Building Highly-Interactive, Data-Intensive, REST Applications: The Invenio Experience

Michelle Annett, Eleni Stroulia
{mkannett,stroulia}@cs.ualberta.ca

Department of Computing Science
University of Alberta

Abstract

With the explosion of Web 2.0 ideas and technologies such as XML, REST, and RIAs (Rich Internet Applications), developers are now creating "mashup" applications that aggregate numerous sources of information and promote rich user interaction. Although many innovative mashups are being created, there has been little research systematically examining which technologies to use and how to design and implement such applications. We describe the features and complexity inherent within a data-intensive, REST-based, RIA entitled, *Invenio*. Invenio combines a variety of different technologies (Yahoo! Maps, Amazon Associates Web Service, REST, and the Flex framework) to geographically visualize aggregated music chart information. We report on our experiences in designing and authoring Invenio, use Invenio's requirements as a case study to examine relevant technologies and recommend a set of possible "best practices" for developing other RIAs.

1 Introduction

As the size of the Internet dramatically increases each year, so do its user's demands. An Internet user is no longer content interacting with static HTML web pages or simple searching portals; instead, users want applications and services that promote collaboration and provide them with rich content that can easily be manipulated. Given these changes in Internet user's usage preferences and practices, it should come as no surprise that the number of applications utilizing Web 2.0 technologies and ideas such as REST, "mashups" and Rich Internet Application (RIA) frameworks has dramatically increased.

Of these ideas, mashups and RIAs are probably foremost in the minds of users. Mashups are web applications that aggregate multiple data sources into one application. Although different in many ways, most mashup applications contain the following three elements: (a) one or more data sources, (b) a website to display an interpretation of the data sources, and (c) a client web browser, which acts as the user interface for the website. On the other hand, RIAs are web services that use the web client to process a user interface and use an application server to store program states and user data that is generated. The proliferation of accessible REST APIs and rich user-interface frameworks that can run with zero installation requirements on browser sandbox environments greatly supports the development of highly interactive, web-based, data-exploration tools.

Given the fact that every web service or applications stores, retrieves and modifies data to some capacity, it is surprising to find that there are few recommendations or instructions in existence that explain which frameworks or APIs work best. Without such information, deciding upon the types of data visualizations to use, the data sources (and their organizational structures) and the affordances (or constraints) of user interaction can lead to mass confusion and frustration for developers and users alike. Many developers end up utilizing a bricolage approach to design and implementation, which ultimately wastes valuable time and resources. Not only does the amount of time spent on a project increase, but the number of "workarounds" or "hacks" that are needed also increases (thereby making the project's code unnecessarily complex).

Sharing one's experiences, either by providing documentation, recommendations, guidelines or instructions, can potentially save everyone time and effort. Providing detailed examples and justifications for one's work – as we do in this paper – can not only help novel developers, but it can also increase the quality of an expert's work and inspire other developers to increase creativity within their own applications. As well, we can use these guidelines and recommendations to help push the evolutionarily pace and optimality of technologies forward, by constantly demanding more improvements and innovations in Web 2.0 technologies and illustrating the importance of cohesion and interoperability between resources, frameworks and APIs.

The remainder of this paper discusses the design rationales of a REST-based RIA service, Invenio, and the experiences encountered during its development. We discuss our motivation in choosing the application domain and review related visualization and mashup research (Section 2), we then discuss the Invenio software architecture (Section 3) and workflows (Section 4). Section 5 concludes with our reflection on the lessons we learned from the experience and our advice on "good practices" for building REST-based RIA applications using complex compositions of existing resources.

2 Motivation

Under the Web 2.0 umbrella, we now recognize a collection of technologies that flexibly and creatively support information sharing on the web. On-line wikis, blogs, and RSS feeds have had an immeasurable influence in various domains, as they provide an easily accessible platform for word-of-mouth advertising, online retail, product research, and media sharing.

On the marketing/advertising front, it is interesting to consider the celebrity gossip blog PerezHilton.com [1] whose founder, Mario Lavandeira, started posting audio and video links of his favorite unsigned or overseas artists such as Leona Lewis, Amy Winehouse, and Mika on his blog. Before his postings, each of these artists had local "cult" followings, but after they were blogged about, they obtained number one hits, top album and singles downloads on Amazon and iTunes, and cross-continental success. With respect to retail and media sharing, the ever-growing popularity of online digital media mechanisms, such as peer-to-peer networks, torrents, and online music stores have shifted the importance of the Internet in music consumer's purchasing habits and the borderless selling of products. From 2005 to 2007, there was a steady decline in physical CD sales (-7.8%, -4.9%, and -9.5%), and an astronomical increase in digital album and single sales (+150%, +20.8%, and +45%) [2,3,4]. Given these sales figures and the compelling evidence for the impact of on-line word-of-mouth, it should come as no surprise that consumers are now, more than ever, looking at the Internet as an influential source of music information and recommendations.

Coupling the importance of the Internet for consumers within the music domain with the popularity of RIAs and mashups, the authors have chosen to create Invenio, a highly interactive, data-intensive mashup service that uses geovisualization techniques to facilitate the discovery of new music by its users.

In looking at the geovisualization and mashup work occurring in academia and industry, there are a number of innovative projects that have been pursued. Wood *et al.* have utilized a geographic map with data dial overlays to demonstrate the power of geovisualization mashups [5]. They have taken cell phone text message queries sent to a popular cell phone provider and combined them with Google Earth to assess trends in text message queries. A data dial displays all of the times of day that a query was made from a specific location (Figure 1A). The angle of a line leaving a specific point indicates the time that a query was made and the length of the line indicates the number of queries made at that time. In contrast to Wood *et al.*'s "typical" geovisualizations, Kwan and Lee used two non-traditional visualization techniques (linear and topography inspired) to visualize a geo-coded (geographical coordinates) data set that was composed of 129,000 human activities [6]. Using horizontal and vertical lines (Figure 1B), as well as peaks and valleys (Figure 1C), Kwan and Lee have extended the boundaries of geovisualization research in a very creative direction. Although both sets of researchers use different ways to express temporal data (one in 3D and the other using 2D dials), neither contain capabilities which might allow one to discover trend patterns that occurred

Service	Purpose	Visualization Types	Interactive Visualizations	Zooming / Filtering
TubeMogul [7]	Deploys and tracks user created videos across multiple websites (AOL, Google, MySpace, Yahoo!,YouTube, Viddler)	Line chart, bar chart, pie chart, Google Map	None	N/A
QLikView Musiq Tracker [8]	Visualizes online and radio station airplay from Mediaguide service	Line chart, bar chart, pie chart, geographic map	Charts – yes; Geographic map – no	On some visualizations, yes
HealthMap [9]	Visualizes disease outbreak bulletins (Google News, curated personal accounts, World Health Organization official alerts) geographically	Google Map	Users click on map markers to view additional information	Filter by disease type , resource provider

Table 1 - Commercial Mashup Services.

over long periods of time, such as days or weeks. This temporal dimension is something that we have included within Invenio.

