The characteristics of Ajax applications

The underlying technologies behind classic Web applications (HTML) are pretty simple and straightforward. This simplicity, however, comes with a certain cost. Classic web pages have very little intelligence and lack of dynamic and interactive behaviors.

Ajax, Asynchronous JavaScript and XML, changes the landscape. Ajax is not a specific product or technology. Instead, it refers to a set of technologies and techniques that allow web pages be interactive like desktop applications. Different from classic HTML web applications, Ajax applications have the following characteristics.

The Web Page as Application

Ajax blurs the boundary between web pages and applications. In classic web applications, a web page is an HTML document that can be rendered by a browser for information display purpose. It has limited or often zero intelligence on its own.

In an Ajax application, the HTML page the server sends to the browser includes code that allows the page to be a lot “smarter”. This code runs in the background acting as the “brain” while the HTML document is rendered in the browser window. The code can detect events such as key strokes or mouse clicks and perform actions responding to these events, without making a round trip to the server.

Through Ajax, a web page feels like a desktop application. It responds fast, almost immediately to user actions, without full page refresh. It can further continuously update the page by asynchronously fetching data from the server in the background, achieving desktop application experience.

Servers are For Data, Not Pages

Ajax changes the role of web pages from being merely HTML documents into “applications” that have both HTML markup as well as code. Similarly, Ajax changes the role of “server” from merely serving HTML pages to serving data as well.

In classic web applications, web servers serve HTML web pages. Some of the pages are static while others are generated dynamically by server side logic. In particular, when the application has dynamic data, the server must convert such data into HTML markup and send them to the browser for display as HTML pages. In this regard, the server is merely serving “screen images” to the client side while the client side browser is merely a screen images rendering engine.

In Ajax web applications, servers do not need to convert data into HTML markup. Server can send data to the client side directly. The client side code will process the data inside the browser and dynamically update the HTML display accordingly. This eliminates significant overhead on the server side for generating HTML pages from dynamic data, leverages the client-side processing powers and delivers better performance and scalability, as shown in Figure X.
In fact, having the server serving data instead of generating and serving HTML pages is the right way to architect Ajax applications. Putting data on the client side gives the client side a lot more flexibility, avoids unnecessary network request/responses and improves performance. Such architecture can further enable offline capable applications.

**Dynamic and Continuous User Experiences**

An important characteristic of Ajax is in its first letter “A” – a user experience that is “asynchronous”. “Asynchronous” means that users continue to interact with the application while the browser is communicating with the server. No more “click, wait and page refresh”, the Ajax user experience is dynamic and continuous.

Classic web applications deliver a “click, wait and page refresh” user experience. Because the Web was originally designed for browsing HTML documents, a Web browser responds to user actions by discarding the current HTML page and sending an HTTP request back to the Web server. After performing some processing, the server returns a new HTML page to the browser, which then displays the new page. The cycle of “browser requests, server responds” is synchronous, meaning that it happens in real-time rather than “in the background” so that the user must wait and cannot perform other tasks. Figure X illustrates the traditional HTML “click-wait-refresh” paradigm.

In AJAX-based applications, partial screen updates replace HTML’s “click-wait-refresh” and asynchronous communication replaces synchronous request/response. This model decouples user interaction from server interaction, while updating only those user interface elements that contain new information. This more efficient application architecture eliminates the wait so users can keep working, and also makes possible non-linear workflows. Likewise, it reduces network bandwidth consumption and server load for improved performance and scalability. Figure 2 illustrates the AJAX asynchronous/partial update paradigm.
Chapter 1: AJAX continuous user experience

The Ajax Software Platform: Requires Real Design

Ajax development and Maintenance Challenges

Ajax has certainly raised awareness of the high potential of web applications. At the same time, Ajax has also brought the usage of JavaScript and DHTML to a much deeper and broader level in application development. There are significant benefits of Ajax, but there are also significant challenges.

The biggest challenge of using Ajax is its scripting-based approach. Ajax requires developers to write a lot of JavaScript code. JavaScript code is hard to develop, debug, test and maintain. JavaScript/DHTML is not standardized and there are incompatibilities between different browsers, different browser versions, and different operating systems. There is a severe lack of tools for developing, debugging and testing JavaScript/DHTML code. There is also a lack of a robust component marketplace. By its definition, a scripting language emphasizes “quick and dirty” at the cost of code maintainability. In contrast, a real programming language like Java and C++ emphasizes formality and strictness. For example, unlike Java or C++, JavaScript is not strongly typed. Programming errors can only be uncovered at runtime. JavaScript object properties and methods can be easily (and arbitrarily) modified regardless of a predefined interface – none of which is allowed in Java or C++.

