TAKE ME BACK: A STUDY OF THE BACK BUTTON IN THE MODERN INTERNET

a Thesis
Presented to
the Faculty of California Polytechnic State University
San Luis Obispo

In Partial Fulfillment
of the Requirements for the Degree
Master of Science in Computer Science
by
Bryan G. Estrada
June 2011
COMMITTEE MEMBERSHIP

TITLE: Take Me Back: A Study of the Back Button in the Modern Internet

AUTHOR: Bryan G. Estrada

DATE SUBMITTED: June 2011

Clark Savage Turner, J.D., Ph.D.
Committee Chair

Franz J. Kurfess, Ph.D.
Committee Member

Gene Fisher, Ph.D.
Committee Member
Abstract

Take Me Back: A Study of the Back Button in the Modern Internet
Bryan G. Estrada

The web browser has become one of the most recognizable software applications on consumer desktops. Yet its utilization and capabilities are often misunderstood. Recent innovations in the web have evolved the Internet into a network of sophisticated applications that defy historical uses of the “browser”; a term that itself has become somewhat of a misnomer. This research studies the evolving set of user expectations for the browser as an application platform and challenges certain anachronistic features, specifically the “back” button, that are unnecessary and confusing given the new environment that browsers are used in. Because of this shift, implicit new user requirements arise around the browser’s user interface. The back button, like other elements in the browser have already demonstrated, should be de-emphasized in modern iterations of web browsers. The study is qualified by an analysis of user behavior within a popular, modern, web application.

Keywords: web apps, back button, browsers
Contents

List of Tables vii

List of Figures viii

1 Introduction 1

1.1 Research Questions .............................................. 3
1.2 Contribution to Scholarship ..................................... 3

2 Background 5

2.1 Evolution of the Web and Browsers ............................ 5
2.1.1 Anatomy of a Browser .......................................... 6
2.1.2 Web 1.0 .......................................................... 6
2.1.3 Web 2.0 .......................................................... 8
2.2 App Stores ......................................................... 9
2.2.1 Google Chrome Web Store .................................... 10
2.3 Back Button Ubiquity .............................................. 10
2.3.1 Browser History Stack ......................................... 11
2.4 Web Applications vs. Web Sites ................................ 13

3 Related Work 15

3.1 Back Stack ........................................................ 15
3.1.1 Recency Model ........................................ 16
3.1.2 Recency with Temporal Ordering .................... 17
3.2 Document Fragments ..................................... 18
  3.2.1 Search Engine Optimization ........................ 19
  3.2.2 Hash Bang ........................................ 19
3.3 HTML5 History API .................................... 20
3.4 Click Tracking .......................................... 21
3.5 User Interaction Design ................................. 22

4 Methodology .................................................. 24
  4.1 The App ................................................. 25
  4.2 Hypothesis .............................................. 27
  4.3 The Experiment ........................................ 27
    4.3.1 Initial Findings .................................. 27
    4.3.2 A/B Testing ........................................ 30
  4.4 Re-examination ......................................... 32

5 Conclusion .................................................... 36
  5.1 Drawbacks and Weaknesses .............................. 36
  5.2 Future Work ........................................... 37

Bibliography .................................................... 38
List of Tables

2.1 navigation represented by stack ........................................... 12
3.1 navigation with temporal ordering ........................................... 17
List of Figures

1.1 an early version of the NCSA Mosiac browser for Windows . . . . . . . . 2
2.1 interaction with a traditional web site . . . . . . . . . . . . . . . . . . . . . 7
2.2 interaction with a rich internet application . . . . . . . . . . . . . . . . . . 9
2.3 sample web site document structure . . . . . . . . . . . . . . . . . . . . . . 11
3.1 web browser wireframe, circa 1995 . . . . . . . . . . . . . . . . . . . . . . . 23
4.1 absolute unique visitors . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
4.2 home screen of the Flixster Chrome app . . . . . . . . . . . . . . . . . . . 26
4.3 back navigation distribution . . . . . . . . . . . . . . . . . . . . . . . . . . 28
4.4 page navigation entrances . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
4.5 page navigation exits . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30
4.6 control case, standard back panel . . . . . . . . . . . . . . . . . . . . . . . 31
4.7 test case 1, highlighted back panel . . . . . . . . . . . . . . . . . . . . . . . 32
4.8 test case 2, highlighted back panel and hint notification . . . . . . . . . . 32
4.9 A/B test results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33
4.10 histogram of daily percentages of A/B tests . . . . . . . . . . . . . . . . . 34
4.11 back panel usage over time . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
Chapter 1

Introduction

A web browser is an application that helps users navigate the Internet\textsuperscript{1}. Many users, however, would not be able to describe the browser this way. This is because they don’t use browsers just to “browse” the Internet. The browser today is a platform that hosts sophisticated applications with features that equal, or may even exceed, traditional desktop software. In this sense, asking someone to define a web browser is akin to asking them to define an application development platform. Understanding the underpinnings of the web browser is unreasonable to expect of end users and unnecessary for productive use of it. The interface that users are confronted with should be simple and optimized for what they want to use it for.