As illustrated in Table 1, there are a variety of successful, online mashups that have been created to visualize aggregate data. TubeMogul [7] provides users with a Google Map on which their demographics are shown. Unfortunately, TubeMogul does not utilize the full power and capabilities that the map can provide: instead of enabling interactivity with this map, the TubeMogul designers have chosen to leave it static and to not provide any way to determine the regional popularity of one's videos over time. The QLikView Musiq Tracker [8] service does provide useful information, but most often there is too much information presented in awkward or confusing ways. The data is largely static and the interactive functionality is very slow, leading to a very ineffective service. The use of color coding techniques, additional pop-up tool tips and zooming and filtering capabilities made the HealthMap service [9] an excellent reference system to use while designing and implementing Invenio.

3 Software Architecture

As Invenio is a Rich Internet Application, it uses several technologies to visualize a rich set of spatial data: the Flex framework, the Amazon Associates REST service, and the Yahoo! Maps API.

Currently, the Invenio architecture consists of the Web Interface, an external database, a REST web service and two commercial web services. The design and interactions between these components will be discussed in Sections 3.1 – 3.5, and the work processes and workflows of the system will be discussed in Section 4.

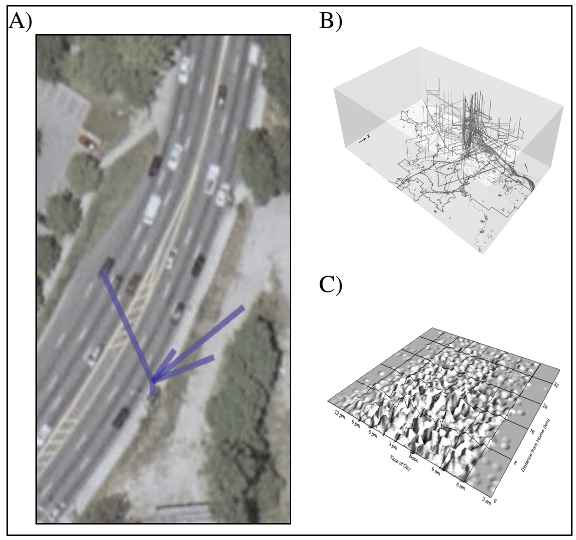

Figure 1 - Geo-Spatial Visualizations from [5,6].

3.1 The Data Resources

Because the most reputable source of music chart information is not freely available (Neilson SoundScan [10]), and there was not an alternative free source, we had to collect our own data. To this end, we researched which radio stations Nielsen SoundScan monitors to create the Billboard Top 100 (freely available at [10]), checked each of these radio stations to determine if they had a weekly request chart on their website, and mined the resulting 190 radio station's websites for their music charts each week, for a period of six weeks.

After each of the weekly chart files was retrieved, they were entered into a four-table MySQL database. The first table is the *producer* table; every artist that had a song on a chart during the six-week period of data collection can be found in this table. The next table was the *product* table, which contains each of the songs that appeared on a chart, accompanied by the artist who performed the song. The third table in the schema is the *vendor* table. This table was composed of radio station name, code (which is a four character unique identifier assigned to each North American radio station), and location information (state, city, longitude and latitude). The last table in the database schema is the *chart event* table. Each row in the table contains a 'chart event', composed of a song, an artist, a station, the date the chart event occurred, and the chart placement of the song.

3.2 XML Schema and REST APIs

Invenio was designed with the explicit objective of being extensible, so that it can be utilized to visualize other products, such as DVDs, books, technology etc. To this end the XML Schema, representing the underlying resource and supporting the REST API, was purposely created with this extensibility in mind.

The schema is organized around five different REST APIs. The *GetProducers* API returns an XML document with a *producerSet* element as the root. Contained within the root element are an unbounded number of *producerEntities* (or artists) from the producer database table. A *producerEntity* is a "wrapper" element that contains a *producerId* and artist or *producerName*.

The resulting XML of the *GetProducts* API mirrors the result of the *GetProducers* API. The top-most element is a *productSet* and it contains multiple *productEntities*. A *productEntity* contains the *productName* or title of the product, (i.e. the song title) and an identification number for the product, which is used within the chart event table.

In keeping with Shneiderman's Visual Information Seeking principle [11], the *GetGenre*'s API is used to "filter" the number of artists that one can choose from. The schema specification of this API's XML results in a *genreSet* root entity with a number of *genreEntities*, each of which only contains a single element, the genre name.

In the same vein, the *GetVendors* API is specified by a *vendorSet* root element, populated with *vendorEntities* (*vendorId*, *vendorName*, *vendorCode, vendorSRC*, and *vendorLocale)*.

The last REST API, *GetChartEvents*, returns a *chartSet* element, with an unbounded number of *chartEntity* elements. No matter what products are being monitored, whether it is book sales, search results, or movies, there are four 'generic sales' elements, which must be present: the producer, the product, the vendor and the date on which the chart event was captured. A *chartEntity* element captures each of these attributes in the XSD schema.

The Invenio schema provides one with the minimal amount of information necessary to geographically plot and deduce novel products or trend information for a variety of product domains.

3.3 The GIS Framework: Yahoo! Maps

We chose to use the Yahoo! Maps Developer API and Yahoo! Maps Flex SWC (Version 3) as the GIS framework underlying Invenio's geography-based music-trends tracking. We needed to have the ability to recognize the user's interaction with the map and respond to it (by relaying the activity and its parameters to the Flex control listeners), and to create a variety of custom markers for different information-communication purposes (such as artist album covers or multi colored circles to illustrate different metrics). The Yahoo! Maps API supports both of these requirements.

In a typical usage scenario, the user may select a variety of options, such as artist, song and date. Next, Invenio accesses the data resource through the REST *GetChartEvents* API to obtain an XML list of chart events. This list is parsed to obtain the radio stations latitude and longitude for each chart event and the resulting values are passed to the Yahoo! Map for plotting.

3.4 Amazon Associates Web Service

The Amazon Associates Web Service is offered free of charge by Amazon.com, to enable developers to query Amazon's product catalog using a variety of protocols: SOAP, REST and WSDL.