Secondly, Ajax itself does not provide rich user interface or incremental update capability. Developer must code such functionality using JavaScript and DHTML. There are various JavaScript libraries available that alleviate this issue to some degree, but they still require developers to write JavaScript and thus do not solve the fundamental challenge.

Thirdly, Ajax development model tends to break the “markup + logic” design pattern. “Markup + logic” is a well established design pattern that separates user interface from application logic. User interfaces are described as markup documents and application logic are written separately in a procedure language to control the behavior. In order to create rich user interface, a lot of Ajax developers embed significant amount of JavaScript inside their web pages. Mixing JavaScript with presentation breaks the clear separation and makes application even more difficult to develop and maintain.

In a typical application life cycle, the most expensive part is not the initial development but the ongoing maintenance. How to overcome the above challenges and thus enable “manageable and maintainable Ajax” should be a topic of high priority and importance for any significant Ajax development project.

Ajax Application Architecture

Given the challenges associated with Ajax, it is particularly important to architect an Ajax application properly. Otherwise the result can be either lackluster performance or code maintenance nightmare, or even both.

There are two items impact Ajax application architecture significantly: the choice of an Ajax engine and client-side application logic implementation.

Ajax Engine

From the point of view of software architecture, the significant difference between an Ajax application and a classic HTML web application is the introduction of a client-side engine. This engine, which runs inside the Web browser, acts as an intermediary between the application’s UI and the server. User activity leads to calls to the client-side engine instead of a page request to the server. Likewise, data transfer takes place between the server to the client-side engine, rather than directly to the Web browser.
Ajax engine is the key to the AJAX application model. Without it, every event generated by user activity must go back to the server for processing. Figure 3 illustrates this latter case, while Figure 4 illustrates the more efficient AJAX model.

![Figure 2: Classic web application architecture.](image1)

![Figure 3: AJAX Architecture.](image2)

There are many different ways to implement the client-side Ajax engine. One approach is to write it from scratch based on the application need. Another approach is to use an Ajax toolkit that is available in the market today. There are many Ajax toolkits today, a lot of which are open source. Some toolkits are communication libraries, some of them are rich user interface components and some of them provide both. Choosing the right toolkit would significantly lower application development and maintenance challenge.

Application logic partition

Regardless of the client-side Ajax engine is implemented, how to partition application logic directly impacts application performance and maintainability. "Application logic partition" refers to the amount of application logic that runs on the client side versus the amount of logic that runs on the server side. On the one side, putting more logic on the client side would deliver better application performance. However, client-side logic can easily result in a lot of hard to maintain JavaScript code. For example, GoogleMap is a relatively simple application and has limited functionality, but it still has more than a hundred Kilobytes of JavaScript logic on the client side (after obfuscation and compression). On the other side, putting more logic on the client side can potentially create application maintenance problem that is expensive and hard to scale to large development teams.

What kind of logic should be put on the client side, how much logic and how the logic should be implemented? These are key questions developers must evaluate carefully in order to build manageable and maintainable applications.

Ajax development model offers a lot of flexibility in application logic partition as shown in Figure XX. On the left side of the figure, most of the application logic as well as data are on the client side. This is a client-centric model that resembles closely to the typical desktop application model. On the right side of the figure, all application logic resides on the server side. This is a server-centric model that is very similar to the classic HTML web application model except for the "RIA" Ajax engine on the client side. Obviously, developers can decide to partition their application anywhere between these two extreme cases.

What is worthy of pointing out is that the server-centric model is fully capable of delivering a rich user experience such as rich UI and asynchronous partial updates. The reason is the introduction of the RIA Ajax engine. In this model, the number of round trips is not necessarily reduced comparing with the classic HTML application model, but the amount of data to be transferred is much smaller. The asynchronous nature of the Ajax engine still enables the "continuous" user experience. The popular JavaServer Faces (JSF) model is such a server-centric model that encourages all processing happening on the server side. The benefits of this model include not only much enhanced user experience than a classic HTML application by the introduction of a client-side Ajax engine, but also good application maintainability. Because all logic stays on the server side, it is much easier develop and maintain application code on the server side than dealing with JavaScript code on the client side.

