From: www.cio.com

You Used JavaScript to Write WHAT?

– Michael Morrison , CIO

January 25, 2008

The word "JavaScript" has become a lightning rod in the Web development community. Depending on who you listen to, JavaScript is either the shining beacon of light leading us toward Web 3.0 or it's an insidious plot to bring the Web to its knees, one hover button at a time. If you eliminate the radical points of view on each end of the spectrum, you are left with very real questions about when it makes sense to use JavaScript, and when it doesn't. There is no doubt that overzealous scripters have built applications that stretch the limits of JavaScript...and in some cases, reason.

More on Dynamic Languages & Web Dev
You Used Perl to Write *What*?!
You Used PHP to Write *What*?!
Beyond Ajax: Software Development, Two Years from Now
Five Compelling Reasons to Use MySQL

What kind of JavaScript development scenario qualifies as stretching the limits of reason? For one, attempting to build complex multimedia applications such as action games. Sure, in many ways it's technically possible to accomplish given that JavaScript is a powerful technology with the capability to manipulate images, carry out animation timing, etc. But high-performance multimedia is in no way JavaScript's strong suit. Even if performance wasn't a critical consideration, JavaScript still falls short when compared to other options, such as Adobe Flash. If you're trying to create complex multimedia software with JavaScript, you're ultimately reinventing several wheels; specialized tools exist for the very purpose of empowering Web developers to build rich, online multimedia experiences.

Okay, so maybe Halo 4 in JavaScript was never really on the table at your company. Does that mean everything else is open game? Not hardly. There are still other pitfalls awaiting the overconfident JavaScript developer who insists on using a JavaScript hammer to whack away at every interactive Web nail in sight.

But the problem of abusing JavaScript as a Web development technology is ultimately more subtle than just pointing out a handful of less-than-ideal coding techniques or application categories. Yes, it's true that nifty little Web effects such as image roll-overs should be carried out using Cascading Style Sheets (CSS), not JavaScript. It's also true that it's generally a bad idea to craft hyperlinks using JavaScript code that breaks down when JavaScript is absent. And don't even get me started on using JavaScript to detect browser versions or, most horrific of all, hijack the browser's status bar to display a cute animated message. The real issue of assessing the role of JavaScript in a specific Web application comes down to the role of the application. Is it a page, a program, or some combination of the two?

So now you're thinking, "Wait a minute, I thought this article was going to give me a handy bulleted list of when to use JavaScript and when not to, and now I'm having to think philosophically about what it is I'm building?" It's true. The key to understanding when and when not to deploy JavaScript has as much to do with the intent of the target application as it does JavaScript itself.

You already have a clue that JavaScript is a client-side scripting language that can add interactivity to Web pages in lots of different ways. But are you truly deploying a collection of Web pages, or are you deploying a full-blown client application that just happens to run in a Web browser? And is this application being designed for internal use within a corporate intranet (where you have a degree of control over the browser), or is it for the general public to consume? These are important questions because, as you may also know, JavaScript is both powerful enough and flexible enough to be used in just about any Web application. The question we're trying to answer here is when, why and why not.

So let's go back to the notion of a Web-based version of Halo 4 as a potential JavaScript application. Why wouldn't it make sense for such an application to be built in JavaScript? First and foremost is performance. Action games are always on the bleeding edge of multimedia technology, and typically they need to squeeze every spare cycle out of the processor. As an interpreted language, JavaScript just isn't cut out for such hardcore performance. Such games are typically built in compiled languages that run natively on a given processor.

But for argument's sake, let's say performance isn't a deal breaker. What then of JavaScript and Halo 4? There are still problems, and one of them has to do with the fact that you would be targeting a mass market with a video game, but it's a market that doesn't necessarily have universal support for JavaScript. For example, some people have JavaScript disabled for security reasons. And since the entire game is dependent on JavaScript, there is no way to "gracefully degrade" the Halo experience for users without JavaScript. They're just left out. Maybe that's okay, though. And that leads back to the question of page versus program.

Let's shift gears to an entirely different JavaScript application: a product review service where people post reviews of products. This is sounding more like the Web we know and love, a web of pages. In this application, each product has its own page; that page contains a list of all the reviews for the product. Although portions of the application could possibly benefit from JavaScript, nothing about it screams "I need JavaScript!" The product reviews can be built using a traditional Web form that is posted to the server for storage in a database. And each product page can be "assembled" on the server using a server scripting language such as PHP.