At present, the browser is host to a many of useful applications, each offering different functionality to accomplish different tasks. Developers are taking advantage of the newest practices in web technology to the traditional role of web pages. Improvements in browser capabilities have enabled things like real-time panning and zooming on a map with Google Maps\textsuperscript{2}, music recommendations and radio in Pandora\textsuperscript{3}, social collaboration in Facebook\textsuperscript{4},

\begin{footnotesize}
\begin{itemize}
\item \textsuperscript{1}The Mosaic Browser shown in Figure 1.1 was one of the first web browsers in the market[1].
\item \textsuperscript{2}http://www.maps.google.com/
\item \textsuperscript{3}http://www.pandora.com/
\item \textsuperscript{4}http://www.facebook.com/
\end{itemize}
\end{footnotesize}
Figure 1.1: an early version of the NCSA Mosiac browser for Windows and feature-parity with its desktop counterpart in the Microsoft Office Web productivity suite\(^5\). Content in today’s Internet is dynamic, customized, and interactive.

But the web wasn’t always this way. Browsers are still designed for navigating documents rather than hosting applications. Any web browser on any operating system has evidence of this in its user interface: back and forward buttons, a refresh button, sometimes a home button, and an address bar. Supporting these interfaces is an imposition on developers that utilize the browser to serve custom applications to their users. Preservation and prominence of these UI elements is arguably necessary to keep a familiar feel to the Internet, but it makes little sense to support traditional navigation schemes—like “back”—within the context of a modern web application.

\(^5\)http://office.microsoft.com/
1.1 Research Questions

Broadly, this research explores what are the user expectations for a web browser as a container for web applications and where do these expectations come from. Given the new context of the Internet and its many uses beyond page browsing, is it even necessary to include traditional browser elements, like the back button, that are based on conventional document navigation? More specifically, given the choice between the browser’s native back button or an app-optimized alternative, which does the user choose?

1.2 Contribution to Scholarship

Several studies have been done around the browser [2, 20] and user behavior [16, 17, 14], but the web is a field that innovates and updates very rapidly [3]. New frameworks, development techniques, and even browsers are released early and often. Advancements in technology evolve expectations from users, and this research draws a distinction between the new user requirements in a web application and a the traditional expectations of a web site. Behavior within a specific application is studied and analyzed.

Studies specifically around the back button have also been conducted [8, 4, 13, 18]. But those studies focus on the back button within a document-browsing context, rather than looking at the browser as an application platform. Furthermore, the web’s rapid pace of innovation beckons updates to research around the Internet.

This particular investigation is conducted within the context of a single application. The web has only recently been popularly recognized as a proper platform to host applications [6]. Before certain advancements in web technologies, applications in the browser couldn’t come close to native desktop functionality, a premise that is becoming less and less true. Moreover, use of the Internet is no longer reserved solely to researchers and academics. The broad public of consumers use the Internet as an important means for communication, commerce,
and entertainment. This shift in perspective should provoke changes in the browser’s user interface. But browsers today still resemble early browsers from the Internet’s inception; optimized for document navigation and hypertext links.
Chapter 2

Background

Innovations in web use motivate this study; chiefly the recognition that the Internet has become mainstream and browsers the primary means to consume content on the Internet. The following sections describe the evolution from web sites to web applications, the proliferation of apps, and ubiquity of the ever-present back button.

2.1 Evolution of the Web and Browsers

In 1974, researchers invented what is known as the Internet Protocol Suite, or TCP/IP: a set of rules that computers follow so they could communicate with each other. In subsequent decades, the protocol eventually matured into HTTP, the Hypertext Transport Protocol [22]. Software programs called browsers used the protocol to download content from servers. The servers returned data in HTML, the Hypertext Markup Language [22]—a format which browsers understood—and rendered to users. Thus born was the World Wide Web.
2.1.1 Anatomy of a Browser

The browser is a software application that knows how to navigate the web. The ability to interpret three technologies, HTML, CSS, and JavaScript, is principle for any web browser implementation.

Historically, HTML was born first in this trinity of web development. HTML is a markup language that initially added structure to data. Instead of delivering mere plain text, publishers could organize their content with markup using tags like `<head>`, `<title>`, `<body>`, and `<p>`. One tag crucial to the standard is the anchor `<a>` tag. Using anchors, publishers created links to different pages on the Internet, even pages that they themselves did not own [10].

The second piece necessary for browsers to understand web pages is Cascading Style Sheets, or CSS. CSS decouples the visual presentation of data from the content itself. Developers can specify things like colors, fonts, borders, and backgrounds using CSS to change the appearance of the data without altering its structure.