In comparison, a server-centric model will not be able to deliver the same performance and availability as a client-centric model. In client-centric models, there are significant application logic runs on the client side. As a result, most of user interactions can be
processed locally without incurring a round trip to the server. Further, the application can be more “resistant” to sporadic network connectivity drop off. Application availability is improved because of this reduced network dependency. The draw back of such a client-centric model is the challenges associated with developing, sharing and maintaining the amount of client side JavaScript code.

Some of the Ajax toolkits provide frameworks that would facilitate the appropriate partition of application logic between the client side and server side. For example, JSF is such a framework that encourages all logic on the server side.

Figure xx: Ajax model offers significant flexibility in application logic partition
Alternatives to Ajax

Ajax is certainly a viable way of developing richer, more interactive web applications. These kinds of applications are typically referred to as "Rich Internet Application (RIA)". RIA is a term that describes the next generation web applications that combines the performance, functionality of desktop software with the universal deployment advantages of the web.

Though it is still evolving rapidly, today’s RIA marketplace is already rich in choice, and IT is challenged to match technology options with business goals. But while there are a variety of approaches and products available for building and deploying RIAs, they nearly all fall into one of only two basic categories:

- Object-oriented programming (OOP) based approaches: Java and .NET
- Scripting based approaches: Ajax and Flash.

The comparative strengths and weaknesses of the different RIA approaches center largely on the programming model and application execution environment they employ. The programming model impacts development and maintenance effort, availability and cost of developer skills, availability of industry and development community support, and such. The execution environment significantly impacts not only application performance, functionality, and reliability; but also the deployment model.

**Strength and Weakness of RIA Approaches**

In general, OOP approaches have the advantages of object oriented programming and are better suited for enterprise-class applications. Scripting based approaches have the advantages of scripting, which is better suited for getting simple tasks done quickly but not necessarily good for application maintenance.

The following table summarizes advantages and disadvantages of the four approaches:

<table>
<thead>
<tr>
<th></th>
<th>Strengths</th>
<th>Weaknesses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ajax (JS)</td>
<td>Most compatible with existing HTML infrastructure and content; Built-in support in most browsers - easy to try without the need for additional software;</td>
<td>JavaScript/DHTML code is difficult to maintain; JavaScript/DHTML code is difficult to develop; Not designed for team development; Performance and functionality limitations</td>
</tr>
<tr>
<td>Java</td>
<td>Large and broad industry support for Java; Large Java developer community; Widely adopted in the enterprise; Robust performance, scalability and reliability; Designed for team development; Maintainable and manageable code</td>
<td>Requires higher programming skill set than scripting; Requires a Java Virtual Machine to run the application</td>
</tr>
<tr>
<td>Flash</td>
<td>Supports rich UI features like animation and video; Flash engine is small and widely available</td>
<td>ActionScript code is difficult to maintain; Not designed for team development; Performance and functionality</td>
</tr>
</tbody>
</table>
Large Flash designer community

<table>
<thead>
<tr>
<th>.NET</th>
<th>Supported by Microsoft</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Robust performance, scalability and reliability</td>
</tr>
<tr>
<td></td>
<td>Robust OOP</td>
</tr>
<tr>
<td></td>
<td>Designed for team development</td>
</tr>
<tr>
<td></td>
<td>Maintainable and manageable code</td>
</tr>
</tbody>
</table>

Limitations:

- Flash designers are not developers – lack of enterprise developer mindshare
- .NET Supported by Microsoft only; Requires a .NET virtual machines in order to run applications
- Requires higher skill sets than scripting

One Size Doesn’t Fit All

In typical enterprise environments, there are always different applications profiles. On one side, some applications are large scale and business critical, whose performance and reliability are of paramount importance. Such applications are typically written by a large development team. Further, maintainability of such applications weighs more than the initial development. On the other side, some applications are small and not critical that are written by one to two developers. Maintainability of such applications is not crucial. And there are many applications that would fall between these two.

