Notewise: A Case Study in Migrating Desktop Applications to the Web

by
SCOTT T. ALLEN

Department of Computer Science
2006
# Table of Contents

Introduction ................................................................................................................................. 3  
Application Domain: The Personal Knowledge Base............................................................... 3  
  Popcorn................................................................................................................................... 4  
  Notewise: An Enhancement of Popcorn ................................................................................. 5  
Ubiquitous Access ....................................................................................................................... 6  
User Interface Modifications ......................................................................................................... 7  
Implementation Challenges in Notewise .................................................................................... 9  
  Browser-Server Interaction ...................................................................................................... 11  
    AJAX..................................................................................................................................... 11  
    REST................................................................................................................................... 12  
    JSDBI................................................................................................................................. 13  
    Catalyst............................................................................................................................... 15  
Dynamically Drawing Vector Graphics ...................................................................................... 16  
Supporting Multiple Browsers .................................................................................................... 21  
Performance ............................................................................................................................... 26  
Tutorial Setup and Lazy Registration ......................................................................................... 30  
Assessment ................................................................................................................................... 31  
  Feedback from Early Users .................................................................................................... 32  
    Strengths............................................................................................................................ 32  
    Weaknesses....................................................................................................................... 33  
  Architectural Assessment ....................................................................................................... 34  
    Strengths............................................................................................................................ 34  
    Weaknesses....................................................................................................................... 35  
  Summary of Assessment ......................................................................................................... 36  
Conclusion ................................................................................................................................. 36  
Appendix A - Notewise REST API documentation ................................................................... 38  
  Status codes.......................................................................................................................... 38  
  Authentication....................................................................................................................... 38  
  Interacting with objects ....................................................................................................... 39  
Objects........................................................................................................................................ 40  
  Kernel..................................................................................................................................... 40  
  Note......................................................................................................................................... 41  
  Relationship............................................................................................................................ 42  
  Visible Kernel......................................................................................................................... 42  
Search........................................................................................................................................ 44  
Appendix B: JSDBI API ............................................................................................................... 44
Introduction
This thesis discusses the process of rewriting the personal knowledge management application Popcorn as a web application, using AJAX. It will first give a brief overview of the idea of the personal knowledge base, going on to describe Popcorn, an initial implementation of a desktop personal knowledge base. It will then go on to discuss some limitations of Popcorn, and the improvements made in those areas in Notewise, a web based version of Popcorn. It will then go on to discuss the challenges faced in building Notewise, and the solutions that were found. In particular, it will discuss a library built to draw vector lines by stretching diagonal images, and JSDBI, a library that provides object mapping for REST web APIs in javascript – both these libraries proved to be very valuable parts to the Notewise architecture. Lastly, it will take a brief look at early user feedback and reflections on the strengths and weaknesses of the Notewise architecture.

Application Domain: The Personal Knowledge Base
The personal knowledge base[5] represents a refinement of many core ideas in the area of knowledge management. A personal knowledge base allows an individual to store their own knowledge in ways that closely mimics the way they think about it in their mind. Its purpose is to provide the individual a flexible but powerful framework to collect, organize, refine, and store their own thoughts. The personal knowledge base can be broken down into three core ideas:
● **Personal** - It is intended not as a means to publish or communicate ideas to others, but rather as a personal storage and retrieval system. It is only editable and viewable by the owner, and stores concepts in the manner that they think about them.

● **Knowledge** - It stores knowledge, as opposed to information. That is, it only stores the knowledge that a user has obtained from the things they've encountered, as opposed to storing everything they encounter.

● **Base** - The human mind stores knowledge based on associations between pieces of knowledge. Any piece of knowledge can be connected to any other piece of knowledge. Thus, a personal knowledge base must allow linking between any two concepts. To make this possible, the individual must be encouraged to incorporate all of their knowledge in one, unpartitioned data store.

**Popcorn**
The prototype personal knowledge base system proposed by Davies (called Popcorn [5]) also makes a distinction between supporting the long term memory and supporting the short term memory. The system is based upon a database which mimics the long term memory - it stores all the knowledge the user has ever entered. This long term memory is accessed via a series of "views" that mimic "snapshots" of the short term memory.

Each view is a two dimensional, restricted-size canvas, in which concepts and
snippets of text may be positioned arbitrarily by the user, according to their own mental model of the subject matter. Relationships may optionally be created between any combination of concepts and snippets of text.

This model of interaction is modeled after concept maps [10][9]. Concept maps are methods of capturing and developing knowledge proposed by Dr. Robert Novak. They feature granular, atomic concepts usually represented as nouns in drawn boxes, connected together by drawn arrows between pairs of concepts. Each arrow may be labeled by a verb phrase, describing the relationship between the two concepts. Each set of two concepts with a relationship between them may be read as a sentence in the direction of the arrow, i.e., noun verb noun.

Concept maps have significant backing from psychological research, and have been proven very effective in learning environments. When extended into a full personal knowledge base system as in the Popcorn prototype, real world user trials have proven it a highly effective model for fluid knowledge capture, formulation and retrieval.

**Notewise: An Enhancement of Popcorn**

Notewise is a web based personal knowledge base tool, intended as an evolution of Popcorn. An attempt has been made to improve the ease of discovery of the user interface actions, to make new users less reliant on documentation and training to get up to speed with the tool. However, the primary goal is to provide
a similar interface to Popcorn, while providing ubiquitous access for users to access their data, by allowing them to access it from any web browser (without using any browser plugins or other installed software), on any computer connected to the internet.

