

Techniques for Developing Highly Interactive Web Applications

John T. Kucera, Ph.D.
Founder and CTO
Asperon® Corporation
(September 26, 2003)

The software industry has seen a major revolution over the last few years as many programs have been converted from being installed on a desktop, to being run over the web. This has tremendously increased the ability to access our own information from anywhere, and has allowed us to share data and programs with colleagues in distant locations. However, despite this tremendous advance, web applications are still very expensive and time consuming to develop, and are significantly less productive to use than software that is installed on the desktop. This article discusses how to quickly and simply develop web applications that can be as productive and easy to use as desktop software.

Productivity limitations of web applications

If you have ever been frustrated by the time required to make an airline reservation online or have been a "power user" of any web-based business application software then you are familiar with the user interface productivity limitations of conventional web applications. They include:

1. The inability to show lots of data on the screen at once. For example, if you do a search that returns many results you have to click through page after page of results to find what you are looking for.
2. The inability to receive real-time notification and updates of events. You have to refresh the web page every time you want to see if something changed. For example, a page displaying stock quotes needs to be refreshed constantly to track the continuously changing prices.
3. The inability to do in-place editing of data in a table. Typically tables are read-only, and you have to click to one or more other pages to make a simple change or to add a new row.
4. The need to wait for a new page to be downloaded and rendered before any action can be accomplished. Even a small change to one piece of data in the application requires an entire page to be downloaded.

The underlying cause of these productivity problems is that the HTTP and HTML protocols that form the basis of the internet were designed primarily as a means to display relatively static information. They were not designed to run interactive applications. We have not been able to evolve to newer, more powerful client-side

technologies because of the difficulty of adopting new standards that are backwards compatible with the current technologies.

Until now these limitations have been largely tolerated because of the tremendous advantages of having software available from any browser-enabled computer. However, as more and more applications move to the web, people are beginning to realize that we have taken a large step backwards in the usability of software.

Development problems

Not only do web applications have less interactivity than their predecessors, but also they are more complex and costly to develop and maintain.

One of the main contributions to this cost is the complex set of interacting technologies that has been created to overcome some of the limitations imposed by a user interface that runs in the browser. Some of these technologies are listed in Appendix A.

These technologies are primarily designed to return (mostly) static pages in response to a client request to a server. While each individual technology was designed to simplify development, most applications require the combination of several of these technologies. The net result is that most software developers have to learn a complex set of technologies that are constantly changing, difficult to test, and that interact with each other in unpredictable ways.

Development is made even more difficult because of the proliferation of pages that result from the request-response paradigm of the web. Every action that a user takes requires navigating to a new page. This linking of actions to pages causes the number of pages to grow rapidly with the complexity of the application. Because the navigation flow is often incorporated into the pages themselves, every page needs to know how to link to almost every other page. This results in a web application whose complexity grows quadratically with the number of pages. Although it seems relatively easy to develop the first few pages, the complexity of a large application grows surprisingly quickly and is soon beyond the point of maintainability.

The proliferation of technologies and complexities of page interaction and navigation usually result in what I like to call the "Version 2 Site Problem." This problem results when an initially simple web application grows beyond maintainability because the original application was not designed for the explosion of pages that inevitably result. Eventually, the application has to be entirely rewritten. This happens frequently because of the perceived simplicity of developing the first few pages of the application. Many companies that face the Version 2 Site Problem have to spend millions of dollars and many man-years of effort to completely rebuild their application.

Another difficulty in developing web applications is the problem with browser compatibility and testing. Despite the best attempts by the Worldwide Web Consortium, the HTML language has not been truly standardized. Each browser vendor implements its own implementation of "street HTML". This results in features that do not work on all browsers. Even those features of HTML that are commonly supported by different browser vendors may function slightly differently from

browser to browser. Further complicating this picture is that most web applications rely on some form of client-side programming, such as JavaScript, to implement even the simplest of interactive functionality. These client-side interpreters are infamous for their incompatibilities, even when the language they are running is well defined.

The net result is that web applications have to be tested on every different browser that they might be used in. Even different versions of the same browser may interpret HTML or one of the client-side programming languages differently. This results in a huge proliferation of browser variants in which the resulting application must be tested.

The interactive client

Many of the problems with conventional web applications can be solved by using a client-side program that replaces or extends the functionality of the browser's HTML rendering engine. These types of programs, called interactive clients or "rich clients", allow the user to break out of the request-response model of interacting with the server. They allow users to use the application in a manner that is similar to a desktop application, and hide the request-response model of communication with the server.

