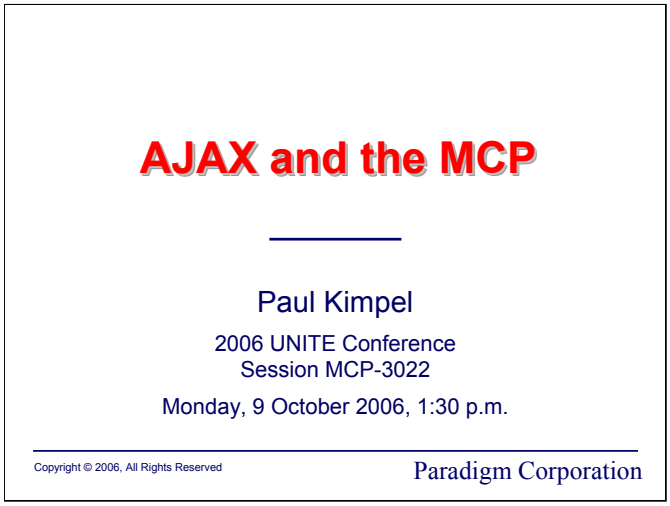


AJAX and the MCP

2006 UNITE Conference
Garden Grove, California

Session MCP-3022

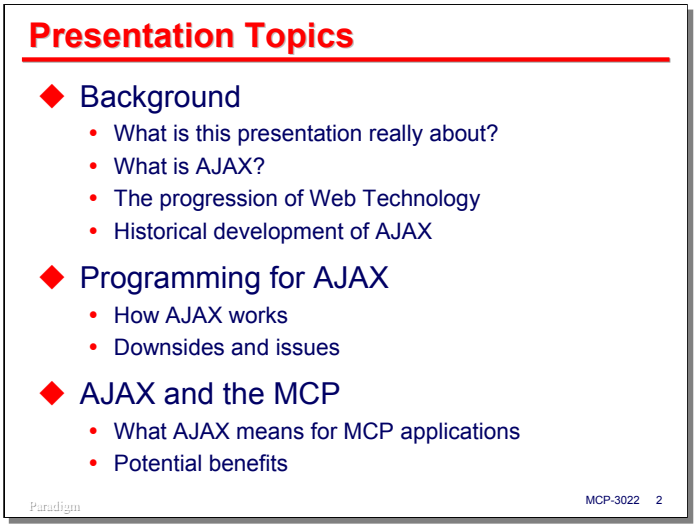
Monday, 9 October 2006, 1:30 p.m.

Paul Kimpel

Paradigm Corporation
Poway, California

http://www.digm.com

e-mail: paul.kimpel@digm.com



Today I am going to talk about AJAX, a relatively new technique for building web-based user interfaces.

I'll begin by discussing some background material – the core issues behind AJAX, what AJAX is, how it fits into the ongoing progression of web technology, and a little about the history behind the development of AJAX.

With that background, I will next discuss how you program using AJAX. I'll describe how AJAX works, and show some simple code examples. I'll also talk about some of the downsides and other issues involved with using AJAX.

Finally, I'll discuss what AJAX can mean in the context of legacy MCP applications and some of the benefits we might expect from deploying this technique in web front ends to the MCP environment.

While the title of this presentation indicates it's about something called AJAX, and I will be talking quite a bit about AJAX itself, there is a deeper issue which AJAX is helping to expose.

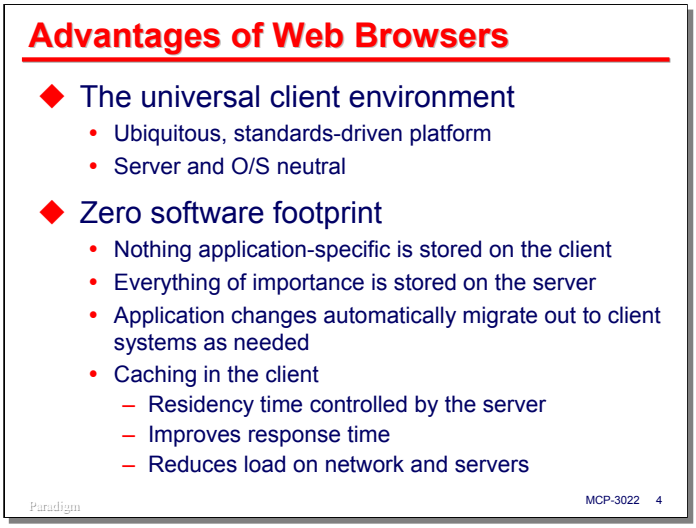
The real subject behind this talk is Rich Internet Applications, or RIA. The web began essentially as a way to deliver documents to users, but over the past several years it has begun to be used as a way to deliver a *user interface*, too. In terms of raw functionality, the basic HTTP/HTML mechanism of the web can do a pretty good job of delivering a user interface, but it can be a little clunky and ponderous compared to the GUI interfaces we have come to expect from PC applications. What we would like to do is make the web-based interfaces look and work a lot more like those PC applications, while retaining all the advantages that make the web approach so valuable. The main things we are looking for in RIA are:

- A much more interactive exchange of data and events between the client and server ends of the interface. I'll discuss this in more detail over the next few slides.
- A finer-grained update of the user's view. I'll discuss this in more detail as well.
- Potentially faster response to user requests and a lower burden on servers and networks.

There are several trends pushing the development and refinement of the technologies behind RIA. The first is a long-held desire to move away from application-specific "thick client" architectures. As we found to our dismay in the client/server initiatives of the 1990s, thick clients work great – until the application has to change. Then we have the problem of migrating the changes out to all of the clients and synchronizing the client changes with those on the server. Change management was a lot simpler in the days of the green-screen terminal, and the web is delivering a way to get back to that state of affairs.

The second trend is that almost everyone in the first and second worlds now has (or has access to) an Internet connection and a browser, and they all know how to use them. This gives us a common base upon which to build user interfaces and application environments that are relatively easy for both sophisticated users and the general public to access and become adept at using.

Finally, I am convinced that thin-client devices will rise again. By "thin client" here, I don't mean Citrix. Most of us simply don't need full-featured PCs to do what we need to do, and suffer because of the administrative overhead required to keep a sophisticated operating system, installed applications, and a data storage system running properly. The battles with malware are just one facet of this problem. Instead, what I think most of us need is a "browser in a box" – effectively the modern-day green-screen terminal. It will have circuitry much like the current PC, but without a disk. It will have a network connection, a web browser, JavaScript, and possibly Java. The thing it will *not* have is that which causes us most of our PC-related grief today – installed applications or a general-purpose operating system.

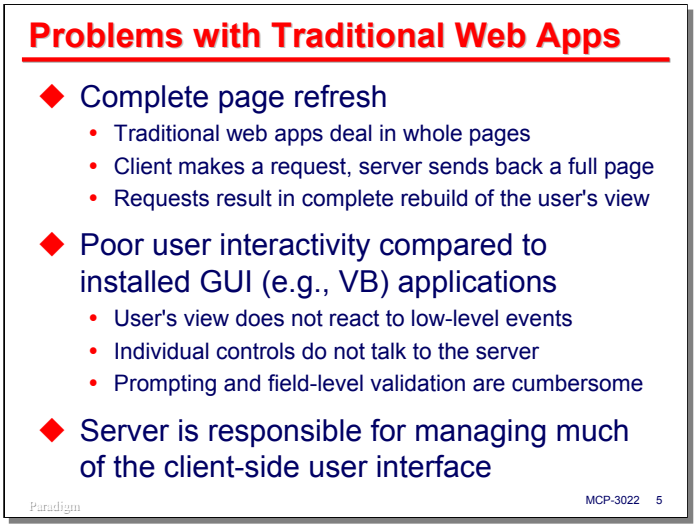


This resurgence of the thin-client approach is based primarily on the capabilities of the modern web browser, which brings a number of significant advantages to the table.

As Marc Andreesen, of Netscape fame, pointed out several years ago, the browser can be a *universal client*. As I mentioned a moment ago, browsers are nearly ubiquitous, and despite the best efforts of Microsoft to the contrary, are becoming closer in both concept and practice to being a totally standards-driven platform.

For the purpose of supporting user interfaces to typical business applications, whether they are internal to the business or accessed publicly, the most appealing characteristic of the web browser is that it presents a zero software footprint, i.e., there normally needs to be no additional software available on the client to support the user interface. Where additional functionality is required, it can generally be provided through plug-ins and similar extensions to the basic browser.

It is highly desirable to have nothing application-specific installed on the client. That way, as the application changes, nothing on the client needs to be changed. Everything of importance that is customized to the application is stored on the server (or servers), all application administration is done on the servers, and application changes automatically migrate out to the client as they are deployed on the servers. Performance can be enhanced by caching elements of the user interface on the client. This both improves response time for the user and reduces the load on the network and servers by not having to retransmit user interface elements each time a transaction is performed. Cache residency (expiration) time can be controlled on an element-by-element basis by the servers, and all modern browsers and web servers efficiently handle the detection and (if necessary) replacement of expired elements.

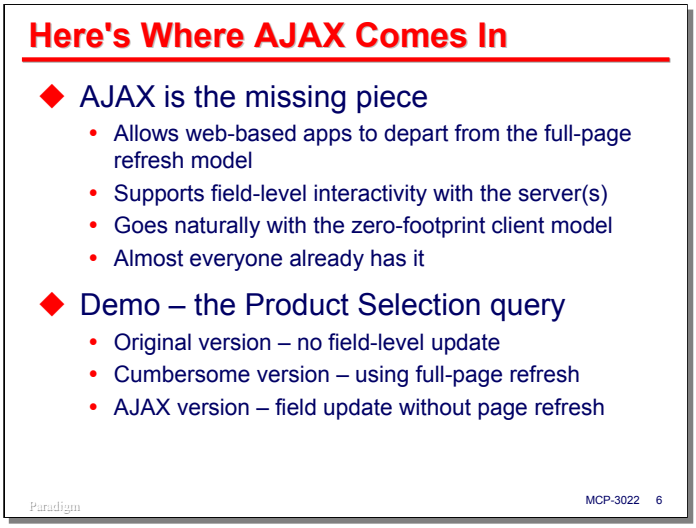


There are, however, some problems with browsers and traditionally-designed web applications as user interfaces.

The first of these is that the web's original model of user interaction calls for the user's view to be completely refreshed every time there is any interaction with the server. Traditional web applications deal only in whole "pages," or documents, as they have been designed around the document-centric model that the web began with. The client makes a request, and in response the server completely reconstructs the user's view and sends back a full page, even if the request produces only minor changes to the view. This typically generates a lot of work on the part of the server, along with what I call the "blink" – the noticeable delay to the user as the current view is cleared and the new one is transmitted by the server and rendered by the browser.

This complete page refresh approach also results in poor interactivity with the user, especially when compared with the type of interactivity we expect to see with installed GUI applications on Windows and Macintosh systems. The user's view either does not react to low-level events (such as list selections or checkbox changes), or if it does, that low-level event triggers a complete submission of the current page and reconstruction of it on the server. In other words, the individual user interface controls on the page do not talk to the server – only the whole page talks to the server. If you want to go beyond only the most simple fill-in-the-blanks types of forms, the traditional HTTP/HTML web approach does not give you much support. Therefore, things like prompting and field-level validation are cumbersome (you can only do them by completely refreshing the page), and inter-field dependencies (where an entry in one field or control affects what you can see in another field or control) are even more awkward and difficult.

Finally, with the traditional web application approach, the server is responsible for managing much, if not all, of the client-side user interface. The server must build the user's view and be concerned with layout and positioning of the interface elements. This can make it difficult to achieve a good separation between user interface details and business rules on the server side. It can also be difficult and tedious to code for, since the only events the server sees are those which result from a complete submission from the user's view, and which will require a complete reconstruction of that view in reply.



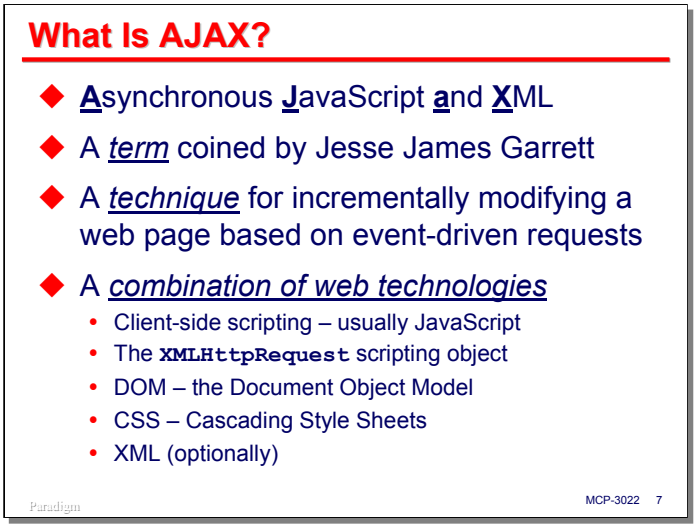
The very coarse-grained interactivity between client and server for traditional web application designs is where AJAX comes in to play. AJAX provides the missing piece – the ability for the user interface to depart from the full-page refresh model and to communicate with the server based on lower-level client events – clicks, list box selections, changes to field contents, and so forth. This finer-grained interactivity can occur asynchronously, while the user is viewing the page and interacting with other controls. In fact, multiple client/server interactions can be taking place simultaneously, provided the server can handle multiple requests in parallel.