Before the revolution of what is now referred to as “Web 2.0”, a web page was just that: a page that was loaded from the web. The Internet was a collection of documents accessible by their Uniform Resource Locator (or URL) and all that browsers could do was navigate this web of content nodes. But the technology evolved, and regular users started to realize the value of the World Wide Web.

2.1.2 Web 1.0

User interactions in the traditional web are simple. The specific history of browser releases and implementations are less important than the underlying execution of what happens when a user makes a web request.

Figure 2.1 sequences the interactions between a user, their browser, and a web server.
Since the browser client is the area of concern, server-side processes have been simplified. Basic interaction in a Web 1.0 world can be described as follows:

1. user initiates some action to load a page in the browser
2. browser makes request to server
3. server interprets request, connects to database, generates markup, etc.
4. data returned to markup thread with Content-Type: text/html
5. markup is rendered, additional resources (css/javascript/etc.) discovered
6. requests for auxiliary data made to server
7. server returns stylesheets to css thread with Content-Type: text/css
8. server returns scripts to JavaScript thread with Content-Type: application/x-javascript
9. styles received, interpreted
10. scripts received, execute
11. engine restyles markup
12. JavaScript engine applies modifications to DOM, runtime
13. document is ready
14. browser returns with feedback to user

In a traditional web site, performing actions like clicking on a link, pressing a button, or submitting a form, would trigger the same series of events over again\(^1\).

### 2.1.3 Web 2.0

The term AJAX was first used in 2005 [7], but the techniques that developers used to create rich Internet applications existed before the term was coined. Asynchronous JavaScript and XML describes the approach that web developers use to update content in a web application without refreshing the entire page. The sequence in Figure 2.2 depicts the same interaction as shown in Figure 2.1, but updates in an AJAX environment come quicker.

1. user initiates some action on already-loaded web application (clicks button, types something, etc.)
2. browser fires events for javascript listeners
3. javascript executes, makes request for new data from the server
4. server responds with response data (XML or JSON) to javascript runtime
5. javascript manipulates DOM with data

\(^1\) See generally [3]
Figure 2.2: interaction with a rich internet application

6. document modifications loaded into browser
7. browser returns with feedback to user

2.2 App Stores

The proliferation of app stores has made discovery and installation of apps extremely easy [5]. Consumers have entered a mindset where specialized software programs (apps) are well-understood and expected on devices. Ironically, well-established platforms like the traditional desktop and even the web have yet to see the same shift in perspective that mobile phones seem to have generated on a platform that didn’t exist a few years ago.
The rise of mobile application platforms have been a key influence. Largely due to mobile devices, consumers now understand the utility of single-purpose, narrowly scoped, and focused applications; i.e. “there’s an app for that”. Success of apps can at least be partially attributed to the shallow history of the mobile web, a platform with no legacy features or user habits to accommodate. During its initial wave of innovation, users in the mobile web space had no preexisting expectations of what the platform should be or how it should behave. At the same time, developers had a large foundation of knowledge around computing and the web in general. The result was a fresh take on how users should do computing with connected devices.

2.2.1 Google Chrome Web Store

Google advocates their Chrome browser as a platform for apps—programs that run in the browser with a dedicated interface and rich user interaction. Developers have been creating rich applications on the web for some time. Google’s effort in the browser and its web store is a formal organization of these services to those familiar with using apps on phones and other similar devices [11].

2.3 Back Button Ubiquity

The back button is arguably the most recognized user interface element of a web browser (barring perhaps the address bar). Over the course of the browser’s evolution, prominence of the back button has prevailed for a number of reasons:

- allows users to return to recently consumed content rapidly;
- can be used even with a naïve, possibly incorrect, perception of how it works;
- thus far, is always accessible, i.e. never hidden in most popular browser UIs;
Figure 2.3: sample web site document structure

- requires little energy to achieve the sought effect, e.g. a user can simply click it repeatedly until the desired page is recognized.

In the traditional web, the mental model of the back button mattered less because the design of content was static and simple. Publishers created content that was mostly hypertext documents that required little understanding of how to navigate the web.

### 2.3.1 Browser History Stack

Traditional document-based web navigation involves users visiting pages by clicking links or pushing buttons. Whenever the user visits a new page, the URL changes and a corresponding entry is pushed onto a one-dimensional stack. The browser maintains a cursor that points to the current position on the stack. If the user clicks back, the cursor moves down the stack, changes the URL to the corresponding address, then loads the new page content.

