Keeping the Web in Web 2.0

An HCI Approach to Designing Web Applications

CHI 2007 Course Notes

Steffen Meschkat, Google, Inc., Senior Software Engineer
Joshua D. Mittleman, Google, Inc., Senior Software Engineer
{mesch,jmittleman}@google.com

CHI 2007, April 28-May 3, 2007, San Jose, California, USA.
ACM 07/0004.

Table of Contents

0. Editorial Note 1
1. Web Application UI versus Desktop Application UI 1
2. A General Caution 2
3. Examples of State Management in Web Applications 2
   (a) Bookmarking versus Browser History 2
   (b) Hard State and Soft State 2
   (c) Ajax Applications 3
      aa) Google Maps 4
      bb) Gmail 6
      cc) Google Web Search 7
4. Concepts for State Management in Web Applications 8
   (a) Taxonomy of Interaction State 8
      aa) Hard State 8
      bb) Soft State 8
      cc) External State 9
      dd) Transient State 9
   (b) Designing Interaction State 9
   (c) History State versus Bookmarked State 9
      aa) Hard State in Bookmarks 9
      bb) Soft State in Bookmarks 9
      cc) External State and Versioning 10
      dd) Transient State in Bookmarks 10
5. Two solutions for browser history management and bookmarking 10
   (a) The Hidden Iframe Method 10
      aa) Recording Hard State – Creating Browser History Entries 11
      bb) Recording Soft State – Storing Values in the Current Browser History Entry 12
      cc) Obtaining a Bookmark URL 12
      dd) Initializing the Application State from a Bookmark URL 12
      ee) Browser History Navigation 12
   (b) The Fragment Identifier Method 13
      aa) Recording Hard State 14
      bb) Recording Soft State 14
      cc) Obtaining a Bookmark URL 14
      dd) Initializing the Application State from a Bookmark URL 14
      ee) Browser History Navigation 15

6. Some Further Implementation Considerations....15
(a) Background Data Retrieval and Browser History....15
(b) Encoding Formats for the State....16
aa) URL parameter name-value pairs....16
bb) URL-escaped JSON....16
cc) websafe-base64-encoded binary data....17
(c) Behavior on Page Reload....17
(d) Updating External State....18
7. Conclusion....18
(a) Paging through Lists....18
(b) Multistep Dialogs and Transactions....19
8. Glossary....20
9. Resources and References....21
(a) Course website....21
(b) Learning Javascript and Ajax....21
(c) Development Help and Tools....22
(d) References....22
(e) Browser History tools....22

Instructor Biographies

Steffen Meschkat joined Google in 2004 and currently works on maps. He earlier co-founded ART+COM AG and datango AG. At ART+COM, he worked on industry funded application research projects of Virtual Reality and, since 1993, the WWW. For datango, he built the client side components of the navigation suite, a technology that augments web applications by simulated user interaction Fragments. He has an MSc ("Diplom") in Physics from Humboldt University in Berlin.

Josh Mittleman received a Masters degree in Computer Science from Rutgers University; and worked in advanced 3D graphics at IBM T. J. Watson Research Center from 1990 to 2002. At Airgator, he led the development of NavAir, a consumer air navigation system. He is currently a Senior UI Software Engineer at Google Inc., working on interactive search results.

Agenda

Thursday, 2007-05-03

Course 41	9:00 am - 10:30 am
Break	10:30 am - 11:30 am
Course 42	11:30 am - 1:00 pm

Objectives of the Course

- Systematic analysis of application state and its relationship to browser history and bookmarking.
- Introduction to the Javascript programming language.
- Introduction to DHTML: HTML Document Object Model (DOM) and event handling.
- Introduction to background server communication with AJAX.

0. Editorial Note

These notes cover the content of both parts of our course (scheduled as courses 41 and 42). Other material, including coding examples and demonstrations, will be included in the course presentation and will be available as handouts and online. These notes present the fundamental ideas we will discuss in the course, but are not intended as a complete, self-contained discussion independent from the course presentation.

1. Web Application UI versus Desktop Application UI

The development of the WWW in the early 90s introduced the use of HTML documents as the remote graphical user interface for network-based interactive applications. The advantages of this platform were manifest, but it initially represented a huge step backward from desktop applications in terms of the visual and interactive quality of the user experience: every significant user interaction required a full refresh of the screen – the download and display of a new HTML page in the web browser. Interactions such as drag-and-drop were impossible, pull-down menus were initially impossible and later difficult and clumsy. The return of the client-server architecture resulted in a return to client-server UIs.

However, the advantages of the web outweighed these shortcomings. First among them was the almost universal, platform-independent access to applications from anywhere on the network. Two other user interface properties emerged as important, too: browser history – universal, uniform, and unlimited undo functionality that allows free navigation of the user's history of interaction with the application – and bookmarking – the ability to remember the state of an application in the form of a URL and to resume interaction later, possibly even from another terminal. These two features were provided by web browsers generically and independent from any specific web application.

The Ajax architecture restored to web applications the interactive and visual quality of desktop applications, while preserving the lightweight, universal, remote access of web applications. However, because of the specific ways in which user interaction is processed in Ajax applications, a naive implementation breaks both browser history and bookmarking. Because maintaining these two interaction patterns is so desirable, they need to be explicitly built into Ajax applications; but the richer user interactions supported by Ajax make this goal especially challenging to achieve.

It is helpful at this point to define these two interaction patterns more precisely.

(a) A **bookmark** is a URL that allows the user to return to a web application, with the state it had at the time he saved the bookmark. In classical web applications, every significant user interaction is either a click on a hyperlink or the submission of a form, both of which load a new document from a new URL: the content of the document is unambiguously represented by its URL. In this way, the user interaction state is fully represented in the URL of the page. This architectural principle is also called *REST*, for Representational State Transfer. A widely applied yet architecturally inferior alternative is Server-Side Session State, which opaquely accumulates state changes from user interactions on the server side, and sends back documents representing the current interaction state. The returned document is not based exclusively on the requested URL, which merely encodes the event, but also on the current session state stored on the server. This approach has different problems, not least that it breaks bookmarking. The user cannot reliably back up to a previous state because the server may not offer it. A server side session usually has to expire to free resources on the server, and thus deletes the context in which to resolve a bookmark URL.

(b) **Browser history** is the stack of states recorded as the user browses through one or more web applications. In the simplest case, it is the stack of web pages that the user has visited, each with its distinct URL. More generally, it can include more than one state of a single page, if the page contains code that generates new states, or if the page is a frameset or contains iframes. We will discuss several ways in which scripts on a page can create new history states in response to user interactions. It is also possible for a script to generate new states automatically, though that behavior is rarely encountered on the web.

2. A General Caution

Web applications run inside a browser, generally using the browser's Javascript interpreter. Although the language – Javascript – and the APIs – most notably the Document Object Model (DOM) – are standardized, different browsers implement them in different ways, because of bugs, differences in understanding of the standards, or by deliberate choice to provide non-standard features. Some of the behaviors we discuss in this course vary from one browser to another. We have tried to mention the important differences between the important browsers (i.e. Internet Explorer vs. Firefox vs. Safari vs. Opera); and the bibliography at the end of this document includes references to books and websites that document many of these differences. However, if you develop complex Ajax applications, you will undoubtedly encounter undocumented weirdnesses, especially as the various browsers undergo upgrades.