**Ubiquitous Access**

A major barrier for some Popcorn users was that they often needed to record or retrieve knowledge when the system was not immediately at hand. While for some this was simply due to being away from the computer, for others, it was more of the case that they used more than one computer on a daily basis, while their data only resided on one of those machines. Because the database is directly integrated with the client application, Popcorn can only be installed and used from one machine at a time.

Thus, one of the criteria of a truly effective personal knowledge base is that it may be accessed anywhere. A personal knowledge base that is not ubiquitously accessible results in users being cautious about committing information that they may need when the personal knowledge base is not accessible.

To address this issue, Notewise is implemented as a web based system. It may be used from most modern web browsers, from any machine that is connected to the internet. In addition, it provides a web services API that allows users and developers to easily develop other means of interacting with the knowledge base,
and to easily integrate the knowledge base with other existing tools and devices.

**User Interface Modifications**

When redesigning the interface, one of the largest goals was to improve the ease with which users learn the interface. Because it is a web-based application, the ability to train users is significantly reduced.

In Popcorn, there is a tendency to overload multiple actions onto the same area, often causing modal interactions in very subtle ways. For instance, in Popcorn, to move a note, you click and drag it. However, to select text in a note, you click in the note to get a cursor, and then click and drag. So the same action (click and drag) has two different effects, based on whether you first clicked on the note. This proved far too subtle a modal distinction, even for experienced users.

In Notewise, using the “one region, one purpose” paradigm, for every area on the interface, only one action was assigned. Thus, the confusion about how to select text in a note versus how to move a note was greatly reduced. Notes in Notewise have their own title bar, which serves as a dedicated area for moving the note. You can no longer move a note by dragging in the text area. Instead, the text area acts as a standard text area in every way, matching the user's expectations.

The concept of “one region, one purpose” naturally leads to the use of
progressive disclosure. Progressive disclosure in interface design makes the most common actions in an application the most prominent, and successively less common actions successively less prominent. It played a significant role in the redesign of the Notewise interface. Relatively uncommon actions such as renaming a kernel were relegated to be modal, such that the user had to make two mouse clicks (one to select the kernel, and one to enter renaming mode) to start the action. Other, more common actions, such as navigating to a kernel, or expanding it, can be reached directly.

Other interface improvements included making actions that were implicit in Popcorn more explicit. One good example of this is search. In Popcorn, to search, the user just starts typing when another text box does not have focus. This causes a search dialog to pop up in the center of the screen that returned results as the user typed. It turns out that this was a very hard paradigm for new users to remember. It also continued to trip up even fairly experienced users, who often attempted to search while another text field was selected, causing instead for the search term to be added to that text field.

The solution was to add an explicit search field, mimicking most other common applications. While Notewise still searches as you type, it is now much more obvious when the application is in “about to search” mode, and when it is in some other mode. To retain the feature that made search the default action
when nothing else was in progress, in Notewise, focus is given to the search field by default.

**Implementation Challenges in Notewise**
The implementation challenges presented by Notewise proved to be significant. There exist to date few web based interfaces that implement the extent of the desktop-like functionality of Notewise. Notewise allows users to directly drag and resize browser elements via javascript, and draws arbitrary vector lines (for relationships) on the fly. Thus, many of the issues encountered were not standard web development challenges. We will give a brief overview of these challenges here, and then address them in greater depth further on.

Notable similar applications in terms of surface functionality include protopage.com[11] and Webnote[2], which demonstrate the ability to drag and resize browser elements (similar to sticky notes), and the line stretching demonstration presented by Henri Mathieu[8], which demonstrates the ability to draw arbitrary vector lines by stretching diagonal lines.

Perhaps the largest development challenge was allowing the user to interact with the application without having to reload the page for every user action. However, at the same time, it was very important that any action the user takes be immediately and automatically saved to the server, without further user interaction. Thus, it was necessary to allow the browser to send data back to the server using AJAX. The extent to which AJAX was used, both to save data, as
well as to retrieve data, necessitated designing a more robust API for interacting
with data on the server. More specifically, there seemed to exist no good tools
for extending the server object model onto the browser, in a seamless, yet
flexible fashion. Thus, a framework to do this was developed.

Another significant challenge presented was that javascript does not natively
provide a mechanism to draw arbitrary vector graphics. Vector graphics
capabilities were used extensively when drawing the relationship lines in
Popcorn, and thus, a workaround had to be found. A library to draw arbitrary
images on the fly quickly and smoothly was developed, which will be discussed in
more depth below.

Because Notewise is intended to provide ubiquitous access from any computer, it
was important to support multiple web browsers. Supporting multiple
web browsers proved to be the most persistently problematic issue faced during
development, and unfortunately also a problem that is not easily solved in a
uniform fashion. We will discuss some of the problems faced by particular
browsers, and what workarounds may be used to combat them.

Once the basic barriers to functionality were overcome, performance proved to
be somewhat of an issue. Users experienced slow page load times, and the load
on the server to generate pages was excessive. Several strategies to address
this were developed, and will be discussed more extensively.
Lastly, we will discuss a system that was developed to allow users to seamlessly transition from knowing little about the system, to walking through a tutorial, to using the system, all without initially registering. We discuss the application of the lazy registration user interface pattern to Notewise, and why it proves beneficial for signing up new users.