As most of the visualizations in Invenio would benefit from the inclusion of details on-demand, we felt that Amazon, which is one of the largest online retailers, would be the most fitting service to use to obtain this information. Every time an artist's album cover or additional album information was needed, Amazon's REST support [12] was used to make an ItemSearch HTTPService call (to Amazon's Digital Music category of products). Invenio is thus able to retrieve image URLs and additional artist information such as album release dates, prices, and editor's reviews using Amazon's protocol.

3.5 The RIA Framework: Flex

The web interface was created using the Adobe Flex programming environment and language. A relatively new product, released in March 2004, Adobe Flex is a collection of technologies that support the creation of Rich Internet Applications using the Macromedia Flash platform. A typical application is built using either Adobe Flex Builder 3, or the free Flex SDK. As pictured in Figure 2, the Flex architecture and framework is composed of three main components, roughly corresponding to the three layers of a model-view-controller architecture: MXML, ActionScript and a Class Library.

The MXML or 'Macromedia eXtensible Markup Language' is used to declaratively create and manage all of the visual components that are present in the user interface. Each of the components in the MXML application layer is derived from the Class Library. The Class Library contains a plethora of display components (Accordions, Menus, Data Grids, Titled Windows, and Buttons) data types, effects (Zoom, Blur, and Dissolve), and charts that can be instantiated or extended to meet most developers' needs. The Class Library also contains elements that support HTTP service calls, image loading, WSDL calls, and the parsing of XML streams and RSS feeds.

Once an application has been created, it is compiled into a .swf executable and deployed. Using Flex there are two ways in which this compilation and deployment can occur. As Figure 2 illustrates, all of the Flex Application Framework components can be placed on a presentation server. This server contains Flex runtime services which compiles the entire collection of framework components present on the server into a .swf file and then sends it to the client. The alternative to this procedure is to create a .swf file within the Flex Builder then place it on a pre-existing server.

Within the Invenio web interface (.swf file), there are three main components: the display shelf visualizations, the Yahoo! geographic map visualization, and the Flex charting elements.

3.5.1 Display Shelf Components

Popularized by the Apple Mac OS operating system, a display shelf is a horizontal (or less frequently, vertical) visualization of a collection of

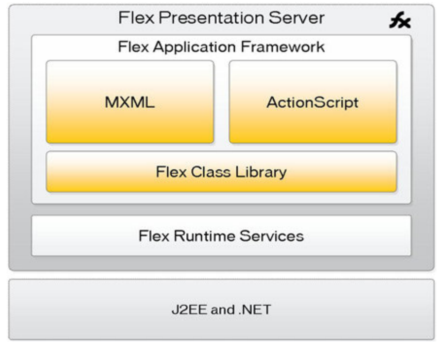

Figure 2 - Flex framework from [13].

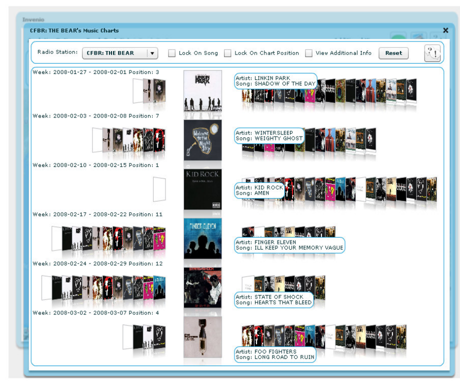

Figure 3 – Invenio display shelf window.

images displayed on animated tiles. The tile that is largest and facing forward is the currently selected tile; all of the unselected tiles are presented in a smaller, semi-profile view (Figure 3). Through the use of keyboard arrow keys, a control component (such as a slider or buttons) or the mouse, a user can "flip" through each tile. Instead of implementing textual tables or charts within Invenio and having a consumer view an endless array of forgettable textual information, the visual prowess that display shelves possess was used with the hope of boosting recall and assisting visually minded people in discovering new music. This unique, visual medium clearly communicates to the user (and music consumer) the temporal movement of an artist's song across many weeks in an innovative, intuitive manner.

3.5.2 Geovisualization

Using the Yahoo! Map and Flex framework, Invenio offers a variety of temporally static and dynamic geovisualizations. Each geovisualization was designed to provide a user with (a) a general overview of the spatial data that was collected,

Figure 4 – A) 'Track By Artist' view, B) 'Track by Success' chart position view, C) 'Track by Success' genre view.

and (b) the opportunity to facilitate understanding by "drilling down" and manipulating the visualizations using Yahoo! Map controls or the provided Flex components. We hypothesize that using a geographic dimension to browse this type of information should help users to discover patterns that they would not normally see in a textual print out of the information or by viewing a large collection of web pages. A user can potentially mentally cluster all the information they see and then create a rich mental model of the data that has been visualized. This mental model can help one to conjure hypotheses and ideas about trends that are occurring over time with respect to new or popular artists, songs or genres.

As depicted in Figure 4A, the first geovisualization depends upon a user selecting an artist, song and date from within the 'Track By Artist' tab in the main application window. Once this has occurred, all of the chart events that correspond to this triplet of information are plotted on the Yahoo! Map. A circular marker is placed at the location of the radio station that created each of the resulting chart events. This initial mapping is intended to help users determine which areas of North America their selection is popular in.

The second mapping visualization, as seen in Figure 4B, is based upon the most and least popular songs on each station's chart. By navigating to the 'Track By Success' tab and selecting a date, users can elect to view the top (or bottom) song for each station on this date. The resulting mapping has the circular markers change into tiles that display the top (or bottom) artists' album cover. The purpose of this second visualization is to assist a user in determining the short-term trends that occur across different regions. Using this information, one can look at the past few weeks of charts to determine if the selected artist has been building a strong national fan base (and steadily climbing up the charts) or if they have a local "cult" following.

The last visualization is designed to help enhance the previous two visualizations by illustrating the long-term trends of different genres (Figure 4C). By visualizing the top or bottom genres for each station, a user can determine which genres are continually producing top songs across all regions and which genres continue to be ranked at the bottom. After selecting a date from within the 'Track By Success' tab, the corresponding chart events are displayed on the Yahoo! Map. The markers that are used for this visualization are circles with fixed radii, whose fill color depends on the genre of the top (or bottom) song that was played on the respective station. Although it is generally thought that most radio stations only play one genre, in looking at the data we obtained, this was not the case; many radio stations had numerous genres within their rotation.

