The Ajax Papers

Part III: Why, When, and What

In parts one and two in this series on Ajax we examined in detail how Ajax communication works and how JavaScript is used to update the page after an Ajax callback. We looked at some of the specific ways RadAjax and ASP.NET AJAX handle Ajax operations and, in short, removed (or at least greatly reduced) the mystery of how these frameworks do their magic.

In this installment, we'll move past the "how" and "where" of Ajax and start to look at the "why", "when", and "what" (I don't think we have any need to look at the "who", so we'll have to be satisfied with 4 of the 5 w's). Why should you use Ajax in your web applications? When is it appropriate to use Ajax? When is it not? What is the *real* value of adding Ajax to your application? What are the drawbacks?

To answer these questions we'll start by looking at common scenarios on the web that have benefited greatly from the use of Ajax. Then we'll explore when you should- and shouldn't- use Ajax in your applications. Finally, we'll wrap up by comparing the data that is sent over the Internet when you use Ajax callbacks instead of traditional page PostBacks to quantify Ajax's value.

Why use Ajax?

Up to this point, we've assumed throughout this series that you actually need or want to use Ajax, no questions asked. And for many of us, that's how we first added Ajax to our applications. We kinda knew what it was (we'd used Google Maps, after all). We knew RadAjax made it very easy to add Ajax to our applications. We knew it would make our applications "cool" and "Web 2.0-ish" (and impress the boss). So we embraced Ajax and ajaxified our applications without much more deep consideration for why we should use it.

The most common benefits of Ajax are pretty easy to list:

- No more <insert negative adjective here> full page refreshes or PostBacks
- Enables responsive web applications that feel like Windows applications
- Reduces the amount of data that must be exchanged between the server and client

While it is easy to measure how much bandwidth can be saved by using Ajax (which we'll do later in this article), it can be much more difficult to quantify the benefits of "responsiveness" and "familiar UI". Fortunately, many have already tried to address this challenge by conducting research to provide some level of expectation of the actual benefit derived from "soft" concepts like responsiveness and familiarity.

One such attempt was made by Alexei White in an article titled "Measuring the Benefits of Ajax". In that article, White explains how two identical applications- one built with traditional PostBacks and the other with Ajax- were tested by a handful of users while their bandwidth and productivity were measured. The results of White's simple tests suggested the Ajax application (on average) improved bandwidth performance by about 70% and total task time by about 30%, which translates into over $10,000 in annual savings (based on an assumed 36 second improvement on a task completed 50,000 times in a year by a $20 per hour employee). Clearly, White's research may not translate directly your application, but it does provide significant evidence that the simple process of ajaxifying your application can save your company (or clients) thousands of dollars.

Trivia: Test your own ROI

If you want to provide the ultimate "why" for adding Ajax to your application, perform your own Return on Investment (or ROI) tests. Using simple tools like Microsoft's Fiddler (www.fiddlertool.com) to measure bandwidth and HTTP traffic and a stopwatch (the hold-it-in-your-hand-and-push-the-button variety) you can conduct your own tests with users to see how much time and bandwidth is saved when Ajax is used instead of PostBacks. Simply create a set of tasks for your users to complete and have them peform that set of tasks on the PostBack and Ajax-enabled versions of your application. Compare the results and apply this equation to come up with the "management friendly" number:

Ajax Cost Savings = **Hourly Labor Rate** * ((**Seconds Saved Per Transaction** * **Transactions Per Year**)/3600)

Furthermore, with RadAjax in your toolbox, there are even more reasons to add Ajax to your web application:

- It's *very* easy to ajaxify your application (no code changes necessary)
- It's very easy to *remove* Ajax from your application if you don't like it
- You have Best in Class support to help you solve your Ajax problems

So whether you added (or are going to add) Ajax to your web application because you thought it was "cool" or because you measured the potential ROI like Alexei White, there should be no doubt that Ajax (even simple implementations) can add lots of value your project.