**Browser-Server Interaction**
One of the main goals when designing Notewise was to make the user experience very similar to that of a desktop application. It was crucial not to force the browser to reload the web page every time the user performed an action. Thus, the newly popular technology AJAX was used to provide asynchronous interaction with the server, without refreshing the page. However, using AJAX led to a need to seamlessly extend the server object model onto the browser, simplifying interactions with the server. Thus, a library called JSDBI was developed to mimic the object oriented to relational database mapping library used on the server. Lastly, the decision was made to use a web development framework called Catalyst, based on the model-view-controller pattern, to help organize the server code.

**AJAX**
Ajax stands for Asynchronous Javascript And XML. It was coined by Jesse James Garrett of Adaptive Path, in his article “Ajax: A New Approach to Web Applications”[6]. While the technology is not something new, the term to
describe it, and the approach to effectively using it, is. Simply described, AJAX allows a web page to make asynchronous callbacks to the server, allowing the user to interact with the server without reloading the page.

This seemingly simple change stands to change the face of web interaction, by taking the realm of rich, real time application interaction, and pushing it into the world of the web. No longer are web applications relegated to being high tech versions of “choose your own adventure”, in which every user action results in flipping to another page. Now, the user can accomplish very complex interactions, in short amounts of time, without leaving the page they are on.

This opens great possibilities for developers to build web based applications that would have previously only been possible as rich client side applications. By bringing these applications to the web, the barrier to usage is significantly reduced, as the user doesn't have to install anything on their local machine, other than a web browser and an internet connection. This allows the user to store their data on a web server, and access it from any computer with an internet connection.

REST
REST stands for Representational State Transfer[3]. It describes an architectural style for building web services that aligns itself with the strengths of the web and utilizes those characteristics of the web that made it successful.

Contrary to many other web services protocols and methodologies, REST places
a focus on the data that is accessed via the API, rather than simply exposing a series of functions to a remote procedure call interface. Thus, the interaction becomes centered around the nouns of the dialog, rather than the verbs.

Interactions are modeled around the four HTTP request types – GET, POST, PUT, and DELETE. Semantically, these can be equated with the four major SQL query types – GET is similar to SELECT, POST is similar to UPDATE, PUT is similar to INSERT, and DELETE is similar to DELETE. Each of these operations is performed against an endpoint URL, that represents a particular resource.

Thus, in Notewise, to retrieve information about a kernel, you might request a GET from http://beta.notewise.com/rest/kernel/42. That asks the server to return the data that represents kernel with id 42. To update part of that same kernel, you would sent a POST to that same url. Similarly, to delete it, you would send a DELETE to that url. To create a new kernel, you would send a PUT to http://beta.notewise.com/rest/kernel, which then returns you the xml for the new kernel, which contains the assigned id.

For further details about the Notewise REST API, see Appendix A.

**JSDBI**

The JSDBI library was developed as a means to extend the data object system on the server all the way to the browser. Using a meta object model, it allows users to define objects that are interacted with via a REST API. For each type of
object, the user defines a URL endpoint, a series of fields, and a series of relationships to other object types. This results in a javascript class with basic CRUD methods (Create, Retrieve, Update, Delete), which transparently interact with the REST interface.

JSDBI extends the basic model presented by object oriented RDMS systems onto the web. An example of such a system is Class::DBI, the OO-RDMS system used by the Notewise server. Through introspection on the database schema at system startup, it provides a class per database table which supports basic CRUD methods, and is extensible by the programmer through supplementary classes. Users build classes that override Class::DBI, and which call a series of class methods at startup which, using a meta object model, add appropriate getters and setters to each class according to the database schema.

JSDBI uses a similar model. During class initialization, each class that overrides JSDBI uses a series of class methods to set the URL endpoint for the class, and the various object attributes and relationships to other objects. If relationships are defined, JSDBI automatically handles the hydration of the associated objects.

One large stumbling block for JSDBI is that any synchronous operation blocks the entire browser UI. Thus, whenever possible, it is necessary to perform actions asynchronously. Currently both updates and deletes are asynchronous, while creates and retrieves are synchronous. In theory, creates could be handled
asynchronously. However, Notewise requires a unique id for all objects displayed on the screen at any given point in time. This is necessary so that updates can be performed, as well as so that there is a distinct mapping from html elements to javascript objects.

This requirement could possibly be mitigated by queuing updates while a creation is still pending, and binding html elements and javascript objects by properties alone, however, this introduces additional complexity to the application, and hasn't yet been attempted.

See Appendix B for a detailed look at the JSDBI API.

Catalyst
Catalyst[12] is a web development framework for Perl which uses the model-view-controller (MVC) design pattern. At its core, it provides a very flexible mapping from URLs to controller methods. It also provides a significant amount of scaffolding that is common to most web applications, such as cookie handling, parameter parsing, form validation, session handling, and interfacing with web servers. It can currently interface with mod_perl, fastcgi, normal cgi, and its own standalone server used during development. It also provides the ability to swap out various other existing perl modules for use as models and views (and even supports using multiple different view or model modules at once).

In my estimation, using Catalyst speeds up web development by as much as 2-3
times. I believe this is caused by a combination of not having to rewrite the standard web application functionality it provides, as well as by being forced to follow good MVC design habits, which help structure the code as the application grows.

The Notewise code uses Catalyst as the core of the server side application, both for generating html that the user sees, as well as for the REST API that the javascript code on the browser interacts with to retrieve and update information. REST controllers were automatically generated based on the database schema using a helper script. They were then modified by hand to add proper authentication and authorization control, and to provide more sophisticated error handling.