3. Examples of State Management in Web Applications

Before we discuss the technological solutions to browser history and bookmarking in Ajax applications, we need to develop our understanding of user interaction state in web applications in general and in Ajax-based web applications specifically.

(a) Bookmarking versus Browser History

Bookmarking is one way to save application state, specifically by storing it in the URL. Not all of the application state is stored in the bookmark. For example: Load a web page that is longer than your screen height (e.g. http://www.chi2007.org/welcome/pastchi.php). Scroll down the page and then bookmark it. Leave the page (e.g. with your Back button) and then load the bookmark: You will return to the top of the page.

Scroll down again, leave the page, come back to it through the browser history. You come back to the scrolled-down page. Caveat: This is browser dependent, but most browsers behave as we have described.

Thus, the page scroll position is an element of the interaction state that is captured in the browser history, but not in bookmarks.

(b) Hard State and Soft State

We saw in the previous example that the browser history preserves the scroll position of a page. But scrolling to another spot on the page does not create another history entry: If you scroll down a page and then click the back button, you are taken to the previous page, not to the previous position on the same page.

Thus, the state elements that are recorded in the browser history are recorded in different ways: some changes create new history entries, such as the URL of the page; some just override the previous value in the same history entry, such as the scroll position. We will refer to the second type of elements of the interaction state as **soft state**, and to the first type as **hard state**.

Another example of soft state is the contents of an HTML form input field. Go to a page that contains a form (e.g. http://portal.acm.org/). Enter some text into the input field; then click any link to leave the page (e.g. click the *Subscribe* link on this page). Now return by clicking the Back button. Your text is shown in the input field. If you bookmark this page and return to it via the bookmark, though, the field will be blank. The contents of form inputs, therefore, is part of the soft state of the application. Caveat: This behavior also varies from one browser to another. Most modern browsers preserve form inputs – and we will exploit that fact later – but e.g. Opera 7 does not.

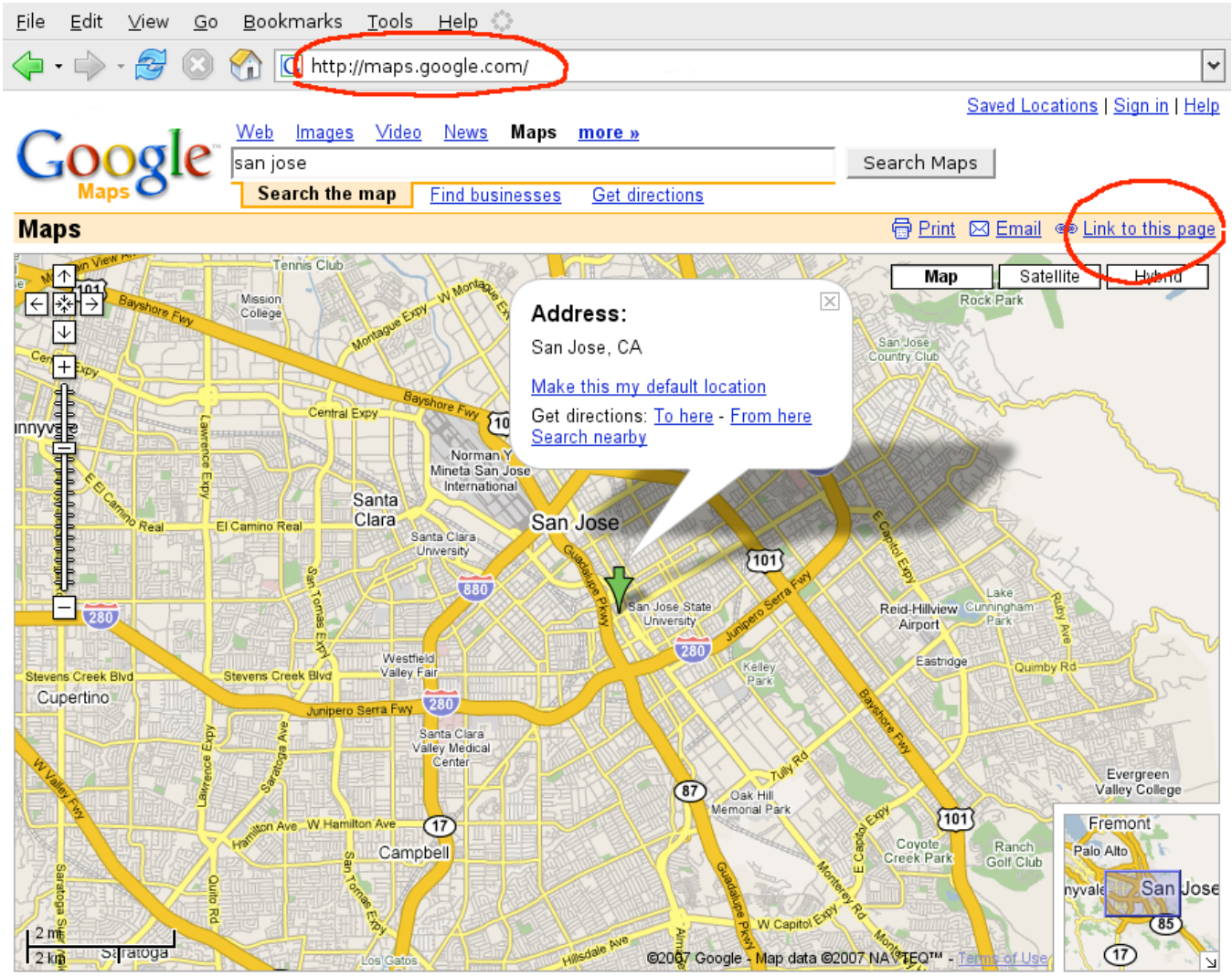
(c) Ajax Applications

In the examples above, we have considered classic HTML applications. As we mentioned, browser history and bookmarks are implemented for such applications by the browser. Let's see how some simple Ajax applications handle both.