It is important to point out that Ajax is not the solution to all applications. In general, large scale business applications that require team development and long term maintenance are much better served by OOP based approaches like Java and .NET. Scripting based approaches are more suited for applications where the tasks are simpler, development team is smaller and maintainability is a less concern.

The following table shows how different approaches fit different application profiles and developer skill sets:

<table>
<thead>
<tr>
<th>Suitable Application Profile</th>
<th>Developer Fit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ajax (JS)</td>
<td></td>
</tr>
<tr>
<td>HTML-centric or web content oriented applications</td>
<td>JavaScript developers (CSS, DHTML, JavaScript, cross browser skills)</td>
</tr>
<tr>
<td>Casual usage pattern applications</td>
<td></td>
</tr>
<tr>
<td>Fast application loading and startup is important</td>
<td></td>
</tr>
<tr>
<td>Limited client side logic (lower maintenance requirement)</td>
<td></td>
</tr>
<tr>
<td>Java</td>
<td></td>
</tr>
<tr>
<td>Transaction oriented applications</td>
<td></td>
</tr>
<tr>
<td>Responsive user interaction and runtime performance are important</td>
<td>Java developers</td>
</tr>
<tr>
<td>Expert usage pattern applications (frequent usage, long duration usage)</td>
<td></td>
</tr>
<tr>
<td>Performance, scalability and reliability can not be sacrificed. Applications that must be maintained for many years.</td>
<td></td>
</tr>
<tr>
<td>Flash</td>
<td></td>
</tr>
<tr>
<td>Casual usage pattern</td>
<td></td>
</tr>
<tr>
<td>Limited client side logic (lower maintenance requirement)</td>
<td>Flash developers</td>
</tr>
</tbody>
</table>
Rich media oriented applications

| .NET (suitable application profiles are similar to Java) | .NET developers (C#, XAML, etc.) |

Given the diverse application profiles and developer skill sets within a given enterprise environment, and each RIA approach has its own strength and weakness, the inevitable conclusion is that “one size does not fit all”. Not one particular RIA approach (Java, Ajax, .NET or Flash) will exclusively own the enterprise environment. There are some applications better served by a scripting based approach like Ajax, some are better served by Flash, some are better served by .NET while others are best served by a Java based RIA solution. All these four technologies will co-exist in any significant enterprise environment.

Alternative Products to Ajax

There are quite a few alternative products available today for building rich internet applications. Each of them fit into one of the approaches mentioned above. Some of the solutions come with tooling that can simplify development and maintenance. The following table shows a list of solutions available today:

<table>
<thead>
<tr>
<th>Runtime Solutions</th>
<th>Tooling</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java</td>
<td></td>
</tr>
<tr>
<td>Java</td>
<td>Nexaweb Platform</td>
</tr>
<tr>
<td>.NET</td>
<td>XAML (Microsoft)</td>
</tr>
<tr>
<td>.NET</td>
<td>Thinlet</td>
</tr>
<tr>
<td>AJAX</td>
<td>Open source: Dojo, Apache XAP, Apache Kabuki, Rico, DWR</td>
</tr>
<tr>
<td>Flash</td>
<td>Adobe Flex</td>
</tr>
<tr>
<td>Flash</td>
<td>Laszlo</td>
</tr>
</tbody>
</table>

Alternative Products to Ajax

There are quite a few alternative products available today for building rich internet applications. Each of them fit into one of the approaches mentioned above. Some of the solutions come with tooling that can simplify development and maintenance. The following table shows a list of solutions available today:

<table>
<thead>
<tr>
<th>Runtime Solutions</th>
<th>Tooling</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java</td>
<td></td>
</tr>
<tr>
<td>Java</td>
<td>Nexaweb Platform</td>
</tr>
<tr>
<td>.NET</td>
<td>XAML (Microsoft)</td>
</tr>
<tr>
<td>.NET</td>
<td>Thinlet</td>
</tr>
<tr>
<td>AJAX</td>
<td>Open source: Dojo, Apache XAP, Apache Kabuki, Rico, DWR</td>
</tr>
<tr>
<td>Flash</td>
<td>Adobe Flex</td>
</tr>
<tr>
<td>Flash</td>
<td>Laszlo</td>
</tr>
</tbody>
</table>

Though these are different RIA solutions and they are all based on different underlying technology platforms, the general RIA programming model is actually converging into a common declarative UI development model.