**Dynamically Drawing Vector Graphics**

One of the significant technological barriers posed by porting Popcorn to the web was the issue of how to draw relationship lines, and the end arrows. Up until very recently, there was no native browser support for drawing arbitrary vector graphics via javascript. Any attempt at drawing vector graphics had to be accomplished through one of two workarounds.

The first method uses many one pixel by one pixel `<div>` elements with a background color, to act as individual pixels in the drawn image. This allows great flexibility, at the expense of speed - creating and managing each individual html element represents a significant load on the browser. This technique can
be taken a step further, by conglomerating all pixels of the same color in a given row into a single `<div>`, that is wide enough to span all of the pixels. This significantly reduces the element count, although the number of elements is still significant. Walter Zorn developed a particularly good, general purpose library for this technique[13].

The second method uses images, either statically built beforehand, or compiled on the fly by the server. Google Maps[7] uses this technique to dynamically build drop shadows and road outlines. This reduces the load on the browser, as browsers are optimized to display images. In addition, it provides a great amount of flexibility and power to generate complex graphics. However, it may introduce latency if the applications requires very fast redrawing of the images.

Relationship drawing in Notewise uses a combination of these techniques. The arrows are drawn using Walter Zorn's vector graphics library. The lines are drawn using a library I developed, which uses a variation of the second technique.

The general idea is to take an image with a one pixel wide diagonal line, with a transparent background, and stretch it to fit the exact required length. The simplest version of this requires two images, one with a line that stretches from upper left to lower right, and one that stretches from lower left to upper right, as neither javascript nor CSS provides a means to rotate or mirror images. This
basic concept is well demonstrated by Henri Mathieu[8].

However, this naive approach presents several problems. First, many browsers, when stretching or shrinking an image, do not antialias it. Instead, they eliminate individual rows and columns of pixels, evenly distributed across the image. When shrinking our one pixel diagonal line images, this causes gaps in the line where a row was eliminated, resulting in a sort of dashed line effect (see Figure 1). In the case of stretching the image, segments of the line grow from one pixel to two pixels. The latter of these two side effects is preferred, from a position of visual aesthetics.

Thus, the decision was made to only stretch images, and never to shrink them. This raises another problem though. As you stretch the image more and more, the number of segments that are two pixels wide continues to increase, until you reach double the size of the original image, in which case the entire line is two

Figure 1: Shrinking a line without antialiasing. a) The line without shrinking b) The line shrunk vertically
pixels wide. Thus, it's important to set a limit on how much to stretch a given image. To do this you need a set of images to switch between, as the image grows and shrinks.

One more optimization can be used, by looking at the way a diagonal line is structured. A diagonal line that is wider than it is tall, is comprised of a series of short horizontal line segments. As you stretch the image horizontally, those short line segments grow, but remain one pixel high. Thus, as long as you only stretch the image horizontally, the image can be infinitely wider than the original image, while still looking "correct". The same goes for vertically stretching images that are taller than they are wide.

Thus, it is possible to use “square” diagonal images (images that have a matching width and height), and choose which image to use based on the smaller dimension of the line desired, always choosing a line that is equal to the smaller

![Figure 2: Stretching a line. a) The original line b) The line stretched horizontally c) The line stretched vertically](image-url)
dimension or slightly smaller (never larger). This results in a line that may be stretched significantly in the larger dimension, which is fine, as it results in an entirely normal looking image, made up of line segments that are one pixel high (or wide), and multiple pixels wide (or high) (see ).

Modern browsers provide several other, more robust solutions to this problem of drawing vector graphics dynamically. Unfortunately, none have enough wide spread adoption to be considered viable for a publicly available web application.

Scalable Vector Graphics provides a declarative approach to displaying vector graphics in a portable manner. It's an XML based markup language for describing two-dimensional vector graphics, which can both describe static images, as well as animated and interactive graphics via either declarative or scripted means.

The recent Canvas html element provides another option for displaying scriptable vector graphics. Originally introduced by Apple for use in their Mac OS X Webkit component, it has been recently adopted by both Gecko based browsers (including Firefox and Mozilla), and has been standardized by the WHATWG working group through an attempt to propose specifications for next generation web applications. Rather than being declarative like SVG, the canvas component is procedural, and can be interacted with via javascript.
While both SVG and the canvas element provide an enormous amount of flexibility and control over the image displayed, neither have wide enough adoption in currently deployed web browsers to be considered usable in mainstream applications.

**Supporting Multiple Browsers**

From the outset, it was clear that for Notewise to be considered a success, it would have to support multiple mainstream browsers. At a very minimum, it had to support Internet Explorer 6.0 and Firefox 1.0 and 1.5. Later, Safari was also added to this list (though support for Safari is not yet complete).

Writing a complex web application for multiple web browsers proves to be a fairly difficult task. While browser compatibility and standards support has gotten significantly better in the past 5 years, compatibility and standards support is by no means complete yet.

The best strategy to supporting multiple browsers seems to be to start with standards compliant code which should do what you want, and then gradually relax standards compliance until it works properly in the target browsers. This proves to be a fairly time consuming and frustrating process. Because a lot of the functionality presented in Notewise pushes the envelope of what browser based applications are capable of, the issues encountered were different than those encountered by many web designers and programmers.
One of the farthest reaching issues was that of a general lack of support for most CSS selectors in Internet Explorer 6. CSS selectors provide a myriad of ways to apply CSS styles to specific html elements in a declarative fashion. When used properly, this provides enormous value by allowing proper separation of semantics and display code, which results in more flexible, easier to maintain code.