Consider the document structure of the sample web site shown in Figure 2.3. The example site is collection of pages arranged in a tree; links between pages only exist where an arrow is drawn. A user can take the following actions to navigate between pages:
1. clicking on a link (∫) within a page; pops all entries above the cursor off of the stack, pushes a new entry to the top, then moves the cursor to the top

2. clicking the back button (⇐); moves the cursor down the stack one entry

3. if available, clicking the forward button (⇒); moves the cursor up the stack

Given these actions, a browser implementing a history stack would represent navigation as shown in Table 2.1. All modern browsers offer this implementation of back (and forward, incidentally), with no alternative implementations of navigation behavior [8, 4]. The stack-based model is arguably an oversimplification of a complex data structure. A multi-dimensional graph of links is reduced to a one dimensional history stack. A most notable consequence is that any navigation via links (i.e. not clicking back or forward) while the cursor is positioned in the middle of the stack will clear all history above the cursor, making those pages impossible to revisit through back/forward navigation. From the example in Table 2.1, steps 8 and 10 demonstrate the history loss when performing this type of action. The stack-based implementation seems to be a design compromise between the complete ability to return to recently viewed pages and the desire of an easy-to-understand user interface for end users.

Within the context of a modern web application, even more problems arise when a user
clicks on a link or button that asynchronously alters the state of the web application’s content without changing the URL. Since the content loads with AJAX, the browser’s URL does not automatically change and no entry is added to the history stack. But if the content on the page is drastic enough, a user without a concrete understanding of the browser’s model could perceive the change as a new “page” and believe a new entry has been added to its history. Even users with a clear understand of the history stack could reasonably expect that clicking back would return the browser to its former state. Clicking back, then, would not load previous contents of the application but rather refresh the page to whatever entry is below the current cursor on the history stack.

Supporting back in interactive applications becomes even more complicated when data parameters are introduced. The HTTP protocol allows more than simple retrieval of web documents. Since its inception, HTTP support GET and POST operations (among other methods) [22]. When URLs are posted, additional data can be sent to the server, traditionally using the <form> element. But since these data parameters are not always encoded in the URL, post parameters wouldn’t be included. Furthermore, the expectation is unclear whether or not to include extra data parameters in a POST in the first place [23].

2.4 Web Applications vs. Web Sites

The Internet has been the host to countless web sites. Sites, in the traditional sense, are simply documents that provide static information about a certain topic. But expectations of the web have grown and applications on the web are now more commonplace. Google lays out certain design principles that successful web apps comply with [12].

Tight Focus. A web app will do exactly one thing (or one category of things) very well. Extraneous functionality outside the scope and purpose of the application confuses users and upsets the feeling of the application.
**Big Screen.** Web apps make the best use of the screen real estate that they are run in. Moreover, since a user consciously decided to use an app (rather than browse a site) extraneous navigation elements are unnecessary.

**Rich Experience.** The user experience of a web app is engaging, rich, and indistinguishable from native desktop applications. Modern browser technology and standards like HTML5 make this possible.

**Beauty.** Technologies like CSS3, SVG, and JavaScript canvas make it possible for applications to be visually stunning without sacrificing usability.

**Speed.** Web apps are extremely responsive. JavaScript runtimes are becoming more and more performant and bandwidth is only getting faster. Users should not have to wait for feedback from a well written web app.
Chapter 3

Related Work

The web has been the subject of study for since its inception, and researchers have already explored user interactions within the browser. Though it shouldn’t, the user interface and implementation of today’s browsers still resemble those of early prototypes. Specifically features like the back button prevail despite being misunderstood by users in the context of web applications. Depending on the user’s perception of the browsers, the back button could mean any number of things: “go back”, “go up”, or “undo” to name a few. Due to the nature of today’s modern web applications, the output a user expects from clicking back won’t always align with the actual action of the browser. Scholars determined that the imposed interfaces for navigation on the Internet are not suitable for a web of applications and interactivity [8, 18, 24].

3.1 Back Stack

Though studying a different problem, researchers have already proposed alternative implementations of the back button to solve the history-pruning problem of stack-based navigation. As described in Section 2.3.1, the typical stack-based behavior underlying back is problem-
atic because previously seen pages are not always reachable through it. This is the effect of representing a multidimensional graph of pages as a one-dimensional stack.

Greenberg and Cockburn [8] offer several alternate behaviors of the back button based on a recency model to overcome this dilemma.

### 3.1.1 Recency Model

A “pure recency” implementation would push an entry onto the stack for every page visit. Pressing *back* would also trigger a stack push. Consider the navigational path $A \rightarrow B \rightarrow C \rightarrow D$, the history stack would be $[A, B, C, D]$. Now clicking back in a pure recency model would move the user back and push $C$ onto the stack. Now the stack looks like $[A, B, C, D, C]$.

While this implementation would maintain a complete history of visited pages, it would be impossible to navigate back to any page beyond the last two. This is because a cycle between the last two pages is always created whenever back is pressed. For example, in the path $A \rightarrow B \rightarrow C \rightarrow D$, a user would not be able to click back twice to return to $B$ because any time back is clicked, an extra entry is added for the page you are currently on.

The “spokes only” approach solves the cycling problem that the pure recency technique is culpable. The implementation is simple and similar to the traditional stack-based model. A “spoke” is a child page that is one of many siblings to a common parent, or “hub”. Clicking back and forward will not modify the stack contents, as usual. But clicking a link to a new page will push an entry to the top of the stack without popping any other entries. Additionally, elder duplicates of the stack are removed.

