let’s look at the code for the ready states so you can conceptualize what is happening.

```
if (x.readyState == 0)

{
  preload.innerHTML = "initializing";
}
else if (x.readyState == 1)
{
  preload.innerHTML = "processing request";
}
else if (x.readyState == 2)
{
  preload.innerHTML = "request acknowledged";
}
else if (x.readyState == 3)
{
  preload.innerHTML = "loading data..";
  setOpacity(0, 'galleryLYR');
  setOpacity(0, 'captionLYR');
}
else if (x.readyState == 4 && x.status == 200)
{
  preload.innerHTML = "Standy: Fading Image In...";
```

That's the basic logic for a preload routine. You may have noticed on the readystate == 3 branch that we are calling a function named setOpacity(). We do this to ensure that the image opacity is set to 0 before it is rendered by the browser after the readyState == 4 branch has been parsed. The function it hooks into enables us to set opacity for the browsers that support it. Opera will simply just display the images without using opacity. The setOpacity() function looks like this:

```
function setOpacity(opacity, id)
{
        var el = document.getElementById(id).style;
        el.opacity = (opacity / 100);
        el.MozOpacity = (opacity / 100);
        el.KhtmlOpacity = (opacity / 100);
        el.filter = "alpha(opacity=" + opacity + ")";
}
```

Let us take a closer look at the rest of the readyState==4 conditional branch as there is a bit going on there.

Line 1: getImageSize = document.getElementsByTagName("img")[0] .getAttribute("width");

Line 2: gallery.style.left = (pageWidth - getImageSize) / 2+"px";

Line 3: preload.style.left = (pageWidth - getImageSize) / 2+"px";

Line 4: document.getElementById('captionLYR').style. left = (pageWidth - getImageSize) / 2 +"px";

Line 5: document.getElementById('accessLYR').style. visibility = "hidden";

Line 6: setTimeout("fadeIn('galleryLYR', 0, '99.99')", 1200);

Line 7: setTimeout("fadeIn('captionLYR', 0, '99.99')", 1300);

Line 1: With this statement we create a variable named getImageSize. As the variable name implies we are going to the dimensions of the image being imported. Specifically its width attribute. In order to do this, we grab the image tag using document.getElementsByTagName("img"). The [0] indicates to the browser to obtain the first image contained within the document. If we wanted to obtain the second image, we would use [1] and so on. Then we obtain the width attribute by employing getAttribute. All this information is now stored in the variable.

Line 2: The element galleryLYR left position is then set by using the existing document width minus the imported image width and then divided by 2 so as to center the image.

Line 3: We do exactly the same thing for the preloadLYR element.

Line 4: Again here we are doing the same thing for the captionLYR element.

Line 5: We set the accessLYR visibility to hidden for each subsequent page load. Both the main page and those being imported have an accessibility menu so we turn them off with JavaScript.

Line 6: This line calls a function named fadeIn. As the name implies it is the function that drives the fading effect. We delay the fading by 1300 milliseconds by using a setTimeout method to avoid flickering as the image renders on the page. The arguments used for this function are fadeIn(Element ID, Fade Start, Fade End)

Line 7: Same as Line 6 except this time we target the element with an id attribute of galleryLYR.

Our fadeIn function looks like this:

```
function fadeIn(id, startfade, endfade)

{

  timer = 0;

  if (startfade < endfade)

  {

    for (i = startfade; i <= endfade; i++)

    {

      setTimeout("setOpacity(" + i + ",'" + id + "')", (timer * 20));

      timer++;

    }

  }

}
```

Only a couple of more things and we are done for part 1. One thing we want to do is to cycle through the images as a user clicks through the forward or previous buttons. In order to do that we use the following case.

```
case "cycle_gallery":

  if (args[1] == 'forw' && gallery_index != galleryArray.length - 1)

  {

    gallery_index++;
```

```
        description_index++;
        ajaxManager('load_page', galleryArray[gallery_index], 'galleryLYR');
        ajaxManager('load_page', descriptionArray[description_index],
          'captionLYR');
      }
      else if (args[1] == 'rev' && gallery_index != 0)
      {
        gallery_index--;
        description_index--;
        ajaxManager('load_page', galleryArray[gallery_index], 'galleryLYR');
        ajaxManager('load_page', descriptionArray[description_index],
          'captionLYR');
      }
      break;
```

