The Ajax Papers

Part II: Updating the Page

In part one of this series on Ajax we looked at Ajax basics. What it is. How it works. Where it executes. We learned that Ajax communication (in its most basic form) only takes a few lines of JavaScript to work. If Ajax is so easy, what's all the fuss about Ajax being so hard?

Even though Ajax in its purest sense just defines a method for communicating asynchronously with the server, it is relatively useless unless you do something with the information returned from the server. That "something" usually means that you need to update portions of your web page with new HTML. And unlike Ajax communication, updating the page with new HTML after an Ajax callback is far from easy, especially when implemented in generic Ajax framework. In this installment, we'll look at what it takes to update a page after an Ajax callback and what frameworks like RadAjax and ASP.NET Ajax do to make the process very easy for developers.

Document Object Model (DOM)

Just as JavaScript and the XMLHttpRequest are critical for Ajax communication, the ability to manipulate the browser's document object model (or DOM) is critical for Ajax page updates. Most modern browsers support a standard interface to manipulate the browser's DOM that can be programmed using JavaScript. There is nothing special about the JavaScript written for Ajax DOM updates (vs. any other DOM manipulation), but it can be tricky to determine *which* portions of your page need to be updated.

Unlike PostBacks (which update the entire page), Ajax callbacks only update the portions of the page you programmatically update with JavaScript. That means you have to handle changes to all the page elements, the page head, the hidden page elements (like ViewState) and even the page title. Furthermore, since the page is not being reloaded, you also have to handle the execution of any page load JavaScript manually. You can see how this quickly gets tricky.

Trivia: Why is manipulating the DOM hard?

Every browser has its own JavaScript rendering engine *and* it has its own DOM layout engine. The W3C standardized the DOM in 1998, but browsers have been slow to provide uniform support of the W3C standards. Microsoft's IE uses a DOM layout engine called Trident (or MSHTML), all Mozilla based browers (including Firefox and Netscape) use the open source Gecko layout engine, Safari uses Apple's WebCore, and Opera uses its own proprietary Presto layout engine. All engines support DOM 1.0, but support for the later DOM 2.0 and 3.0 is still spotty.

Two Approaches for Updating the Page

When it comes to updating the page after an Ajax callback, there are two basic approaches: 1) parse information from the server and build the updated controls on the client, or 2) build updated controls on the server and simply "swap" the old with the new on the client. The first approach can reduce the amount of information that is sent over the Internet during an Ajax update, but the savings is usually offset by the browser's generally slow handling of DOM manipulations.

Both RadAjax and ASP.NET AJAX take the approach of rendering the updated controls on the server and then removing the old controls and adding the updated controls on the client. There are some frameworks that parse XMLHttpRequest's response to create the updated controls on the client (such as zumiPage), but for the most part Ajax libraries leave the heavy lifting on the server.

Basic "Swap" Approach

An Ajax library that is built to generically handle most page update scenarios (like RadAjax or ASP.NET AJAX) uses complex code to iterate through updated controls and perform the necessary client updates- more complex than we'll look at in detail in this article. But the basic process of "swapping" a control in the browser DOM is actually fairly simple.

Let's say we have a simple HTML input element on the page with an ID set to "lblTest". Assuming we had code to extract the updated HTML from our XMLHttpRequest reponse, the first step would be to *add* the new control to the page, like this:

```
if (nextSibling != null)
{
        return parent.insertBefore(element, nextSibling);
}
else
{
        return parent.appendChild(element);
}
```

Where "nextSibiling" and "parent" are relative to the old control and "element" represents the new control you are adding to the DOM. You add the new control before removing the old control to avoid the screen "flickering" in Mozilla browsers- in Opera you must remove before adding. That's just one of the many nuances of working with the DOM in different browsers. After you've added your new control to the page, you can remove the old control like this:

```
var parent = element.parentNode;
if (parent != null)
        parent.removeChild(element);
```

