Cloud Computing And Equal Access For All

T. V. Raman
Google Inc.
raman@google.com

ABSTRACT

Web-2.0 applications use the Web as a platform for delivering end-user applications. This transformation has a profound impact on how applications are authored, deployed and consumed. Software applications in this environment are no longer monolithic — instead, they are naturally separated into distributed components that implement application and interaction logic. The application logic along with user data resides in the network cloud; the user interface made up of presentation and interaction logic is delivered in a form best suited to the user's needs, *e.g.,* via a universal client such as a Web browser.

The advantages of this usage/delivery model for mainstream users has been widely explored in the last 18 months. This keynote focuses on the impact of this transformation on users with special needs. Today, the potential for universal access presented by applications delivered via the Web remains largely unrealized. This is partly due to the impedance mismatch that results from trying to treat interactive Web applications as static Web documents. Eliminating this impedance mismatch requires innovation at all levels of the technology stack with respect to:

- How Web applications are authored and deployed,
- How Web applications are consumed,
- How Web interaction is augmented by adaptive technologies.

This keynote will describe some of these challenges and the accompanying opportunities.

1. INTRODUCTION

The Web was originally designed as a global hypertext system for the interchange of electronic documents. But in comparison to their print counterparts, electronic documents are interactive. The interaction facilities initially built into HTML have been augmented over time by the addition of a universal scripting language (JavaScript) and a programmable Document Object Model (DOM) that have together helped realize the early vision of *"The Document Is The Interface"*. But the basic architecture of the Web continues to be underpinned by the following:

HTTP A simple stateless protocol for client-server interaction.

URL A universal means for addressing resources published on the Internet.

HTML A hypertext markup language for authoring content and to aid in bringing together content addressable via URLs.

As Web documents evolve into applications, the distributed nature of the Web necessitates a clear separation between application and interaction logic. This is because application logic and user data mostly reside in the *Web cloud* and manifest themselves in the form of tangible user interfaces at the point of interaction, *e.g.,* within a Web browser or mobile Web client.

This separation has enabled newer Web applications to separate the form of the underlying data being manipulated from its representation within the user interface. As an example, Google Calendar allows users to access and maintain their calendar on the Web. By separating user data (in this case, the user's list of appointments) from a given user interface, *e.g.,* a table displaying the user's appointments, this calendar application can now manifest itself in a form that is best suited to the user's needs. Thus, Google Calendar provides distinct desktop and mobile interfaces, both of which manipulate the same underlying calendar. Notice that an immediate consequence of this design is automatic synchronization of the user's data across the various clients with which the user might access the calendar.

A not so immediately obvious follow-up consequence is that it no longer matters as to which specific client the user chooses for accessing the calendar — as long as the accessing client is capable of manipulating the single underlying representation. The key advantage here is that people that one works with need not be aware of the calendar client being used — notice that this brings Web applications like Google Calendar on par with email — typically, reading an email message does not require one to ask the sender what email client he or she uses.

2. CREATING WEB APPLICATIONS

W4A2008 - Keynote April 21–22, 2008, Beijing, China. Co-Located with the 17th International World Wide Web Conference.

This section sketches out what is involved in creating Web applications as a precursor to outlining the accessibility challenges and opportunities presented by this evolution.

Creating Web applications that deliver application services such as calendaring, word-processing and email involves building software components that can be classified as:

Server Components that implement application logic and manage user data.

Client UI components that *bind* to this underlying application data-model.

The rest of this paper will focus on the components that implement this UI *binding.* The *binding* and *interaction* layers form the crux of the final user experience and are the natural focus when it comes to creating user interfaces that match a given user's needs and abilities.

The complexity of a given Web application determines its underlying data-model — and this in turn determines the *binding* and *user interface* layers. Notice that a sophisticated visual representation does not necessarily mean a complex application data-model — to make this concrete, consider the following examples:

Maps Online map applications such as Google Maps provide a rich user interface for browsing and navigating maps. But the underlying application data-model is very simple — the core of the data-model encapsulates map coordinates of a given location. This is augmented by additional services such as geo-coding used to map human-readable addresses to machine-consumable coordinates. User operations such as:

- Zoom in, zoom out
- Get directions,
- Scroll,

are exposed as a set of verbs, and the result of these operations is an updated map view.

Spreadsheet In contrast, the underlying data-model of a spreadsheet application is a lot more complex. The visual presentation is a table displaying a range of cells from the spreadsheet. But the underlying data-model captures the relations among the cells, rows and worksheets making up a given spreadsheet. User operations are also correspondingly complex, and map to verbs such as:

- Add a work-sheet,
- Insert a row or column,
- Edit a cell-value.
- Add a dependency relation,

Consequently, the user interface has to provide the necessary affordances to enable user-level access to the various operations that are available.