The other nice thing about AJAX is that it fits in with the browser's zero software-footprint model – most modern browsers have what is needed to support AJAX, so almost everyone already can take advantage of it.

I have a demonstration that gives a simple illustration of what is possible with AJAX. This is a query that selects products for a company based on a number of criteria, including brand, reporting category (a sub-brand classification), style (the container or packaging of the product), and UOM (unit of measure). These selection values are in pull-down lists. Based on the criteria selected from these lists, the query displays a list of matching products. This demo is in three parts.

- First is the original version of this page, which simply loaded all possible values into the respective lists. The problem with this is that if you make a selection for the most-major criterion, brand, the remaining lists still have all of the possibilities, whether they apply to that selected brand or not. This is an example of what I term an inter-field dependency (or rather, in this case, the lack of one). A rule of good user interface design is that you should not show the user anything that is not relevant to the current state of the interface. Since the pull-down lists continue to show non-relevant choices for less-major criteria when selections are made for the more-major criteria, this interface violates that rule.
- The second example attempts to address this by filling the less-major lists with only relevant choices when a selection is made in a more-major list. It does this, however, using the traditional web application approach – whenever a selection is made in a list, that causes the entire form to be submitted to the server. The server then rebuilds all lists based on the current selections. This results in a full page refresh of the user interface.
- The third example accomplishes the same result, but uses the AJAX approach instead. When a selection is made in a pull-down list, only the less-major lists are rebuilt. The page requests new values only for the lists that are being rebuilt, and the reconstruction of those lists is done on the client, not by the server. The server simply sends the data necessary to rebuild those lists, not the entire HTML for the <select> elements that define those lists. We will see how this is done shortly.

[*Note: the code for this demonstration is not included with the presentation materials, but can be made available upon request*]

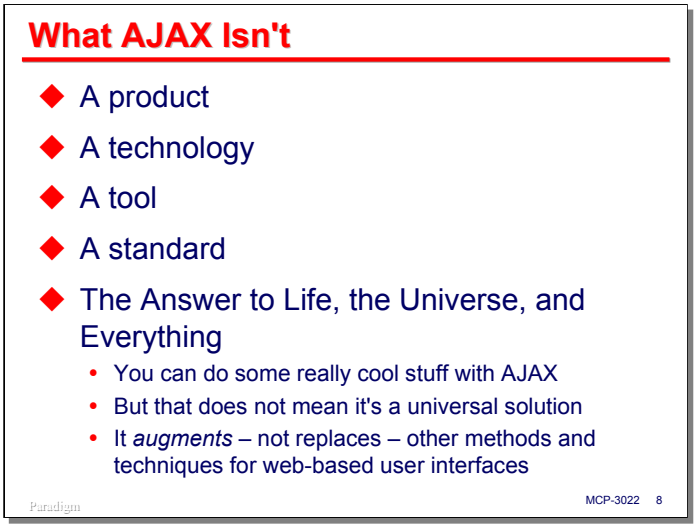


Having seen a simple example of AJAX in action, the question is now, just what is it?

AJAX stands for Asynchronous JavaScript and XML. This is a term coined by Jesse James Garrett of Adaptive Path in early 2005. AJAX has a lot to do with being asynchronous and with JavaScript, but the XML part is optional, and may or may not be associated with a particular AJAX application, as we will see.

AJAX is actually just a *technique*. The term describes the combined use of a number of web technologies, the result of which is the ability to incrementally modify a web page, asynchronously and simultaneously with the user viewing that page. This result is accomplished by means of:

- Client-side scripting. This usually means using the JavaScript interpreter that is implemented by virtually every currently supported browser. Other scripting languages can be used, but the ubiquity of JavaScript means that is is the one that is almost always used.
- The **`XMLHttpRequest`** object. This is the vehicle by which the client can submit asynchronous requests to a server and process the response while the user is viewing the page. Microsoft IE 5 and 6 implement this as an ActiveX component. IE 7 and most other browsers that support **`XMLHttpRequest`** implement the object natively.
- The Document Object Model, or DOM. This is a standard API that allows a scripting language to manipulate a web page from within the client browser. This is the means by which the scripting component can modify the user's view based on the response from **`XMLHttpRequest`**.
- Cascading Style Sheets, or CSS. Styles are the proper means by which the rendering of a web page is controlled. Scripting can modify the styles of page elements to render them differently, controlling such things as font, color, borders, margins, position, size, and whether an element is visible or not.
- Extensible Markup Language, or XML. This is an HTML-like notation designed to represent and transmit arbitrary data structures. It is *one* of the formats with which the server can send the **`XMLHttpRequest`** response. The DOM API has the ability to parse XML and make the structure and values of the response available to the scripting component. Almost any format can be used for the response data, however, and XML is not a required part of the AJAX technique.



Given what AJAX is, there are a number of important things that it is not.

First, AJAX is not a product, although there are a number of products coming to market that exploit AJAX, either as a user interface mechanism, or as development environments to build AJAX-enabled web applications.

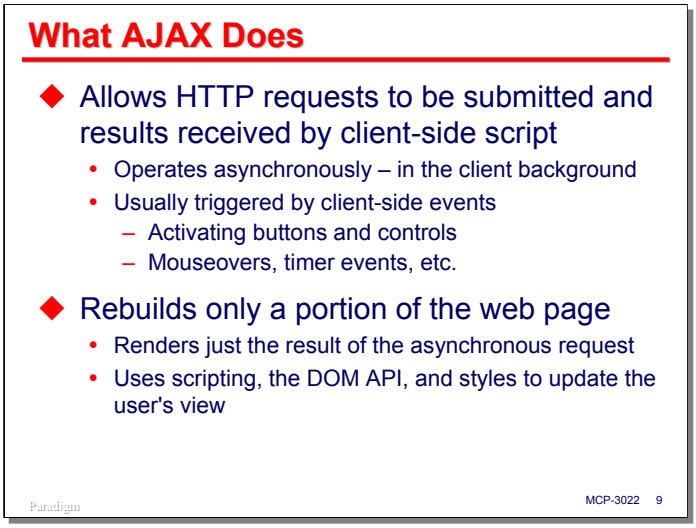
AJAX is also not, of itself, a technology. It is a combination of other technologies used in a particular way.

AJAX is not a tool. It is instead a technique or approach to applying its constituent technologies to a given problem.

AJAX is not a standard, although all of the technologies on which it is based are the subject of standards.

Finally, and most important, while AJAX is a really interesting idea, and you can do really cool stuff with it, it is not the answer to everything. AJAX is just another arrow in the quiver of the web designer and developer. AJAX augments all of the other web methods and techniques that are in use today.

There is a real temptation in our business to take a really good idea and try to apply it universally. Some of the hype currently surrounding AJAX comes from this temptation. We need to recognize AJAX for what it is – a really good idea with some really useful applications, and to recognize where it is the appropriate approach for a given situation. We also need to recognize where some other, perhaps less snazzy, approach would be better.



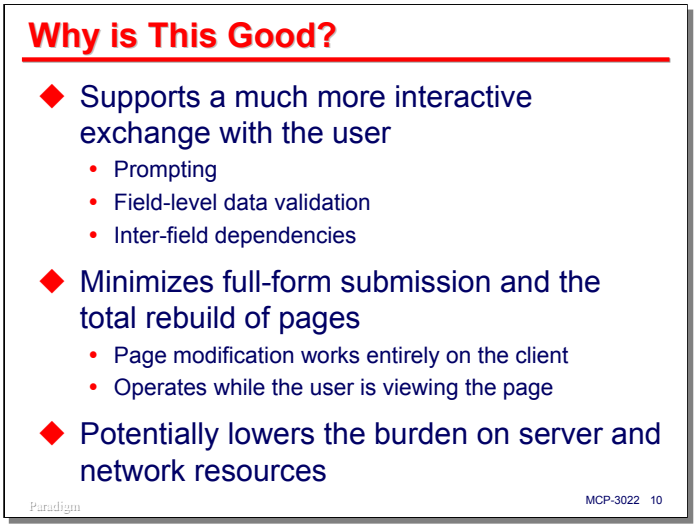
So just what is it that AJAX does? The answer to this is straightforward, although what you need to go through when implementing a particular AJAX solution can get quite complex.

AJAX simply allows a web page within a browser to submit HTTP requests to a server, receive a response from the server, and process that response. The result of processing that response is typically a change to page as it is displayed to the user.

The important part of this is that the request/response/modify sequence happens asynchronously while the user is viewing the page. All processing occurs in the background of the client browser. Changes to the display just seem to happen, similar to the way they do with workstation GUI applications. Several of these interactions potentially can occur in parallel.

These AJAX interactions are usually triggered by user interface events on the client. That could involve the user clicking buttons, selecting an item from a list, checking a checkbox or radio button, changing the contents of a text box, and so forth. They can also be triggered by more subtle events, such as moving the cursor over an element (a so-called "mouseover" event), or the expiration of a timer.

Another nice part of the AJAX approach is that normally only a small portion of the displayed page changes. There is no complete page refresh – no "blink." This results in a much smoother response from the user's perspective. These changes to the view are accomplished through a combination of scripting, calls on the DOM API, and changes to element styles.



Why is the AJAX approach good? The primary answer is that it supports a much more interactive exchange with the user. This enables some user interface techniques that are difficult and awkward to accomplish within traditional web application design. For example, AJAX allows a web interface to support prompting and auto-complete techniques. Using AJAX you can implement true field-level validation – as a user enters a value, that entry can be sent to the server for validity checking. It also supports the enforcement of dependencies between or among elements of the page. Entries in one field may enable or restrict the validity of other fields or the set of values they can contain.

AJAX can minimize the occurrence of full-form submissions and the total reconstruction of pages in response. Page modification works entirely within the client browser environment. The server normally responds with data, not HTML, so the server's job of constructing a response is usually much easier. The job of modifying the user's view of the page is done entirely within the browser, while the user is viewing that page, removing a further concern from the server-side application.

In addition to making the server's job easier, AJAX can reduce the processing burden on the server and the amount of traffic on the network. AJAX tends to produce more interactions between client and server, but these interactions tend to be much shorter and simpler than those required to process full-page submissions and respond with full-page refreshes.

If you are interested in seeing some sophisticated AJAX implementations, here are a few that are currently available on the web.

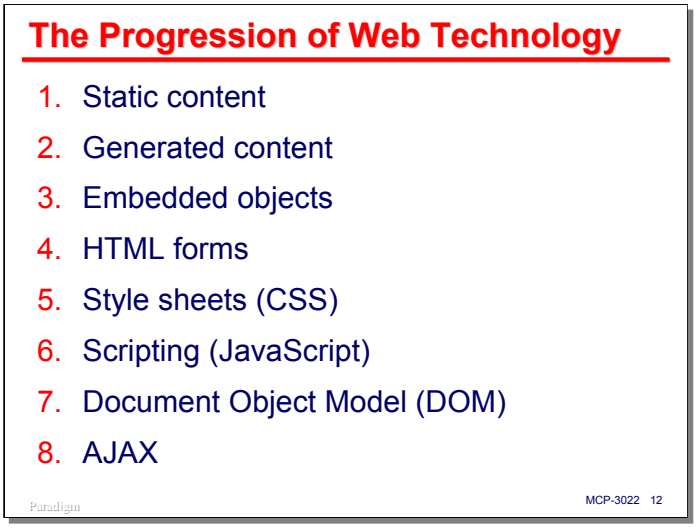
Google Suggest was one of the earliest AJAX implementations to capture wide-spread interest. It showed that you can do a good job of prompting and auto-complete with a web interface. As you begin to type a term in what looks like a typical Google search box, a list of the top ten index keys that begin with what you typed appears in a list below. As you type more characters, the set of index keys is updated to reflect your complete entry thus far. You can, of course, click on one of the keys in the list to display the usual Google list of page hits. The server interactions and list updates occur between keystrokes – there is no "submit" button and no page refresh. This site uses some technology in addition to AJAX, but it is a masterful demonstration of the potential of web-based user interfaces.

Google Maps is probably the best known of the first-generation AJAX-enabled sites. Like Google Suggest, it uses some things in addition to AJAX, but the speed and smoothness of changes in view as you interact with it is truly impressive.

A9 is the Amazon.com search engine. It is another good example of quick and smooth changes to the user's view without overhead and "blink" of full page refresh.