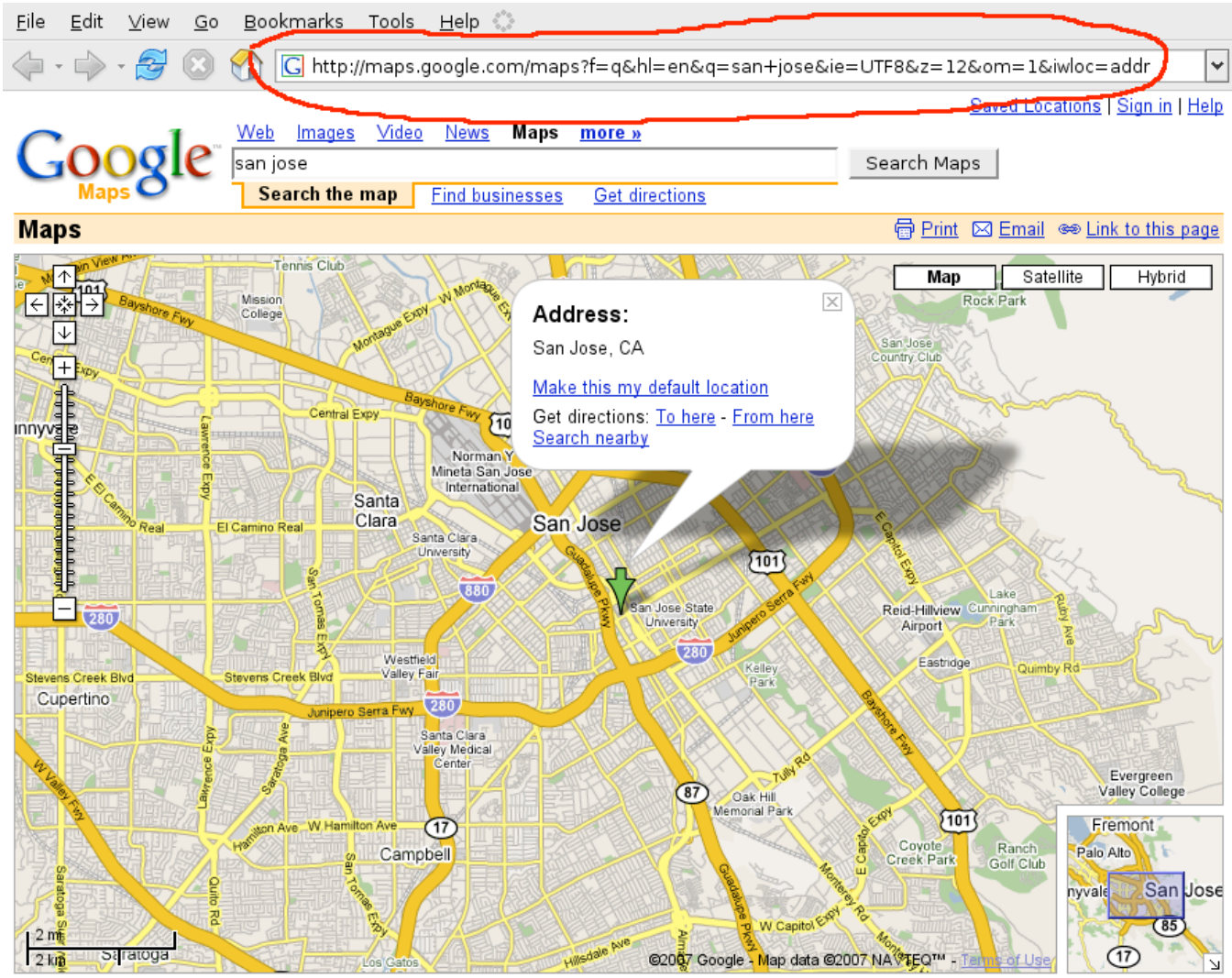
In Ajax applications, interaction logic is implemented on the client side using script that is embedded in the HTML page. Additional information that is necessary to adequately handle a user event can be retrieved from the server in the background and used to update the current page. Because, however, the user event itself is not sent to the server and the current view is not updated by entirely replacing the current document by another one, the current URL does not change during a user interaction with an Ajax application. Therefore, bookmarking, i.e. storing the current URL of the page, does not directly capture the current interaction state of the web application. If the application developer wants the URL to capture the state of the web application, he must implement it explicitly.

aa) Google Maps

Consider http://maps.google.com/ . Into the search input field at the top of the page, type "san jose" and press the search button. The map view changes to San Jose. The URL of the page, however, is still http://maps.google.com/ , which when stored as bookmark will not represent the current map view of San Jose.



The maps application provides a "link to this page". Open this link in a new window directly, or create a bookmark from it and open the bookmark in a new window: The link will reflect the current view of the map. The application is interpretting the new URL and using it restore the application state. This decoding of the application state from the URL could happen server-side or client-side; a priori, we cannot tell which a particular application is doing.



Now search for "san francisco". Change the map type to "Satellite" and pan the map a bit. Now use the back button of the browser to go back to the previous history entry. You are back in San Jose, and back in cartographic mode. Use the forward button: you are in San Francisco, and in satellite mode, at the position you panned to. From this you can conclude two facts (and verify them by additional experimentation):

(1) A new search creates a new entry in the browser history. Going back brings you to the previous search. Thus, a search in the maps application behaves with regard to browser history like a link or a form submission in a classical web application.

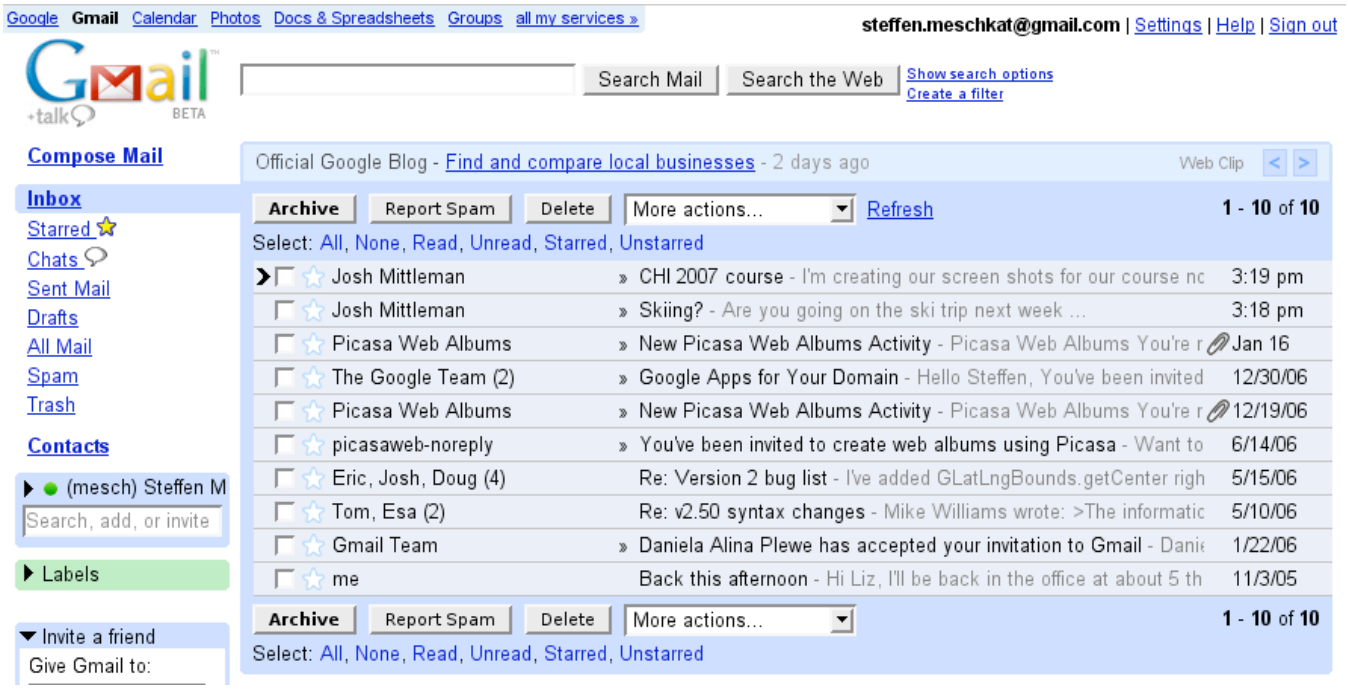
(2) Changing the map state is recorded in the browser history, but it doesn't create a new history entry. Going back doesn't bring back the previously selected map type, but it brings back the last search with the map type that was selected when that search was last seen in the browser. Thus, changing the map type in the maps application behaves with regard to the browser history in the same way as scrolling the page in a classical web application. (So does panning the map, in fact.)

Now, "link to this page" from the view with the changed map type: this captures the search query, the map type, and the position of the map.