Web applications use HTTP as the underlying protocol for connecting the user interface with the application components implemented on the server. In the case of simple data-models such as the one described for maps, this communication reduces to a set of name-value pairs that encapsulate the user's request, *e.g.,* start and end locations when obtaining map directions. The server's response comes back in the form of an interactive web page that holds the relevant content, along with event handlers that implement the right interaction behavior — see **??**.

Sophisticated applications such as spreadsheets or Google Earth use hierarchically structured data — as an example, many Google applications use ATOM Publishing Protocol (APP) layered on HTTP to connect the client to the underlying application. In this usage model, application data is exchanged using ATOM feeds, with standard HTTP verbs — GET, POST, PUT and DELETE — mapping to the basic operations exposed by the application. Notice that everything that has been described with respect to the binding layer is completely independent of the type of user interface being delivered.

3. TANGIBLE USER INTERFACES

Web applications come to life when the underlying data-model is connected to a tangible user interface. When rendered to a Web browser, the user interface is instantiated by constructing an appropriate HTML Document Object Model (DOM). The HTML DOM holds the content to be displayed to the user, along with the code needed to encapsulate interaction behavior. In this context, *content* consists of traditional document constructs such as paragraphs, lists and tables — but with the difference that all of these can be dynamically exposed, hidden, or updated during the course of user interaction. Interaction behavior takes the form of event handlers that are attached at appropriate points in the DOM. Thus, the Web user interface comprising of actionable UI widgets embedded within information that is being presented is ultimately realized as a dynamically updating hypertext document.

This end-result has many of the features of traditional HTML documents including CSS-based styling of declarative markup. This similarity to traditional hypertext documents also leads to some confusion, in that it is tempting to treat Web user interfaces as *documents.* A Web application *snapshot* that captures the state of the user interface at any given point in the interaction can be *serialized* as a document; notice however that from the perspective of end-to-end interaction, a Web application *is not* a document. Said differently, "The document *is* the interface — but that does not make the interface *a document*".

4. CONSUMING WEB APPLICATIONS

Web applications are *consumed* on the client by interacting with the result of rendering the user interface served up as a Web page containing embedded user interface elements. Mainstream browsers often need to be augmented with adaptive technologies such as screenreaders and screen magnifiers to make them usable for visually impaired users.

This mode of end-user interaction has also highlighted the impedance mismatch that exists between traditional desktop adaptive technologies and Web applications. The situation has been made worse by the fact that the Web application model described in the previous section was not designed but discovered — *i.e.,* we started with a document-oriented Web that has over time come to acquire the features needed to implement end-user applications. Adaptive technologies have typically trailed, rather than led this evolutionary process — as an example, adaptive technologies still try to treat most of the Web as *documents* rather than as *applications*.

A direct result of this is evinced by the fact that until recently, screenreaders had a very difficult time handling JavaScript powered Web pages. However, the situation is beginning to change as we start evolving the rest of the AT stack to align with the world of Web applications. An important step in this alignment is the development of W3C ARIA. W3C ARIA defines a set of DOM properties that can be used to implement *reflection* within Web applications as a means of allowing adaptive technologies to query the *role* and *state* of UI elements that appear in the HTML DOM. W3C ARIA also goes farther than traditional desktop accessibility APIs by defining *live regions* — a light-weight mechanism that allows the declaration of observer/observable relationships between portions of the UI.

Emerging support for these DOM extensions within mainstream browsers, and corresponding support within screenreaders and self-voicing browsers has now created the foundation for discovering design patterns that work for users with special needs.

5. USABLE UI PATTERNS

W3C ARIA as outlined in the previous section brings in-page Web widgets such as scroll-bars and menus on par with traditional desktop widgets when viewed from adaptive technologies. However, there is more to using a Web application than interacting with the individual controls making up the user interface — end-to-end usability of an application is finally determined by how effectively and efficiently the user is able to complete a given task. As an example, an email interface whose menus and sliders fail to raise the relevant platform-specific accessibility events can appear as a *black-hole* to adaptive technology — in this sense, adding W3C ARIA support to these interaction widgets is a necessary requirement. However, this in itself is not sufficient to ensure that the entire application becomes usable.

Meeting the sufficiency requirement above requires us to step back and ask how user interfaces have been handled by adaptive technologies, independent of the specific interaction environment used to deploy a given application. Looking at one specific class of adaptive technologies — screenreaders for the blind have always included application-specific customizations to make a given application usable. This form of customization predates today's GUIs, and can be traced back to the early screenreaders that ran on DOS machines in the late 80's. Such application-specific customizations have been variously called:

IBM User *profiles.*

Vocal-Eyes User *set* files.