With interactive clients, end users can see tables or trees that have thousands of rows of data, and can scroll through that data simply by dragging a scrollbar. The data can typically be edited right on the screen in which it is displayed without having to go to a different page made specifically for data entry. In addition, interactive clients allow users to perform actions using menu items or buttons – in many cases without requiring a screen redraw. In interactive-client applications the data that a user enters is typically validated immediately as it is entered in each text field. The user does not have to fill out an entire page of data and then hit a "SUBMIT" button to see if the data is invalid. Some interactive clients can even support real-time updates where the display can be updated immediately as events occur on the server. This makes it possible to create dashboard-like applications for monitoring business conditions (Business Activity Monitoring or "BAM"), monitoring a call center, or monitoring an auction where the prices change in real time as bids are placed.

Not only do interactive clients provide end users with a much more productive user interface, but they also make applications easier to develop and maintain. This is because the number of screens can be reduced, and because the user interaction model does not have to mix actions with navigation.

The decoupling of actions from navigation in interactive clients results in applications with far fewer screens than a traditional page-based web application. In addition, these applications do not need to embed navigation inside each page. These two factors significantly reduce the explosion of pages and interdependencies that results when more traditional web applications grow to be large.

Interactive client user interfaces can also be much easier to maintain because they can more completely take advantage of the Model-View-Controller (MVC)

architecture, which separates the user interface from the business logic. While many web products claim to use the MVC architecture, they cannot get all the advantages of the MVC design because the model cannot immediately send data to the view (user interface) after the data changes. By achieving complete separation of model and view, the user interface of an interactive client can be easily modified without having to change the underlying application. This can greatly reduce the total cost of maintaining an application because the user interface usually changes more frequently than the business logic. Better separation also leads to more code reuse and a smaller overall code size, resulting in development time and cost savings.

Finally, interactive clients can take the burden of complex client interactivity away from the browser, making the client more interoperable. Considerable testing can be saved if the interactive client software is made generic and independent of the application so that when the application changes, the client side code remains the same. These generic interactive clients must be tested in all the browsers, but once tested, there is no need to test every application that is built with them. This separates the responsibility of the interactive client provider from the application developer.

Techniques for developing interactive clients

Table 1 lists some of the considerations for developing an interactive client. Interactive clients can be implemented in JavaScript, Java, Flash, ActiveX, .NET, or custom browser plug-ins.

Multithreaded Programming	<ul style="list-style-type: none"> - Fundamentally a multi-threaded, asynchronous client. - Asynchronous message receipt. - Need to avoid blocking the rendering thread while waiting for user inputs. - Thread synchronization with modal dialogs and blocking operations. - Out-of band cancel mechanisms.
UI Layout and Event Handling	<ul style="list-style-type: none"> - Complex GUI toolkit APIs requires specialized knowledge. - GUI builders can help with fixed layouts, but dynamic layout management is usually required. - IDE can make the skeletons of the event handlers, but you have to fill in most of the functionality by writing code by hand. This makes the IDE less useful for making changes once the first version has been created, because the IDE cannot interpret the parts that you wrote. - Traditional approach has developer code each screen, then applet with all screens is downloaded on startup. The code for the screens can be large, and each different application has to be tested in every possible runtime environment. - Need to watch memory usage from downloaded images if using Java. - Consider developing a screen modeling language and interpreter.
Communications	<ul style="list-style-type: none"> - HTTP tunneling: Must go through proxy servers and firewalls.

	<ul style="list-style-type: none"> - User-driven events must be forced into request-response model. - Use message bundling to reduce number of round trips (latency). - Use message compression to reduce bandwidth. - Encryption and security of communications protocol. - Limitations of HTTP 1.1 on number of simultaneous connections to same server can cause communications layer deadlocks. - Nested messages – a response may result in a new request. When to send it?
Caching	<ul style="list-style-type: none"> - Client needs cache to reduce the number of round trips. - Cache implementation and synchronization. - Client memory size limitations require “forgetting” old data. - Need to know when to outdate cached data.
Validation	<ul style="list-style-type: none"> - Use client and server side validation to reduce round trips.
Deployment	<ul style="list-style-type: none"> - Browser compatibilities. - Must use “least common denominator” client capabilities, or be prepared to install a plug-in or other client-side program. - Computer savvy of application users determines what can be installed on clients. - Diversity of clients’ operating systems and ability to control which users enter the application may limit choices for what can be done on client. - Need strategy to get updated versions of client code to end users.
Server Side Requirements	<ul style="list-style-type: none"> - Need to implement a fine-grained request handing model for possibly validating every input, or sending data to client on request. - Scalability to larger number of users requires very efficient server-side message handling. For example, avoid starting unnecessary threads to process messages. - Need server-side caches to back up client caches and keep them in synch. This forms a 3 layer caching strategy. - Should separate business logic layer from server-side processing related to the UI. This is best done by architecting an application model that puts database, business logic, and client processing in different layers on the server.