This general RIA programming model is centered on the usage of an XML-based UI markup language to create a rich user interface. The XML-based UI markup provides a much higher level of abstraction than HTML for building rich user interfaces. XML UI frees programmers to focus on the application’s core logic and significantly reduces the need for scripting (for more detailed benefit description, see XML Journal: XML for Client-side Computing, March 10, 2004).

Below we will show examples from scripting-based approaches (Adobe Flex and Laszlo Systems) as well as an OOP-based approach (Nexaweb). All solutions are zero-install and thus can be run inside any popular web browser today without any installation or software download. On the client side, Flex require the presence of Flash engine (Flash 8 and above). Laszlo requires the presence of Flash engine (Flash 6 and above) while Nexaweb requires the presence of a JVM (JDK 1.1 and above).

Flex:

Flex is Adobe’s product for delivering Rich Internet Applications. Flex is a Flash-based application. Developers would create Flex application using an XML-based UI markup language called “MXML” (Macromedia markup Language) and write application logic using ActionScript. The Flex Presentation Server will compile the MXML files into SWF (Flash movie format), deliver the compiled SWF to the client side Flash engine and run the application.

```xml
<?xml version="1.0" encoding="utf-8"?>
  <mx:Form>
    <mx:FormItem label="UserId" required="true">
      <mx:TextInput id="userId" width="150"/>
    </mx:FormItem>
    <mx:FormItem label="Password" required="true">
      <mx:TextInput id="password" width="150"/>
    </mx:FormItem>
    <mx:HBox horizontalGap="30">
      <mx:Button label="Logon"/>
      <mx:Button label="Cancel" click="this.deletePopUp()"/>
    </mx:HBox>
  </mx:Form>
</mx:TitleWindow>
```
Laszlo:
Laszlo is also a Flash-based product that is fairly similar to Flex technically, though it is developed by Laszlo Systems and currently offered as an open source product.

```xml
<canvas height="450">
  <window x="10" y="10" width="300" height="200"
    title="my window"
    resizable="true" closable="true">
    <button x="10" y="100">Hello, World</button>
  </window>
</canvas>
```

Figure xx: Laszlo Code Example UI

Nexaweb:
Nexaweb is a Java-based RIA product. Developers would use an XML-based UI markup to create rich user interface, and build client-side business logic by writing client-side Java beans (called "Managed Client Object"), which are standard Java program objects. Nexaweb Client runtime would dynamically render the XML UI markup to present a rich user interface, and dynamically download client-side Java objects to the client side for execution in a "on demand" fashion.

```xml
<xal xmlns="http://www.openxal.org/xal">
  <window title="New Window">
    <boxlayout orientation="vertical" pack="start" align="stretch"/>
    <tree>
      <row expanded="true">
        <cell text="Tree Item 1"/>
      </row>
      <row>
        <cell text="Sub Tree Item 1"/>
      </row>
      <row>
        <cell text="Sub Tree Item 2"/>
      </row>
      <row expanded="true">
        <cell text="Tree Item 2"/>
        <row>
          <cell text="Sub Tree Item 3"/>
        </row>
      </row>
    </tree>
    <button text="OK"/>
  </window>
</xal>
```

Figure xx: Nexaweb Code Example UI
Example xx: Nexaweb Code Example

Figure XX: Nexaweb code example UI

As shown from the above examples, though Nexaweb uses Java and Laszlo/Flex uses Flash, RIA UI development is conceptually identical between different RIA solutions. The XML UI abstraction significantly lowers the complexity and cost for building rich user interfaces, of which Ajax development can certainly learn and benefit from as well.

Cross Technology RIA Solutions
All RIA solutions are fundamentally constrained by its underlying technology: Ajax, Flash, Java or .NET. If a developer picks Flex to develop his RIA, he has to live with the pros as well cons of Flash. Likewise, if a developer picks an Ajax toolkit to develop his RIA, the developer has to live with the various challenges associated with DHTML and JavaScript. As we mentioned earlier, among the four technologies, each has its strength and weakness.

There is some very interesting development in the RIA marketplace recently: cross-technology RIA solutions. Both Laszlo Systems and Nexaweb announced that their products are supporting more than one technology. The same application can be delivered and rendered on different technology platforms. Laszlo supports both Flash and Ajax (DHTML). Nexaweb support Java and Ajax. This is an extremely interesting development. With this development, developers do not have to fight the “religious war” of JavaScript vs. Java, Java vs. .NET or .NET vs. Flash. Such development accommodates not only different developer skill sets, but also open the door of combining the benefits of scripting based approaches with those of OOP-based approaches, delivering optimal results.