To determine how an artists' song popularity has been changing over time, a user can select the "View all weeks" option in accordance with any of the above visualizations. This option dynamically creates and plays a temporal geovisualization animation of the filtered chart events as they occurred throughout the six weeks of data collection. The resulting dynamic animation can be used to view which geographic areas an artist is popular in or where specific genres are emerging. By observing the changes that occur in marker color and radii, the different album covers that appear and the emergence or disappearance of markers, a user can spot changes, in a much quicker and simpler manner than by flipping between pages in a report or navigating amongst a variety of music sites.

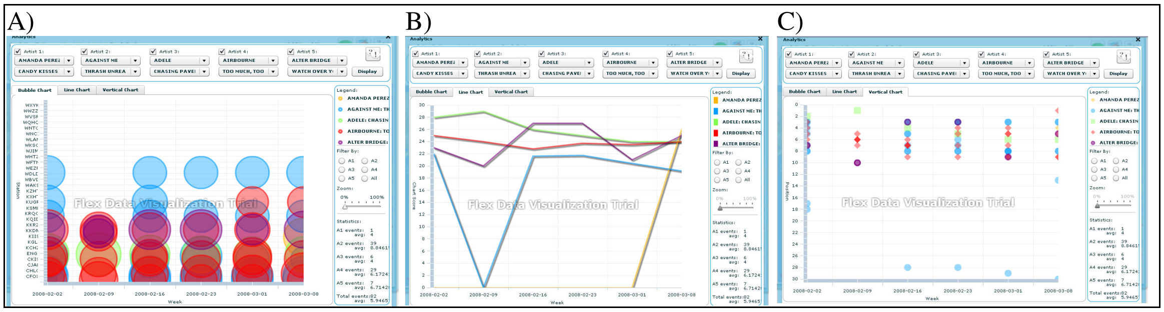

Figure 5 – A) Invenio Bubble Chart, B) Invenio Line Chart, C) Invenio Plot Chart.

3.5.3 Charting

To assist users of the system, multiple charting visualizations have been included in Invenio (a bubble chart, line chart and plot chart). Each of these visualizations are located within the chart window and can help a user to compare the temporal increase and decrease in popularity of different songs. A user can select up to five different songs and compare the movement of the songs against each other.

The bubble chart, seen in Figure 5A, visualizes a song's popularity over time as it relates to different radio stations. By changing the size of a bubble's radius to correlate with a song's chart position, a user can gain a quick overview and general idea about how a song has been embraced by a radio station's audience over time. If there is a small variance in bubble radii, then an artist has a steady audience; if there is a steady increase or decrease in the bubble radii over time, then an artist is either gaining or losing popularity. By using a simple visualization metric such as bubble size, the user is provided with a quick overview which can be further refined using the line and plot charts.

In order to provide a user with more detailed analytics pertaining to the songs a user has selected, a line chart has been created (Figure 5B) to visualize a song's popularity across stations using a 'chart score' metric (the average position of a song across all music charts). By computing this metric for each song across each week, it becomes very easy to compare different songs against each other when the total number of chart events per week may vary across stations.

Invenio's last chart view, as found in Figure 5C, is a plot chart, and its purpose is to provide an enhanced level of visual analytical data. A plot chart graphs a number of data series' using data points instead of bars or lines. This plot chart is used to view the variance that occurs between a song's chart placements. A user can quickly browse the graph's data points and determine if a song's chart positions are clustered together or if they have a large variance. A song that has a small variance indicates that the song had a steady rise or fall to/from popularity, but a larger variance indicates that an artist's popularity is being influenced by an external force, such as a recent concert, album or promotional campaign. As well, a user can use the graph to easily determine how many times a song was placed at position 'X' by looking at the shade of the point's marker: the darker a marker, the more events occurred at that position. Using this last chart view, a user can quickly and easily determine how fragmented a song's popularity is and track this fragmentation over a period of time.

4 Workflows

Because Invenio has the potential to be used by a wide variety of users, it implements several alternative workflows tailored to the information needs of these various potential audiences.[1] Regardless of which workflow is invoked, all the information that Invenio utilizes is dynamically retrieved at run time. The core retrieval APIs that Invenio makes to its web service are issued in the form of HTTP GET requests. They are illustrated in Figures 6 and 7 and have already been discussed in Section 3.2.

One should note that the three workflow categories we have identified for Invenio exemplify the general categories that applications examining

[1] The interested reader can see a viewlet demonstrating the workflows at http://www.cs.ualberta.ca/~stroulia/660/Winter2008/animation.swf

different products or services typically support, namely (a) consumer-research workflows, (b) product-history workflows, and (c) producer-executive workflows.

4.1 Consumer Research workflow

Because products and services such as videos, music, consulting and business research have become borderless commodities, utilizing a geographic representation of a product's popularity may provide a consumer with an alternative way of discovering new or forthcoming products. Within Invenio, the most beneficial user interactions that support this need come via the Yahoo! Map (which is located within the main application window). In the main application window under the 'Track By Artist' tab, five different GET requests can be invoked by a user's actions.

The first request is issued while the web service is initializing. To initially populate the producer's combo box with values, Invenio issues a *GetProducers* request to the web service. The result of this request populates the combo box with all the artists present in the producer database table. If a user so chooses, they can filter this large list of artists by selecting the 'Filter By Genres' check box. Then, Invenio issues a *GetGenres* request to the web service to retrieve a list of all the genres present within the product table. If a user selects one of the resulting genres, another *GetProducers* request is issued and a filtered list of artists whose songs fall within the selected genre populates the artist combo box. Selecting an artist from the artist combo box invokes a *GetProducts* request to the web service. The resulting XML populates the song combo box with a list of songs that were performed by the artist selected in the artist combo box. After a user has selected a date from the date chooser and clicked on the 'Display' button, Invenio issues a *GetChartEvents* request to the web service. Upon completion, this request returns a filtered list of chart events and displays them on the Yahoo! Map.

The selection of the second tab of the main application ('Track By Success') causes a different tab of options to appear. Within this tab, a user can select a date (using a date chooser), click on one of the chart event filtering options and then select the 'Display' button. Just as in the first tab, the selection of this button causes Invenio to request a filtered set of chart events, and plot them on the Yahoo! Map.

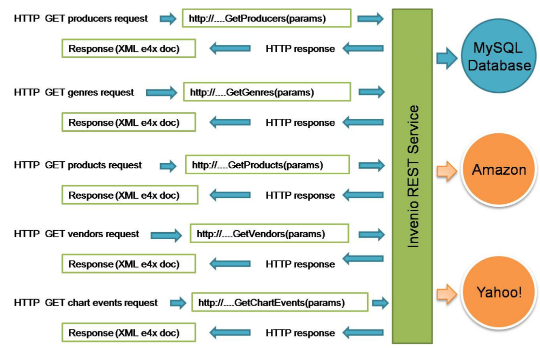

Figure 6 - Invenio MySQL Requests.