The original goal was to use a single div element to represent the each kernel or note. Within that, individual divs would be used for the various parts of the header and the body. The outer div would contain a number of CSS classes, indicating the particular state of the object, such as whether it was selected, highlighted, collapsed or expanded, and so on. Various combinations of states would result in the object appearing in different colors, which meant swapping out the images that made up the header, among other display changes.

The most straightforward method of encoding which header images went with which state was to set the background of each header div based on the class of the immediate parent, which was the outer div, using a child selector. Thus, the selector would be something like '.vkernel.selected > .left'. This selects any element that has a class of 'left' which is a direct descendant of an element with classes 'vkernel' and 'selected'. The upshot of doing this is that CSS classes need to be changed in only one place to affect the display of the object, and the CSS rules that dictate how an object is displayed are very straightforward and simple.
Unfortunately, despite their being a part of the CSS 2.0 specification as outlined by the W3C, Internet Explorer 6 does not support child selectors. Nor does it support selectors that involve multiple classes for the same element (i.e., '.vkernel.selected'). The first problem, a lack of support for child selectors, caused issues because kernels can be nested inside each other, which causes issues if descendant selectors are used instead. Thus, a selector '.vkernel.selected .left' will match both the header of the selected kernel, as well as the header of any kernel contained inside it, making kernels that are not actually selected appear as though they are.

A solution to this issue was reached by using negative classes in addition to the existing positive classes. Thus, the class 'notselected' was assigned to all objects that were not currently selected. This allowed more complex selectors to be built of the form '.selected .vkernel.notselected .left', which protected children of selected elements from appearing as though there were selected. However, this solution was imperfect as well, due to lack of support for selectors involving multiple classes for the same element.

The current solution, which works tolerably well, assumes a brute force approach. It assigns a series of composite classes to each element which needs to vary it's appearance based on the state of the object. The composite classes are of two types. The first contains the object type, and a dash separated list of
each attribute of the object (i.e., 'vkernelselected-nothighlighted-collapsed-contains'). The second set of classes is virtually the same, but consists of pairs of the object type and each attribute individually. Lastly a class is added with just the object type. These three sets of classes allow the assignment of styles based either on the object type, a particular attribute for that object type, or the composite of all attributes for that object type. A javascript function manages these classes as the state of the object changes.

Unfortunately, this is not the extent of Internet Explorer's lack of support for the CSS 2.0 specification. It does not support min-width, max-width, min-height, or max-height style attributes, which prove extremely handy when specifying the constraints on an object as it is resized by the user. Nor does Internet Explorer support the :focus selector attribute, which matches elements that currently have keyboard focus. Both these problems can be solved through the use of javascript, at the expense of more complicated code.

Another major stumbling block was the lack of support for specifying both a left and right offset (or top and bottom offsets), and allowing the browser to calculate the width (or height). This proved particularly frustrating when building the outer interface frame, as one of the design constraints was to size the interface frame to the browser size, without using scrollbars. While there are existing techniques to force a web page to use up the available horizontal space, there are few techniques to do the same vertically, as few conventional
web pages have a similar constraint. By specifying a top and bottom offset for an element, the browser will calculate the height necessary to make it match those offsets. However, since Internet Explorer only allows the specification of either a top or a bottom offset, but not both, this technique was not viable.

There were two solutions that eventually emerged, each which contained both strengths and flaws. The first technique is to use the correct markup according to the CSS 2.0 specification, and then resize the element manually using javascript with a trigger on the browser resize event. While this has the advantage of using entirely correct markup, it has the disadvantage of not rendering properly if javascript is not active. The second technique relies on quirks of the way tables are rendered if no doctype for the page is set. One can create a table which is uses 100% of the screen height, and then specify a specific row within that table that uses all the remaining available space.

Internet explorer also does not support alpha transparency for png files. Instead, it uses an opaque neutral gray background for the image. Notewise uses alpha pngs extensively for the relationship creation halo around objects. However, Internet Explorer can be made to display alpha pngs properly by using the “filter” CSS directive[4]. As described, this must be combined with a CSS selector that uses the [attribute] selector feature to set the background for all other browsers, as Internet explorer ignores styles with the [attribute] selector. If this is not used, Internet Explorer displays both the background image (with
an opaque gray background), as well as the transparent filter directive.

Firefox is alas, not without its own peculiar bugs. One of the more frustrating bugs has been a Macintosh only bug that causes scrollbars to always render on top of any other content, regardless of whether the element they are part of is on the top layer. This caused problems for the corners of notes that had one of two scrollbars, as the corner resize widget and one of the scroll buttons (in the lower right corner) were competing for the same space. The eventual solution was to hide the scrollbars when the user places the mouse over that corner – this task is made easier by the fact that the corner still receives mouse events as though it was rendered on top, even though it is not visible.

**Performance**
A major source of concern towards the end of the development cycle was performance, particularly the speed of page loads. There are two main concerns when talking about performance. The first, and more obvious issue, is that of latency for the user, particularly the speed of page loads. The higher the latency, the more sluggish the application feels to the user. The second, less immediately obvious performance concern is that of server load. The more CPU time required to generate a given page or perform a given action, the more server horsepower is required to serve the same size user base.