Figure XX is an architecture diagram shows how Ajax and Java can co-exist within the same programming model for the same application. This is primarily accomplished by using an XML abstraction layer that can be processed and rendered by either Ajax or Java on the client side.

The following is a sample application written using such an XML abstraction layer. This sample application is an RSS reader that would read RSS feeds from Yahoo and display all the feeds in a table. The code is shown in "Example 5" and the UI screen display is shown in Figure 6.

```xml
<xal xmlns="http://www.openxal.org/xal">
  <data:documentDataSource id="yahoo"
    source="http://rss.news.yahoo.com/rss/topstories"
    xmlns:xs="http://www.openxal.org/data" />
  <rootPane>
    <borderLayout/>
    <table borderPosition="center">
      <column>
        <header text="Title"/>
      </column>
      <column>
        <header text="URL"/>
      </column>
      <column>
        <header text="pub Date"/>
      </column>
      <data:iterator xmlns:xs="http://www.openxal.org/data"
        dataSource="yahoo" type="ONE_WAY" name="newsIterator" select="//item">
        <row>
          <textView>{"title"}</textView>
          <link text="{"link"}" alignHorizontal="left"/>
          <cell text="{"pubDate"}"/>
        </row>
      </data:iterator>
    </table>
  </rootPane>
</xal>

Example 5: A Simple RSS Feed Application

<table>
<thead>
<tr>
<th>Title</th>
<th>URL</th>
<th>pub Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>European Fed-Res Bank Rate Hike Caution</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 6: Example 5 Screen Display
Real-World Ajax: Some Classic Examples

Ever since the phrase "Ajax" was coined, Ajax has been spreading like wildfire in developer communities. There are lots of applications are developed using Ajax, in fact, a lot of applications were developed using Ajax long time before "Ajax" was phrased as "Ajax". In this section, we will go over some Ajax application examples to give the reader some sense of what Ajax is capable of and what applications have been built using Ajax so far.

AJAX Chat Applications
Chat application represents an excellent AJAX experience that is not possible with classic web model. Chat requires asynchronous communication and can not afford "full page refresh", which is one of the reasons that many chat/IM applications showed up since Ajax became popular.

Gabbly: Live Chat for Any Website
Gabbly is a new application that embeds a chat window in any web page. As a user, you see the target website as the way it is except for an additional chat window. From the chat window, you can have real-time chat with other users.

Gabbly uses Iframe to display the target website. Then in a separate Iframe, it displays the chat window. The chat window uses XmlHttpRequest object to communicate chat messages with the server asynchronously. Figure xx shows how Gabbly works with CNN.com:

Figure xx: Gabbly running on CNN.com As an Example

Gabbly is a great web 2.0 application that can add significant value to various websites. For example, it would enable all CNN.com readers to be able to interact with each other...
in real-time. Such real-time interaction between random web visitors turns the web from a static, passive media into an interactive social environment.

AJAX IM, the AJAX instant messenger

**AJAX IM** is an AJAX instant messaging client. It has a clean rich interface featuring multiple windows that feels like a normal desktop application.

AJAX+PHP CSS Popup Chat

**AJAX+PHP CSS Popup Chat** is another AJAX chat application that implements one-to-one chat using popup windows. The application is written in PHP, MySQL and is available for download free under GPL license.

Meebo: Connecting All Popular IM Systems

Meebo is a web-based Instant Messaging service that connects with all major IM systems such as AOL, Yahoo and MSN. A user can login using his/her account from any of these IM systems, retrieve the buddy list and chat with them.

By the end of 2005, Meebo averages about 250,000 logins per day. On December 7, twelve weeks after launch, Meebo had 236,000 successful logins, 6,534,948 messages sent and approximately 13,069,896 total messages carried.
AJAX Office Applications

Office applications are another category of applications that are not possible before without Ajax ("Ajax" being defined in a broad sense as DHTML and JavaScript). There are word processors, spreadsheet, and slide show and so on.

Ajax Word Processors: Writerly, AjaxWord

Writerly (http://www.upstartle.com) is an Ajax-based word processor, recently acquired by Google. Writerly enables online document editing from a browser, share documents instantly with authorized users, collaborate with people and store document securely online.

AjaxWord (http://www.ajaxword.com) is an open source word processor mimicking the Microsoft WORD look and feel, written using JavaScript and DHTML. It features server-side file storage that is dedicated to each user. Users would use a familiar file dialog to open or save files. When creating a new file, a user will be prompted to select from a list of templates from a modal dialog.