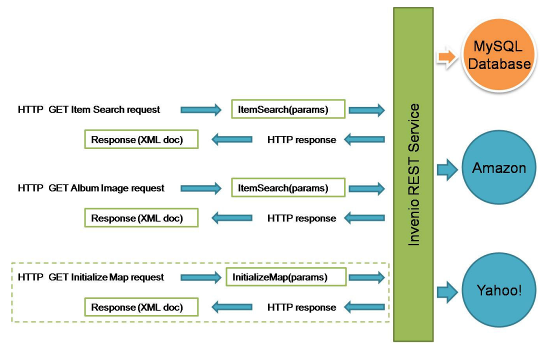

Figure 7 - Invenio Amazon and Yahoo! Requests.

It should be noted that an alternative to the date chooser method of selecting a date in both tabs is to select the "View All Weeks" checkbox. This action will return filtered results for all of the weeks of data collection and display a temporal animation of the chart events on the Yahoo! Map.

As interactivity was a very important feature of Invenio, all of the resulting Yahoo! Maps respond to user interactions in many ways. First off, whenever a user hovers over a marker (and the "View Additional Information" checkbox has been selected), Invenio issues an *ItemSearch* request to Amazon to retrieve the album cover of the song, the release date, price, number of Listmania! lists on which the album appears, and an Amazon editor review. The resulting information is displayed in a pop-up window within the Yahoo! Map. Another type of interaction involves the selection of a Yahoo! Map marker, which causes the display shelf window to open and a *GetVendors* call to be issued.

4.2 Product History workflow

When looking at the history of a product, one is usually concerned with temporal increases and decreases in its popularity, utility, gross margin, replacement rate, etc. Within Invenio, one can look at a product or song's temporal popularity using the display shelf window. This window can be opened in two ways, each one involving different types of user interactions (and subsequent GET requests).

If a user has elected to use the 'Display Shelf Window' button to open the display shelves, then the creation of this window invokes a *GetVendors* request. The result of this request populates the station's combo box with a list of radio stations that they can choose from. Once a user has selected a station from this combo box, a *GetChartEvents* request is sent to the web service. The result of this request is a collection of chart events that occurred on the selected radio station. After these events are obtained, a *GetAlbumImage* call is issued for each artist in the chart events set. The result of these requests populates the display shelves with the appropriate artist album covers.

If a user has clicked on a Yahoo! Map marker all of the above requests are made, except for the *GetVendors* request (as this information is already available via the marker that the user has clicked on).

The last type of API request that is invoked is the *ItemSearch* request, issued each time a user clicks on a display shelf tile. The results of this request populate a pop-up window with information pertaining to the selected artist.

4.3 Producer Research workflow

Producers are extremely concerned with monitoring the successes and failures of their competitors. Although a user can compare different products, or songs, quite easily by interacting with the Yahoo! Map and display shelf window, the most informative data for this audience can be found via their interactions with Invenio's charting components.

Within the charting window, there are only a few interactions that take place. Once a user clicks on an 'Artist X' check box (where X is limited to 5), Invenio issues a *GetProducers* request to the web service and the resulting artists are added to the relevant artist combo box. Just as within the main application window, the selection of an artist from an artist combo box results in a *GetProducts* request. After the request returns a list of songs, they populate their respective combo boxes and are available for a user to make a selection. Once a user has selected their desired number of artists and corresponding songs and clicks on the 'Display' button, Invenio makes a *GetChartEvents* request to the web service. The resulting chart events are then converted into chart series' and are displayed on the three different chart visualizations present in Invenio. The rest of the interactions a user can make within this window (filtering which series are displayed, hovering over nodes, zooming in/out) do not cause Invenio to issue any other requests to its server, unless one is switching their artist or song combo box selection.

We should note here that each of the HTTP GET requests made to the data resource returns a special-purpose format of XML, called e4x (ECMAScript for XML). Using e4x, an XML document is not treated as a DOM object and as such does not require lengthy accessor functions or specific XML libraries to parse or create it. Instead, XML is treated as a set of primitive objects that can be easily created and traversed using dot (.) or attribute (@) operators.

5 Development Practices

In this section, we describe the specific requirements that we had established for Invenio, the technology choices that were available to us while developing our application, the decisions that we made regarding its design and development, and some lessons and recommendations that we believe may be useful to others who engage in similar endeavors in the future.

5.1 Requirements

Because Invenio is a rather complex service with numerous components and visualizations, there were many requirements that needed to be considered before the implementation of Invenio could begin.

Due to the large amount of data present within Invenio (14,000 chart events), one of our requirements was that it needed to be able to aggregate and accommodate a large quantity of data and be able to obtain a small subset of this data dynamically and on-demand. Although Invenio's data was only collected for a six week period, the data storage medium needed to be able to store data that could potentially be added at a later date, and also be able to manipulate its structures fairly easily to accommodate new domains or types of data.

In line with this, we needed to be able to retrieve the data from storage in a manner that would allow for low user-perceived response latencies. If a user had to wait for data to appear, they could become frustrated and navigate away from our web service. Our mechanism of component intercommunication also needed to be very rapid and have zero data loss while being transmitted. If the data received from storage was incomplete or missing, this could result in malfunctions in the web service, which would lead to a decrease in user experience and satisfaction.

Another requirement that was inherent within Invenio was the necessity of a geographic visual representation of the data. This geographic visualization needed to be in the form of a highly interactive and customizable map. Because one of our goals was to handle runtime user choices and selections in effective, interesting ways, we also needed the map to respond to dynamic sets of data that were modified at runtime. It would not be sufficient to have a static map visualization of only one artist or song; we needed to allow the users to change their selections quickly so as to compare the results efficiently. As well, we needed to be able to allow users to overview, filter, and zoom their selections using very easy to learn techniques contained within a highly intuitive interface.

Because there were many different target audiences with varying levels of skill sets that could be using Invenio, there also needed to be many, different, highly interactive and visually appealing visualizations. We strongly believe – although we have no formal usability proof in support of this claim – that static, non-animated representations of data would not be sufficient for this web service to be effective; instead, we needed to find novel, unique ways to represent the data in ways that appealed to a large number of potential audiences. We hypothesized that creating representational mediums such as charts and display shelves would greatly increase a user's experience and also allow us to utilize external APIs and services to aggregate even more data within Invenio.