There are several components that contribute to the page load time as perceived by the user. The server must generate the necessary html and javascript for the
page, it must be transmitted to the browser, and the browser must parse the html and execute the initial javascript. While some of these actions may be performed in parallel (the browser starts parsing the page before it is full transmitted, for instance), they all contribute to the overall load time of the page.

Optimizing the process of generating the html involves several components. One of the easier aspects to address is that of database performance. Properly indexing the database and optimizing slow queries (and elimination redundant queries) goes a long way, and is fairly straightforward. Early on in the development of the Notewise server code, there was a number of queries that were not optimized, and there was a huge performance gain in optimizing several of them.

Another method for improving the time necessary to generate pages is caching. Caching can be utilized at various levels of granularity, both at the data level, and at the html/page level. For instance, one simple cache of data that had a huge performance gain was caching the object associated with a particular object id. Because it never changes, this proved to be not only relatively failsafe, it dramatically reduced the number of unnecessary queries.

Caching html and other web objects (images, CSS, and javascript) proves to be a much stickier problem. Notewise uses a unique, semantic URL for each view. The advantages of this are significant, as each view is now also a valid web page,
with its own URL, which can be bookmarked, emailed, or written down. Moreover, URLs follow a very simple scheme (http://beta.notewise.com/username/name_of_kernel), and thus, in theory, a savvy user could hand generate a URL for themselves, which should prove to be useful as users attempt to integrate Notewise with other services and websites.

However, the disadvantage to this scheme is that as the user navigates between pages, the browser loads entirely new html for each page. When a user interacts with a page, making changes that are sent back to the server via AJAX, that ends up changing what the contents of the html will be when the page is loaded next. Thus, if the page is cached, when the user makes a change, navigates away from it, and then navigates back to the cached version, it will appear as though the change was never made. However, if the user didn't make any changes, the html would still be valid.

The simplest approach is to only cache content that will never change based on a user action, such as images, javascript libraries, and CSS style sheets. All html is marked to never be cached, even when using the forward and back buttons. However, it may be possible to allow html to be cached, and use the “must-revalidate” directive in HTTP 1.1 to force caches to validate the freshness of the cache for each page load. This validation is done by having the cache contact the server and compare the last modified date of the version in the cache with the last modified date of the resource on the server. While this has yet to be
attempted, this could prove to be a fairly significant speed improvement, provided it proves significantly cheaper to generate the last modified date than it is to actually generate and transmit the html. The downside to this is added complexity.

Another means of decreasing the perceived load time of the page is to load the page in sections. That is, generate only the html necessary to display the most prominent portions of the page initially, and ship that to the browser. Then, using javascript, fetch and fill in the less prominent portions of the page. While this does not decrease the total page load time, or the load on the server (it probably slightly increases the server load), it does decrease the amount of time before the user sees portions of the page, and in fact, may decrease the time to actually being able to interact with the application.

Notewise currently uses this technique to load portions of the sidebar, specifically the parents panel and sandbox. As these two panels provide ancillary information, it is not necessary for the user to see them immediately, and generating the parents panel in particular is relatively time intensive, so delaying loading proves to be a significant win. Similarly, the loading of html for the body of collapsed kernels is delayed until the kernels are expanded (and thus, the body is displayed). This alone can potentially reduce the task of generating the html for the view from $O(n^2)$ to $O(n)$, where $n$ is the average number of kernels per view.
Tutorial Setup and Lazy Registration

One of the goals of Notewise was to make the process of starting to use the tool as easy as possible – from first going to notewise.com, to signing up for an account, to using the tool regularly. A major piece of this process is to allow the user to start learning how to use Notewise, and indeed to actually start using it, as quickly and effortlessly as possible.

Once potential Notewise users wish to try using it, they click on a “try it now” link. This immediately takes them to the Notewise tutorial. The tutorial is actually an automatically generated user account, prepopulated with the standard tutorial instructions. A new account is generated for each user, on the fly. The user is then automatically logged in under their tutorial account. They now have their own version of the tutorial, which they can modify to their heart's content, and which no other users may see.

Once the user is finished with the tutorial, they may then choose to start their own personal knowledge base by clicking on another link at the top of the interface. This essentially causes the same process to occur again. Another account is created, and the user is automatically logged in under that account. However, this time no contents are copied into the account, as was the case for the tutorial – the account is empty.

This account will become the user's permanent account, when they choose to
register. However, until they choose to register, the information they enter will still be stored, and will be accessible as long as their browser retains the cookie. When the user does choose to register, they need only give their desired username and password, and their email address, which are simply added to their account. They may now login to their account from any browser.

This process follows the recently named lazy registration pattern[1]. More generally, the lazy registration pattern allows a website to start to accumulate pieces of information about a user, allowing them to customize their interaction with the site, while delaying formal registration until a later date. Often discussed in an e-commerce context, it is a useful pattern to smooth the process of setting up an account for a user into a seamless, natural interaction.

Assessment
It is currently possible to assess the strengths and weaknesses of Notewise based on two primary areas: the end user experience, and the success of the technical architecture. The strengths and weaknesses of the end user experience can currently be evaluated in terms of initial, informal feedback from a small handful of beta testers that are using Notewise. While formal user testing has not been done, the informal feedback may give some insight as to what issues users are encountering, and what the relative success of a larger user test would be. The strengths and weaknesses of the overall Notewise architecture are perhaps best judged not only by looking back at the
development process, but also looking forward at the possible ease of future development.