When you *should* use Ajax

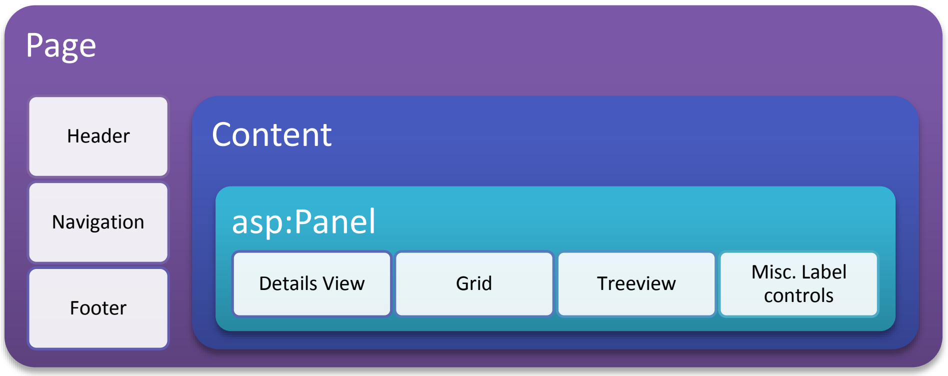
With tools like RadAjax and ASP.NET AJAX, it is not hard to find places in your application that can benefit from instant ajaxification. In general, any UX (short for "user experience", a more holistic approach than "user interface" or UI) in your application that does not involve navigation is a good candidate for Ajax. Common examples of these interactions are:

- A form that validates values with some server process
- A drop down list that loads values in response to another element's action

- Voting or rating input elements
- Multi-tab interfaces
- Any grid operations (such as sorting, selecting, editing, filtering, etc.)

It is important that you identify actions in your application that can be ajaxified at a very granular level. In other words, if you have a page on which most of the contents are updated after a PostBack, applying Ajax to the page will not provide much (if any) benefit. Remember from our earlier examination of Ajax in part's one and two of this series that Ajax is good at quickly updating *small* portions of the page. If you apply your ajaxification too broadly, you'll lose many of the underlying benefits.

Let's look at a practical example. Let's say we have page that conceptually looks like this:



If we decide to ajaxify all of the actions on this page that normally use PostBacks, we might be tempted (especially if we're using ASP.NET AJAX UpdatePanels) to simply ajaxify the "asp:Panel". After all, that would automatically convert any action from the Treeview, Grid, or the Details View into an Ajax Callback and allow those actions to update any of the other controls in the Panel. Doing that, though, would probably eliminate most of the value we could extract from Ajax.

Instead, we should focus on defining Ajax interactions that update the fewest elements on the page as possible. For instance, if our Grid has an event that updates the Details View, our goal should be to create an Ajax action that updates *only* the Details View and leaves all other controls (including the Grid) untouched. If our Treeview loads nodes from the server as it is expanded, we should only update the Treeview and no other control on the page to maximize our Ajax benefits. The RadAjax Manager from Telerik (and the soon to be released Manager for ASP.NET AJAX from Telerik) makes it very easy to define these granular relationships and use Ajax in a way that adds value to your projects.

When you *shouldn't* use Ajax

Ajax is *so* easy to add to applications these days, it is just as important that you know when *not* to use it as it is to when to use it. While it improves many page interactions, the overhead and complexity that Ajax can introduce to a page sometimes outweighs the potential benefits.

To understand when you shouldn't use Ajax, it is important that you understand some of the problems Ajax introduces into otherwise working applications:

- **Back button breaks**
  This is probably one of the biggest challenges for applications that were not designed with Ajax specifically in mind. Usability gurus will tell you that the most used feature of any modern Internet browser is the back button and any application that breaks it should be considered "unusable". Applications designed for Ajax overcome this limitation by using special tricks to populate a browser's History (thus re-enabling the back button) or by adopting a single-page model that breaks the users "I need to go back" thought process.
- **Users cannot bookmark pages**
  If you're foolish enough to use Ajax for a site's navigation (which I obviously recommend against, unless you've fully adopted the single-page application model), Ajax will also make it impossible for users to bookmark pages in your application.
- **Server errors get lost**
  Not properly handled with client-side code, unexpected server responses during Ajax operations can result in the appearance of a "hung" application. Sometimes extra consideration must be made when moving from PostBacks to Ajax to properly redirect your users when an error occurs.
- **Longer initial page load time**
  If you think Ajax is designed to make your initial page load time faster, you're wrong. By its very nature, Ajax will likely increase your initial page load time since it must load client libraries (JavaScript files) to handle all of the Ajax operations. This impact is usually relatively small when weighed against the improved responsiveness on subsequent Ajax requests, but if your page is relatively simple and the Ajax is providing relatively little value you may want to consider if the page load penalty is worth it. (To be fair, browser caching makes this less of an issue if you have other pages in your application that use the same Ajax libraries, but something to consider nonetheless.)
- **Search index problems**
  If you decide to use Ajax to load content in your application that may have previously been loaded on the initial page load (for instance, ToolTip content), remember that search indexers will not be able to see or index that content. That means you need to be careful about using Ajax on content that is key to your search indexing or you need to add code that automatically recognizes a search engine crawler and serve a version of your page that includes the Ajax content.
- **Browser feedback breaks**
  When you use PostBacks, users know the page is doing something when they see the little loading graphic spinning in their browser. When you use Ajax, this critical feedback that the page is working disappears. Fortunately, products like RadAjax make this an easy problem to solve with easy to use LoadingPanels that take the place of the browsers progress indicator.

Obviously, you shouldn't rush into adding Ajax to your application until you understand fully how each of these issues could potentially affect your application's performance and usability. You should always make sure you have a good reason for adding Ajax to your application (or page) and never add it simply because it looks cool. If you do, you may end up running into one of these problems unexpectedly and creating more trouble than necessary.

Trivia: Fixing the back button

There are several different ways you can fix the browser back button on Ajax-enabled pages. The two most popular methods are the "hidden iFrame hack" and the "window.location" hack. Each method has drawbacks (for instance, the iFrame hack doesn't work in Safari 2.0 and the window.location hack doesn't have full support IE), but they can help you restore some level of back button support to your Ajax applications. Several libraries exist that give you easy access to these hacks, such as the Really Simple History (or RSH) JavaScript library that is designed specifically for this task. If you're using ASP.NET AJAX, Nikhil Kothari has created an "UpdateHistory" control that enables you to (more) easily handle the task of selectively populating the browser's history and thus re-enabling the back button. We will cover some of the back button hacks in detail in a future installment in this series.

What is the difference: PostBack vs. Callbacks

Here is where the rubber meets the road. So far we've assessed in general terms why and when you should use Ajax, but all of that is empty rambling if Ajax does not *really* improve your page performance in a measurable way. To prove that Ajax makes a meaningful difference, we'll examine the bandwidth used and responsiveness of a sample web page in three different scenarios:

1. Traditional PostBacks for all operations
2. RadAjax to ajaxify all operations
3. ASP.NET AJAX to ajaxify all operations