Different from all other web-based word processors, AjaxWord features a multiple windows interface (MDI) that enables a user to work on multiple documents at the same time.
<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Datacenter Prices</td>
<td>1993</td>
<td>2005</td>
<td>2013</td>
<td>$1100</td>
<td>$120</td>
<td>$128/megabit/month</td>
</tr>
<tr>
<td>2</td>
<td>Bandwidth:</td>
<td>$175</td>
<td>$25</td>
<td>$25/month</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Cage Space:</td>
<td>$130,000+</td>
<td>$6500+</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Mashups**

A mashup is a web application that delivers an integrated experience by seamlessly assembling content from more than one source and displaying them in an integrated user interface.

Mashup sounds similar to traditional "application integration". In fact, application integration developers have been assembling data from multiple sources and present them in one integrated application for many years. The main difference between Mashups and traditional "application integration" is where the "integration" takes place. Traditional "application integration" integrates data on the server side ("back end integration"), which would typically require server side programming skills (Java, C++, etc) and access to enterprise server-side resources. "Mashups" typically perform the integration at the browser layer without touching server-side at all ("front end integration"), which only requires JavaScript and HTML coding skills, and data access is readily available from eBay, Amazon, Google etc via the public Internet. ([http://www.programmableweb.com/](http://www.programmableweb.com/) lists close to 200 public APIs as of April 2006).

Ajax (JavaScript and DHTML) is a major reason that enabled the popularity of mashups. Without the wide popularity and support for Ajax, it would have been very difficult, if not impossible, to "integrate" data from multiple websites at the browser layer.

Similar to how blogs revolutionized online publishing, mashups are revolutionizing web development by allowing anyone to combine existing data from sources like eBay, Amazon, Google, Windows Live and Yahoo in innovative ways. The greater availability of simple and lightweight API's have made mashups relatively easy to design. They require minimal technical knowledge and thus custom mashups are sometimes created by unlikely innovators, combining available public data in new and creative ways. Today there are many mashups available on the web ([http://www.programmableweb.com/](http://www.programmableweb.com/) tracks a total over 600 mashups as of April 2006), though a lot of them are simply "cute" without significant real value. The interesting trend to watch is what mashups mean to the enterprise. Would it enable a "new" way of "enterprise application integration"? Would it enable enterprises to think...
of a "new" way of "service orientation", for example, making data not only available via SOAP but also via REST?

**Mashup Example: HousingMaps**

Almost immediately after Google published GoogleMaps, programmers started building mashup services atop Google's map infrastructure. HousingMaps (http://www.housingmaps.com) is one of the earliest and best known one.

HousingMaps is a site that pulls real-estate listings off the popular classified-ads site craigslist (http://www.craigslist.org), uses the addresses of the listed homes and apartments in a given neighborhood to figure out their latitudes and longitudes, and lets users view the properties on a Google map. Each listing is shown as a pushpin, and clicking on the pushpin pops up a small window with the price and sometimes a thumbnail image of the property. A list of the visible properties runs down the side of the screen, each linked to the original Craigslist posting. And because results are filtered into price categories, users can easily steer clear of high-rent districts. HousingMaps has no affiliation with craigslist or Google, but is merely accessing both websites via public APIs.

---

**Summary**

Ajax enabled a lot of interesting applications. This section talked about traditional "office productivity applications" such as word processors, and chat/IM. What is important but was not cited in this section are Ajax applications in the enterprise environment. There are many companies that are actively using Ajax for their business applications, dramatically enhancing user experience and productivity.

Ajax also enables a new kind of application: mashups. Mashup opens up new possibilities of how applications can be built and how web applications can be consumed. Combing data from multiple web sites, mashups bring significant additional value to users. In the enterprise environment, mashups and the traditional integration approach would go together as two complementary ways of enterprise integration.