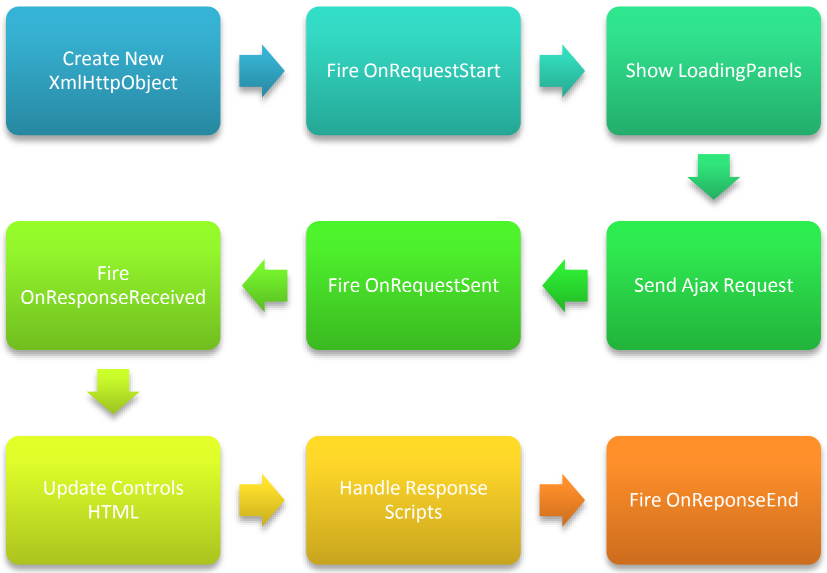
Where "element" is the HTML element you want to remove from the DOM. This approach works well in Mozilla browsers, but it can create serious memory leak problems in IE (go figure). To solve that problem, RadAjax uses a "garbage bin" approach to collect old controls and then dispose of them in a way that avoids IE's memory leak problems. We'll look at this method in more detail later.

Trivia: UpdatePanels vs. RadAjaxManager

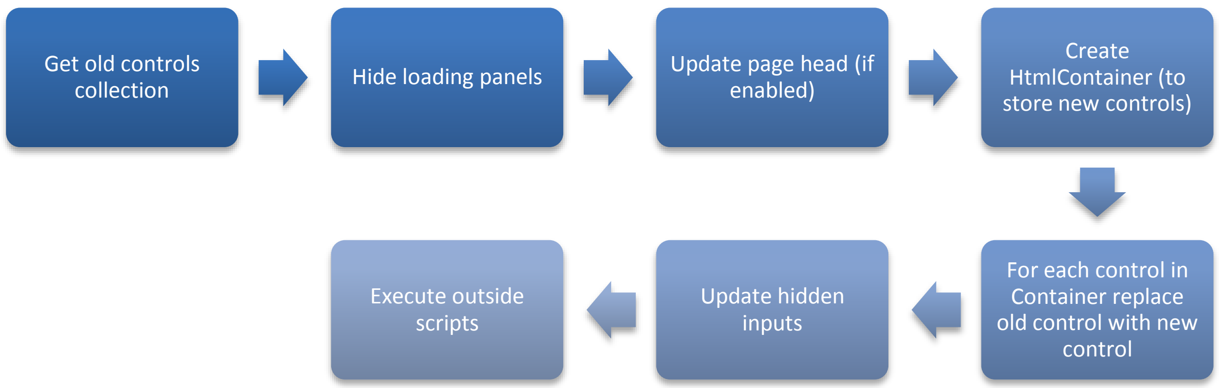
Unlike RadAjax, ASP.NET AJAX does not provide a control that allows you to define specific controls that should be updated as the result of an Ajax operation. Instead, ASP.NET AJAX relies on UpdatePanels (similar to RadAjax's AjaxPanel) that update *all* controls inside of the UpdatePanel. For this reason, you won't find any "parent.removeChild" or ".appendChild" code in Microsoft's ASP.NET AJAX Libraries. What you will find is code like this: "updatePanel.innerHTML = updatedHtml;", that replaces all of the HTML in an updated UpdatePanel with the server generated response.

How Does RadAjax Do It?

When you execute an Ajax callback using RadAjax (or any RadAjax based controls, such as RadGrid), the framework performs these major actions:



The step in this process that handles updating the page controls is obviously "Update Controls HTML". If we look at the step in detail, we see it performs the following steps:


Get old controls collection
Hide loading panels
Update page head (if enabled)
Create HtmlContainer (to store new controls)
For each control in Container replace old control with new control
Update hidden inputs
Execute outside scripts