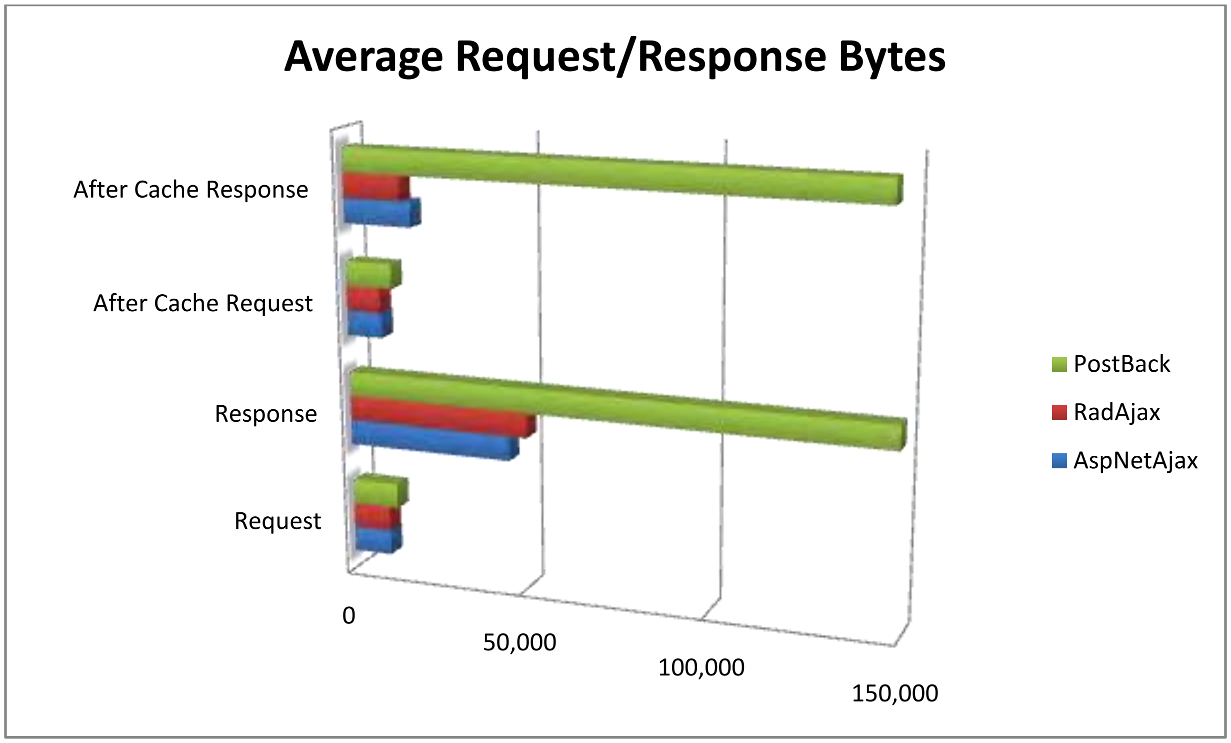
The Test

To conduct our test we'll use a real world site: converter.telerik.com. This online code converter from Telerik allows people to easily convert VB to C# and C# to VB, and it is a perfect place to put our scenarios to the test. Tests will be run by pasting a C# code snippet into the Code Converter and then clicking the "Convert Code" button. Using IE 7 and Fiddler, we will measure the size (in bytes) of the request and response for each conversion, and we clear the browser cache before each set of tests. The test will be run for each of our three scenarios, resulting in 18 data points (page load + 5 clicks for each scenario).

Since this is a single step process, measuring actual time savings will be difficult. To measure time savings in your own tests, you should design a test with multiple steps and try to find several users that you can time as they complete the steps of your test. We don't have that luxury for this paper, so we'll be focusing on primarily on the bandwidth savings Ajax provides (or does it?).

The Results

After running all of our tests, logging the results, and then averaging the request and response bytes for each scenario (PostBack, RadAjax, and ASP.NET AJAX), we discover that Ajax does in fact deliver a *huge* bandwidth savings over traditional PostBacks (see chart below). In the case of our Code Converter, the act of simply switching from PostBacks to RadAjax callbacks (which we can do without writing any code…but that's another demo) reduced our average response size by over 65%! If we look at the average response size after the first response (in other words, after most JavaScript files have been cached), the improvement made by RadAjax is just shy of 90%. That means the server has to send 90% less data over the Internet and the user has to wait on 90% less data to arrive when the Code Converter uses RadAjax instead of PostBacks.



In business terms, this experiment tells us that Ajax is a real money saver. Let's say the Code Converter processes 1000 conversion requests a day and that the full PostBack response size of those pages is about 200KB. In a month's time, the Code Converter could burn through 6**GB** of bandwidth with traditional PostBacks. By implementing Ajax, though, we can cut that number to just 600**MB.** Scale that out for larger and more popular sites and you can see how the savings really start to add up (*especially* for the *users* on the other end of that number, waiting for the response to download).