However, while *spokes* does provide a complete representation of the user’s history, the ordering of the list it preserves can be severely different from the user’s expectations. Returning to the example in Figure 2.3, if a user navigates to the hub $C$, then wants to navigate to $D$, $E$, and $F$ by backing up to the hub, the resulting stack would look like $[A, B, C, D, E, F]$. From this point, backing up to the hub $C$ would take three clicks of the back button, whereas
Table 3.1: navigation with temporal ordering

<table>
<thead>
<tr>
<th>Navigation</th>
<th>Cursor</th>
<th>Temporal Stack</th>
<th>Secondary Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A</td>
<td>[A]</td>
<td></td>
</tr>
<tr>
<td>A → B</td>
<td>B</td>
<td>[A, B]</td>
<td></td>
</tr>
<tr>
<td>B → C</td>
<td>C</td>
<td>[A, B, C]</td>
<td></td>
</tr>
<tr>
<td>C ← B</td>
<td>B</td>
<td>[A, B, C]</td>
<td>[B]</td>
</tr>
<tr>
<td>B ← A</td>
<td>A</td>
<td>[A, B, C]</td>
<td>[B, A]</td>
</tr>
<tr>
<td>A ← H</td>
<td>H</td>
<td>[A, B, C, B, A, H]</td>
<td></td>
</tr>
</tbody>
</table>

a traditional stack based view would only require clicking back once. Such “hub-and-spoke” browsing behavior is inefficient in this technique.

An enhancement to spokes technique could overcome both the complete history problem while maintaining efficiency for hub-and-spoke browsing behaviors. The enhanced approach behaves similar to spokes, except that instead of just adding the newly visited page to the stack, it pushes both the current and new page to the top (again with elder duplicates removed).

But even with this technique, true temporal ordering is not maintained.

3.1.2 Recency with Temporal Ordering

Temporal ordering is a modification to the pure recency model. It works by keeping a secondary stack of back navigations that get added to the main history stack if and when the user clicks on a new forward link after a sequence of backs. Consider again the example in Figure 2.3. The Recency Model with Temporal Ordering will produce navigation behaviors as described in Table 3.1. When back actions are used, page visits are added to a secondary list maintained transparently from the user. If the user clicks on an unvisited forward link, the temporal stack gets appended the the main stack (erasing duplicates) in addition to the new page visit.

This model can improve navigation because it makes all historical page visits reachable.
via the back button. Users can now truly use the “click-until-the-desired-page-is-recognized” strategy without the risk that the desired page has been pruned. The disadvantage is that the new model introduces longer navigation paths that may be unfamiliar to seasoned web users.

Cockburn, et. al. studied the feasibility of the temporal model against traditional stack-based navigation [4]. Thirty-four participants completed a variety of navigation tasks; some were given a customized version with a temporal back button, none of which were given any sort of training on what it was or how to use it. They found that there was no significant difference in performance between the stack and temporal interfaces, but noted the promise that temporal users with no training performed equally well in completing tasks as stack-based users.

3.2 Document Fragments

The proliferation of AJAX applications on the web stimulated developers to innovate around the browser’s limitations and overcome this history stack obstacle in asynchronous applications. Notably, developers realized that document hash fragments, originally intended for local document navigation, could be used to communicate information with the server.

AJAX applications can exploit document fragments to give the perception of response to a navigating the back stack. The original purpose of the document fragment (the part of a URL that comes after the #) was to navigate to different elements within a page. A paragraph with and identifier (<div id="content">, for example) can be linked to with an anchor. So, clicking a link that looked like <a href="#content" ...> would cause the browser to scroll to the element with the id content without sending a request to the server. Visits to document fragments are stored in the browser’s history stack and are never sent as requests to the server.
Since JavaScript executes locally in the browser, it has access to this fragment. Some implementations will run a timer loop to check the browser’s address and look for changes in the fragment. More recent implementations of JavaScript engines in browsers will fire an event that notifies its subscribers when a change happens.

### 3.2.1 Search Engine Optimization

One fatal drawback of using document fragments is that blindly using technique breaks SEO. Search engine optimization is already a well-written topic, but it is worth mentioning that search engine crawlers will not execute javascript — an exploit that document fragments rely on to load content asynchronously.

### 3.2.2 Hash Bang

Google came up with a “hash-bang” convention that resolves the SEO problem for AJAX applications [15]. This is not a specification, but rather a convention that only Google follows. The convention works by reading specially formatted hash-bang links and translating them to query parameters.

Take, for example, a twitter URL with `href="#!/aplusk"`. Publishers who expose their document fragment links with an appended exclamation mark can optionally respond to requests to the corresponding query string parameter `?_escaped_fragment_=/aplusk`. Googlebot will translate the hash-bang links to the escaped fragment parameter and make that request to the server instead. It is the publisher’s responsibility to adhere to the convention, but if they do then Google will still index their content.
3.3 HTML5 History API

The recently specified HTML5 History API is a standardized way to manipulate the browser’s history via script [9]. The spec introduces two key interfaces that allows script developers to use truly canonical URLs in AJAX applications: the `history.pushState()` function and `popstate` event on `window`.