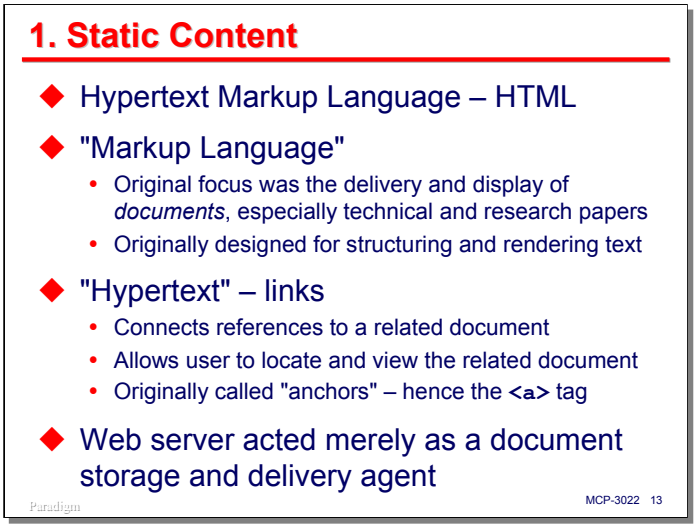
Netflix Top 100 is a nice demonstration of the usefulness of partial page update in response to subtle events occurring on the client. As you move the cursor over a movie title, a balloon pops up with detailed information about that title. Unlike a "tooltip" pop up (which is easy to accomplish in HTML with the `title` attribute of an element) the contents of the balloon are the result of a query sent back to a Netflix server.

Writely is simply amazing, and a powerful demonstration of where web-based user interfaces are destined to go. It is a fairly full-featured word processor that works within a web page and has a very Windows-like user interface. It's not quite Microsoft Word, but it probably does 100% of what 90% of us use Word for. Google recently acquired the start-up company that developed Writely, so we can assume that we will be seeing more applications of this sophistication and power coming along.



To understand AJAX and how it fits into the scheme of web-based applications and user interfaces, you need to view it within the context of existing web technology and how we have gotten to where we are today. Over the next several slides, I'll talk about the progression of web technology and how it has developed over the past 15 years.

The sequence of development presented here is not a strictly chronological one, but is intended to show how the technology components of AJAX build on each other and how their combination resulted in the development of the AJAX technique.

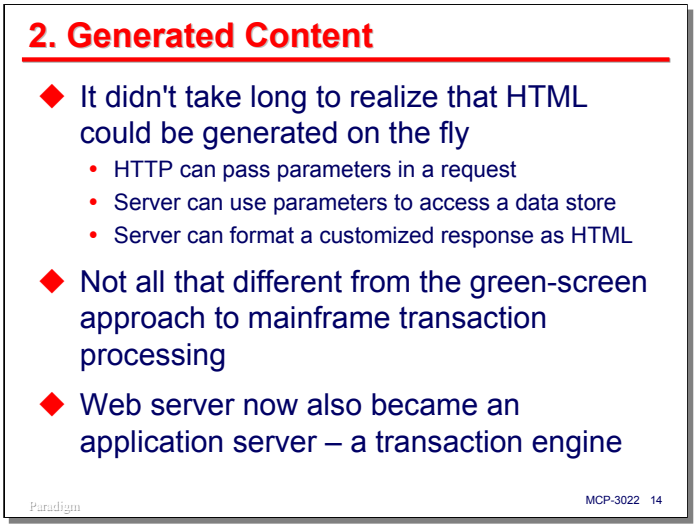


The first use of the web was to deliver static content to clients. To implement that capability, Tim Berners-Lee and his colleagues developed two standards which are still today the underpinnings of the web. The first of these is the Hypertext Transport Protocol, HTTP, which describes the message formats and rules for clients and servers to interact. The second is the Hypertext Markup Language, or HTML, which describes the content and structure of a document to be delivered over HTTP.

I call this first stage of web technology "static content" because it was oriented to delivering documents – something that was prepared in advance and simply recalled on demand. The original web was a paperless way of delivering, well, papers – especially technical and research papers that Berners-Lee was involved with at CERN, where he was working at the time.

The original HTML was composed of two main parts, a markup language derived from SGML, which wrapped a structural description of the document (paragraphs, headings, bullet lists, etc.) around its textual contents, and hypertext links embedded in the document, which allowed the user to efficiently call up and view related documents from the point at which they were referenced in the original document. These links were originally called "anchors," which is why HTML still uses the **<a>** tag to represent a link.

It is important to note that in this first stage of web technology development, the web server acted merely as an agent to store previously-prepared documents, recall them on demand, and deliver them to the client. It was an application by itself, and the user interface on the client was limited to locating documents, rendering them for the user's view, and activating links. This mode of web usage is still very much alive today, and may still account for the majority of interchanges between web clients and servers.

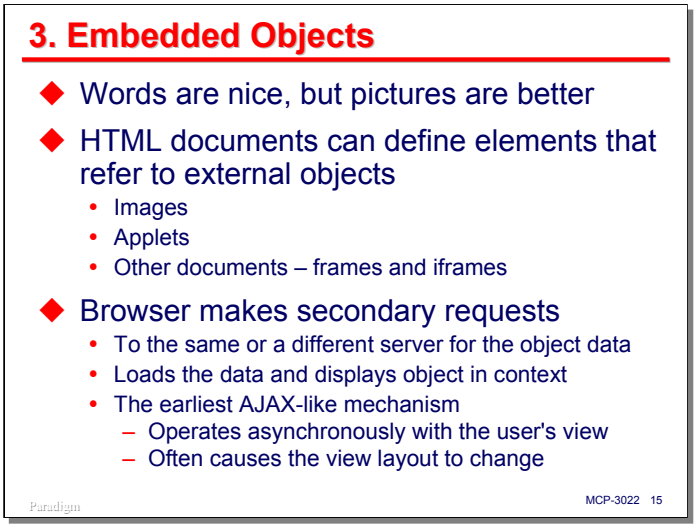


The next step in the evolution of web technology was the advent of generated, or dynamic, content.

It did not take long for people to realize that HTML and the other types of data that are delivered to a web browser could be generated on the fly. HTTP has the ability to pass parameters as part of a request, and the server can use these parameters to access a data store and format a customized response, usually in the form of an HTML-formatted page.

Those of us who have spent our careers with mainframes recognize what is going on at this stage – it's transaction processing – except that instead of having to build a user interface within an 80x24-character box, HTML and the graphical browser give us a lot more flexibility in how the parameters of a request can be entered and how the output can be formatted.

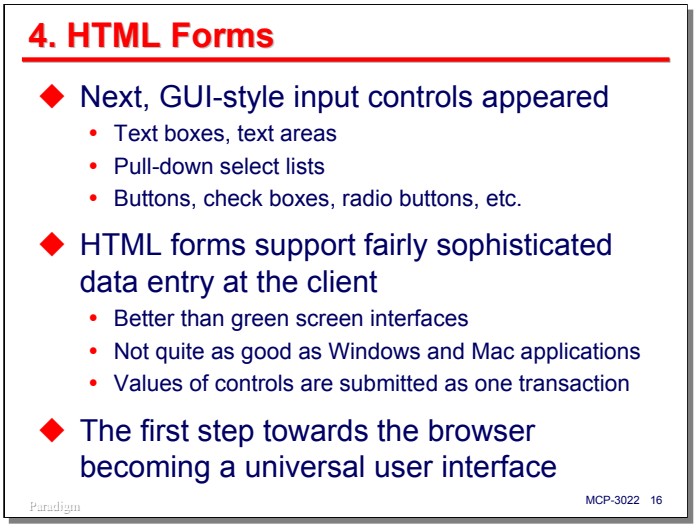
Thus, at this stage, what the web server has become is an application server, or a transaction engine. There are a number of interfaces available to connecting web servers to application programs, including the original Common Gateway Interface (CGI), Active Server Pages (ASP), Java Server Pages (JSP), and the AAPI and WEBPCM interfaces used within the MCP environment.



The next stage in the evolution of web interfaces was that of embedded objects. The most common type of such object is an image. Other examples include applets, frames, and iframes.

The standards formally refer to these type of objects as *replaced elements*, since the content of the element is not contained within the HTML description of the page, but rather a reference to the content, which is stored elsewhere and is accessible through the web. As the web page is being parsed and loaded, the browser issues a separate, secondary request to fetch the content, which it then substitutes in place of the original referencing element – hence the *replaced* designation. We have all seen a web page load, especially over a slow connection, where the text appears first, and the images appear much more slowly. This is a result of the image data being requested separately by secondary submissions to the server, arriving asynchronously, and being rendered by the browser in the background while you view the page.

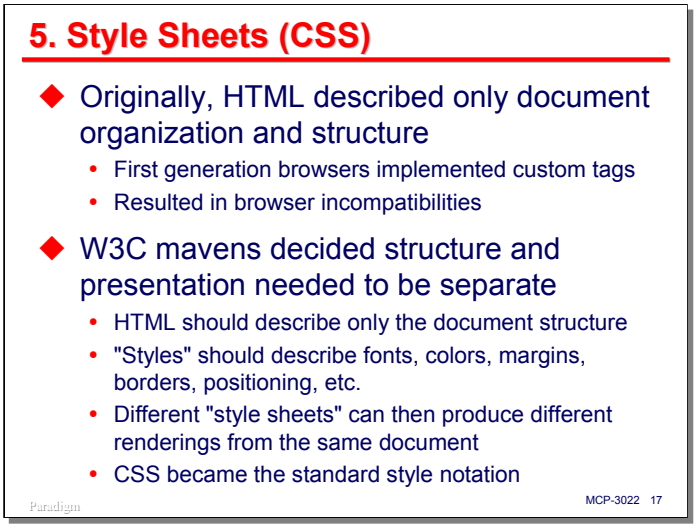
In a way, this is the earliest AJAX-like mechanism for web interfaces. The idea that the browser can issue HTTP requests and process them in the background is at the heart of the AJAX technique.



In the next stage of this conceptual progression of web interface technology, forms and GUI-style input controls appeared. A set of tags were added to HTML to define text boxes, pull-down lists, buttons, check boxes, radio buttons, and other basic user interface controls, along with a form element to enclose the controls. Each control element can be associated with a value (or in the case of select lists, multiple values). The user is able to initiate an event that "submits" the values of all active controls to the server as one request.

The controls supported by HTML are basic, but they permit the designer to create a fairly sophisticated data entry capability on top of a standard browser. HTML forms are not quite as good as the dialogs and controls we are used to seeing in Windows and Macintosh applications, but experience has shown that they are good enough for all but the most demanding user environments.

Forms and GUI controls were the first step towards the browser becoming a universal user interface, and opened the door for the web to become a vehicle for serious query, reporting, and transaction-processing applications.

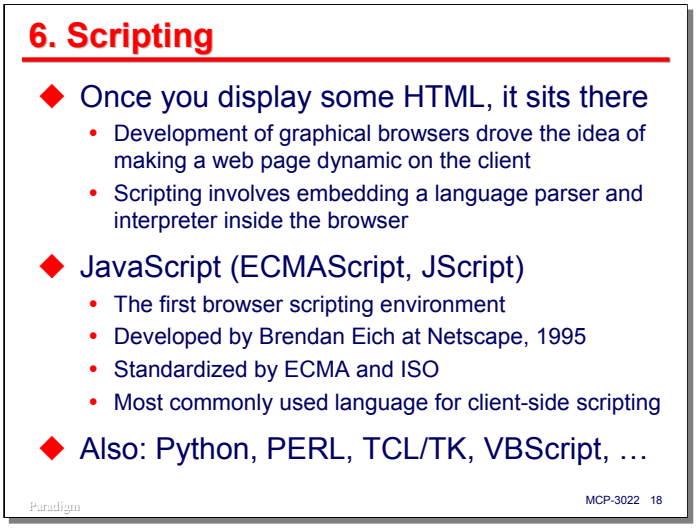


Thus far in this progression of web technology, we have been talking about adding functionality to the server and browser, and supporting user interfaces, but early HTML left a lot to be desired in terms of document presentation. Originally, HTML was intended to describe the organization and structure of a document, not the details of its layout and presentation. You got the font, color, and spacing the browser developer thought you should have.

As a result, the first generation of graphical browsers starting introducing special tags, such as **`<font>`**, and attributes, such as **`bgcolor`**, to control presentation. The problem is that these browser extensions were vendor-specific and somewhat incompatible with other vendors' browsers. Other problems arose as people began to realize that web pages weren't just for CRT screens. Cell phones, PDAs, and specialized browsers for the disadvantaged all started using HTML in ways it was not well suited to support.

Eventually some ideas from the printing profession took hold, and those involved with standards development at the World Wide Web Consortium (W3C) decided that document structure and document presentation needed to be separate. HTML should go back to its original purpose of describing organization and structure, and a new (to the web) concept of "styles" would describe the details of presentation – fonts, colors, margins, borders, positioning, etc. Style rules could be collected in documents called "style sheets", which could in turn be referenced from HTML pages. If the structure and presentation of a document were properly separated, one HTML document could then be rendered in a number of different ways simply by changing the style sheet (or sheets) associated with it.

Only one style sheet notation has at this point gained wide-spread acceptance, Cascading Style Sheets (CSS) from the W3C. CSS 1 has been a standard since the late 1990s, and most modern browsers support it fairly completely. CSS 2 is supported less completely, but that situation is improving rapidly with the latest browsers, such as Firefox 1.5 and IE 7. CSS 3 is currently under development. Many of the old HTML features, such as the **`<font>`** tag, have been standardized, but are now officially deprecated.