Last, make a mental note of the zoom level at which you see San Francisco. Then search for "san jose" again, and halve the size of the browser window, and come back (using the browser history) to San Francisco. At which zoom level do you see San Francisco now?

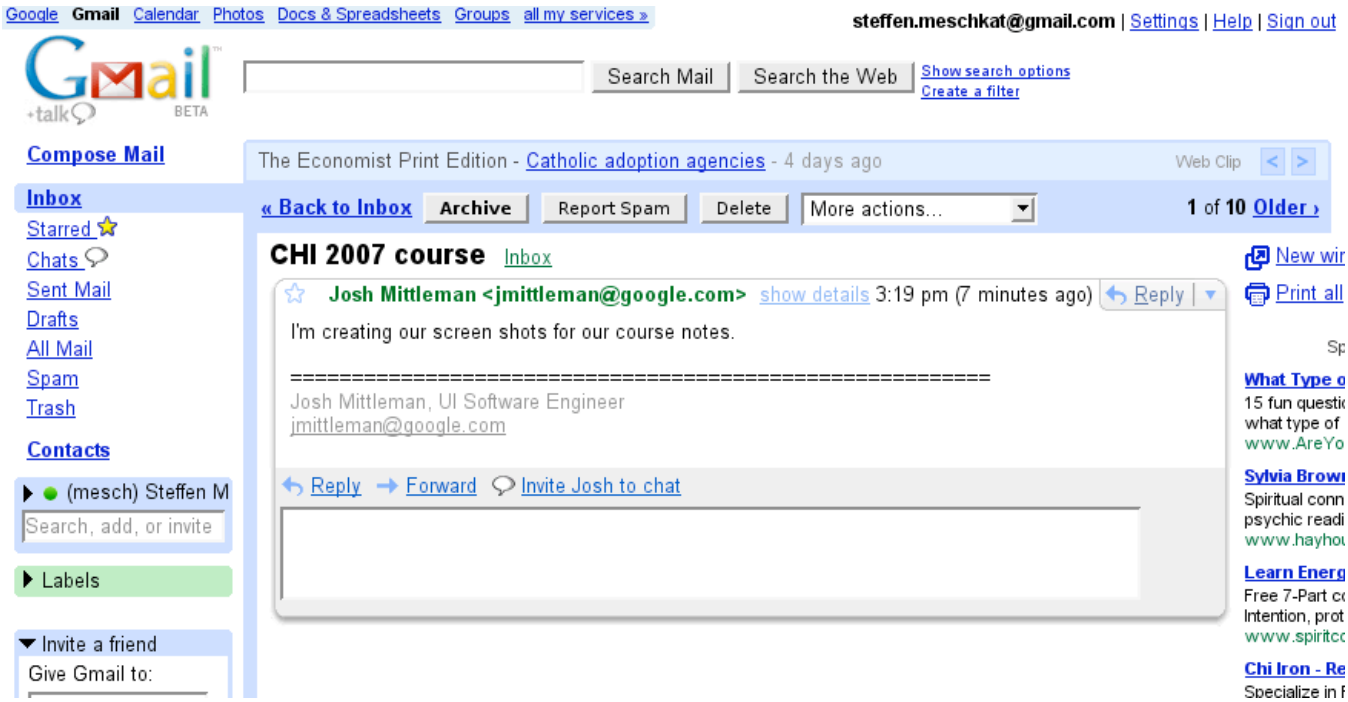
Note that these behaviors are application design choices, built into the Javascript running in the browser.

bb) Gmail



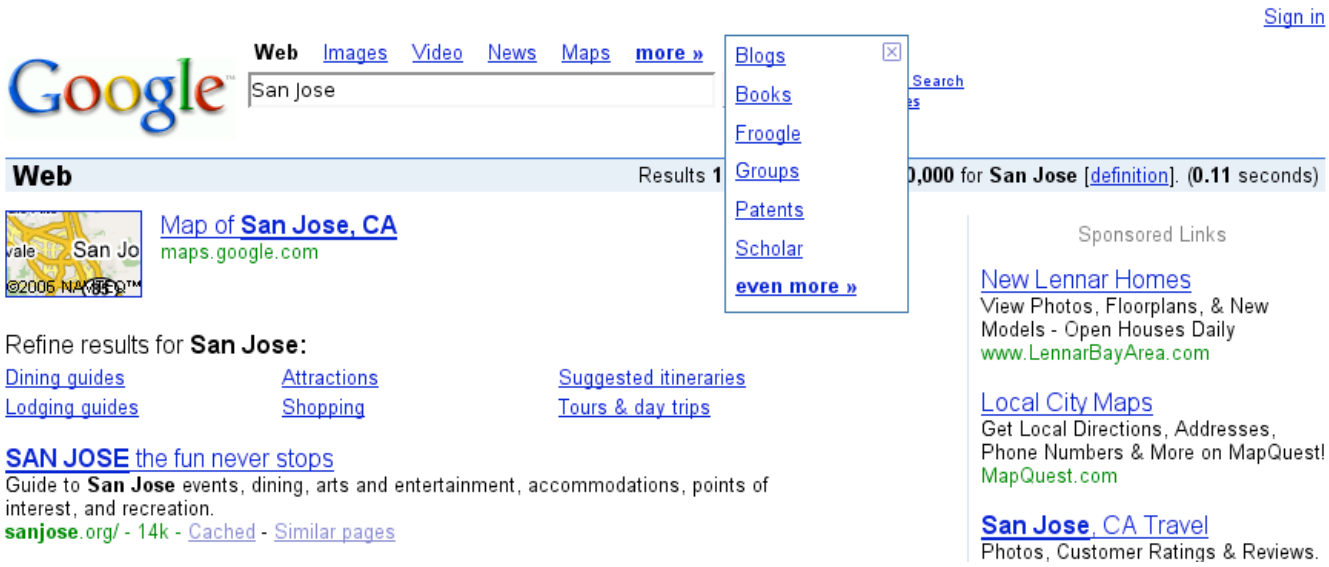
In Gmail, open the first conversation in the inbox. Notice that you can go back to the inbox using the link "Back to inbox" provided to the left of the conversation, but also using the back button of the browser.

Again, select the first conversation; and from another mail client send yourself a message to the Gmail account you are looking at. Wait for the message to arrive, and go back to the inbox, using the back button. Do you see the new message? You should. Apparently, the set of messages shown in the inbox is not state that is stored in the browser history, but the current view is, i.e. inbox vs. conversation.



cc) Google Web Search

In Google, search for "HCI"; and then search for "San Jose". Click on the link "more" above the search box at the top of the page, but don't select any of the options. Leave the "more" box open and use the back button to go back to the previous search result page. Use the forward button to come back to the page on which you left the "more" box open. Is it still open? If you're using Internet Explorer, it isn't. The opened more box is not recorded in the browser history. If the "more" box is still open, read the next paragraph. Open the "more" box again. Bookmark the page. Open the bookmark in another window: The opened "more" box is also not recorded in the bookmark.



Depending on your browser, you may actually see something different. One of the facts of life in Ajax development is that every browser has quirks, and your applications will need to accommodate them. (Considerable expert help is available to solve the more common examples of this problem; see the Bibliography for pointers.) In particular, some modern browsers (e.g. Firefox 2, Opera 9) preserve the entire DOM in browser history, so that changes applied to DOM elements by scripts in the page are preserved across navigation. So if you try our example in Firefox 2, then you will find that the "more" box actually is open when you click "forward". This "DOM preservation" or "fast back" function can be useful, but can also often interfere with consistent UI design in your applications. We'll discuss that problem in more detail later.