As described in Section 2.3.1, the browser maintains a history of URLs that the user has visited. Pre-HTML5 techniques that modified the browser’s history by changing URLs would cause a full page refresh. The `history.pushState()` function changes the URL in the browser’s address bar without refreshing the page. History entries that are added via this function are marked and will never cause a page refresh. Instead, when these marked entries are visited (i.e. via the back button), a `popstate` event is fired on the `window` object.

Use of the History API is preferred because it solves two key problems that accompany document fragments:

1. First order data cannot be included in a single request. The application is slower since document fragments are never communicated to the server in a standard HTTP request. An application must download the initial content first, then use Javascript to interpret the hash fragment and fetch the rest of the data in a second request.

2. Search cannot access AJAX loaded data. Because Internet-crawling bots used by search engines will not execute Javascript, any data that is loaded after the initial page request will not be indexed. Unless the developer implements conventions (like Hash-Bang in Section 3.2.2), their application will lose search-engine discoverability.

The History API overcomes these problems because applications utilizing it have stateful, canonical URLs, so all the information that is needed to load the data is communicated to the server in the initial page request.
Forthcoming web applications with a stateful representation of data can be expected to use this API in the future. Unfortunately, the spec is so new that the majority of browsers on the Internet don’t even support it! Between the major desktop browser implementations in current use, only Chrome 8+\textsuperscript{1}, Safari 5+\textsuperscript{2}, and Firefox 4+\textsuperscript{3} implement the HTML5 History API\textsuperscript{[19]}.

### 3.4 Click Tracking

Studies have also been conducted to analyze user behavior when revisiting content. In 2007, Obendorf, et al. \textsuperscript{[18]} performed an extensive click-stream study of web browser usage focusing on page revisitation, updating previous studies in light of the new, dynamic nature of the Internet. A revisit can be described as the consumption of content from before. He identifies different types of revisits users engage in; short-term revisits to backtrack or undo, medium-term revisits to re-utilize or observe, and long-term revisits to rediscover.

Weinriech, et al. \textsuperscript{[24, 18]} recognized the evolving trends in web technology prompted the need for new investigations on user behavior. Their long-term study tracked client usage through individual browser logging with a select group of subjects, comparing their findings with older studies and noting the changes in user navigation behavior. They identify the increasing use of web applications, noting the increased number of new window events that suggests a mode of interaction where users keep dedicated windows open for special, application-like services. While their browser-agnostic study does not advocate any specific UI modifications, they note certain features like backtracking and history “were not designed for dynamic pages and online applications with volatile contents”.

\textsuperscript{1}released December 2010
\textsuperscript{2}released June 2010
\textsuperscript{3}released March 2011
3.5 User Interaction Design

The principles of design suggest that there is a generalized set of abstractions that can be applied to software improve their interfaces and make using applications easier. Donald Norman’s timeless publication [16] lays out common principles.

**Visibility.** Visual cues are important for users to know what can be done. The relationship between visual indicators, their position in the interface, and what they do make a difference for users to find the appropriate control for the task at hand. Visual cues for less important controls, or for controls that happen automatically should be de-emphasized [21].

**Feedback** is about returning information to users about their action and the resulting accomplishment. Feedback that is too subtle, too late, or not present at all would make performing tasks frustrating and perhaps impossible. Additionally, constructive use of feedback can also help visibility for user interaction.

**Constraints.** The concept of constraining refers to ways to restrict how a user can interact with an interface at a given moment. A common application of this principle appears in user interfaces that disable or de-emphasize elements that cannot or should not be used. This prevents users from selecting incorrect options and reduces mistakes.

**Consistency** refers to interfaces that have similar operations and use similar elements for achieving similar tasks [21]. Consistent interfaces are easier to learn and use because users need only to remember a single mode of operation that is applicable across the interface.

**Affordances.** Norman defines affordances as “the perceived and actual properties of [a] thing, primarily those fundamental properties that determine just how [it] could possibly be used.” [16]. Arguably, though, interactive elements’ perceived and actual behaviors should align. A button that looks like a back button should actually go back.

Consider, for example, the evolution in web browser user interface. Early implementations of Internet Explorer (and similar browsers of the time) resembled the wireframe in
Figure 3.1. The interfaces exposes certain visual cues, namely a toolbar of icons to perform several different actions. Without a history of usage patterns around web browsing, user interface designers didn’t know what actions were most important. Thus everything in the interface received equal treatment — the back button has no more emphasis as the print button or the increase-font-size button.