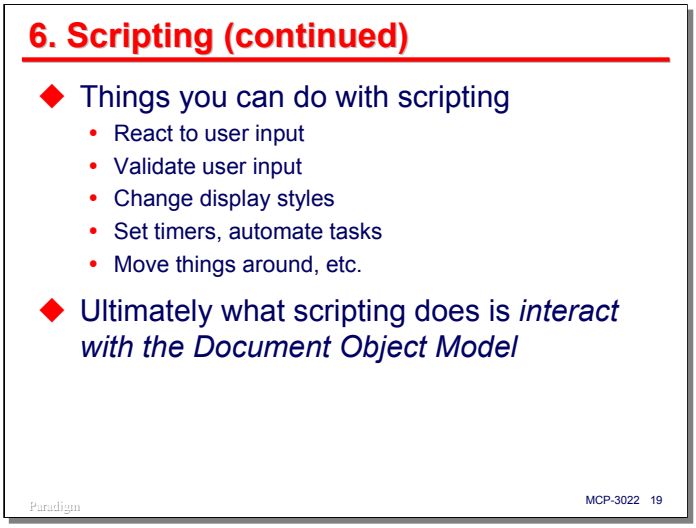


The next evolutionary step in the progression of web interface development was a profound one – scripting. With plain HTML, once you display a web page, it just sits there. Nothing can change until you submit something to a server and it responds with a replacement page.

Development of the graphical browser drove the idea that a web page could be a dynamic entity, and to do that, the browser had to become programmable. Scripting involves embedding a programming language parser and interpreter in the browser. HTML documents can have programs ("scripts") embedded within them. These scripts can be activated when the document is loaded in the browser, or in response to a user interface event (e.g., changing the contents of a text box).

The first such scripting language was JavaScript, originally developed by Brendan Eich at Netscape in 1995. It has been standardized by the European Computer Manufacturers Association (ECMA), and more recently by ISO. It's official name is ECMAScript, but everyone still calls it JavaScript.

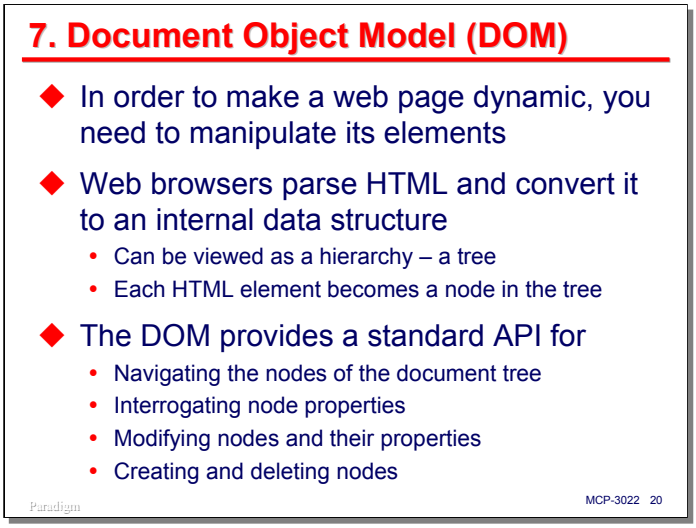
JavaScript is still by far the most prevalent scripting language used with web browsers, and every modern web browser supports it. Depending on the browser, there are other scripting languages that can be used as well, including Python, PERL, TCL/TK, and (for Microsoft only) VBScript.



Now that we have this nice programming language embedded in our browser, just what can you do with it, and how does that help make the web page dynamic? In fact, what does it mean for a web page to be dynamic? Here are some examples:

- You can react to user input.
- One of the ways you can react to user input is by validating or normalizing it.
- You can change the styles of elements so that they are rendered differently. For example, you can change the color of an element.
- You can set timers and automatically do something when that timer expires. You can automate tasks, so that one event triggered by the user accomplishes many thing on the page (e.g., validate *all* the fields on a form).
- You can move things around and make them appear and disappear.

Ultimately, though, all of the things you can do in a browser with scripting revolve around a single, central concept – interaction with the browser's Document Object Model.



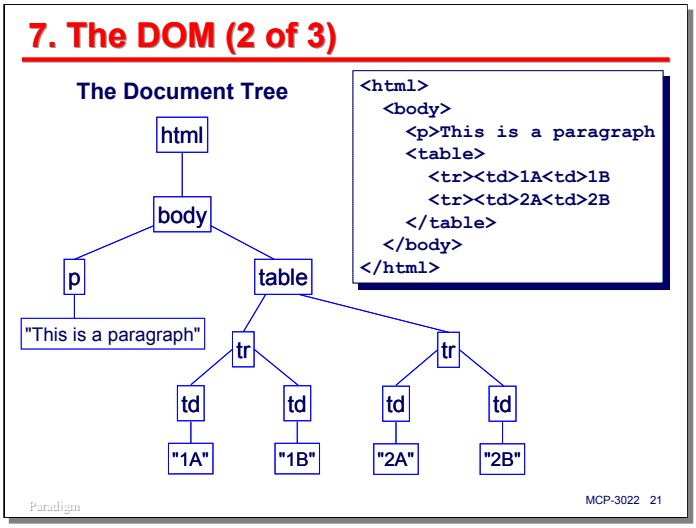
In order to make a web page dynamic, you need to be able to manipulate the elements of that page. What are the elements of a web page? Initially, they are all of the things that were specified in the HTML description of the page – paragraphs, bullet lists, images, tables, forms, text boxes, buttons, and so forth.

Web browsers do not actually display HTML documents. What they do is parse the HTML into an internal data structure, and then use that data structure to drive the rendering and presentation of the document to the user. HTML is just a way to get that data structure loaded into the browser in the first place.

This data structure can be viewed as a hierarchy, or tree. Each HTML element becomes a node in the tree. Therefore, to manipulate the view that the user sees, you need to be able to traverse this tree and manipulate the individual nodes.

The Document Object Model, or DOM, is a brilliant concept. It is a standard API for navigating the nodes of the document tree and manipulating those nodes. You can interrogate node properties, modify nodes and their properties (including styles), rearrange nodes, and even create and delete nodes.

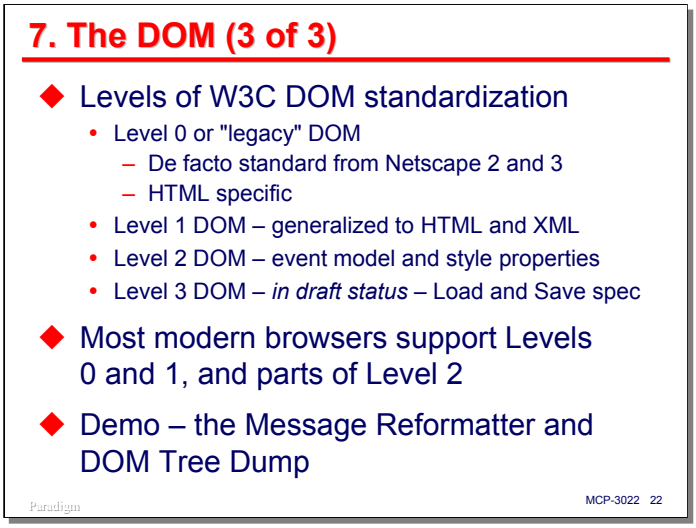
In principle, you can build a web page by not sending any HTML to the browser at all – just send some JavaScript and have it use the DOM to build the elements of the page. This radical an approach is seldom taken, but the idea that you use scripting and the DOM to alter the elements (or nodes) in a portion of the document tree is one of the two main enablers for AJAX (the other being the **`XMLHttpRequest`** object).



This slide shows a very simple HTML document containing a short paragraph and a table with two rows and two columns. It also shows the tree of document nodes that would result from parsing the HTML. Note that each HTML *element* (as opposed to tag) becomes a node in the tree. Strings of text contained within elements also become nodes of the tree.

HTML elements and text are the two primary types of nodes that exist in a document tree. There are a few other types, but they are not as common.

Consider, though, that HTML and XML have a lot in common (both are derivatives of SGML), and that the elements of XML documents are also hierarchical and can be parsed into a tree structure. The same DOM API that is used with HTML documents can be used to navigate, interrogate, and manipulate XML documents. The ability to use the DOM API in a browser to pick apart an XML document is where the "**X**" in AJAX comes from, even though it is not necessary to use XML with AJAX.



The DOM is standardized by the W3C and currently has three levels. A fourth level (3) is currently in draft status.

- The Level 0 DOM is also called the "legacy" DOM. This represents the de facto standard that was introduced in the Netscape 2 and 3 products. It is still widely used, but is specific only to HTML documents. It is usually implemented as a set of JavaScript objects.
- The Level 1 DOM is the first version created by the W3C. It is designed as a true API and looks completely different from the Level 0 DOM, although functionally it is a superset of Level 0. Aside from this syntactic difference, the main advance with Level 1 was its generalization to handle both HTML and XML documents.
- The Level 2 DOM is the current standard. The main extensions over Level 1 were a generalized event model (which Microsoft does not currently support) and properties for styles.
- The Level 3 DOM is presently in draft status. It has a number of extensions to Level 2. One of these is the Load and Save specification, a part of which is the standardization of the **`XMLHttpRequest`** object.

Most modern browsers to a fairly good job of supporting Levels 0 and 1 of the DOM, and all have some support for Level 2. Due to the complexity of implementing these DOM standards, comprehensive browser support seems to lag finalization of the standards by at least four to five years.

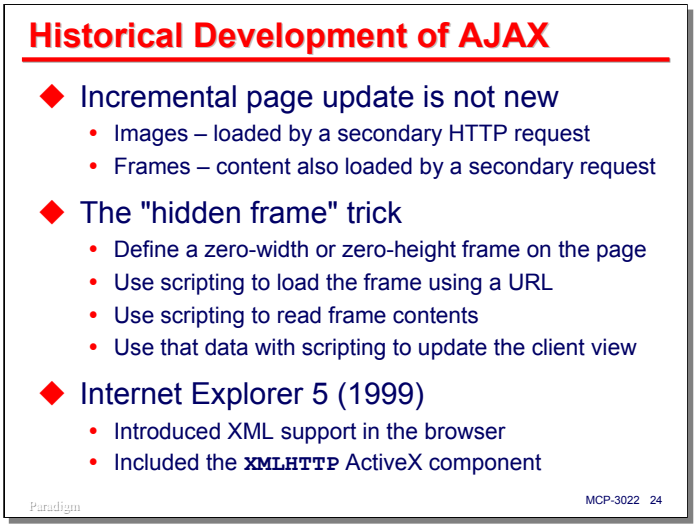
The next demo I want to show you illustrates how the DOM API can be used to completely rearrange the contents of a web page and display it in an entirely different form. It also illustrates the document tree and how it can be traversed.

8. AJAX

- AJAX is just the latest step in the progression of web technology development
- Has nothing really new
  - Uses a combination of existing technologies
  - Most have been around since at least 2000
- What is new is the *approach* – the realization of a new way to use these technologies together

This brings us to AJAX, which is just the latest step in the progress of web interface technology development that I have been discussing over the past several slides. There is nothing really new that is added at this stage. AJAX is simply a combination of the technologies just covered, most of which have been around since at least the year 2000.

What *is* new is the approach that AJAX takes – the way these technologies have been combined and the synergy that results from that approach.



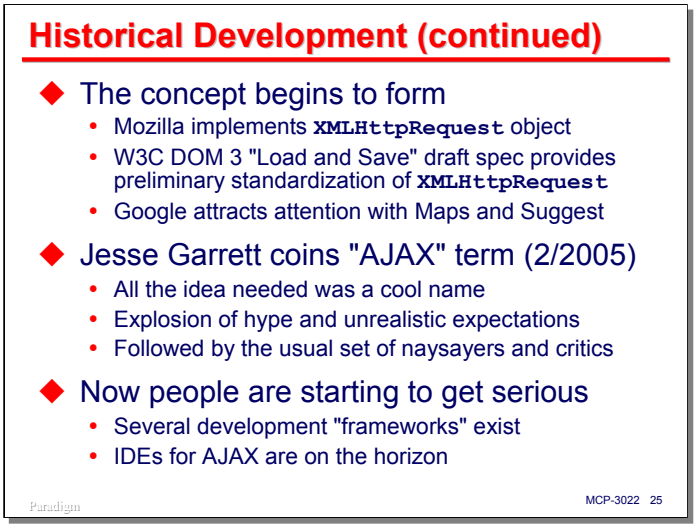
To better understand AJAX and how it takes advantage of the technologies on which it is based, it's instructive to examine how the concepts behind AJAX developed historically.

First, the idea of incrementally updating the page as the user is viewing it is not new. We have all seen that happen with slowly-loading images, which are the result of a secondary, behind-the-scenes request by the browser to fetch the image data. Incremental update can also be done with frames, as through scripting you can change the **`location`** property of a frame window to submit an HTTP request to a server, which will load the window with the server's response.

This led to a pre-AJAX technique used by some designers that involved the use of "hidden frames." You can define a frameset in HTML that contains multiple frames, one of which has a zero width or zero height. Since one of the dimensions is zero, this frame not visible to the user, but it exists as an object within the browser and can be manipulated by scripting and the DOM API. You can set the hidden frame's **`location`** property to a URL, which will cause some data (in whatever form the server sends it, perhaps plain text) to be loaded in the frame. Using scripting, you can read the data in the window, and via the DOM use it to update the visible parts of the frameset. All of this can happen in the background while the user is viewing the parts that are visible.