Because Invenio is a prototype application, one of our other requirements was that it should be relatively easy to add and remove components to and from the service. Due to this constraint, we needed to find a way in which the communication between the different components in our program (i.e. data, web service, geographic map, charts, display shelves, Amazon data and user interaction) could easily be extended or modified as needed, using the simplest protocols possible.

As well, we needed to find a framework or programming language that had many highly visual components (such as buttons, sliders, combo boxes), that could be used with a variety of different web browsers and operating systems, and would support the creation of a visually rich, intuitive application interface.

5.2 RIA Framework Choices

Given the project requirements discussed in Section 5.1, one of our biggest challenges was determining what type of RIA framework to use. Currently, there are three rather popular frameworks: Adobe Flex (described in Section 3.5), Open Laszlo [14] and SmartClient [15].

5.2.1 OpenLaszlo

OpenLaszlo is an open source web service framework that was created by Laszlo systems in 2001. One of its major selling features is its "write once, run everywhere" mantra. An application developed using OpenLaszlo is compatible with all web browsers and operating systems, allowing one to author their program on one system, and have a 'guarantee' that it will be compatible on all others. OpenLaszlo is actually very comparable to Adobe Flex. As depicted in Figure 8, OpenLaszlo is composed of two main components, the LZX programming language (akin to MXML and XAML) and the OpenLaszlo web server. The LZX language utilizes XML tags to create OpenLaszlo components (just as MXML does within Flex), which support rapid prototyping, long-term code maintenance, and collaborative software development. The LZX library contains a plethora of components, debugging properties, animation capabilities, data binding mechanisms, and web service support (XML, SOAP, XML-RPC and JavaRPC).

Typically, an OpenLaszlo program is authored using either an Eclipse IDE plug-in or a text editor (just as Flex, with the Flex Builder or SDK support). After the program has been written, it is passed to the OpenLaszlo web server. This web server is a Java or J2EE servlet that compiles LZX applications into a DHTML executable binary. This executable binary is then viewed in a web browser using DHTML. The one downside to this approach is that one needs to install a Java or J2EE servlet with the appropriate OpenLaszlo servlet extension in order for this DHTML compilation to work. If one already has a web server installed then they can choose to compile their OpenLaszlo program into a Flash .swf file and view it in a web browser using Flash player (just as Flex does). Although this second method of compilation can be useful, it does have one major flaw: one cannot consume SOAP web services or use XML remote procedure calls. While both methods of compilation can be useful, it seems that their pitfalls could cause a potential user to switch to another framework such as Flex or SmartClient.

5.2.2 SmartClient

Although not as widely used as Flex or OpenLaszlo, SmartClient is a similar framework, which is not completely open source. Distributed by Isomorphic in 2000, it uses DHTML and AJAX to create smart, client-side SOA clients for web applications. As illustrated in Figure 9, the SmartClient architecture was designed with a very modular style in mind; its SDK comes with a variety of services, data components, implementation blocks and GUI components. For a fee, one can download a variety of visual creation tools (builder, runtime console, component editor, admin console) and the SmartClient Server.

Of most interest to developers is SmartClient's focus on MVC architecture, making authoring applications across multiple developers a simple task. Specific constructs were built into the SmartClient API to support and encourage individuals to utilize this architecture, no matter how small or large their web service is. SmartClient also provides one with a seamless integration with RSS, REST, SQL, SOAP messages and XML namespace handling.

As with Flex, one uses XML tags to create, modify and populate a variety of visual and foundation components, such as grids, trees, buttons, and scrollbars. Just like Flex and OpenLaszlo, SmartClient provides support for a variety of web service protocols such as WSDL and REST (in either XML or JSON) One interesting difference between the two previously discussed frameworks is the fact that the deployment of SmartClient applications is not server specific; SmartClient can run off of a J2SE Container, or Spring, Hibernate, JSF or Struts. While this does make life on

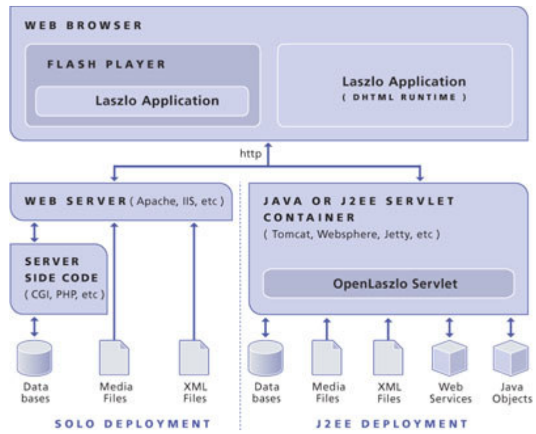

Figure 8 - OpenLaszlo Architecture [14].

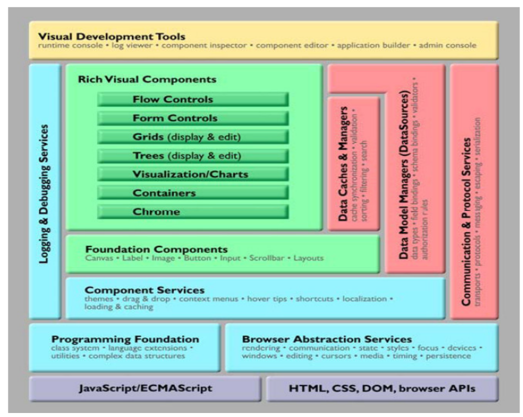

Figure 9 – SmartClient Architecture [15].

the client side easier (because there is zero-install for programs such as Flash), it does make things more complicated for the developer on the server side, especially if they are new to web service development.

In looking at these three frameworks, it appears that they each focus on one of two things: either providing a method to organize and aggregate data well, or providing a rich interface design experience. SmartClient is an example of the former approach with a web services organization, with its multiple server options and MVC architecture; it almost appears that interface design is nothing but an afterthought. Opposite to this is Flex, which provides seamless integration with external APIs, components and flashy interfaces but places less of an emphasis on architecture organization: components can be created in either MXML or ActionScript, and there is little emphasis on data aggregation or organization. Unlike SmartClient and Flex, it appears that OpenLaszlo is focused on creating a happy medium for developers, weighing organization and interface design equally by providing multiple deployment methods and a rich interface library.