4. Concepts for State Management in Web Applications

In the examples above, we have seen how different elements of the user interaction state behave differently with regard to browser history and bookmarks. In classical web applications these distinctions are hardwired in the browser. In Ajax-based web applications, contrariwise, the management of interaction state must be implemented explicitly. Handling interaction state sensibly and consistently requires a clear understanding of the different properties of elements that affect the application state. Moreover, because user interaction state can be accumulated in the browser, and application-specific behavior that allows the manipulation of the interaction state is implemented in the browser, the taxonomy of elements that constitute the state is more complex for Ajax applications than for classical web applications.

(a) Taxonomy of Interaction State

aa) Hard State

Some state elements, when they are changed by user interaction, create a new entry in the browser history, so that navigating back restores the previous value of that state element. Examples of this are the query in Google Maps, or the view (inbox vs. conversation) in Gmail. We call the set of state elements that create history entries when they change the **Hard State** of the application. Note that if the page consists of several documents loaded into separate frames, replacing any one of these documents will create a new history entry. Thus, a history entry can be thought of as representing an n-tuple of URLs. This interpretation will be useful later.

bb) Soft State

Other state elements are recorded in the browser history, but changing them does not create a new entry in the browser history. An example of this is the selected map type in Google Maps. Such state elements are referred to as **Soft State**. Notice that in the case of the map type or pan position of the map in Google Maps, the soft state is also part of the bookmark of a page. This need not have been the case: It was an application design choice.

cc) External State

In the examples above, we have encountered two more types of state elements which are important but less prominent in classic web applications. When I return to the inbox of my web mail application, whether by bookmark or through browser history, I don't expect to see exactly the same set of messages that were in my inbox previously. Indeed, that would be an error: I expect to see the current state of my inbox, with any messages that have arrived since my last visit. This is an example of **External State**: it changes over time, due to both user interaction and external events, but cannot be represented in bookmarks or browser history. Instead, it is stored outside the application and fetched by the application as needed.

dd) Transient State

Finally, we have seen examples of elements affected only by user interactions, but whose state is not preserved at all. When a pull-down menu is opened, that fact is not recorded anywhere and the user does not expect its state to be restored when he reloads the application. These elements of the application are **Transient State**.

(b) Designing Interaction State

When designing the user interaction for an Ajax-based web application, the classification of state elements into these classes must be made consciously and explicitly in order to implement logical, consistent behavior of the application. Notice that in some cases the right choice of behavior follows with necessity from the application. In other cases, there is no canonical choice and you have to make a deliberate design decision to select the behavior of a state element.

Sometimes the right choice of the state elements with regard to state invariants is subtle. In one of the above examples of the "link to this page" in Google Maps, the current viewport is encoded into the URL using the center point and the span in degrees of latitude and longitude. If the bookmark is loaded in a browser with smaller screen window, the original field of view is recreated, potentially by changing the zoom level. An alternative would have been to encode the center point and zoom level, which would result in a smaller map view in a smaller browser window, but at the same zoom level, resulting in the same level of detail in the map view. Both alternatives are sensible, but it is important to use one approach consistently throughout the application.

(c) History State versus Bookmarked State

The classification above, into Hard, Soft, External, and Transient state, pertains mostly to browser history. The state as stored in a bookmark is closely related, as follows:

aa) Hard State in Bookmarks

Hard State is represented in bookmarks.

bb) Soft State in Bookmarks

Soft State may be represented in the bookmark or may not, depending on the application developers' choices. However, for those elements that are included in the bookmarked state, there is no distinction between hard and soft state elements.

cc) External State and Versioning

External state is usually not meaningfully represented in bookmarks any more than in browser history. Notice, though, that in many situations it makes sense to assign specific URLs over time to certain stages of the external state. These URLs can provide versioning. For example, a word processing application could save a copy of a document – automatically or by user choice – and assign each copy a distinct URL so that the user can access past versions of his work. Notice that versions (i.e. changes in external state elements) are usually not accessible through the browser history.

dd) Transient State in Bookmarks

Transient State is not represented in bookmarks, by definition. But of course the distinction between soft and transient state is an application design choice.

5. Two solutions for browser history management and bookmarking

In Ajax-based web applications, user interaction does not cause the page to be replaced by another page, thereby creating an entry in the browser history. Instead, interaction logic is implemented in the page and causes the page to be changed based on the user actions. This change may involve the retrieval of data from a server and its inclusion into the page content; but this retrieval usually happens in a way that does not create an entry in the browser history nor change the bookmarkable URL of the page. (The basic mechanism for fetching data with an Ajax application is the same as the one used by the browser to load a new page, but the effect on the state is delegated entirely to the application rather than managed automatically by the browser.)

Therefore, in order to provide history and bookmarking for state changes in our application, we have to consider ways to explicitly change the page URL, and to explicitly create browser history entries. The two main methods we will discuss are the use of hidden iframes, and the exploitation of the fragment identifier part of the URL. Both are described in the following sections. Their applicability relies, however, on two properties of the application architecture that need to be explicitly established:

(1) The ability of the application to describe its own state as a URL, and hence to capture a complete picture of the user interaction state elements of the application in terms of soft, hard, transient, and external state. That is, the application must both be able to represent its current state as a URL as well as to initialize itself into the state given by a URL.

(2) The availability of a complete picture of possible state transitions, at which it would potentially be necessary to update either the URL, the browser history, or both. That is, it must be possible to update the description of the application state in a URL whenever any element of this state changes.

(a) The Hidden Iframe Method

An iframe is an HTML element that allows one HTML page to be embedded in another. A hidden iframe is simply an iframe that is not visible to the user, e.g. because it is positioned outside of the browser window.

The current interaction state is stored in the document of this hidden iframe; and a Javascript function in this document is used to notify the application that it should configure itself to the stored state. The notification of the application must be triggered when the page (including the iframe) is reloaded due to a history navigation event (like pressing the back button); that can be implemented with an onload event handler or a callback to a function defined on the application's main page. An advantage of this method is that the hidden iframe can be used not only to store the current interaction state in the browser, but also to contact a server in the background to retrieve data associated with this state.

A hidden iframe might be implemented in HTML like this:

```
<iframe style="border:0px;width:1px;height:1px;
   position:absolute;
   bottom:0px;
   right:0px;
   visibility:visible"
  id="HiddenIframe"
  src="/mapfiles/home3.html">
</iframe>
```

Here is a typical document loaded into a hidden iframe that is used to store state:

```
<html>
  <head>
    <script>
      function loadstate() {
        var hardstate = location.query.substr(1);
        var softstate = document.getElementById("softstate").value;
        window.parent.loadstate(hardstate, softstate);
      }
    </script>
  </head>
  <body onload="loadstate()">
    <input name="softstate"/>
  </body>
</html>
```

aa) Recording Hard State – Creating Browser History Entries

A new browser history entry is created by loading a new document into the hidden iframe. As mentioned above, loading a new document in one frame causes the browser to automatically create a new history entry. The application state is encoded into the URL of the iframe, usually in the query part of the URL. Notice that the file part of the URL could also be used to store the application state, but this is less common. The host part of the URL cannot be used, because then the resulting response document in the iframe would not be able to access the main page because of cross-site scripting security restrictions.