The hidden frame trick is essentially AJAX, but the way you submit the request and retrieve the data via the hidden frame is a bit of a kludge. Microsoft (somewhat unwittingly, it now appears) paved the way for AJAX by releasing a better way to interact asynchronously with a server in the background. As part of the XML support for IE 5, which was released in 1999, they implemented an ActiveX component named **`XMLHTTP`**. Their version of JavaScript (uh, sorry – *J*script), has the ability to instantiate an ActiveX component as a scripting object. This object allows a script to submit an HTTP request asynchronously, monitor its status, and retrieve the response when the request completes.

All of the pieces for AJAX were now in place, but the potential was not realized until a few years later.

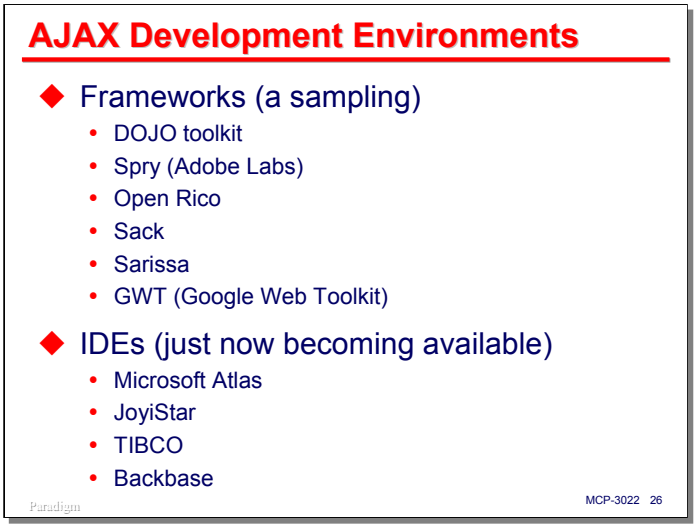


There seems to be some dispute as to who actually pulled the pieces together and did what we now call AJAX first, but it's clear that the concept was forming during the 2003-2004 timeframe. Mozilla implemented the equivalent of Microsoft's **`XMLHTTP`** component as a native JavaScript object, **`XMLHttpRequest`**. Other browser developers followed suit and the native object was formalized in the W3C's draft Load and Save specification for DOM Level 3.

The thing that seems to have caught everyone's attention, though, was the work Google was doing with its web-based applications, especially Google Maps and Google Suggest. Suddenly here were web-based user interfaces that didn't look like they were running in a web browser at all – except for having a slightly slower response time due to the latency of the Internet, they looked a lot like installed Windows applications, and really nice ones, at that.

Then Jesse Garrett coined the AJAX term in early 2005, and interest in the technique reached a fever pitch. It seems that all that this approach needed was a cool name. That name simultaneously validated the concept and gave people a convenient way to talk about it. You could say it was very clever marketing, except that it apparently happened by accident. Of course, with the explosion of interest and a few slick implementations available, there was also an explosion of hype and unreasonable expectations, which is still taking place. Right after the hypemeisters (you could almost hear them saying "AJAX changes everything") came the naysayers and critics (and probably a few simply spreading FUD) saying it was a bad idea and wouldn't work, and so forth. As usual, the truth is somewhere in between.

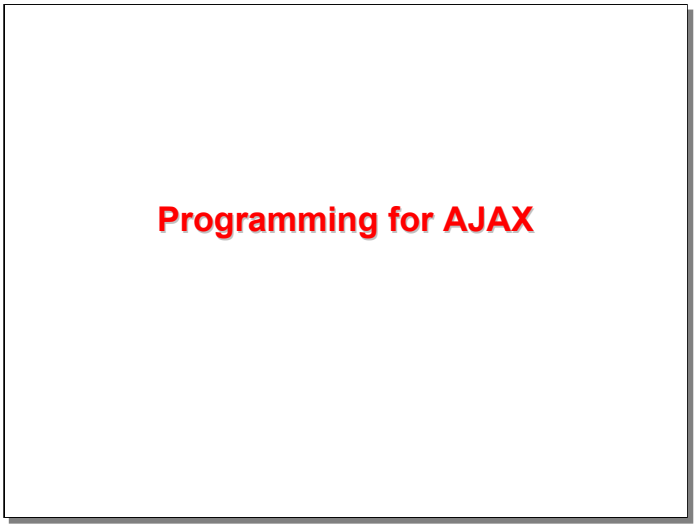
At this point the hype is starting to calm down somewhat and people are starting to get serious about figuring out where and how AJAX fits in with the rest of web development technology and technique. There are a number of frameworks (or toolkits) that are currently available for AJAX development, and a number of IDEs oriented towards AJAX are on the near horizon. If nothing else, AJAX has triggered a resurgence of interest in, and respect for, JavaScript as a programming language.



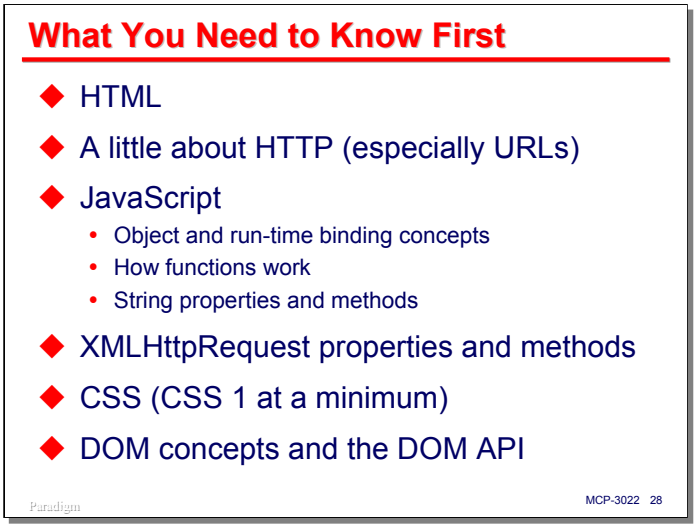
This slide lists a number of the frameworks that are currently available for AJAX. I have not used any of these, and cannot comment on their features or suitability.

GWT, the Google offering, seems especially interesting, though. With GWT, you program your user interface in Java (not JavaScript), and test and debug it using standard Java tools. Then you run the Java through the GWT translator, and it spits out a JavaScript version that will run in the browser environment.

I have also listed a few IDEs that are either out or are nearing release. Again, I have not used any of these, and show them here simply as examples of what is currently happening in this technology space.

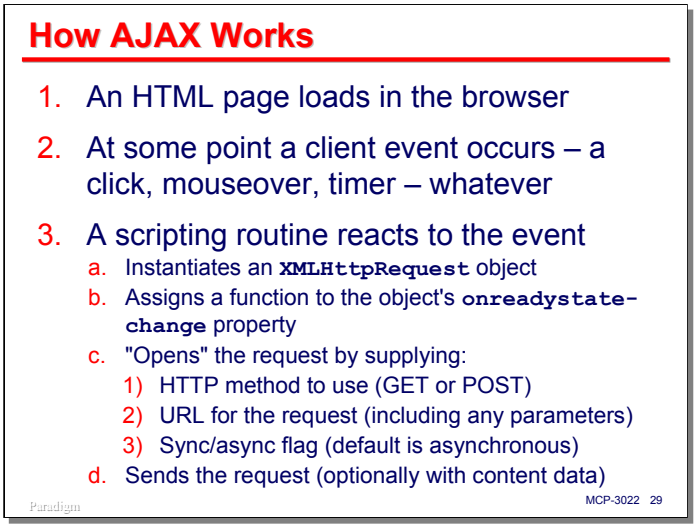


Having covered all of this background on AJAX and considered how it developed, it's now time to see how you actually apply the technique and look at some code.



One of the challenges of AJAX (and something that the frameworks and IDEs mentioned earlier all try to address) is that you need to know quite a bit of stuff about web development in order to use the technique.

- You should have at least a passing familiarity with HTML, the types of elements your web page will be using, and how those elements relate to each other.
- You need to know a little about HTTP, especially the format of URLs, query string parameters, and the difference between GET and POST requests.
- AJAX implies scripting, so you are going to need to have a working knowledge of some scripting language, probably JavaScript. You need to understand how JavaScript objects work, and how names are bound to objects and properties at run time. You also need to know how JavaScript functions work, and how functions can be manipulated as data. Finally, you are probably going to be parsing and formatting data, so you should have a strong familiarity with the properties and methods of JavaScript strings.
- You will need to understand the **`XMLHttpRequest`** object, its properties, and methods, as this is how you will submit asynchronous requests to the server.
- You should be familiar with styles and style sheets, and how the style cascade works. At a minimum, you should be familiar with Level 1 of the CSS standard.
- Finally, since you will be manipulating the user's view of the page at run time, you need to understand DOM concepts and the relevant portions of the DOM API. You should be familiar with at least the Level 1 DOM specification.



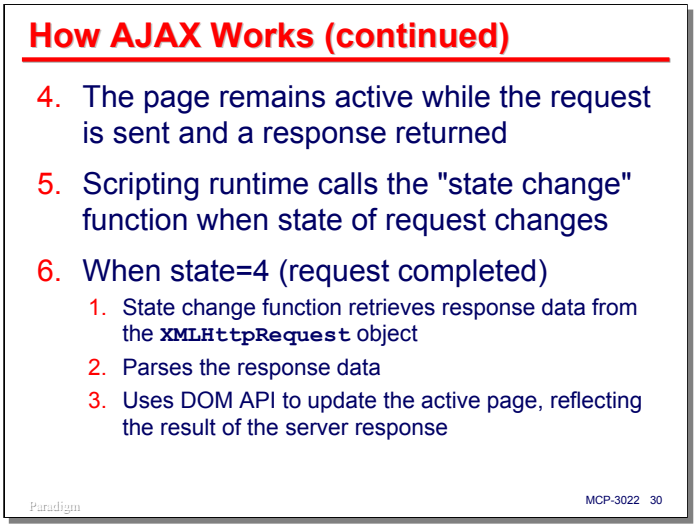
Over the next several slides I am going to describe in detail how AJAX works and show the basic coding involved in implementing an AJAX interaction.

It all starts when an HTML page is loaded into a browser. The browser parses the HTML, creates a document tree, and from that tree renders the document for the user's view.

At some point after that, a client event occurs. That event could involve clicking a button, changing text in a field, or selecting an item from a list. It could also involve a more subtle event, such as passing the cursor over an element (a "mouseover"), or the expiration of a timer.

In response to that event, the browser run time environment activates some script code (we'll see how this is done shortly). The script code recognizes that it needs to communicate with the server, so it does the following:

- It creates an instance of the **`XMLHttpRequest`** object. Alternatively, it may reuse an instance of this object that it had previously instantiated.
- It assigns a function to the object's **`onreadystatechange`** property. The job of this function is to react appropriately whenever the state of the request changes. The object will automatically call this function whenever it notices one of these state changes.
- The script code then "opens" the request by calling the object's **`open()`** method with the following parameters:
  - A string containing the HTTP request method, usually "GET" or "POST".
  - A string containing the URL for the request. If the request method is "GET", the URL will normally contain a query string for the parameters of the request.
  - A Boolean value indicating whether the request will be made synchronously or asynchronously. A value of **`true`** implies asynchronous, and is the default if this parameter is omitted.
  - There are also optional parameters that will pass credentials to the server – a user name and password.
- Finally, the script code calls the object's **`send()`** method to initiate the request. If the request method is "POST", a parameter to this method supplies the body, or content, of the request.



At this point the request has been sent to the server, and assuming the request is an asynchronous one, the page remains active in the browser and the user can continue interacting with it.

As the state of the request changes, the object will call the function assigned to its **`onreadystatechange`** property. An asynchronous request goes through five states:

0 = uninitialized

1 = open (set after the **`open()`** method call)

2 = sent (set after the request has been completely sent to the server)

3 = receiving (set as the response is being received from the server)

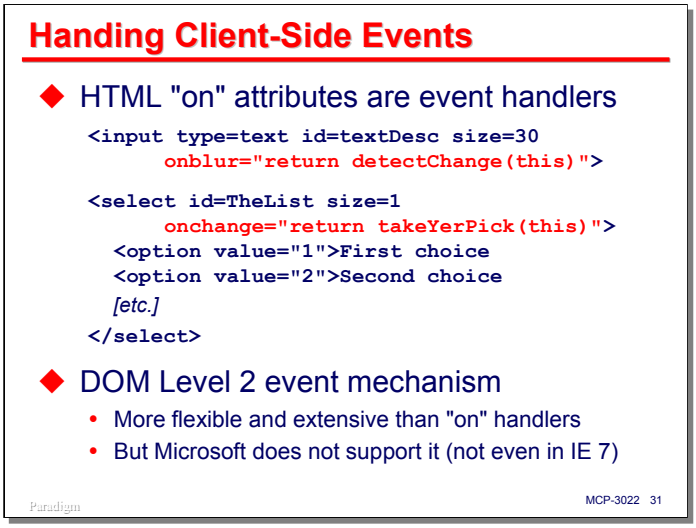
4 = completed (set after the response has been completely received)

In most cases, the function handling state changes is only interested in the final state, completed. When this state is signaled, the function typically then does the following:

- It retrieves the response data from the **`XMLHttpRequest`** object. It can always retrieve the data as raw text. If the response has been sent as XML (and the response's HTTP Content-Type header indicates the response is in XML format), the function can also retrieve the response as a DOM document object.
- The function then parses the response data. If the response is in XML, it can use the DOM API to traverse the document object and extract the relevant information. Otherwise, there must be some agreement between the client and server as to the format of the data, and the function will need to parse the raw text response.
- Using the results from parsing the response, the function manipulates the appropriate parts of the web page DOM to update the user's view.