5.2.3 Rationale for Choosing Flex

In deciding which of these frameworks to use, we looked at our specific requirements and came to the realization that we needed a framework that would support a relatively simple source of data, but would be able to create a remarkable user experience through the use of sharp, innovative visualizations and user interface design. External data integration and deployment methods were also major concerns.

Looking at the examples of SmartClient applications, it appeared that SmartClient's GUI was very "clunky" and bloated and the integration of a geographic map would be a very time consuming process. As well, the applications did not suggest that SmartClient was geared towards visualizations, animation or charting, instead they made SmartClient appear to be a glorified form builder – which did not fulfill our objectives with Invenio. The extension or customization of components appeared to be more complicated than necessary, making the communication between components such as the geographic map and the display shelves unnecessarily complex and unwarranted. As charting was an important part of our web service, the flexibility and user interaction capabilities that were provided by the application would not meet our needs and we did not have the extra development time that would be needed to bring them to the appropriate level. In our opinion, zero-install clients are desirable; however, this "positive" feature of SmartClient is not necessarily decisive. According to the Milward Brown survey conducted in March 2008, 98.8% of Internet-enabled PCs use Flash [16]. Given this statistic, the benefit of SmartClient's zero-installation client does not outweigh its potential problems on the server end.

Although OpenLaszlo does appear to try and integrate both component organization and user interfaces equally, a further inspection into their components did not prove to be promising. The interfaces that we saw as examples were lackluster. The components were poorly designed, the graphics were unimpressive and even with the addition of skinning or themes, the interfaces did not have much fluidity. Because Amazon and Yahoo! Map integration appeared next to impossible and the creation of custom components such as charts and display shelves did not appear to be supported, we opted to not use OpenLaszlo. Because we would need to use the Java / OpenLaszlo server to utilize external data sources such as Amazon and our Invenio Web Service (XML HTTP calls are not supported in the compiled SWF deployment option), we would have to use JDBC to access our database, and we felt that this was unnecessary and would greatly complicate our work. Instead of setting up a separate J2EE web server and using JDBC, we could use Flex and create a PHP-based web service that would use HTTP GET requests to obtain our data, all of which could be set up in a fraction of the time that OpenLaszlo's method would require. Given the unnecessary work that would be needed and the unintuitive nature of the OpenLaszlo components, the usage of Flex seemed to be the right choice for our web service.

Another factor that played a part in our decision process that was only tangentially related to our project requirements (to the extent that we had to meet development deadlines) was the amount of support and help available to the developer. As this was a first time prototype design, it was vital that support be easily available to consult and that the integration of all the external API's and custom components be as seamless and fluid as possible. Once again, Flex proved to be the solution for us. Flex has a vast quantity of user submitted examples, cookbooks and tutorials that make it a very easy for a new developer to get started with their framework in a very short period of time.

5.3 Recommendations Checklist

The decision about which frameworks and external sources to use, how to combine them and how to design them depends upon a variety of factors:

(a) Intent behind the RIA service

Web services that are designed to be information providers need to utilize frameworks that support information visualization, either graphical or textually, in a clear manner. Although data storage is important, one should place more value on frameworks that create interfaces with low cognitive load and protocols that allow large amounts of data to be easily aggregated. Information-gathering services (i.e., use forms and search engines) should look towards frameworks with protocols that encourage low latency and quick response rates. Such services are typically designed with future extensibility in mind and as such, protocols and frameworks should be designed so that the aggregation and search ability of new, up and coming information sources can easily be incorporated into such services.

(b) Types of data visualization needed

Static visualizations of data are currently very easy to create and render, but the components utilized to create these displays should be extended to promote user interaction and knowledge discovery. A static 2D line chart of a product's success is no longer sufficient; instead, users want to view visualizations that they can interact with, "drill down" into, and view additional external information about. Services that heavily depend upon dynamic visualizations need frameworks that can support substantial quantities of data and rapid user interactions and manipulations. Such visualizations need to be supported by mechanisms that allow users to obtain overviews of the data being displayed and then facilitate deeper exploration and discovery. Currently, the RIA frameworks that are used in practice do not support these ideas to the desired degree. The current frameworks are slow to react to user interactions, have long pre-loaders and require data to be broken down into very small subsets in order for rapid transmission to occur.

(c) Deployment contexts required and desired

The target audience of one's web service plays a major role in the deployment contexts that one requires for their service. Users in the 20-30 year old demographic generally use different operating systems and demand rich visually appealing interfaces. Providing this group with a visually watered down service so as to support zero-installation principles would create a mismatch between user expectations and delivered product features. In other cases, such as global web services, one's target audience spans a wide range of demographics, making the types of deployment one uses hard to determine. In this case, using a framework that supports multiple deployment mediums may prove to be effective. Web services designed for educational purposes are typically used by school-aged children familiar with only one operating system (that usually does not have administrator install privileges). Creating a web service that provides a zero-installation client would be beneficial for this group; a multi deployment framework would be wasteful.

(d) Extent of component intercommunication

It is fairly intuitive that the more components (either GUI or external API created) one has, and the more user interaction present in an application, the larger and more complicated its inter-component communication demands will be. Web services that are highly visual and graphical, as opposed to those that are information gatherers, benefit most from unstructured frameworks. Choosing a framework which has little to no imposed structure can be a benefit for programs with externally created components, such as those using JavaScript to relay user interactions to a web service, or those requiring validation from an external server or source. Under these conditions it can be very easy to create and extend custom user and component interactions. On the other hand, structured frameworks which have built-in architectures are a perfect choice for those services which need to be duplicated multiple times, need to follow specific protocols, and deal with sensitive information. Services such as online stores, banks and email providers are excellent examples. Without a standard mechanism of user interaction, it would be very difficult to add new features to a service or repair damaged or vulnerable components of a service process.

(e) Amount of unnecessary customization needed to compensate for API or framework insufficiencies

Of the lessons we learned, this was one of the most valuable. Frameworks that do not support component or protocol extensibility are practically useless. Given the popularity of mashups and data source integration, frameworks should be designed and chosen so as to facilitate the integration of other programs or information; if integration appears too complicated, a developer would and should choose another framework or protocol. The more "workarounds" and "hacks" one needs to create to compensate for missing features and functionalities, the less valuable and helpful a framework is. By creating frameworks that support and encourage component extensibility, developers encourage individuals to create novel mediums of communication, interaction and visualization, which could be incorporated into further releases, pushing the success and popularity of RIA frameworks forward. Not only will less code be needed in the end, but developers will not have to spend hours tinkering with SDKs or APIs trying to perform simple component customizations. The less time spent tinkering, the more time is available to tackle more challenging problems such as data retrieval, manipulation, and visualization.