So far, we're talking about the traditional Web 1.0 way of doing things with HTML and good old forms. But what about Ajax, that clever merger of JavaScript, XML and some kind of asynchronous something or other? With the help of a little Ajax, the product review application could be made more responsive, eliminating the traditional banter between client and server as forms are processed and data is sent back and forth in small chunks on a need-to-know basis. In many ways, the Ajax version of the application represents a minor improvement but, in terms of usability, it could have a significant payoff. And there you have it, a perfect justification for JavaScript in an otherwise traditional Web application. Case closed...or is it?

The problem with casually injecting Ajax (JavaScript) into traditional Web applications that don't desperately need it is that you run the risk of making the application dependent on JavaScript, and perhaps unintentionally limiting the pool of users who can access it. Remember, there will be people who don't have JavaScript support. You already had a perfectly good (and functional) Web application, even if it was categorized as oh-so-passé Web 1.0. But what's worse: dull and boring but consistent, or hip and flashy but exclusive? Would you rather all users have a satisfactory experience, or most users have a superior experience at the expense of a minority of users? These are important questions, and they lead back once again to whether your application is a page or a program.

Okay, fine, then let's officially declare the product review application a set of pages. Does making that declaration eliminate JavaScript? Not at all. But it does impact how JavaScript is used. More specifically, by committing to a page mentality, you're saying that interactivity is secondary to content. And that means the accessibility of the pages cannot diminish if JavaScript is unavailable. What this boils down to is that JavaScript must be used as an enhancement to the application, not as a necessary component. The beauty of this philosophy is that it still leaves the door open for slick Web 2.0 effects like dynamic data exchange with Ajax, while at the same time preserving the traditional Web development techniques that still work as a lowest common browser denominator.

The other side of the coin is the mentality of viewing a Web application as a *program*, as opposed to a *page*. In this scenario, the application is utterly dependent on the active functionality made possible by JavaScript, which means it's okay to forego users who lack JavaScript support. Google has embraced this philosophy in several marquee products, two of which are extremely popular: Gmail and Google Maps. Both applications make extensive use of Ajax (JavaScript), and neither apologizes to users who can't run them due to a lack of JavaScript. If this article had been written just a few short years ago, I might have used an e-mail application as the ridiculous example of when not to use JavaScript, instead of Halo. But Gmail has pushed through that barrier.

What has yet to be fully shaken-out is the viability of Web-based applications for carrying out tasks we've grown accustomed to doing in native client applications, such as e-mail. Gmail has come a long way and a lot of people are using it, but you still hear some grumblings about that intangible usability difference between a browser-based application and a stand-alone application.

Even if JavaScript-powered, web-based e-mail ultimately takes hold, surely there are other stand-alone applications that will just never make sense in Web form. Two such applications that come to mind are video and photo editing. Similar to games, these are such media-intensive applications that they just can't make sense in JavaScript, right? Yet Adobe has already released Premiere Express for online video editing and is putting the finishing touches on Photoshop Express for Web-based photo editing. What's interesting about these applications is that they aren't technically built in JavaScript; they're built in ActionScript, a close cousin of JavaScript used in Adobe's Flex development environment. But the ActionScript in these applications is compiled, so the net effect is more akin to a native application. Adobe may be foreshadowing the future of Web scripting to some degree, at least in terms of building more feature-rich applications. And in doing so, they're forcing us to rethink just what is possible with scripting languages.

Adobe aside, it's difficult to draw lines in the sand when it comes to the usage of JavaScript on the Web because the sand is still shifting under our feet. So while I'll stick to my guns about not building Halo 4 in JavaScript, over time it will become increasingly difficult to rule out entire classes of applications as being viable in Web form. Halo 7 may be a different story!

For the foreseeable future, the more important decision is accessibility and making sure your Web application

works—and works well—for the most users as possible. If you err on the side of viewing JavaScript as an "enhancement technology," you're unlikely to go wrong.

Michael Morrison is a writer, developer and author of a variety of books covering topics such as Java, Web scripting, game development and mobile devices. Some of Michael's notable writing projects include Head First JavaScript *(O'Reilly, 2007)*; JavaScript Bible, 6th Edition *(Wiley, 2006)*; Teach Yourself HTML & CSS in 24 Hours, 7th Edition *(Sams Publishing, 2005)*; and Beginning Mobile Phone Game Programming *(Sams Publishing, 2004). In addition to his primary profession as a writer and technical consultant, Michael is the founder of Stalefish Labs, an entertainment company specializing in games, toys and interactive media. When not glued to his computer, skateboarding, playing hockey or watching movies with his wife, Masheed, Michael enjoys hanging out by his koi pond.*