Obviously, the state-change function may not be doing all of this processing itself, and may be calling other script functions to perform some of these tasks. It is common practice for the state-change function to worry only about state changes, and to call separate function that takes responsibility for all processing of the response data.

Let us now look at these steps in more detail, and the type of HTML and JavaScript coding necessary to implement them.



The first thing to look at in detail is how script code can be attached to client events. The traditional, and still most common, approach is to associate event handlers with specific elements on the page. In HTML, these event handlers are specified by the so-called "on" attributes of HTML tags. The value of these "on" attributes is a string containing script code. When the event related to that "on" attribute occurs, the corresponding script code is executed by the browser's run time system.

There are a large number of these events, and several (such as **`onmouseover`** and **`onclick`**) can be associated with almost any HTML element. Form controls support an additional set of events (such as **`onchange`** and **`onselect`**) related to their function has data entry elements.

The examples on the slide show two common types of events. The first, **`onblur`**, is attached to a text box. This even fires when the user moves the current focus away from the control, perhaps by tabbing or clicking another part of the page with the mouse. In this case, the event handler simply calls some other scripting function, which is presumably embedded elsewhere in the page. The "**`this`**" parameter to that function refers to the input box's DOM object. Event handlers can return a Boolean result. If the result is false, the run time system ignores the event and acts as if it never happened.

The second example illustrates the use of an **`onchange`** event attached to a select (or pull-down) list. This event will fire whenever the user changes the currently-selected item in the list, usually by clicking on the item.

The DOM Level 2 standard defines a more flexible and extensive event mechanism than these "on" attributes, but at present it is seldom used. The primary reason is that Microsoft browsers support an entirely different, Microsoft-specific event mechanism, which means that the standard DOM mechanism is not usable by perhaps 90% of the browsers currently on the Internet. Microsoft has continued this incompatibility in Internet Explorer 7, which is now in late-beta status.

Sending the Request

```
var xhr = null;
var url = "/tran101?Param1=5&Param2=ABCDEF";

try {
  if (window.XMLHttpRequest)          // Mozilla (et al)
    xhr = new XMLHttpRequest();       // also works in IE 7
  else {
    try {                             // Microsoft IE 5/6
      xhr = new ActiveXObject("Microsoft.XMLHTTP")}
    catch(e) {
      xhr = new ActiveXObject("Msxml2.XMLHTTP")}
    }
  }
catch(e) {
  throw new Error("XMLHttpRequest object not available")}

if (xhr) {
  xhr.onreadystatechange = xhrCallback;
  xhr.open("GET", url, true);         // true ⇒ asynchronous
  xhr.send(null)}                     // send with no content
```


The next item to consider in detail is the instantiation of the **`XMLHttpRequest`** object and submission of the request.

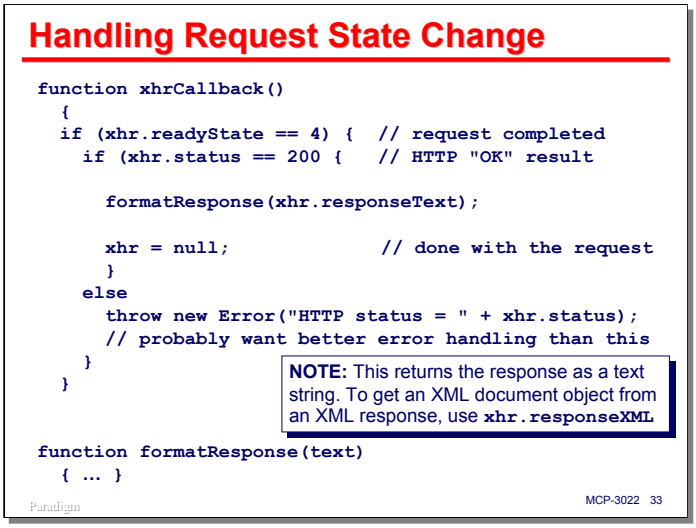
Microsoft has another incompatibility with the standards (and the other modern browsers) in that it still implements this object as an ActiveX component. All other browsers implement it as a native object. IE 7 will also implement the object natively. Therefore, a little extra coding is needed at present to instantiate the object based on how it is implemented in the browser.

- We first declare a variable, **`xhr`**, that will be used to refer to the object once it is instantiated.
- Next, we check to see if **`XMLHttpRequest`** is defined as a property of the browser's **`window`** object. If it is, we can do a Mozilla-style (standards-based) instantiation of the object.
- If the object is not native to the browser's environment, we next try two different methods of instantiating it according to the Microsoft convention.
- If none of these attempts succeeds, we throw a run-time JavaScript error.

Once the object is instantiated, we next need to perform the following initialization of a request:

- We assign the name of a function to the **`onreadystatechange`** property of the object. This function is discussed on the next slide.
- We then "open" the request. In this case we pass the following parameters:
  - "GET" to indicate this will be an HTTP GET request (as opposed to a POST request).
  - A string containing the URL to which the request will be sent. Since this is a GET request, we include two parameter values (starting after the "?") that will give the server additional information about the request.
  - The value **`true`**, which indicates that this is to be an asynchronous request. Once the request is sent, control will return to the scripting run time environment. If the value were **`false`**, the **`send()`** method would block until the response was received.
- Finally, we call the **`send()`** method to transmit the request to the server. Since this is a GET request, the request has no content (body), so we pass a null as the parameter to **`send()`**.

At this point, control returns to the browser run time, and the user can continue to view and interact with the displayed page.



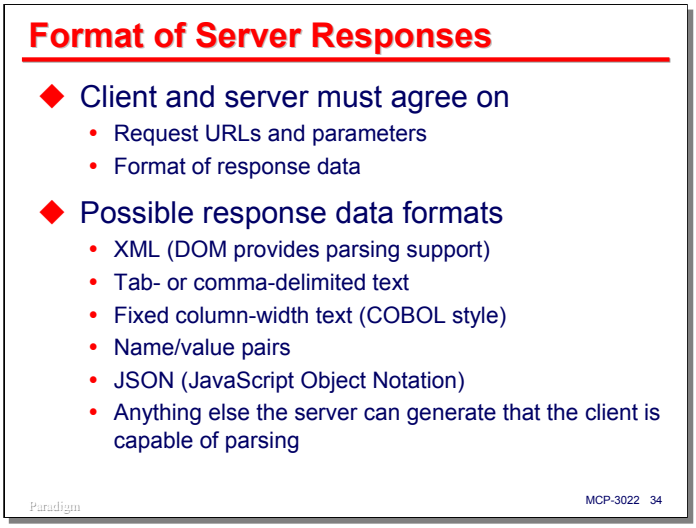
Nothing further happens with this AJAX interaction until the state of the request changes. When a state change occurs, the **`XMLHttpRequest`** object calls the state-change function we supplied to it. As is typically the case, all we care about is the completed state, indicating the server's response has been completely received.

The current state is indicated by the object's **`readyState`** property. As mentioned earlier, the value 4 indicates the request is complete. When the function is called in the completed state, it checks the status of the request as reported by the server, using the object's **`status`** property. 200 is the HTTP status value for a successfully completed request.

In this example, the state-change function delegates all responsibility for processing the response to another function, **`formatResponse()`**. It passes to that function the value of the object's **`responseText`** property, which contains the raw text of the server's response.

If the response were in XML, the state-change function could access the raw XML text using this same property. It could also obtain the response using the **`responseXML`** property. Accessing this latter property causes the **`XMLHttpRequest`** object to first parse the XML text, build a document tree from the XML elements, and return that tree as a DOM document object. The DOM API can then be used to traverse the tree and extract values and properties from the tree nodes.

The routine shown here is a very minimal example. A production-grade implementation would certainly need to have more extensive error checking and error handling code.



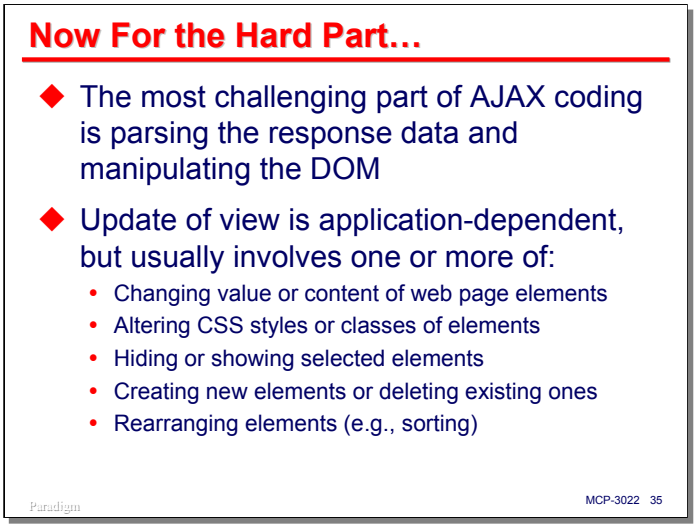
As in all client/server interactions, the client and server need to agree in advance on the format and content of request parameters and response data. In most cases, parameters can be encoded in the URL query string and sent with a GET request. You can, however, embed parameters in the body of a POST request, perhaps emulating the format of a form submission by using URL-encoding.

The possibilities for the format of a response are much more varied.

- XML is a very general-purpose and self-defining way to format data, but generation and parsing of XML can carry a lot of overhead. In addition, not all browsers support the parsing of XML responses.
- You can do something very simple, such as tab- or comma-delimited text.
- You can get even simpler and send COBOL-style fixed-length fields with no delimiters between the fields at all. While this is very easy to generate from COBOL applications, the client will need to know the exact offsets and lengths of each field in the response.
- Another option is to use name/value pairs. There are several conventions for this, with URL-encoding being the one most commonly used for web applications (e.g., **`name1=value1&name2=value2&`**…).
- There is a relatively new format that is quite popular with AJAX designers called JSON (JavaScript Object Notation). This is essentially a fancy way of representing name/value pairs, but has more general capabilities. The JavaScript language allows you to code something called an "object literal". This is a series of name/value pairs contained within curly braces and delimited by commas. The nice part is that the "value" can be any type of JavaScript literal, including arrays, functions, and other object literals. This literal is transmitted as ordinary text in the request response. Scripting then passes the text of the literal to the JavaScript **`eval()`** function, which compiles the literal at run time and (if successful) adds the resulting object to the run-time name space. Script code can then access the properties of the object using standard JavaScript syntax. A JSON string representing a numeric value, a string, and a literal array might look like this:

```
{num:25, str:"This is a string", ary:[1, "abc", null]}
```

- Finally, the response can consist of anything else the server can generate and the client is willing to parse, including binary data.

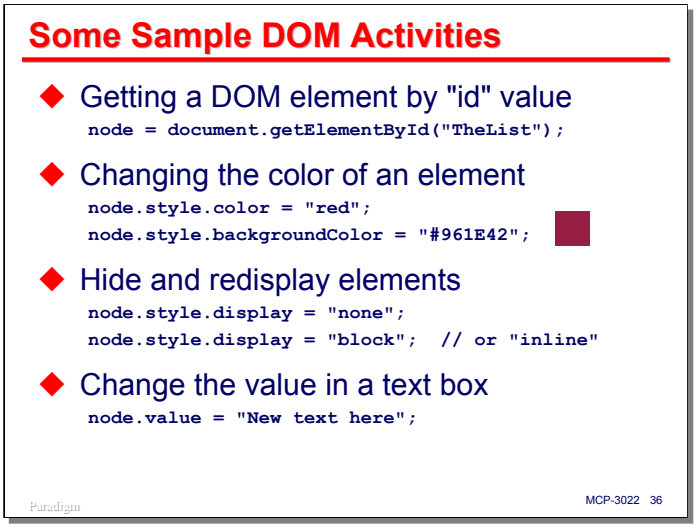


Thus far we have looked at how to submit a request and retrieve the response. Once you learn the **`XMLHttpRequest`** object's properties and methods, though, this part of AJAX coding is normally not very difficult.

The hard part is doing something with the response once you have received it and parsed the data. This usually involves manipulating the DOM to change some portion of the user's view. Web pages can have a complex structure, and the DOM is an extensive API, so most of the design and coding is usually invested in this part of the implementation.

What you need to do in this step is, of course, entirely dependent on your application and the design of the page you are going to modify. Typically, though, you do some combination of the following:

- Change the value or content of web page elements. This could include changing the text in a text box control, checking or unchecking a check box control, or changing the contents of a table cell.
- Alter the CSS styles or classes of page elements. This could include changing color, font, border, or any of the other style characteristics of the element.
- Hide or show elements. CSS has two styles, **`display`** and **`visibility`**, which control whether an element is visible on the page.
- Create new elements or delete existing ones. Elements can be added or deleted from the document tree using calls to the DOM API. New elements can also be added using HTML injection, by means of the **`innerHTML`** property of an element node.
- Rearrange existing elements. Using the DOM API, you can move element nodes from one part of the tree to another. This can be used to rearrange a table, e.g., by sorting rows on the value of one of its columns.