This script isn't as complex as it first appears. Consider this line for a moment:

```
ajaxManager('load_page', galleryArray[gallery_index], 'galleryLYR');
```

It is the same line that we learned earlier on, except instead of naming a page like so

```
ajaxManager('load_page','somepage.html', 'galleryLYR');
```

We use the galleryArray as an argument. We then increment or decrement the array depending upon which direction is clicked by using gallery_index++. The reverse of that is gallery_index--;

All we have to do then is just condition out the directions based on an argument and the galleryArray.length and we are good to go. The final thing to look at as far as this part of the tutorial goes is the start_up case

```
    case "start_up":
      ajaxManager('hide_access');
```

```
  ajaxManager('array_setup');

  ajaxManager('load_page', galleryArray[gallery_index], 'galleryLYR');

  ajaxManager('load_page', descriptionArray[description_index],
'captionLYR');

  break;
```

Here we want to hide that accessibility on the main page so we do that by calling a case via the ajaxManager function. We then want to store the arrays so that we can use them as a user cycles through the gallery. The rest is loading content as was discussed previously.

The whole JavaScript file for the fading galley widget is 3.76 kb which is light enough in itself. We could shave a considerable amount of that by using shorter variable and function names and removing all the white space. But, since this is a developer's / designer's orientated site, I like to leave my code readable.

If you think about it that small piece of JavaScript does quite a lot. It loads in both html / image files and xml files, centres those files automatically, provides a mechanism where the user can continually load images and navigate either forward or backward and provides a fading FX. Not bad going is it?

Part of what I am attempting to do here with this tutorial and the ones to follow is to demonstrate how Ajax works in principle. By no means is this set of tutorial meant to be end all and be all of Ajax, but rather is an introduction to how Ajax operates. Obviously I haven't covered everything that Ajax can do in this tutorial. But we are going to expand into different areas in subsequent tutorials.

- Gallery optimization. Currently the way the gallery is set up it loads all the JavaScript on load up. In the next tutorial, we are going to us the importing JavaScript feature and for that matter introduce speeding up of CSS. If you clever enough and willing to rollup your sleeves you can take the code from this interface and integrate it into the gallery widget.
- Create a preload widget to integrate with the Gallery using header requests.
- Create a mechanism for delivering different Galleries. For example, Galley 1, Gallery 2 etc..
- Develop a cross browser FX animation library that we can use with the Gallery.
- Take a look at some form processing and some of the benefits of using ajax with forms. We haven't touched this side yet, but we will.
- Provide a back and forward bookmarking fix for browsers. That's a biggie that a lot of people struggle with, so keep an eye out for that.

- Look at some ajax templating.
- Integrate some RSS feeds.

Eventually we will build a CMS and integrate a lot of what we have learned here into a CMS web application. But that is in the future, for now you have this tutorial to play with. I hope you find it of some value in your development endeavours.

As I mentioned earlier you can download all the files. What you get in that download is the sliding book like interface that you can view from here, plus the Fading Image Gallery Widget and all examples used in this tutorial.

Also keep an eye out on DHTML Nirvana as I will update and make available a few standalone widgets based on what we have learned here.

If you think this tutorial is of value then feel free to donate a few dollars. It keeps food on my table among other things! You can donate via paypal from the following link. Note that is one of the alternate email addresses I use in case you're wondering who's email address that is.

Donate Via Paypal

If you want to send feedback, suggestions or flames etc here is my email address

Email Me

Finally here are all the files you need to play with this offline. Don't forget you will need a local server to view the files properly if you to intend to view them offline.

Download Files

Over and out. Enjoy!