**Feedback from Early Users**

Roughly 20 user accounts have been given out to users, including the members of the research team. Of these, roughly 10 have used the tool any significant amount (for instance, it appears that 4 users never even logged on). Some of these users have sent unsolicited feedback to various members of the research team. While a good portion of this feedback consisted of bug reports for broken functionality, it also consisted of higher level discussion of the relative merits of the tool and the interface. Out of the active users, 6 had also previously used Popcorn, and thus were able to compare the relative strengths and weaknesses of Notewise as compared to Popcorn.

**Strengths**

For users that used multiple computers on a regular basis, the ability to access their data from any web browser proved to be very useful. However, a significant portion of the users only use one computer on a regular basis, and thus, the ability to access their data from multiple computers does not provide any significant added value.

Overall, the user interface appears to be more discoverable to users. There no longer seems to be any issue with users being able to figure out how to perform searches, or how to create relationships, two features that were difficult for
Popcorn users to discover. One user noted that this added discoverability came at the expense of ease and speed of use of some features for experienced users.

For example, to create a relationship in Popcorn, the user merely had to move the cursor over the outer edge of an object, and then drag. In Notewise, the user must click on the object to select it, and then drag from the outer "halo" surrounding the object that appears when it is selected. The latter adds an additional click to the workflow. However, the former proved to be relatively indiscernible for new users, and experienced users were observed to have frequent problems with finding the correct distance from the edge in which to drag.

**Weaknesses**
Perhaps the largest weakness that emerged from user feedback was the slow page load time when navigating between different views. This was particularly evident to former Popcorn users, as Popcorn's time to navigate between views was, on average, several times faster than that of Notewise.

One of the largest user concerns was that their knowledge is inaccessible when they are not connected to an internet connection. While it is unclear how often this is actually the case, for some users, the possibility of this meant they were reluctant to commit any significant amount of knowledge to Notewise, for fear of not being able to access it.
Several users also cited uneasiness with the fact that their data was being stored on a centralized server. In particular, there was concern that this server might either be compromised, allowing other people to view their sensitive information, or that the server might suffer a catastrophic failure, resulting in data loss.

Lastly, while some features were easier for users to discover, others became harder. For example, some users had problems discovering the search dropdown that appears as soon as search results are returned, as the user is typing in the search field. This was largely due to the slow speed at which search results were returned. While this speed can definitely be improved, the long term solution should be to provide the user some feedback as soon as they start typing, to let them know that search results are in the process of loading.

**Architectural Assessment**

**Strengths**
JSDBI, the javascript library that was developed to extend the server object model onto the browser, proved to be an enormous win in terms of providing a useful abstraction for server interaction and in terms of speeding up development. While it took several iterations of the JSDBI code to correctly implement it, and to discover and develop a full set of features, this has proved to be one of the areas of the code that was modified the least. Not to mention, it made retrieving and sending data to the server almost entirely seamless, and
something that was rarely thought about once JSDBI was mature.

**Weaknesses**

One of the biggest weaknesses of the architecture of the javascript code on the browser was the lack of a strong abstraction that encompassed the shared qualities of kernels and notes. While a superclass for these two objects emerged, it emerged organically during development, and served primarily as a means to share code between the two objects, without hiding much complexity.

It seems that a stronger abstraction for this behavior is possible, though it is not immediately obvious what it would look like. If this were successfully achieved, it would likely be significantly easier to add new types of objects to the system.

Notewise currently loads a new page every time the user navigates to a new view. One advantage to this is that each view has its own url, allowing it to be easily bookmarked like any other web page. However, the downside is that for each page, the interface and associated javascript and css code must be reloaded and evaluated from scratch. This is a significant source of time during page loads.

An alternative architecture would be to have only one url for the entire application, and to dynamically load the content for each new view without reloading the page. This is common among some larger web applications (such
as gmail.com), and provides significant speed increases. However, this comes at the expense of the ability to seamlessly bookmark pages within the application. This might be combated slightly by using the fragment portion of the url (the part after the # sign in the url http://foo.com/bar#baz). However, the fragment is usually used to signify a particular section of a document, and thus, this solution is not ideal.

**Summary of Assessment**
Performance was not addressed until quite late in the development process. However, it proved to be one of the larger issues that impacted the user experience. In retrospect, it would have been advantageous to address this earlier in the process, particularly during the design phase, before significant code was written. However, because there are so few applications of this kind, it is unclear how much could have been known about performance at the start of development.

**Conclusion**
Notewise successfully provides the basic functionality in Popcorn in a web browser environment. While the challenges in migrating such an application to the web are significant, such an endeavor is certainly possible. In particular, the use of various libraries and frameworks proved to be one of the biggest gains to development of Notewise. Catalyst proved to be a big win in terms of overall web development, while JSDBI and the line stretching library that were developed for Notewise did a good job of solving particular domains of problems. However,
there still remains work to be done to make web development of this nature as straightforward as current desktop application development.
Appendix A - Notewise REST API documentation

The Notewise REST API is built around the REST methodology. You can find more about REST at http://www.xfront.com/REST-Web-Services.html.

**Status codes**
Status about the success or failure of the requested action will be returned via the HTTP response code. We use the following status codes:

- 200 - Ok - no errors were encountered, and the task was successfully performed
- 201 - Created - this is only given during a PUT, and signifies that the resource was correctly created
- 400 - Bad request - There was a problem with the way you specified your request. Make sure you are passing in all necessary information, and that the request is properly structured.
- 403 - Forbidden - You do not have access to do what you were trying to do
- 404 - Not found - The object you referenced doesn't exist on the server
- 500 - Internal error - something went wrong in the server code. Please report this to a server admin.