This slide shows some simple DOM manipulation activities. One of the most basic things you do with the DOM is locate a node and obtain a reference to it. There are a number of ways to do this. One of the most common is to search for the node by its element "id" attribute using the **`document.getElementByID()`** method.

Once you have a reference to an element node, you can read and modify the properties of that node, and call methods of the node's object. A common thing to do is change the color of the element's text or background. Colors can be specified using a set of standard names or by a hexadecimal RGB value.

Hiding and showing elements on the page is quite simple. Setting the **`display`** property of an element's **`style`** property to **`none`** causes the element to disappear from view and the space formerly occupied by it to be closed up. This will cause the browser to rearrange and redisplay the page as if the element never existed. The element can be made visible again by setting the **`display`** property to **`block`** (for block-level elements like paragraphs and tables) or to **`inline`** (for in-line elements such as spans and bold text).

Most form controls support a **`value`** property. This can be used to read or set the value the control will transmit on submission of the form. In most cases is also causes the value displayed by the control to change. The example on this slide shows how the value of a text box control might be changed.

Sample DOM Activities (continued)

- Deleting all rows from a table body

```
var kid = tabBody.firstChild;
while (kid) {
  tabBody.removeChild(kid);
  kid = tabBody.firstChild}
```

- Appending a row to a table body

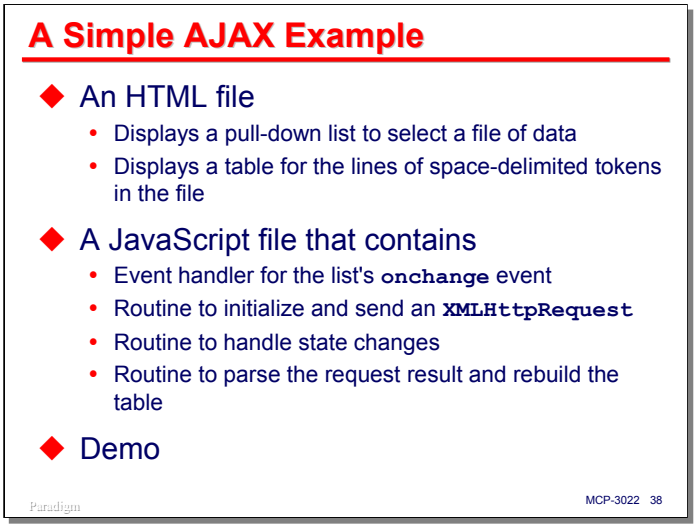
```
var cell;
var row = document.createElement("tr");
cell = document.createElement("td");
cell.appendChild(document.createTextNode("1A"));
row.appendChild(cell);
cell = document.createElement("td");
cell.appendChild(document.createTextNode("1B"));
row.appendChild(cell);
tabBody.appendChild(row);
```


This next slide shows a couple of more advanced DOM techniques.

The first example deletes all rows from the body of a table. Assume we have already obtained a reference to a **<tbody>** node of the table and stored that in the **tabBody** variable. The **firstChild** property returns a reference to the first subordinate (child) node of that node. For a table body, the first child is the first table row (**<tr>**) node in the body. The **removeChild()** method deletes that row from the table, along with any nodes that are subordinate to the deleted one. Thus, that method call also deletes the table cell (**<td>**) nodes for the row and all of the cell contents. Then the code again obtains the first child node of the body (which, since the first row has been deleted, is now what used to be the second row), and loops, continuing to delete rows until there are no more. At that point the **firstChild** property will return **null**, which evaluates as **false** in the **while** statement, thus terminating the loop.

The second example adds a row to a table body. This is more complex, since we must create not only the row, but all of the cells and the content of each cell. Again assume that **tabBody** contains a reference to the table body. The **document.createElement()** method creates a new element node based on the tag name that is passed as a parameter. The **document.createTextNode()** method creates a new text node based on the value of the string that is passed as a parameter to it. The **appendChild()** method adds the node object passed as its parameter to the end of the end of the list its object's child nodes. Finally, after having built up the table cells for the row, the row is appended to the table body object.

This is a simple example, creating a row with just two cells, each containing a text element ("**1A**" and "**1B**", respectively), but it illustrates the general technique of creating nodes and appending them as children of other nodes to build up a document sub-tree. That sub-tree is then attached to some existing node in the web page's document object to add it to the page and make it visible. Simply adding a node to the document tree causes the browser to rearrange and redisplay the page according to the modified document structure.



To pull all of the foregoing discussion on the AJAX technique together, I've tried to come up with a simple example that illustrates using the **`XMLHttpRequest`** object and the response from a request to manipulate the DOM of a web page.

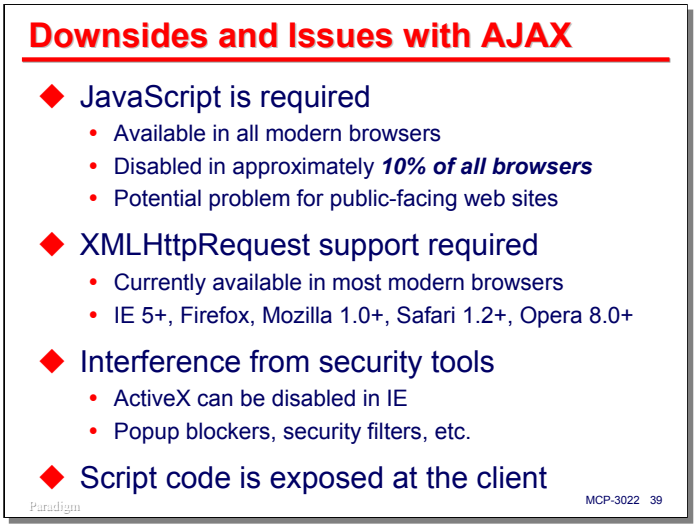
The web page consists of a pull-down list and a table. Selecting an item from the list causes a request to be sent to the server. Based on the selection made, the server responds with a standard text file. The file is assumed to contain lines delimited by new-line characters. Each line is assumed to contain a series of space-delimited tokens. The JavaScript code parses the response into lines, and each line into an array of tokens. It then deletes the heading and body of the table, replaces the heading row with the tokens from the first line in the file, and rebuilds the body rows from the remaining lines in the file, each line representing one row, and each space-delimited token in the line becoming one table cell in the row. The code is oblivious to the number of lines in a response and the number of tokens on a line – it will format whatever it finds in the response text. The first selection in the pull-down list is a dummy. Selecting it will delete any existing table body, but not load a new body.

This example consists of three parts:

- An HTML file that defines the basic web page.
- A JavaScript file which is referenced from a **`<script>`** tag in the HTML file. This JavaScript file contains the event handler code that is activated when a selection is made from the list, the code to set up the **`XMLHttpRequest`** object and send a request, and the code to react to state changes and process the server's response.
- Three text files that provide the response text for the three entries in the pull-down list.

Note that the last selection in the list ("Text Lines") does not contain tabular data and that there are a variable number of tokens on a line. This results in a table having rows with varying numbers of cells, but this is the correct result, and is a product of the JavaScript code blindly creating rows from whatever tokens it finds on a line.

These files are included as an attachment to this presentation and are available from our web site using the link shown on the References slide at the end. If you use Internet Explorer to view this demo, you do not need to have a web server involved -- just put all five files in the same folder on your local system and open the HTML file in IE. This may not work with other browsers, since strictly speaking, fetching a file from the local file system is not done with an HTTP request, and hence does not work with the **`XMLHttpRequest`** object.

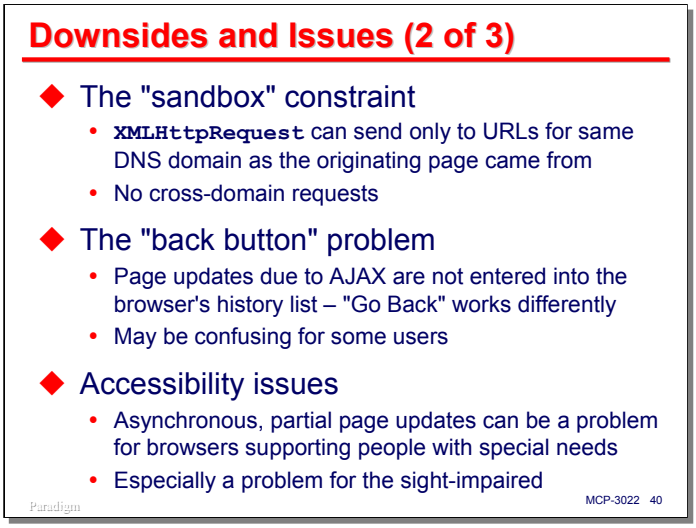


AJAX is a wonderful technique, but it comes with its own set of downsides and issues to be considered. The first is that it requires JavaScript (or whatever scripting language the web page invokes). JavaScript is available in all modern browsers, but most browsers allow the user to disable scripting. Some recent studies estimate that approximately 10% of all browsers on the Internet have JavaScript disabled. For an intranet application, browser settings and the availability of JavaScript are fairly easily controlled, but the potential lack of JavaScript in a subset of clients is a potential problem for public-facing web sites.

Second, support for the **`XMLHttpRequest`** object must be available in the browser. All of the modern browsers with a noticeable user base now support this object, but there are still plenty of copies of older versions of these browsers out there, so this is a consideration, especially for public-facing sites.

Another potential problem is interference from security tools. The instantiation of ActiveX components can be disabled in IE, which can prohibit use of the **`XMLHTTP`** ActiveX component. Popup blockers, security filters, and some of the popular browser toolbar plug-ins can also interfere with the mechanisms upon which AJAX depends.

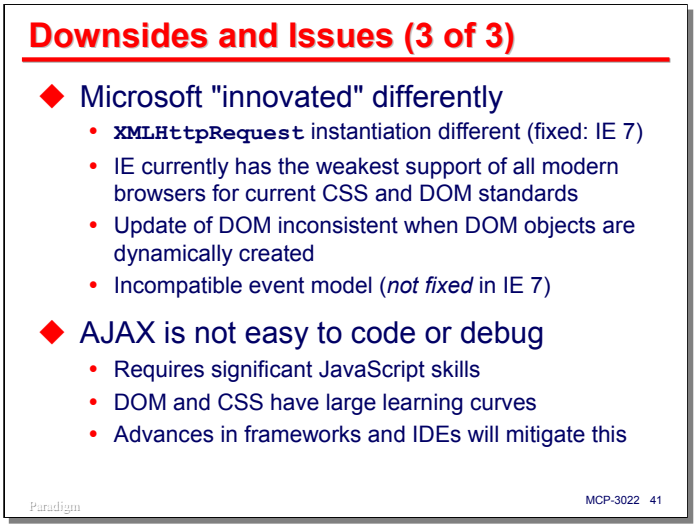
Something else for the designer and implementer to consider is that all JavaScript code is transmitted in clear text to the client and is visible at the client. Script obfuscation tools can mitigate this somewhat, but they do not provide complete protection. This means that there are some things that you need to avoid doing on the client, particularly security-related activities. These are usually better done on the server, anyway, where the code is not visible.



Another issue to consider is the security that the **`XMLHttpRequest`** object enforces. You can send requests only to servers within the same DNS domain as the one from which the original web page came. Attempts to make a cross-domain request will be rejected by the object. This is called the "sandbox" constraint, and is intended to keep malicious web sites from using the browser as a pathway for accessing other, perhaps secured, sites.

Then there is the "back button" problem. Browsers keep a history list of recently-visited pages. Users can use this list to revisit prior pages. This history list is updated only on full page refreshes, however, and changes to the user's view due to scripted DOM updates are not represented in that list. Therefore, using AJAX often modifies the meaning of "Go Back," and this can be confusing for some users.

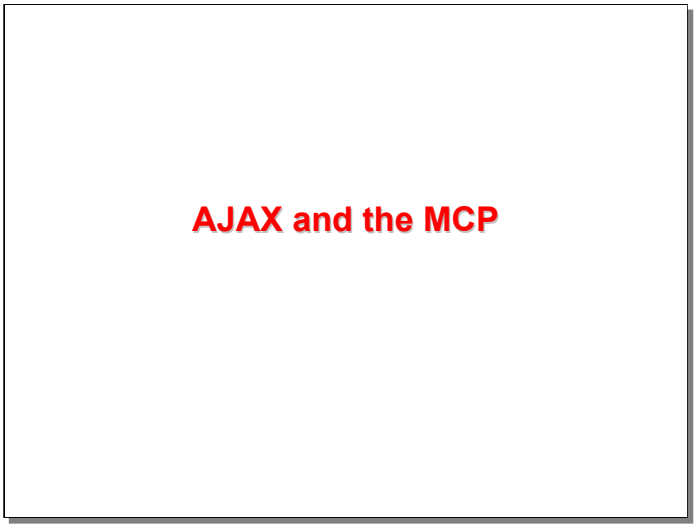
A potentially serious problem with AJAX concerns accessibility for disadvantaged users. Asynchronous, partial-page updates work well for most of us, but can be hard to detect and difficult to understand for some others. This problem is especially crucial for the sight-impaired, who often use aural browsers that can only traverse the document linearly.