Window Eyes User *set* files inherited from above.

JAWS Application-specific scripts.

ORCA Application-specific Python extensions.

Emacspeak Application extensions and task-oriented Web wizards.

Such application extensions typically implement the following:

- The logic needed to automatically speak relevant updates to a portion of the user interface.
- Augment visual elements such as icons with relevant metadata that is found to be missing.
- Add special navigation keys to enable the user to gain immediate *random* access to distinct portions of the visual interface.
- Introduce additional user-level commands that allow the user to query for specific items of information.

Commercial screenreaders bundle such extensions for major applications. In this context, the arrival of Open Source screenreaders opens up the possibility of a world where such extensions can be developed by application authors, screenreader developers and end-users coming together — it may well turn out that such extensions are much easier for application authors to create given an open API to the speech layer as managed by the screenreader.

Coming back to Web applications, we're now transitioning from a world where users worked with a small number of large software applications to one where the required user affordances are delivered at the point where they are needed. Thus, rather than using a single monolithic word-processing application to author *all* documents, users depend on having access to the needed authoring functionality at the point where content is being created. This model of software deployment where end-user services are delivered on-demand has created a world where there are a large number of applications services developed and deployed *across* the Web. The primary accessibility challenge raised by this evolution is then the question:

> Given that Web applications are developed *across* the Web, how can we develop the desktop world's equivalent of application extensions at *Web* scale?

In general, such extensions can be delivered on the Web by:

- Web application providers injecting the needed augmentation based on user preferences. Such *injection* has to be independent of any given adaptive technology to avoid a combinatorial explosion, and to avoid the risk of locking users into any given adaptive technology.
- Adaptive technology vendors injecting such augmentation that is specific to the AT being used.
- Web user agents relying on declarative metadata to identify relevant portions of a Web user interface, and exposing such metadata via *reflection* to the adaptive technology.
- The Web community at large experimenting with different approaches that work at *Web scale.* Such large-scale experimentation can be used to discover and codify design patterns that work as a means of arriving at the next generation of accessibility techniques.

The power of Web 2.0 in this respect is that the various approaches outlined above are not mutually exclusive — they can all be realized in parallel. Here are specific instances where each of the above is being implemented:

Injection Project AxsJAX is an example of a framework that enables the injection of application-specific augmentation. The injected code is independent of any given adaptive technology — this neutrality is achieved by building on W3C ARIA.

Screenreaders Mainstream AT vendors are beginning to bundle scripts for popular Web applications.

Browsers Implementations of W3C ARIA automatically turn metadata present in the HTML DOM into metadata that can be retrieved by adaptive technologies via the relevant reflection APIs.

Community Google-AxsJAX was conceived as an Open Source framework for disseminating the relevant expertise needed to access-enable Web applications. As we *AxsJAX* various Google applications, these examples are being used to explain how W3C ARIA can be leveraged to build end-to-end accessibility. In the process, we are extracting a library of common reusable functions that can be used across the Web for quickly augmenting a given Web application.

6. WEB APIS AND SPECIALIZED BROWSING

Finally, one of the biggest opportunities opened up by the design of Web 2.0 applications is the arrival of high-level Web APIs that enable the creation of *custom* clients. Such custom clients can be thought of as specialized browsers where the specialization happens along one or more of the following axes:

Task Customized for a task, *e.g.,* displaying the weather. Examples include Google Gadgets and Apple Desktop Widgets.

Environment Specialized for use in a given environment, *e.g.,* mobile access from small screens. Examples range from custom mobile applications, *e.g.,* Google Maps for Mobile, to custom Web interfaces.

User Specialized to match a given user's needs and abilities at a given time. As an example, Emacspeak includes a wide array of speech-enabled Web tools.

Notice that the architectural separation of user interface and application back-end described earlier makes all of the above tractable.

7. CONCLUSION

Web applications are here to stay — the advantages they bring in terms of ubiquity are numerous. Accessing Web applications using adaptive technologies from the era of desktop software has been a challenge. The advantages present in a world where user interface augmentation is delivered using the *same* Web technologies used to deploy end-user access to application services opens up a wealth of opportunities for users with special needs. The advent of GUIs in the 90's was often labeled a serious problem for visually impaired users — yet, a better understanding of the underlying issues, along with improvements in the adaptive technology stack has led to a wider level of access to software and information. As we develop this field, I strongly believe that we will look back at this time 10 years from now and come to a similar conclusion about Web 2.0 applications.