**Authentication**
To authenticate, you can pass any URL the additional CGI parameters 'email',
and 'password'. The response will contain a cookie, which you may optionally use to authenticate during the rest of your session. You may also continue to pass 'email' and 'password' for each request.

Example:
http://beta.notewise.com/rest/kernel/1?email=scotty@scottyallen.com&password=password

**Interacting with objects**
All objects in the system have their own URLs. To interact with an object, make HTTP requests to its URL. The standard four actions are:

- Create a new object - Send a PUT with the attributes of the objects as CGI parameters to the base URL for that object type (ie, http://beta.notewise.com/rest/kernel).
- Retrieve an existing object - Send a GET to the object's url. It will return an xml document of the form:
  
  `<response>
    [object specific xml...]
  </response>`

  See below for the xml document format for specific object types.
- Update an existing object - Send a POST request to the object's url, with the attributes you'd like to change as cgi parameters.
Delete an existing object - Send a DELETE request to the object's url.

The standard object urls are:

- `http://beta.notewise.com/rest/kernel`
- `http://beta.notewise.com/rest/note`
- `http://beta.notewise.com/rest/relationship`
- `http://beta.notewise.com/rest/containedobject`

**Objects**

Note that the order of attributes in the XML is not guaranteed

**Kernel**

Sample XML:

```
<kernel id="123"
    name="foobar"
    created="2005-11-18 06:43:09"
    lastModified="2005-11-17 23:43:09"
    source="http://yahoo.com"
    uri="http://en.wikipedia.org/wiki/Foobar"
    object_url="http://notewise.com/fred/foobar/123">
    <containedObjects>
        [xml for visible kernel and note objects...]
    </containedObjects>
</kernel>
```
Things that the user can set (all are optional):

- name
- source
- uri

Note
Sample xml:

```xml
<note id="880"
    container_object="878"
    created="2005-01-01 01:02:03"
    h="20"
    lastmodified="2005-02-02 03:04:05"
    source="myuri"
    w="30"
    x="40"
    y="20">
    a test note
    a new line
</note>
```

Things that the user can set (things with * are required):

- container_object* (the id of the kernel this note is contained on)
- w (width)
- h (height)
- x (x coordinate of upper left corner)
- y (y coordinate of upper left corner)
- source (where the contents of the note were obtained from, if any)
- content (the actual contents of the note)

**Relationship**

Sample XML:

```xml
<relationship id="1000"
    nav="fromleft"
    part1="998"
    part2="999"
    type="shot"/>
```

Things that the user can set (things with * are required):

- nav * (can be 'bi','non','fromleft','fromright')
- part1 *
- part2 *
- type (this is the label on the relationship)

**Visible Kernel**

Sample xml:

```xml
<visiblekernel collapsed="1"
```
contained_object="1096"
container_object="948"
height="30"
width="19.7002"
x="33.94"
y="17.4367">
<kernel name="This is a test"
created="2005-12-18 18:44:33"
id="1096"
lastModified="2005-12-18 11:44:40"
source=""
uri=""
user="210"/>
</visiblekernel>

Things that the user can set (things with * are required):
- contained_object *
- container_object *
- height * (only used when not collapsed)
- width * (only used when not collapsed)
- x *
- y *
- collapsed * (0/1 == false/true)
Search
To search for notes or kernels, send a GET of the form
http://notewise.com/rest/search/mysearchstring replacing "mysearchstring" with
your desired search string. This will search for any kernels or notes
containing words that start with the given search string. The response will
be of the form:

<response>
    [kernel and note xml...]
</response>

Appendix B: JSDBI API
The JSDBI module presents a simple CRUD interface for REST web services.

Requires Prototype.js 1.3.1 - http://prototype.conio.net/

Use it like the following:

    // Basic class setup
    Music.Artist = Class.create();
    Music.Artist.extend(JSDBI);
    Music.Artist.prototype = (new JSDBI()).extend( {
        initialize: function () {
            ...
        },
    });

    Music.Artist.fields(['artistid', 'name']);
    Music.Artist.url('http://someserver/rest/artist');
    Music.Artist.elementTag('artist'); //optional

    // In the calling code
var artist = Music.Artist.insert({name: 'Billy'});
var artistid = artist.id();

artist = Music.Artist.retrieve(artistid);
document.write("name: "+artist.name());

artist.name('Fred');
artist.update();
artist.destroy();

The REST interface must use a single url endpoint, with primary keys for the objects appended to the end of the url, like http://localhost/rest/artist/1 where 1 is the primary key of the artist requested. Retrieve and create will be called on the base url (http://localhost/rest/artist) and update and delete will be called on the object's url (http://localhost/rest/artist/1).

Actions are mapped onto the REST interface in the following manner:

retrieve - HTTP GET
insert - HTTP PUT (accepts CGI parameters for field values)
update - HTTP POST (accepts CGI parameters for field values)
delete - HTTP DELETE

Assuming the docTag set to 'response' and the elementTag is set to 'artist', the xml returned from the server should look like:
<response>

<artist name="Fred" artistid="80"/>

</response>
Works Cited


5. Davies, Stephen. The Efficacy of Personal Knowledge Bases for Materializing Mental impressions. 2005


9. Novak, J.D.. The theory underlying concept maps and how to construct them.. Available at: http://cmap.coginst.uwf.edu/info. 2003