Here is an example URL:

```
http://maps.google.com/maps?q=san+jose+convention+center&f=1#top
```

The parts of the URL are:

protocol	`http`
host	`maps.google.com`
path	`/maps`
query	`?q=san+jose+convention+center&f=1`
fragment	`#top`

The query in a URL may consist of URL parameters, name-value pairs separated by ampersands. In this example, the parameter names are q and f. A URL can also specify a port path on the host; that is not shown in this example.

When the URL of the document in the hidden iframe is changed, the previous document is replaced and a new one is loaded. Therefore it is essential that this happens in an iframe and not in the main page, since the application state stored in the main page would otherwise be destroyed.

The document is loaded from the server (compare this to the document fragment identifier method, outlined below). It can therefore contain server-supplied data that are pertinent to the current application state. For instance, in Google Maps, the search results for a new query are loaded in this way: the current query is encoded in the q URL parameter of the URL of the document in the hidden iframe, and the server supplies the search results associated with the search as the content of the document.

When the document is loaded, a callback – a Javascript function defined in the main frame – is called to both notify the application of the history navigation event, and to pass to the main page the supporting data contained in the content of the document, possibly including soft state (see next section).

Alternatively, the iframe document can be static, and the data that are necessary to support the changed application state can be retrieved independently using an XMLHttpRequest.

bb) Recording Soft State – Storing Values in the Current Browser History Entry

To store soft state, we can exploit the fact that the values of input fields are preserved in the browser history in most browsers. (This is true of IE, Mozilla browsers including Firefox, and Safari; but not Opera 7.) We add an input field to the document in the hidden iframe. Whenever soft state elements change, we record their state in this input field. When the application changes its hard state, loading a new document into the hidden iframe, the value in this input field is preserved in the browser history. When the user returns to a browser history state (e.g. clicks the back button), the saved value of the input field is restored by the browser and can be used by the callback to restore the application soft state.

cc) Obtaining a Bookmark URL

Since the hard and soft state of the Ajax application does not directly correspond to the URL of the main page, a simple bookmark will not allow the state to be recovered. Instead, the application must explicitly construct a URL to use as a bookmark for the current state, which must be updated each time the state changes. In Google Maps, this is done using the "link to this page" URL. This URL is, in fact, the URL of the document in the hidden iframe extended with the soft state values stored in the input field in the hidden iframe document. To make it easy to append the soft state values to the URL, they are recorded in the input field as a list of URL parameter name-value pairs.

dd) Initializing the Application State from a Bookmark URL

The application decodes the data stored in the bookmark URL and uses it to restore the saved state. If the application state is stored in the parts of the URL that are sent to the server, the server can also examine the state specified in the URL and may participate in the initialization of the application. Depending on the application, including the server in this process may be an advantage because it could save additional server round trips.

ee) Browser History Navigation

Any HTML document may specify an onload handler, i.e. a Javascript function that is called when the browser finishes loading the document. A document loaded into a hidden iframe is no exception: It may have its own onload handler, which can be used to notify the main page that a new page has been loaded in the iframe. Since we load a new iframe document to create a new history state, navigating back

through the browser history will result in reloading a previous iframe document. Its onload handler will fire, allowing the application to configure itself to match the saved state.

There is no difference in flow of control between the result of a hard state transition that was caused by direct user interaction, and the hard state transition that is caused by a browser history navigation. In both cases, the main page is instructed to configure the saved application state by way of a callback triggered by the onload event in the hidden iframe document.

One feature of modern browsers interferes with this method: Some browsers, such as Opera 9 and Firefox 2 implement **fast-back**, which stores the complete state of each document in the browser history. When such a document is restored from browser history, no onload event is fired, because the page is supposed to be saved in its last state, with any changes that the onload handler made the first time it was called. As with many smart ideas, this is unfortunate. In Firefox, the fast-back behavior can be suppressed by registering an "unload' event handler. In Opera 9, unfortunately, the hidden iframe method is no longer directly applicable.

(b) The Fragment Identifier Method

The fragment identifier method changes the fragment identifier part of the URL of the current page to store the state. Normally when the URL of the current page changes, the entire page is reloaded, and application state that is stored in Javascript data structures in the page is lost. However, there is an exception for the fragment identifier (also known as **hash**, because it is appended with a hash mark # at the end of the URL). Changing the fragment identifier is intended to scroll the document to the section identified by the new fragment identifier; so it does not load a new document, it scrolls the existing one. In HTML documents, the fragment identifier references the value of a "name" attribute of an "A" element (an HTML "A" element with a "name" attribute, as opposed to one with an "href" attribute, is also known as "anchor").

Example: The URL

```
http://www.google.com/apis/maps/documentation/reference.html#GIcon
```

refers to this element in the reference.html document:

```
<h2><a name="GIcon">class GIcon</a></h2>
```

We do not want the page to scroll, of course: We are exploiting this behavior to save state information in the URL and, in some cases, to create a new history entry. However, some browsers will not actually create a new history entry for the changed URL if the new fragment identifier doesn't match a named tag. Therefore, the procedure is as follows:

(1) Create an "A" element with a "name" attribute of the desired value.
(2) Absolutely position it at the top edge of the window.
(3) Update the hash value of the page URL. The browser will find the anchor identified by the hash value (which we just created), and scroll the page to show this anchor at the top of the window (where it already is, so nothing changed).
(4) Remove this "A" element again.

Sample implementation:

```
// Create the new element.
var newElement = document.createElement("A");

// Give it a name and position it.
newElement.name = "my_anchor";
newElement.style.position = "absolute";
newElement.style.top = window.scrollY + 'px'; // See note, below.

// Add it to the DOM
document.body.appendChild(newElement);

// Update the URL.
location.href = "#" + newElement.name;

// Remove the element from the DOM and delete it.
newElement.removeNode(true);
newElement = null;
```

In Internet Explorer, setting the page URL can have the side-effect of generating an audible click.

The determination of the current scroll position of the window depends both on the browser and on the specific positioning settings of the page level elements in the document.

aa) Recording Hard State

To use the new URL (with the new hash value) to create a new browser history entry, simply assign the new fragment identifier to `location.href`, as in the example above.

bb) Recording Soft State

To use the new URL to save state without creating a new browser history entry, set the page URL with `location.replace()`. As before, the URL must be relative, with only a document fragment identifier part. Note that the application developer can decide when to record a change as hard state, when to record it as soft state, and when to ignore it, allowing the explicit definition of hard, soft, and transient state.

```
location.replace("#" + newHashValue);
```

cc) Obtaining a Bookmark URL

Since we have set the hash value of the URL of the page and encoded the application state in it, the current state can be bookmarked directly. No extra step is necessary.

dd) Initializing the Application State from a Bookmark URL

A bookmark of the application has the application state encoded in the document fragment identifier. This part of the URL is never transmitted to the server; so the decoding of the state data and the initialization of the application into the saved state must occur in the browser (though of course the application may request extra data from the server).

ee) Browser History Navigation

When the user navigates to a history entry, the URL of the page changes (specifically, the fragment identifier part of the URL), but no new page is loaded. Consequently, the page isn't notified that it has to assume the state encoded in the browser history entry. Thus, we need to detect this change explicitly. One solution is to poll the page URL to detect a change; and then call the appropriate function to change the application state.

```
// Track the current fragment identifier so
// we know when it changes.
var currentHash = null;

// Define a function that is called every
// 50 ms to check whether the page URL has
// changed. Note that we define the function
// inline, without naming it.
window.setInterval(function() {
  if (location.hash != currentHash) {
    // Update the current hash.
    currentHash = location.hash;

    // Signal the application to update its
    // state to match the information in the
    // hash.
    UpdateApplicationState(location.hash);
  }
}, 50);
```

6. Some Further Implementation Considerations

(a) Background Data Retrieval and Browser History

There are two main methods for background data retrieval: using a hidden iframe, and using the XMLHttpRequest object. Each has its advantages and disadvantages.