The comparison of RadAjax to ASP.NET AJAX, on the other hand, was fairly equal. While the average ASP.NET AJAX response size with the "first request penalty" (before caching) was 46KB and the RadAjax response was about 51KB, the subsequent “post cache” responses were 19KB and 16KB on average respectively (see figures below). The perceived responsiveness of ASP.NET AJAX and RadAjax were the same for this test, but the RadAjax implementation was much easier to configure and add to the application.

Page Size by Content Type and Scenario

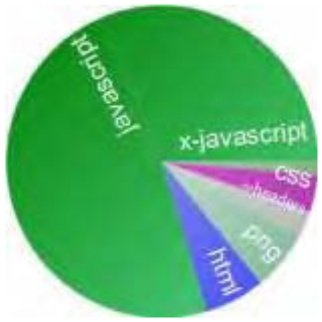


Fig 2.1: No Ajax, original page request (164KB)

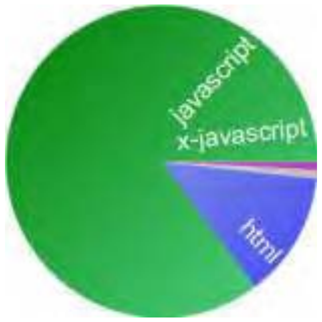


Fig 2.2: No Ajax, PostBack response (146KB)

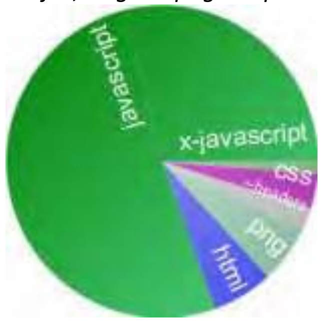


Fig 3.1: RadAjax, original page request (223KB)

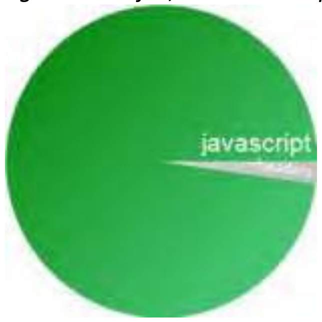


Fig 3.2: RadAjax, Callback response (16KB)

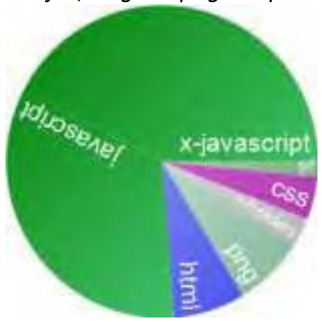


Fig 4.1: ASP.NET Ajax, original page request (179KB)

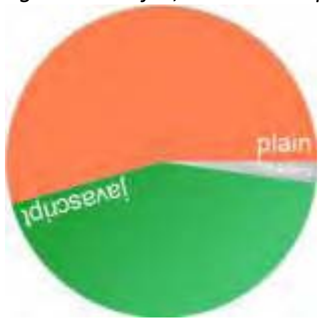


Fig 4.2: ASP.NET AJAX, Callback response (29KB)

These charts highlight some interesting facts about Ajax. First, as I asserted earlier in this paper, Ajax does not make the page smaller on the first uncached request. In fact, we can see both of the Ajax enabled pages are noticeably larger (8% for ASP.NET AJAX and 27% for RadAjax). We see that Ajax’s real benefit is in the subsequent page response size, which leads us to interesting fact number two. Look at what’s missing from the second RadAjax and ASP.NET AJAX charts. There are no HTML, image, or CSS “slivers” in those responses (unlike the PostBack response). And even though there appears to be more JavaScript, we can tell by the total response size that the majority of the JS being returned in the PostBack response has been cached in the Ajax responses.

These tests clearly make the case for Ajax in the Code Converter website. While your tests may not produce numbers that look exactly like these, you should find that the general conclusions and comparisons remain true. At the end of the day, the take away from these tests should be:

- RadAjax and ASP.NET AJAX offer relatively similar performance
- Regardless of which Framework you use, Ajax definitely provides significant, measurable benefits over PostBacks.