Table 1 – Considerations for developing interactive web clients.

Interactive client user interfaces are strongly event driven. They are asynchronous, require multi-threaded programming, and can result in deadlocking problems or unresponsive applications if not carefully designed. In user interface programming it is important to make sure the main system repaint thread is never blocked when a user action is taken, so that the display can continue to respond at all times. However, it is also important to limit what the user can do while an action is completing to ensure the user is always aware of the state of the system. These two requirements mean that there has to be some sort of system to prevent user input without blocking the repaint thread. This requires careful design of the threading

model on the client. Similar considerations have to be made to ensure that modal dialogs work correctly, without causing the user interface to deadlock.

The development of interactive clients is complicated by the need to adapt the event driven user interface to the relatively slow and unreliable connection provided by the internet. In the old days of user interface programming the developer could rely on the proximity of the display to simplify programming. For example, a desktop client can send every keystroke from the display into the processing engine because the display is in the same address space as the rest of the code, and the processing to send a simple keystroke is very fast.

When things are moved to the web, however, the design needs to be modified to take into account both the bandwidth and latency of the connection between the display and the data. The bandwidth limits how much data can be sent per unit time once a connection is made, and the latency determines how long it takes to send the first byte. Latencies of up to 300 milliseconds are common in intercontinental connections. This usually rules out sending every keystroke from the client to the server, because a user interface that responds with such latency appears sluggish. The requirement to be responsive means that changes that affect the user interface must be carefully delayed until they can all be sent at once. Some sort of message bundling architecture is required to ensure that messages get delivered in a timely manner, but do not wait too long when the user is expecting a response.

Internet bandwidth and client-side memory limitations dictate that the client must have some sort of data caching. Client caches must be synchronized with the server so that they are updated or invalidated when the server data is changed. These cache synchronization mechanisms must be made robust in the sense that if a connection is dropped or delayed the client and server have to re-synchronize their cache state without getting confused. An intelligent mechanism for cache clearing is essential since the developer cannot rely of having an arbitrary amount of memory on any individual client.

Bandwidth optimization also requires careful consideration of where user inputs should be validated. In some cases, such as requiring an integer within a certain range, validation can be done on the client. However, other validations, such as looking up a value in a large table of possible values, needs to be done on the server. An interactive client should be designed to support both client and server side validation, and to choose the validation mechanism that is appropriate for each situation.

Another consideration for developing interactive clients is whether the messaging mechanism should be re-entrant. Often, a client request causes the server to respond with something that requires the client to send another request. Should this re-entrant request be done immediately, or be deferred until the original response is complete? The answer is dependent on your implementation architecture, and requires careful thinking on how to handle such situations.

Interactive clients require dynamic layout management to make their user interfaces portable to different screen sizes. This makes it more difficult to use integrated development environments (IDEs) that have simple drag and drop interfaces for laying out the screens, as many of these tools produce fixed size layouts. The UI designer not only has to design for the desired screen size, but also has to plan how the user interface will change if the screen is of a different size.

In addition to layout, the interactive client designer usually has to write custom code to handle the user-generated events from the display. Many IDEs generate nice "skeletons" of code for handling events, but the actual event handling has to be implemented by hand. Then, when the user interface is changed, the IDE cannot invert the process, resulting in the need to re-implement many of the event handlers by hand whenever the user interface is changed.

One way to reduce the complexities of layout and event handling is to implement a generic mechanism for generating screens based on some display specification. For example, a browser renders a screen based on the HTML data that it downloads. Because of this it is relatively easy to make changes in the display. Implementing such a rendering mechanism for an interactive client can also help make the client more general, and reduce the testing that is required because changes to each user interface screen no longer have to be separately tested.

The final consideration that needs to be taken into account while developing interactive clients is deployment. The single most important feature of most web applications is that the end user does not need to install anything to use the application. If you develop an interactive client, you may need to require that users install some client-side software to interpret your display. This requirement, while relatively simple for an engineer, is often prohibitively difficult for non-technical end users, and can result in an application that has very little user acceptance. Also, some techniques for rendering rich user interfaces will only work for users with certain browsers or operating systems. This further limits the application's target market. Most of these deployment problems depend heavily on the nature of the application being developed and on the skills and requirements that can be imposed on end users, so consult your marketing department before finalizing on a design.

Commercially available interactive clients

Recently, some pre-packaged software for creating interactive clients has started to become available. One entry in this market is the Asperon AppProjector™. The AppProjector™ is a prefabricated interactive client front-end for web applications. It comes with a presentation server that is installed on the server, and a generic interactive client that is downloaded to the end user's web browser. The AppProjector™ allows developers to specify their entire application on the server using Java and XML without having to write any client-side code. The AppProjector™ client can be deployed in most existing browsers without a plug-in, and so can support true web-based applications.