Contrast this interface with the UI of Google Chrome’s latest iteration shown in Figure 4.2. Most of the browser’s previous functionality is constrained into a single menu toggled by the “settings” button (the wrench icon). Only back, forward, refresh, and the address bar are immediately visible to the user. The modern browser’s UI is more constrained, hiding or disabling less-frequently used elements. Behaviors are mostly consistent with what users expect. And in the traditional document-based web the iconic metaphors afforded to users (back, forward, refresh, search) do what they should.
Chapter 4

Methodology

The basis of this investigation centers on the Flixster web application in the Google Chrome web store. Certain characteristics of the application make it an ideal testbed for study.

1. It is a web app, not a web site. It’s in a store, so there is a perceived notion of “apps”.

2. Users of the web store are early adopters and familiar with technology.

3. Visitors are guaranteed to be using Google Chrome, a web browser with the latest HTML5 capabilities.

4. Flixster is a moderately popular application in the web store. With around 3,000 daily unique visitors, there is a sufficiently significant amount of data that can be generalized with statistical confidence.

5. The timing of data collection began just as the web store launched into the wider broader audience.

The sections that follow describe the application itself, the nature of the analytics in the experiment, and the findings.
4.1 The App

Flixster for Chrome is an entertainment utility designed to help fans discover movies, read reviews, and find showtimes and release dates for new and upcoming films. While the focus of the research is around the analytics collected, it helps to understand the basic functionality of the app itself.

The Chrome version of Flixster closely resembles the Flixster’s other application platforms (iPhone, iPad, Android, etc.) and takes advantage of the latest features that the Google Chrome web browser offers. Users are presented with a familiar two-panel user interface with a navigation panel on the left and the main content panel on the right. Selecting items (a movie, actor, or theater) from the list on the left loads additional data into the content panel. Additionally, the toolbar on top has navigation buttons and a search bar that change the list on the left to the corresponding set of films or theaters.

The app effectively presents a view of an ontological graph of movie data with three entity
Figure 4.2: home screen of the Flixster Chrome app

nodes: movies, actors, and theaters. Movies have two-way edges between the actors in their cast and theaters that they are playing in. Users can navigate between entities by clicking on the appropriate links or buttons.

When a user drills into a movie or a theater (rather than visiting a sibling data node in the list on the left), they traverse deeper into the graph of data. When this happens, the application reveals a back panel that will return the user to their previous content panel when clicked. The panel only appears when a user clicks into another node within the current node’s context. If the user navigates back to top-level nodes the panel goes away. Pressing
the browser’s native back button, clicking the application’s back panel, and pressing the backspace button on the keyboard all accomplish the same thing — returning the user to the previously visited entity node.

4.2 Hypothesis

The back panel is conditionally activated and context aware user interface element that allows users to quickly revisit content that they have already seen. Given this new, in-app solution to application navigation, will users prefer it over traditional navigation with the native back button?

4.3 The Experiment

Recall that the conjecture of this research is that the built-in UI of the web browser is no longer necessary for users to understand and use a modern web application to complete the task they set out to do. Gathered analytics track user data to measure the feasibility of this premise. Specifically, usage of the browser’s native back button is measured and compared with the usage of the application’s back panel. All data was collected by monitoring use of the application.

4.3.1 Initial Findings

The first set of data was gathered by monitoring clicks and counting the events when the user navigated back; either by use of the back button, the back panel, or the backspace key. The pie chart in Figure 4.3 shows the distribution of back navigation gathered during a period of two months. Whenever a user navigated backwards through one of the three
means, a tally would be added to the corresponding count of events. As depicted, only 15% of back navigations happened by way of the back panel.

There are several possible explanations to such a low representation of back panel use over traditional back navigation. Users are already assumed to be accustomed to conventional web browsing where the back button has prevailed for many years. Because the back button is not gone entirely, users correctly assume that its functionality will continue to work even in the context of a web application. Additionally, the back panel only appears conditionally, putting its discoverability into question. Analytics show that the majority of page views are of first-level content and will thus not include a back panel. Consider the following entrance

\footnote{Because of the way the browser's history worked, a user could click the native back button or press backspace key and the app would load the previously viewed piece of content even if the user currently on a first-level entity node. This type of back navigation is not included in the statistical analysis}
and exit charts. The segmented pie charts show the percentage of navigation that lead to or from pages that will or would have included a back panel. Rings on the outside of the chart qualitatively demonstrate the overall traffic of that page entity type.

The chart in Figure 4.4 shows the entrance sources for theater and movie pages. For theater pages, 81% of impressions came from the user clicking on the sidebar. Of the movie page’s 1.6 million impressions, 88% came from the sidebar. This means that the majority of page these page views were first-level navigation where the back panel does not appear\(^2\).

The chart in Figure 4.5 show where users are going to. Consider again the movie page (the most trafficked entity node). Only 25% of navigation away from the movie page is to a deep link–either a showtimes or actor page. The remaining 75% of navigation leads either

\(^2\)Other entity types like actor and showtimes are not shown in the entrance chart because these nodes can only be visited through deep linking, so the back panel would show 100% of the time in these cases.
to another first-level entity node or is a bounce out of the application.