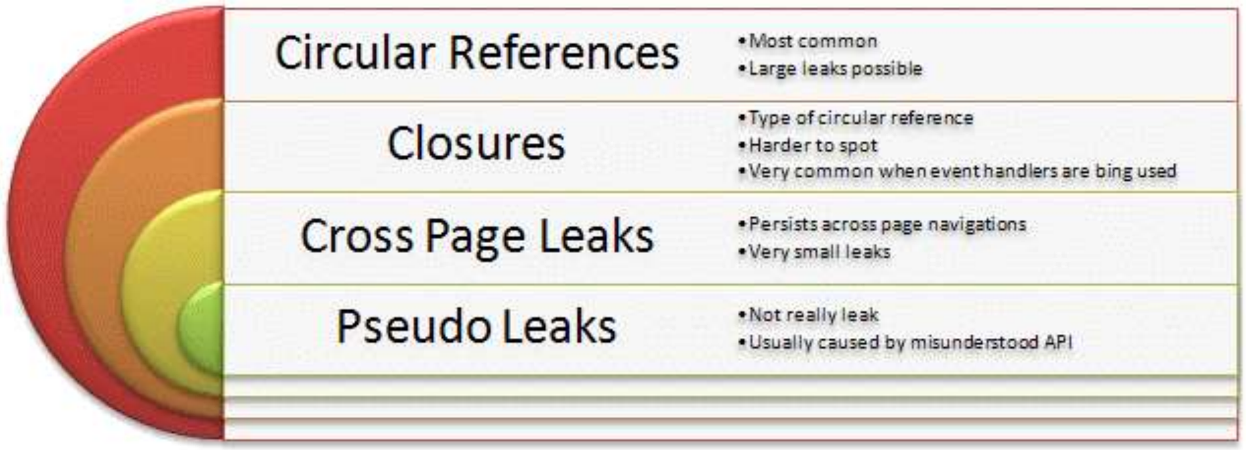
Basically, RadAjax takes the collection of updated controls returned from the server (as defined in your RadAjaxManager or as contained in your RadAjaxPanel) and systematically removes the old version from the page and inserts the new version. To do this, RadAjax must know where a control is located on a page (determined by looking at an element's parent and nextSibling). Optimizing your updates to minimize these page searches is one of the key ways you can improve your Ajax performance.

Ajax Page Update Gotchas

Browser Memory Leaks

Browser memory leaks have always existed, but the advent of extremely complex client-side web applications that don't rely on the traditional page navigation model to update the page have made memory leaks a problem that can't be ignored. Just like .NET code, browsers have garbage collectors that are supposed to release memory from unused DOM elements (such as elements you remove from the page in an Ajax update). Certain coding patterns, though, can foil the browser garbage collectors and quickly turn your Ajax application into a memory hog.

The most common coding practices that result in memory leaks are: Circular References, Closures, Cross-Page Leaks, and Pseudo Leaks. MSDN has a great article on memory leaks in IE and I encourage you to read it for a full understanding of these leak types (link in References list). This chart quickly highlights the frequency in which these leaks occur and their relative impact:


Circular References
•Most common
•Large leaks possible
Closures
•Type of circular reference
•Harder to spot
•Very common when event handlers are bing used
Cross Page Leaks
•Persists across page navigations
•Very small leaks
Pseudo Leaks
•Not really leak
•Usually caused by misunderstood API

The RadAjax framework takes special steps to make sure all DOM updates are done efficiently with the least possibility for memory leaks. Take a look at the JavaScript RadAjax uses to remove an element from the DOM:

```
RadAjaxNamespace.DestroyElement = function(element)
{

        if (AjaxNS.IsGecko())
        {
                var parent = element.parentNode;
                if (parent != null)
                        parent.removeChild(element);
        }

        try
        {
                var garbageBin = document.getElementById('IELeakGarbageBin');
                if (!garbageBin) {
                        garbageBin = document.createElement('DIV');
                        garbageBin.id = 'IELeakGarbageBin';
                        garbageBin.style.display = 'none';
                        document.body.appendChild(garbageBin);
                }

                // move the element to the garbage bin
                garbageBin.appendChild(element);
                garbageBin.innerHTML = '';
     }
     catch(error)
     {
     }
}
```

You can see that a special garbage bin is used to remove controls from the IE DOM to prevent costly memory leaks. It's this type of DOM optimization that makes Ajax frameworks invaluable tools when adding Ajaxifying our applications.

Trivia: How do you fix a leaking IE?

Knowing that browser memory leaks exist is great, but how do you figure out if *your* application is leaking? A great tool exists called Drip that enables you to easily obvserve any IE memory leaks your web application. Originally created by Joel Webber in 2005, Drip is now a SourceForge project that is freely available here: http://outofhanwell.com/ieleak. Even if you're not concerned about fixing memory leaks in your application, it's an interesting exercise to use Drip to see how efficiently your application uses system memory. In any event, just add Drip to that growing Ajax Batman-like bottomless toolbelt to improve your ~~crime fighting~~ code fixing skills for the future.