Using a hidden iframe, the data is retrieved from the server by loading a new document into the iframe. As we have already discussed, this process can create a new browser history entry (by assigning the new URL to `location.href` or `location.hash`) or not (by calling the method `location.replace()`). The onload event handler on the new page notifies the main page of the arrival of the data, which the application can then process. Note that this approach requires that the data be embedded in an HTML document.

An XMLHttpRequest object can be used to make a special request to a server to retrieve any arbitrary data. The data can be encoded however is convenient for the application, e.g. as a JSON literal, without the need of an HTML document to encapsulate it. When the application creates the XMLHttpRequest object, it registers an event handler to notify it of the arrival of the new data. Using an XMLHttpRequest will never create a browser history entry, so it cannot be used to retrieve data from the server and record a state change in the browser history at the same time.

The apparent efficiency of the XMLHttpRequest is sometimes misleading: When the data are formatted in JSON format, sent with an HTTP Content-Type:text/javascript, and also compressed with gzip (Content-Encoding:gzip), then under certain conditions, the data will not be properly decoded by Internet Explorer. Therefore, bare JSON data can not always reliably transmitted in compressed encoding to pages in Internet Explorer. This doesn't happen for HTML pages, even those which contain script. Thus, embedding JSON data in a wrapping HTML document and loading them using a hidden iframe is sometimes preferable to the use of the XMLHttpRequest object.

(b) Encoding Formats for the State

There are many alternatives for the data format used to encode the application state into the bookmark URL. The requirements are:

(1) The size must remain below a certain limit for the size of URLs (browser dependent).
(2) It must contain only characters that are allowed in a URL.
(3) It may be desirable for the application state to be intelligible, for example to allow users to create links to application states from scratch.

Here are a few alternatives with different properties:

aa) URL parameter name-value pairs

Example:

```
http://maps.google.com/?f=q&hl=en&q=76+9th+ave+nyc&ie=UTF8&z=15&ll=40.74
199,-74.00455&spn=0.016713,0.037508&om=1&iwloc=addr
```

bb) URL-escaped JSON

The data encoded as URL parameters in the example above could be written in JSON as:

```
{
  "f": "q",
  "hl": "en",
  "q": "76 9th ave nyc",
  "ie": "UTF8",
  "z": 15,
  "ll": {
    "lat": 40.74199,
    "lon": -74.00455
  },
  "spn": {
    "lat": 0.016713,
    "lon": 0.037508
  },
  "om": 1,
  "iwloc": "addr"
}
```

This representation can be URL encoded and used, for example, as the query part of a URL. (Note that this URL is for illustration only, as maps.google.com encodes its application state in URL parameters only, not in JSON.)

```
http://maps.google.com/?%7B%22f%22%3A%22q%22%2C%22hl%22%3A%22en%22%2C%22
q%22%3A%2276%209th%20ave%20nyc%22%2C%22ie%22%3A%22UTF8%22%2C%22z%22%3A15
%2C%22ll%22%3A%7B%22lat%22%3A40.74199%2C%22lon%22%3A-
74.00455%7D%2C%22spn%22%3A%7B%22lat%22%3A0.016713%2C%22lon%22%3A0.037508
%7D%2C%22om%22%3A1%2C%22iwloc%22%3A%22addr%22%7D
```

This format obviously suffers in readability as compared to URL parameters. Most browsers will accept a less fully URL-encoded form which is more readable:

```
http://maps.google.com/?{"f":"q","hl":"en","q":"76 9th ave
nyc","ie":"UTF8","z":15,"ll":{"lat":40.74199,"lon":-
74.00455},"spn":{"lat":0.016713,"lon":0.037508},"om":1,"iwloc":"addr"}
```

The main advantage of this representation is that it allows the user to enter values normally (e.g. the value of "q" is a string, while the value of "om" is a number); and that it accommodates more deeply structured data (e.g. the "ll" value is a record with two fields, "lat" and "lon").

cc) websafe-base64-encoded binary data

In certain circumstances, it may make sense to encode the state as binary data, for example by applying data compression in order to squeeze a maximum amount of data into the limited length of the URL. Because arbitrary binary data cannot occur in a URL, these data need to be encoded using only ASCII characters that have no special meaning in a URL. This is accomplished using websafe base64, which is a variant of the base64 encoding that doesn't require URL escaping. The above JSON example in websafe base64 encoding:

```
http://maps.google.com/?eyJmIjoicSIsImhsIjoiZW4iLCJxIjoiNzYgOXRoIGF2ZSBu
eWMiLCJpZSI6IlVURjgiLCJ6IjoxNSwibGwiOnsibGF0Ijo0MC43NDE5OSwibG9uIjotNzQu
MDA0NTV9LCJzcG4iOnsibGF0IjowLjAxNjcxMywibG9uIjowLjAzNzUwOH0sIm9tIjoxLCJp
d2xvYyI6ImFkZHIifQ==
```

(c) Behavior on Page Reload

A special case is a page reload, caused, for example, by clicking the "reload" browser button or pressing the CTRL-R key. Browser behavior on reload is highly variable. In some cases (e.g. clicking reload in Internet Explorer or Firefox), the values of hidden iframes and input fields are preserved and so the techniques described here for restoring page state will work correctly. This **soft reload** is essentially the same as navigating to the current state as a browser history entry.

In other browsers (Opera, Safari), clicking reload will discard any changes to the values of hidden iframes and input fields, restoring them to the values they had in the original HTML page. In these cases, any changes to the page state will be lost.

All browsers allow a **hard reload**, which fetches a fresh copy of the page from the server. In Firefox and Internet Explorer, for example, you can force a hard reload by pressing Shift and Ctrl while clicking Reload. The values recorded in input fields and hidden iframes are lost.

In all these cases, though, state encoded in the fragment identifier of the URL will be preserved.

(d) Updating External State.

External state has to be retrieved by the page explicitly. There are multiple ways to keep being notified of the changes of external state:

(1) Polling the server. At regular intervals, send an update request to the server. If the server-side state has changed, the server responds with the data needed to update to browser-side representation of the external state.

(2) Whenever a background data request is made to the server, state changes for the external state elements are piggybacked onto the server response. This is essentially a lazy version of the preceding method.

(3) A two-way channel to the server is established using a mechanism outside HTML/Javascript, e.g. a socket connection is opened from Flash or from a Java applet.

(4) A notification channel is built using a keep-alive HTTP connection that is closed when there is an external state update available. If the response sent through this connection is crafted in the right way, the page is notified of the closing connection by an error event.

The details of these methods are beyond the scope of this presentation.

7. Conclusion

This course gives an overview of how to tackle the challenge of implementing bookmarking and browser history for Ajax-based web applications. While it suggests specific solutions, the main contribution is the development of concepts that allow a better analysis and statement of the problem, based on which it is simpler to find more general solutions. Still, it should be pointed out that the problem still doesn't solve itself, and explicit design decisions and implementation work are required. To illustrate this, we conclude with two more examples that show that the semantics of application state may be non-trivial to analyze and establish.