(f) Importance of data source organization, data visualization, and user interaction

This factor became the most apparent when researching the differences between each framework (Section 5.2). On the surface most of the frameworks in use today appear to be quite different (albeit with differences in deployment and feature support), but in reality they all have one of two main focuses which are mediated by user interactions: data organizer, or data/interface visualization. There are a variety of examples provided for all frameworks that describe how to organize data or how to visualize it using components or interfaces, but there is very little information pertaining to the appropriate ways to fuse and utilize these two ideas together, and even fewer explaining where user interaction should fit in. How does one design their data sources to promote visualization? How does one create complex, interactive visualizations from simple or aggregated data sources? Where does user interaction fit in? All of these questions need to be considered and answered.

(g) An attempt at "best practices"

To summarize and consolidate the lessons we learned through our experience, we formulated the following recommendations:

1. If possible, design visualizations first. It is easier to break down a complex display into a variety of meaningful parts and populate them, than it is to tease meaning out of a complex data source. If, for some reason, the visualizations must come last, one should look first for broad patterns or themes, and then drill down into the complexities.
2. The simplest possible data source or database representation should be designed. All unnecessary data from the basic resources should be filtered away. Complex sections should be broken down into easily pluggable components. As much information as possible – high in quality, low in quantity, should be stored.
3. The design should ensure that users are able to overview, filter and zoom through visualizations or interfaces where appropriate. Information should be revealed on-demand to the user; the more user interaction, the more novel, detailed information should become available.
4. The design should aim to decrease the latency between a user's actions and the service's response. No one is willing to be kept waiting for feedback about their decisions or choices.
5. Viewing the same information should be possible using multiple mediums or methods. Charts, graphs, graphics, geovisualizations (where appropriate), media and interactive components can reveal different information to different users.
6. In RIAs, user interaction with the data should not be an afterthought. The web service being built is for the user; appropriate accommodations should be made to design interfaces and visualizations that will be intuitive for your target audience's skill set.

Given the fairly small amount of research that has gone into best practices for large-scale RIA design, our experiences with the creation of Invenio should be a valuable starting point for further exploration into this area.

6 Conclusion

As most new web technologies fall under the Web 2.0 umbrella, one would expect to find an abundance of documentation or user experience literature detailing how a developer can easily mash a variety of these technologies together. However, through our experiences in authoring a data-intensive RIA, entitled Invenio, we discovered that a very small amount of literature pertaining to this topic actually exists. To this end, we have developed and reported a list of "best practices", or guidelines to use as a launching pad for future research in this area.

Invenio was designed and instrumented utilizing a complex combination of Web 2.0 technologies such as Adobe Flex, the Amazon Associates Service and Yahoo! Maps. This paper has reported on the functionality, system design and technology choice rationales behind Invenio, with the hope that it's descriptiveness can assist others who are developing large-scale Web 2.0 applications.

About the Authors

Michelle Annett received her Bachelor's degree from the University of Alberta and is currently studying there as a master's student. She is interested in developing virtual reality applications and systems to study human spatial navigation and wayfinding. She can be reached at mkannett@cs.ualberta.ca.

Eleni Stroulia, http://www.cs.ualberta.ca/~stroulia holds M.Sc. and Ph.D. degrees from the Georgia Institute of Technology and is an Associate Professor with the Department of Computing Science at the University of Alberta. Her research addresses industrially relevant software-engineering problems with automated methods, based on artificial-intelligence techniques. Her team has produced automated methods for migrating legacy interfaces to web-based front ends, and for analyzing and supporting the design evolution of object-oriented software. More recently, she has been working on the development, composition, run-time monitoring and adaptation of service-oriented applications, and on examining the role of Web 2.0 tools in enabling the practices of collaborating communities.

References

[1] Celebrity gossip juicy celebrity rumors Hollywood gossip blog from Perez Hilton. http://www.perezhilton.com. June 2006.

[2] U.S. Album Sales Down 9.5%, Digital Sales Up 45% in 2007. http://www.dmwmedia.com/news/2008/01/03/u.s.-album-sales-down-9.5%25%2C-digital-sales-45%25-2007. March 2008.

[3] Album Sales Slump as Downloads rise. http://www.usatoday.com/life/music/news/2006-01-04-music-sales-main_x.htm. March 2008.

[4] Nielsen SoundScan Releases Year-End Sales Data. http://www.billboard.biz/bbbiz/content_display/industry/e3iXZLO0IdrWuAOeIRwz3vtYA%3D%3D. March 2008.

[5] Wood, J., Dykes, J., Slingsby, A., Clarke, K., Interactive Visual Exploration of a Large Spatio-temporal Dataset: Reflections on a Geovisualization Mashup, *IEEE Transactions on Visualization and Computer Graphics*. pp. 1176-1183. November/December, 2007.

[6] Kwan, MP., Lee, J., Geovisualization of Human Activity Patterns Using 3D GIS: A Time-Geographic Approach. *Spatially Integrated Social Science*, Michael F. Goodchild and Donald G. Janelle (eds). New York: Oxford University Press. 48-66. 2004.

[7] TubeMogul.com – Online Video Distribution and Analytics. http://www.tubemogul.com/. March 2008.

[8] QLikView Musiq Tracker. http://demo.qliktech.com/qlikview/AJAX/MusiqTracker/. March 2008.

[9] Brownstein, J., Freifeld, C., Reis B., Mandl K., HealthMap: Internet-based emerging infectious disease intelligence. *Infectious Disease Surveillance and Detection: Assessing the Challenges - Finding Solutions.* Washington, DC. pp. 183-204. 2007.

[10] Neilson BDS Stations. http://www.bdsonline.com/stations/index.html. January 2008.

[11] Shneiderman, B. The Eyes Have It: A Task By Data Type Taxonomy for Information Visualizations, *Proc. IEEE Symposium on Visual Languages '96*, IEEE, Los Alamitos, CA. pp. 336-343. September 1996.

[12] Amazon Web Services @ Amazon.com. http://www.amazon.com/E-Commerce-Service-AWS-home-page/b?ie=UTF8&node=12738641. January 2008.

[13] Adobe – Flex Development Center. http://www.adobe.com/devnet/flex/. January 2008.

[14] Deployment Architecture | OpenLaszlo. http://www.openlaszlo.org/deparchitecture. April 2008.

[15] SmartClient Overview. http://www.smartclient.com/technology/basics.jsp. April 2008.

[16] Adobe – Flash Player Statistics. http://www.adobe.com/products/player_census/flashplayer. April 2008.