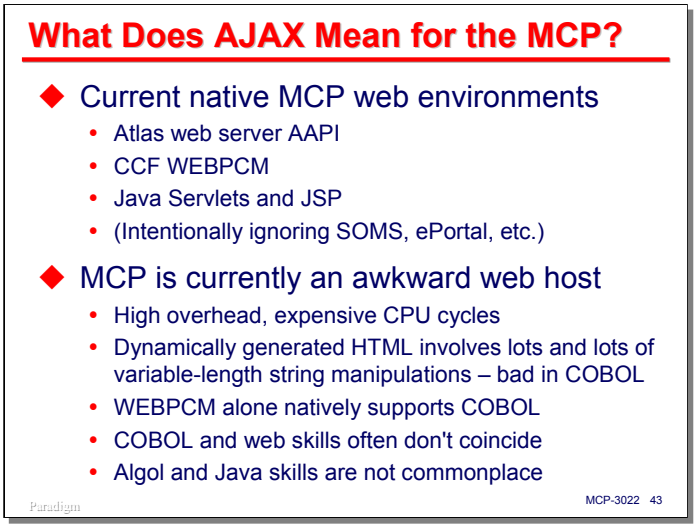
Another significant issue is that Microsoft has gone its own way in several areas, and the behavior of IE with technologies that support AJAX differs markedly from both the W3C standards and other browser implementations.

- I have already mentioned the difference in the way that the **`XMLHttpRequest`** is instantiated. This is actually fairly easy to code around, and is fixed in IE 7.
- IE currently has the weakest support of all the modern browsers for the CSS and DOM standards. I understand that there are significant improvements in IE 7, but even that version is not as compliant as it needs to be.
- One of the big problems I have encountered in implementing AJAX solutions with IE is that the browser does not consistently update the DOM when objects are dynamically created. For example, when adding controls to a **`<form>`** element, the names of the control elements are not added to the JavaScript namespace, and cannot be used as names to reference the controls as properties of the form object, as those names can when the form controls are defined in HTML. Instead, you must reference these created objects by their "id" attribute, or through the **`elements[]`** collection property of the form object.
- As mentioned previously, IE supports an entirely different event model from the one specified by the Level 2 DOM, and this incompatibility is carried over into IE 7.

Finally, something to consider when planning an AJAX-based design is that AJAX is not particularly easy to code or debug. It requires considerable JavaScript skills, along with significant knowledge of CSS and the DOM API. All of these are large subjects and have a considerable learning curve associated with them. There are several frameworks available now that help mitigate this issue, and the IDEs which are coming to market will help this situation even more.



Thus far, we have been talking about AJAX techniques generically, without reference to the type of server with which they communicate. This is appropriate, since web interfaces *should be* server-neutral. AJAX has some characteristics that are of interest to the MCP environment, however, and I would like to spend the last portion of this talk exploring these.



MCP systems have supported a native web server for almost ten years now. There are currently three application environments which support MCP applications providing web services:

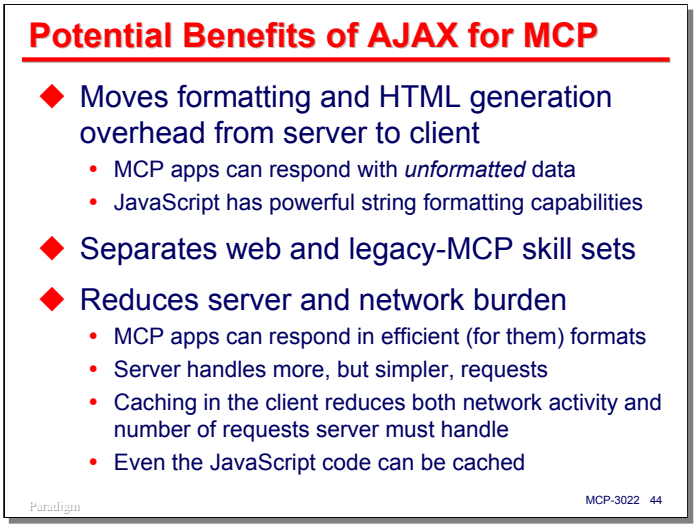
- The native Atlas web server AAPI, which requires connection libraries, and is therefore only used with Algol and NEWP applications.
- The WEBPCM component of the Custom Connect Facility (CCF), which interfaces Atlas to COMS, and allows COBOL and Algol programs to provide web services.
- Java Servlets and Java Server Pages (JSP), which requires the use of Java.

For the purpose of this discussion, I am going to ignore other products where HTTP is not supported within the MCP environment, such as SOMS and ePortal, although AJAX techniques may be appropriate for those products as well.

Given these three native environments for supporting web services, the MCP is currently somewhat of an awkward web host. Establishing TCP/IP connections carries a fairly high overhead, and MCP CPU cycles are quite expensive compared to commodity systems.

Most applications with a web interface must dynamically generate HTML responses to requests. This requires lots and lots of variable-length string manipulations and the construction of buffers of HTML text. Algol is quite good at this sort of string manipulation, but COBOL is not, so dynamic generation of HTML in COBOL is very expensive. Since most MCP applications are written in COBOL, this is a further problem, especially since the WEBPCM is the only environment that conveniently supports COBOL applications.

Another problem is that the skill sets for legacy application programmers and web page programmers often do not coincide – most COBOL programmers do not have strong HTML skills, let alone JavaScript, CSS, and the DOM, and most web-oriented programmers would probably rather be boiled in oil than program in COBOL. Similarly, Algol and Java skills are not that commonplace in most MCP shops.



AJAX has the capability to mitigate at least some of the impact of these issues for MCP applications. First, using AJAX, it is possible to move most, and in some cases *all* of the web page formatting and HTML generation from the server to the client. Using JavaScript and DOM manipulation, the client can construct the user interface locally. The role of MCP applications can be reduced to providing pure web services – responding with raw, unformatted data to AJAX-style requests from the browser. JavaScript has very powerful string parsing and formatting capabilities, so it is well suited to taking unformatted responses, parsing them, and inserting them into the user interface using either HTML injection or direct DOM manipulation.

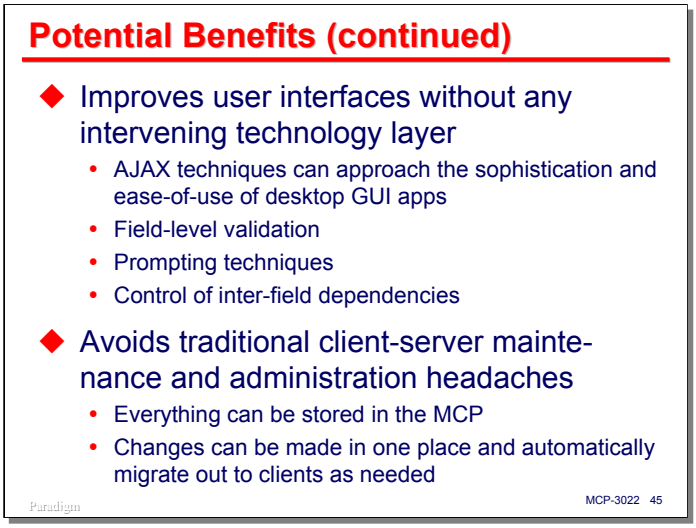
The second thing AJAX can do is nicely separate the skill sets involved. Web designers and programmers can concentrate on the user interface and MCP programmers can concentrate on business rules and the data base. The interface between them when using AJAX is fairly narrow. Writing WEBPCM applications for the MCP is a lot easier if you don't have to worry about HTML formatting and all of that variable-length string concatenation stuff. MCP COBOL programmers can process requests and prepare responses using much simpler data formats – fixed-length fields, tab-delimited fields, or comma-delimited fields. XML and JSON formatting is possible as well, but formatting those types of responses has many of the same problems as formatting HTML responses.

Third, using AJAX can reduce the burden of web services on the sever and network. Since a lot of HTML text does not need to be assembled on the server, CPU and memory resources are lower. Since all of that HTML does not need to be transmitted over the network to the client, less network capacity is required. As mentioned earlier, AJAX techniques generally send more requests to servers than traditional web application designs do, but those requests are usually much simpler, and preparation of the responses on the server can be much easier and more efficient.

Caching in the client can help, too. If GET requests are used to repeatedly fetch the same data (e.g., to load the elements of a pull-down list), the server's response to the first request will be cached, and subsequent identical requests (those having the same URL and query string parameters) will retrieve the response from the browser's cache and not even send a request to the server. Some care needs to be taken that cached responses expire appropriately, but this is easily controlled by the server-side application.

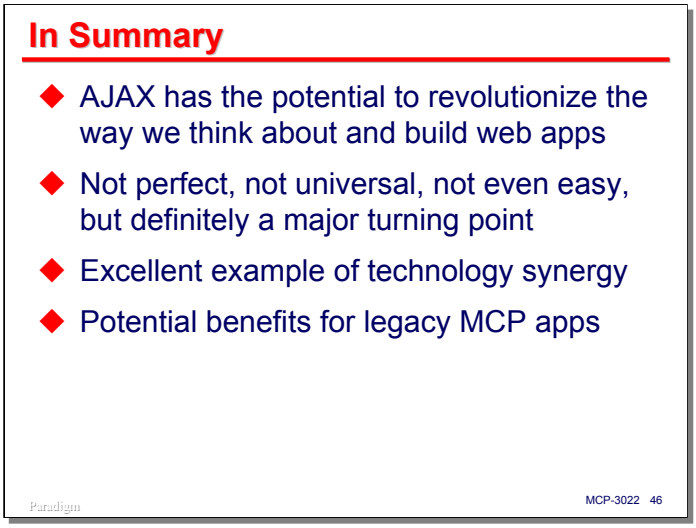
Since AJAX implementations generally involve fairly large amounts of JavaScript code, transmission of this code to the client is a potential source of additional overhead. Script code, however, can be stored in separate files and referenced from web pages, much like a COBOL **`COPY`** statement works. When script code is stored separately from the web page, that code can be cached as well.

The net result of employing AJAX can be to move a significant amount of the application workload away from the server and out to the clients where there are usually computing resources to spare.



Another area where AJAX can add value is by improving the quality of the user interface. Applications built with AJAX techniques can approach the power and sophistication of desktop GUI applications. Further, AJAX can do this without having to acquire and deploy any intervening technology layer. Rich and highly interactive user interfaces can be supported with a simple two-tier design. Several techniques, such as field-level validation, prompting, and controlling inter-field dependencies, that were heretofore extremely difficult to implement with traditional web approaches, are possible with AJAX.

Finally, AJAX can help return the overhead for client-server application maintenance and administration closer to the levels we enjoyed with green-screen interfaces. Everything can be stored on the MCP – data bases, MCP applications, the web server, and the JavaScript code that drives the client-side user interface. Changes to web-based applications can be made in one place, the MCP environment, and will automatically migrate out to clients using standard HTTP cache management mechanisms. AJAX brings us a step closer to having client/server implementations without having a heavy administration burden on the client.



In summary, AJAX has the potential to revolutionize the way we think about and build web applications – not just for the MCP, but for all web-enabled environments. Popular interest in AJAX is at this point little more than 18 months old, but we are already seeing it have a significant impact not only on specific web applications, but what we are able to conceive as possible with web technologies.

AJAX is certainly not a perfect solution, and as I took pains to mention earlier, certainly not a universal one, but it is definitely a major turning point in further development of the World Wide Web. AJAX applications at present are not particularly easy to prepare, but over the next few years we should begin to see some IDEs and advanced frameworks that will address that problem.

One of the things that pleases me most about AJAX is the synergy that comes from its combination of underlying technologies. Somehow AJAX techniques produce something that is more than the sum of their parts, and it is a relatively simple thing – the **`XMLHttpRequest`** object – that really pulled the pieces together and produced that synergistic result.

The interesting thing to me, and the reason why I decided to do this talk, is the potential for AJAX to make the MCP a much better web host than it is today. Moving the details of managing the user interface completely out to the client frees MCP systems to do what they do best – process transactions and manage data bases.

References

- "Ajax: A New Approach to Web Applications"
  - Jesse James Garret, 18 February 2005
  - http://www.adaptivepath.com/publications/essays/archives/000385.php
- "Raw Thought: A Brief History of Ajax"
  - http://www.aaronsw.com/weblog/ajaxhistory
- "The XMLHttpRequest Object" [draft specification]
  - http://www.w3.org/TR/XMLHttpRequest/
- "Introducing JSON"
  - http://www.json.org/
  - See also IETF RFC 4627 at http://www.ietf.org
- *JavaScript: The Definitive Guide, **5th Edition***
  - By Daniel Flanagan, O'Reilly, 2006, ISBN 0-596-10199-6
- This presentation and examples
  - http://www.digm.com/UNITE/2006

END

AJAX and the MCP

2006 UNITE Conference

Session MCP-3022