(a) Paging through Lists

In Gmail, go to the second page of your inbox, and select the last conversation. Now send yourself a new message (make sure it's really new, don't just reply to an earlier message, so that it won't get appended to any existing conversation). When the message has arrived, and you go back to the inbox, what page of the inbox do you see now? It is the same, and the conversation that you selected when you left the page is now on the next page. Assuming you could bookmark the page (which in gmail, alas, you cannot, but in many other applications you can), what would be the right semantics for such a link? Would it be the "second page of the list of conversations in the inbox", whose content changes over time, or would it be "the page of the list of messages in the inbox that contains this specific conversation", which changes over time, too, because the chronological ordering of conversations changes as new messages arrive.

(b) Multistep Dialogs and Transactions

Consider the dialog that lets you transfer money to another account in your bank's online banking web application. Usually, this is a two-step dialog. First you input all the transaction parameters, such as amount and recipient, and in a second step you input a unique confirmation number. Now consider the possibility of bookmarking the page on which you input the confirmation number that validates the transaction: What would be the possible semantics of this bookmark? This depends, among other considerations, on how you construe the semantics of the transaction. If, conceptually, the transaction starts with the first step, and ends after the second, then the bookmark would lead to a page that displays the current status of the same transaction. Since that transaction is no longer pending, it would show the transaction as committed.

If, however, you conceive of the form as merely collecting data for an upcoming transaction, but the transaction as being executed only when the second part of the dialog is concluded, then this bookmark would be a transaction template, and the bookmark would let you create another transaction with the same parameters as the original one.

Neither of the two alternatives is clearly better, and neither would be incorrect to implement. The only clearly wrong decision would be not to clearly decide the semantics of such a bookmark at all.

8. Glossary

Bookmark: A saved URL, used to return to a page previously visited or to restore a previously generated state of a web application.

Browser History: The stack of states recorded by the browser as the user navigates through one or more web applications. A browser history entry corresponds to a tuple of URLs for the main page and each frame within it. Browser History Navigation means moving among these saved states, e.g. with the browsers back and forward buttons.

Cross-site scripting (XSS): A type of computer security exploit in which information an attacker inserts malicious code into a script in such a way that it will be executed by script.

Cascading style sheets (CSS): A language for specifying the visual properties of HTML elements, e.g. color, font, placement on the page.

Dynamic HTML (DHTML): a programming paradigm that combines HTML, CSS, and Javascript.

Fast back (also called DOM Preservation): A feature of some modern browsers, e.g. Firefox 2, Opera 9, which saves the state of the DOM in browser history so that the page can be more quickly re-loaded when the user returns to a history entry.

Hypertext Markup Language (HTML): the XML language used for organizing and formatting most WWW documents.

HTML Document Object Model (DOM): The data structure constructed by the browser to represent the structure and content of an HTML page.

Hypertext Transfer Protocol (HTTP): the basic mechanism for transferring data over the WWW.

Iframe: An HTML element that allows one HTML page to be embedded in another. A hidden iframe is simply an iframe that is not visible to the user, e.g. because it is positioned outside of the browser window.

Javascript: An interpreted programming language used in browsers to modify the page, create complex interactive or automatic behaviors, etc.

Javascript Object Notation (JSON): a format for representing Javascript objects as text in a way that allows them to be restored simply by applying the Javascript function eval(). Thus, for any object x, eval(JSON(x)) == x.

Universal Resource Locator (URL): An address corresponding to a document that can be loaded in a browser. A URL consists of a protocol, a server, a filepath, a query that may include name-value pairs, and a fragment identifier or hash.

XMLHttpRequest: A javascript function that allows a javascript application to request data from a server.

9. Resources and References

(a) Course website

http://steffen.meschkat.googlepages.com/ajax

(b) Learning Javascript and Ajax

W3 Schools Online Web Tutorials, http://www.w3schools.org/. Excellent tutorials on all the basic tools of modern web development. Particularly: Javascript: http://www.w3schools.com/js/default.asp , HTML DOM: http://www.w3schools.com/htmldom/default.asp, Ajax: http://www.w3schools.com/ajax/default.asp

A (Re)-Introduction to JavaScript, by Simon Willison http://simon.incutio.com/slides/2006/etech/javascript/js-reintroduction-notes.html A short tutorial.

Javascript in Ten Minutes, http://javascript.infogami.com/Javascript_in_Ten_Minutes An even shorter tutorial.

Dynamic HTML: The Definitive Guide, by Danny Goodman, 3rd edition (O'Reilly Media, 2006). The bible of javascript programming, it contains detailed discussion of javascript objects and methods, CSS properties, HTML elements, and DOM.objects.

JavaScript: The Definitive Guide, by David Flanagan, 5th edition (O'Reilly Media, 2006). A complete manual of javascript. A better book for learning the language than the previous one; but once I learned the language, I've used the previous book much more often.

CSS: The Definitive Guide, by Eric A Meyer, 3rd edition (O'Reilly Media, 2006). A thorough manual of CSS. The previous book covers some of the same material but not in as much depth.

Head Rush Ajax, by Brett McLaughlin (O'Reilly Media, 2006). An easy guide for a beginner.

Professional Javascript for Web Developers, by Nicholas C. Zakas (Wrox Professional Guides, 2005).

Professional Ajax (Programmer to Programmer), by Nicholas C. Zakas, Jeremy McPeak, Joe Fawcett (Wrox Professional Guides, 2006).

The Visibone Browser Book, http://www.visibone.com/products/browserbook.html A handy 16-page quick-reference cheat sheet.

Ajax in Action, by Dave Crane, Eric Pascarello, Darren James (Manning Publications, 2005).

(c) Development Help and Tools

DevGuru, http://www.devguru.com/. Bug reports, workarounds for browser incompatibility, and the like.

WebReference JavaScript Articles, http://www.webreference.com/programming/javascript/.

Firebug, https://addons.mozilla.org/firefox/1843/. A Firefox plugin that provides a fabulous suite of development tools.

Firefox DOM Inspector. This tool is built into the browser and can be found in the menus at Tools / DOM Inspector. Similar tools are found in other browsers.

Internet Explorer Developer Toolbar Beta 3, http://www.microsoft.com/downloads/details.aspx?FamilyID=e59c3964-672d-4511-bb3e-2d5e1db91038 An Internet Explorer extension that provides a similar suite of developer tools.

JSUnit, an XUnit-style unit testing framework for Javascript. http://www.jsunit.net/.

JSDoc, JavaScript Documentation Tool, http://jsdoc.sourceforge.net/. A tool that parses inline documentation in JavaScript source files, and produces an documentation of the JavaScript code.

Apple's Safari page http://developer.apple.com/internet/safari/ and particularly the developer FAQ http://developer.apple.com/internet/safari/faq.html

Mozilla Developer Center: Javascript, http://developer.mozilla.org/en/docs/JavaScript

How to debug JavaScript problems with Opera, http://my.opera.com/community/dev/jsdebug/

(d) References

CSS 2.1 Reference, http://www.culturedcode.com/css/reference.html A definitive manual of CSS.

Javascript DOM Reference, http://krook.org/jsdom/.

(e) Browser History tools

Really Simple History, by Brad Neuberg, http://codinginparadise.org/projects/dhtml_history/README.html A small javascript module that implements browser history for Ajax applications, using many of the techniques we have presented.