ASP.NET AJAX, on the other hand, uses "dispose" expando methods on objects to handle memory leaks. The "dispose" expando is a method interface exposed by the ASP.NET AJAX client-side library that any custom object can implement and its purpose is to handle the removal of any attached events (or in other words, to remove anything that could cause a closure memory leak). When an UpdatePanel updates the page, it loops through all elements that will be updated and calls the "dispose" methods (if available) and *then* updates the HTML. This snippet shows you how ASP.NET AJAX UpdatePanels handle these dispose methods:

```
function Sys$WebForms$PageRequestManager$_destroyTree(element) {
                if (element.nodeType === 1) {
                                    var childNodes = element.childNodes;
            for (var i = childNodes.length - 1; i >= 0; i--) {
                var node = childNodes[i];
                if (node.nodeType === 1) {
                    if (node.dispose && typeof(node.dispose) === "function") {
                        node.dispose();
                    }
                    else if (node.control && typeof(node.control.dispose) ===
"function") {
                        node.control.dispose();
                    }
                    var behaviors = Sys.UI.Behavior.getBehaviors(node);
                    for (var j = behaviors.length - 1; j >= 0; j--) {
                        behaviors[j].dispose();
                    }
                    this._destroyTree(node);
                }
            }
        }
    }
```

And just for complete understanding, here's what a typical "dispose" implementation may look like:

```
function Sys$UI$_UpdateProgress$dispose() {
       if (this._pageRequestManager !== null) {
             this._pageRequestManager.remove_beginRequest(this._beginRequestHan
       dlerDelegate);

             this._pageRequestManager.remove_endRequest(this._endRequestHandler
             Delegate);
       }
       Sys.UI._UpdateProgress.callBaseMethod(this,"dispose");
}
```

Handling Response Scripts

One thing you may have discovered when you started using Ajax is that the following ASP.NET code no longer works:

```
Page.RegisterStartupScript("key","someScript();")
```

Normally, that code snippet will render some JavaScript to the bottom of your page that will be executed when the page loads after a PostBack. When you use Ajax, the page no longer knows that it needs to update the HTML with this new JavaScript. Remember, with Ajax you have to tell the page about *every* item that needs to be updated *and* you have to manually execute dynamically added JavaScript.

RadAjax provides an easy mechanism for getting around this issue. Instead of using the code above, you will write something like this:

```
RadAjaxManager1.ResponseScripts.Add("someScript();")
```

When you add your scripts to the RadAjax ResponseScripts collection, the RadAjax framework will make sure the scripts get executed after an Ajax callback by executing code like this:

```
var newScript = document.createElement("script");

newScript.text = scriptCode;

var scriptContainer = AjaxNS.GetHeadElement();
scriptContainer.appendChild(newScript);

newScript.text = "";
```

I've omitted some of the cross browser and clean-up code for readability, but you get the general idea. The Ajax engine creates a new "<script>" element and executes your JavaScript.

What's next?

In the next part of our multi-part Ajax series, we'll examine in detail what is being sent over the pipe when we perform an Ajax Callback vs. a traditional ASP.NET PostBack. We'll compare the difference in the HTTP request and response sizes and we'll look at how RadAjax and ASP.NET Ajax handle different Ajax scenarios. In short, now that we know how Ajax communicates with the server and how it updates the page, we'll try to quantify how much "value" Ajax updates have over postbacks. Until then, enjoy using your time saving Ajax framework of choice and go forward confident that you *really* understand what's going on under the hood.

Todd Anglin, Technical Evangelist
Telerik
www.telerik.com

References

1. Wikipedia, multiple articles, www.wikipedia.com
2. Quirksmode, "Benchmark – W3C DOM vs. innerHTML", http://www.quirksmode.org/dom/innerhtml.html
3. zumiPage, Homepage, http://www.zumipage.com/default.htm
4. MSDN, "Understanding and Solving Internet Explorer Leak Patterns", http://msdn.microsoft.com/library/default.asp?url=/library/en-us/IETechCol/dnwebgen/ie_leak_patterns.asp
5. OutOfHanwell, "IE Memory Links", http://outofhanwell.com/ieleak/index.php?title=Main_Page
6. Joel Webber Blog, "Drip: IE Leak Detector", http://jgwebber.blogspot.com/2005/05/drip-ie-leak-detector.html