### 4.3.2 A/B Testing

Even with the explanations offered to justify such a low use of the back panel, 15% still seemed curiously low. The problem initially seemed to be back-panel discoverability. Consider the default panel design in Figure 4.6. The panel perhaps blends in with the rest of the list items and is unclear that it’s actually clickable. Two scenarios were tested against this default control group. Upon opening a new session in the app, users were randomly assigned a test group equally among the two scenarios and the control group. Test A added emphasis to the back panel by highlighting it with a different color, as shown in Figure 4.7. Test B, in addition to the highlighting of the panel like Test A, showed an animated tip box to point
out the back panel the first time the user navigates through a deep link. The experiment ran for one week immediately after the data collection period from Section 4.3.1.

The experiment proved successful. During the week of testing, Test A showed over a 5% increase in back panel usage and Test B almost a 13% increase. The increases in the average back panel use also proved to be statistically significant. Figure 4.10 shows a histogram of the distribution of back panel usage across the different test groups. Scores were counted in increments of five. A statistical t-test between Group A and the control group showed a
p-value of 0.047 with Group B's p-value $5.33 \times 10^{-6}$. But peculiarly, the control group also showed an increase usage of the back panel relative to the initial statistics.

### 4.4 Re-examination

The disparity between the initial statistics and the control group from the week of A/B testing suggests that something may have changed at some point in time. It occurred that examining data as it changes over time could affect the perspective of the results. Recall that the data collected in Figure 4.3 showed only a small minority of back panel usage relative
to the native back button. But the pie chart is an aggregation of all back navigation as it occurred over the course of two months.

The graph in Figure 4.11 plots the day-to-day percentage of back-navigation performed by way of the back panel from the same dataset used in Figure 4.3, before A/B testing begun. As it turns out, users were already discovering and learning to use the back panel on their own! Statistically, the increase appears to grow evenly, as the linear trend line is drawn to demonstrate and simple linear regression shows a correlation coefficient of 0.9.

Usage of the back panel between the beginning of the test and the end of the test grew nearly twenty percent, trending upwards without the assistance of hint text or highlighting. Since traffic remained relatively steady during the testing period, it is safe to assume that the set of users clicking on the back panel were the same set of users throughout the study. The increase in back panel usage could suggest that the users’ preference for in-app navigation, as opposed to browser imposed navigation, is also growing.
The hypothesis is partially confirmed. While the back navigations performed via the application still only make up a minority of all back navigations, use of the panel increased among the same set of users over a relatively short period of time. Moreover it was found that users can be taught to rely on in-app operations like the back panel (rather than native browser ones) with simple affordances in the user interface.
Figure 4.11: back panel usage over time
Chapter 5

Conclusion

Today’s Internet is not ready for the complete removal of the back button. Data shows that the majority of users still depend on the back button, and complete removal of the browser’s chrome could be uncomfortable and inconsistent. But this research demonstrates that application behavior can be discovered, taught, and even self-learned by intuitive users. Elements like the back button are being used less and less, and users are becoming accustomed to the app-like nature of the Internet.

5.1 Drawbacks and Weaknesses

One key drawback of this research is its isolation from the rest of the true Internet. Though the research was conducted under the precondition that data would be gathered from within the context of a web application, more accurate results could be gathered had the study not been isolated to a single app. Access and timing constraints prevented data collection from other apps.

Participants of the study could safely be assumed to be composed of early adopters and expert-users. Google Chrome still holds only a minority of the browser market share and is
not installed by default on the most popular operating systems. The user would have had
to install a new browser on their desktop, been familiar with “app stores”, and installed
the Flixster chrome app; tasks that cannot be easily expected of most mainstream Internet
surfers.

Finally, no subjective data was collected from users. While this research focuses mostly
on measurable, objective information, it could have been helpful to gather user opinions on
usage to help in a qualitative analysis of results.

5.2 Future Work

This work could easily be extended by addressing the weaknesses mentioned previously.
Collecting data about app usage in a different web app, perhaps one that is not focused so
heavily on navigation (like a spreadsheet or mapping application) could provide interesting
results. Additionally collecting subjective data through interviews or surveys to users, espe-
cially “average” users that make up the majority of the Internet’s population, would also be
beneficially.

From the perspective of industry, the landscape of web development is still wide open.
The HTML5 specification is still being implemented and less than a majority of web users
have access this technology. The Internet is evolving beyond the browser. Mozilla is creating
a open app store specification of their own\(^1\). Google is building on Chrome’s success by
building devices that are \textit{just} the web\(^2\). Who knows what innovations the pioneers of the
Internet will bring?

\(^1\)\url{http://apps.mozillalabs.com/}
\(^2\)\url{http://www.google.com/chromebook/}
Bibliography