The AppProjector™ has been in development for over four years, and comes with all of the powerful event management, caching, bandwidth optimization, and layout features described in this article. It is especially well suited for creating forms-based business applications. The AppProjector™ can be used as the basis for a new application, or can be easily integrated into an existing web application that uses Java and a web application server. A free trial version can be downloaded from Asperon's web site at <http://www.asperon.com>.

Conclusion

Interactive clients will soon be a requirement for the next generation of web applications because of the increased user productivity and reduced development cost they provide. You can either develop your own interactive client using some of the design considerations discussed in this article, or use a prefabricated interactive client like the Asperon AppProjector™. We hope this paper has been useful to help you better understand the advantages of interactive web clients and how to develop them, and encourage you to consult the following additional references on our web site (<http://www.asperon.com>) for more information:

- AppProjector™ Demos.
- AppProjector™ Technical White Paper.
- AppProejctor™ User's Guide.

Asperon® is a registered trademark of Asperon Corporation, Palo Alto, CA.

Appendix A – Technologies for developing traditional web clients.

This appendix lists some of the more common technologies that have been used to create traditional web clients. The technologies are often used in conjunction with one another, resulting in a technology stack that is complex and difficult to maintain.

- **JavaScript**

- Adds simple interactivity to client.
- Well suited to overcoming basic limitations of HTML language.
- Good for simple interactions: button rollovers, psuedo-tabs, etc.
- Does not scale to complex interaction very well because it is a scripting language.
- Different browsers interpret the language differently and so all application screens have to be tested on every supported browser.
- Does not help in generating client content from business logic unless you create an interpreter that runs in JavaScript on the client.
- Often JavaScript code is very specific to the page in which it resides because it refers to other parts of the page by name in the document object model (DOM). This makes it difficult to maintain.

- **PHP/Perl**

- Simple server-side scripting languages that help server generate static pages from dynamic content.
- Good for simple projects that have only a few pages. Not designed for maintainability of very complex “enterprise class” web applications.
- Pages have to be interpreted every time they are requested. This results in slower response times from the server.

- **CSS (Cascading Style Sheets)**

- Makes it easier to maintain a consistent set of fonts, colors, and related styles among multiple pages of a web application.
- Makes it much easier for graphics designers to change the style properties across all pages of an application.
- Does not help with content or layout of pages.

- **XSLT**

- A transformation language/engine that converts XML application data to HTML pages.
- Helps to generate a consistent set of pages across the application by centralizing the job of laying out the screens.
- Can also be used to centralize some of the other style information.
- Typically used in conjunction with CSS and JavaScript.
- Disadvantage is that developers end up writing XSLT programs to generate HTML and JavaScript code. Developers need to learn XSLT and have to generate the HTML. Cannot outsource graphics to graphics designers or traditional HTML editing tools.

- **JSP/ASP (Java Server Pages, Active Server Pages)**

- Allows developers to put dynamically generated content into pages that have been created with traditional web authoring tools.
- Content is still “static” in that a page refresh is required to change anything on the screen.

- Web authoring tools don't make it easy to consolidate display styles or layout for a set of related pages.
 - Have to code page navigation logic into pages.
 - Developers need to understand that they are really coding servlets behind the scenes, and must be familiar with servlets programming.
- **Tag Libraries**
 - Non-standard HTML "tags" that are inserted into an HTML page. These are replaced by data from the server when the page is sent to the client.
 - Some examples of this include Cold Fusion, and custom tag libraries.
 - Advantage is that tags are easy to use and intuitive.
 - Often more complex scripting is required to generate a page (e.g. when outputting multiple table rows). This requires making a scripting language out of tags.
 - Developers must learn the often non-standard tag scripting languages.
 - Tag based scripting languages have limited functionality.
 - Pages need to be interpreted and tags parsed every time a page is requested resulting in slower response times.
- **Struts**
 - A framework for building traditional, page-based web applications that helps developers separate some of the business logic from the UI by providing an architecture and a set of helper classes to handle requests.
 - Main benefit is that struts helps you handle incoming requests more cleanly, and helps with page navigation.
 - Most other technologies are focused only on generating new pages, but not on handling requests or navigation.
 - Struts is becoming popular because it starts to address the difficulties in handling requests.
 - Still requires the other traditional techniques such as JSPs to generate pages.
- **JSF (Java Server Faces)**
 - Adds dynamically generated HTML "controls" on top of the struts framework.
 - Controls can be generated for different browsers to accommodate browser incompatibilities, and can improve abstraction of user interface.
 - Next generation after Struts.