Trivia: The ViewState Burden

It's not secret in ASP.NET development that ViewState can quickly double (or more) the size of a page. In the past, developers have often had to accept this or find creative ways to move ViewState off of the page to avoid the size penalty while still being able to maintain state across PostBacks. When you use Ajax, though, you have new opporunities to disable ViewState altogether. Ajax callbacks do not refresh the whole page so the state of most controls is natrually preserved. By eliminating ViewState you can further reduce the size of your Ajax request and response sizes, often realizing improvements of over 50%. In a simple test conducted with a RadGrid and ASP.NET Panel, Ajax response sizes were decreased from 24KB to 5KB by disabling ViewState and using client-side code instead of server-side updates to update the appearance of the Grid.

Designing for Ajax

We've spent a lot of time talking about how you can convert existing ASP.NET applications to Ajax-enabled applications in this paper, but we haven't addressed the scenario in which you design your application from the ground up with Ajax in mind. If you have the luxury to design your application from scratch, there are definitely lots of new interaction models that Ajax enables you to use. With Ajax and a powerful set of UI controls (like Telerik's RadControls for ASP.NET), you can replicate almost any desktop interaction model on the web, in all of the major browsers.

The benefits of modeling a web application after a desktop application often manifest in the form of reduced training and support costs. Users' expectations for how interfaces should work are still largely influenced by the time they spend on the desktop, especially by programs they use often (like Microsoft's Office). If you can provide experiences on the web that support those expectations, you will have to spend less (not *no*) time training your users and your users will likely be much more productive in a shorter period of time.

The reason I often glide past this approach to web application design in these papers is because it is still hard for many experienced web developers to think in the new terms of designing web applications that feel like desktop applications. We have become so used to the multi-page, form element model of designing pages that we simply overlook opportunities to embrace a single-page application model that uses Ajax to support context menus, menu bars, and the like. If you do step back, though, and design an application that maximizes the power of Ajax, you can definitely create applications that do much more with Ajax than simply turning PostBacks into Callbacks.

Trivia: Designed for Ajax

If you found this section to be more "rah, rah, Go Team" than fits your fancy, check out some sites that have fully embraced Ajax from their inception to see exactly what Ajax enables. A great example is PageFlakes.com. Built completely on ASP.NET AJAX, Page Flakes is a perfect example of how you can use an Ajax Framework (vs. custom Ajax code) to implement a unique PostBack-less interface. Also a good example is Yahoo! Mail, which replicates the feel of a desktop mail client in the browser. While it runs on Yahoo!'s own Ajax framework, the site is still a good example of how new web technologies (and wide browser support) enable previously unthinkable interactions on the web. And with tools like RadAjax, you don't have to be a mult-million dollar company or have hundreds of hours to implement them!

What's next?

Coming up next, we'll shift our focus away from a general study of Ajax and begin to look at ways you can specifically optimize RadAjax in your applications. At this point you should have a complete understanding of what's going on under the hood of RadAjax and have a sense for the kind of measurable value Ajax can provide. Now it's time to take that knowledge and unleash it on RadAjax to squeeze out the maximum performance gains.

As I near the end of this series, though, I want to make sure I've addressed all of your unanswered questions about Ajax and RadAjax specifically. If I've missed something along the way that you'd like to know more about, please email your questions to me at todd.anglin[at]telerik.com. If I receive enough questions, I'll do a special "open mic" installment of the Ajax Papers where I answer your questions. Until then, have fun testing your own applications and telling your bosses how much money you're going to save the company by adding Ajax to your next project!

Todd Anglin, Technical Evangelist
Telerik
www.telerik.com

References

1. White, Alexei. Developer.com. "Measuring the Benefits of Ajax". http://www.developer.com/java/other/article.php/3554271
2. Bosworth, Alex. SourceLabs.com. "10 Places You Must Use Ajax". http://www.sourcelabs.com/blogs/ajb/2005/12/10_places_you_must_use_ajax.html
3. ASP.NET AJAX. General use. http://ajax.asp.net