Mutable Data Representations for Lightweight Publishing and Curation

by

Scott B. Ostler

S.B., EECS, MIT (2007)

Submitted to the Department of Electrical Engineering and Computer Science

in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

January 2009

© 2009 Massachusetts Institute of Technology
All rights reserved.

Author .................................................. Department of Electrical Engineering and Computer Science

January 21th, 2009

Certified by .................................................

David R. Karger

Professor, Department of Electrical Engineering and Computer Science

Thesis Supervisor

Accepted by .................................................

Arthur C. Smith

Professor of Electrical Engineering

Chairman, Department Committee on Graduate Students
Mutable Data Representations for Lightweight Publishing

and Curation

by

Scott B. Ostler

Submitted to the Department of Electrical Engineering and Computer Science on January 21st, 2009, in partial fulfillment of the requirements for the degree of Master of Engineering in Electrical Engineering and Computer Science

Abstract

Web site authors frequently exhibit collections of structured data that visitors can contribute to, using specialized interfaces. Website authors also desire powerful tools for managing their website's data, so they can integrate visitor suggestions, search past versions of their data, and perform edits. However, content management frameworks have too steep a learning curve for casual users, and require programming to customize how a site's data is generated, viewed, and edited. Incorporating a curation process into lightweight data publishing frameworks can lessen the burden of data submission and management for web site visitors and authors alike. The proposed thesis investigates a data submission process that is easy, customizable, and portable. It also explores the characteristics of successful data management interfaces as applied to evolving sets of data.

Thesis Supervisor: David R. Karger
Title: Professor, Department of Electrical Engineering and Computer Science
Contents

1 Introduction ........................................ 13
   1.1 Related Work ..................................... 15
       1.1.1 Exhibit ..................................... 16
       1.1.2 Wikipedia ................................... 16
       1.1.3 Freebase ................................... 19
       1.1.4 FileMaker ................................... 20
   1.2 Thesis Contributions ................................ 22
   1.3 Outline ........................................... 22

2 Overview of Exhibit .................................. 25
   2.1 Minimal Setup and Startup ......................... 25
   2.2 Transparent, Extensible Data Representation ....... 26
   2.3 HTML Template System ............................ 28

3 Data-Editing Tools in Exhibit ....................... 31
   3.1 Graphical Data Management ....................... 32
       3.1.1 Editable Lenses ............................. 32
       3.1.2 Item Creation Form ......................... 35
       3.1.3 Web Scraper ................................. 37
       3.1.4 Change List ................................ 40
       3.1.5 Submission ................................ 42
   3.2 Changes to Exhibit’s Data Model .................. 43
       3.2.1 Adding an Item-centric Layer ............... 43
3.2.2 Tracking Mutable Item State

3.3 Summary

4 Evaluation

4.1 Analysis of adding data management tools to exhibits

4.1.1 Adding the item-creation link

4.1.2 Creating the item-editing lens

4.1.3 Adding a rel="exhibit/output" link

4.1.4 Conclusion

4.2 User study results

4.2.1 Dorm Sale

4.2.2 the knit project

4.2.3 my gifs // ur gifs // our gifs

4.2.4 The Wang Clan

4.2.5 User study analysis

5 Conclusion

5.1 Future Work

A Study Preparation Email

B Edit Lens Language Reference

B.1 New Language Elements

B.1.1 ex:role="edit-lens"

B.1.2 ex:edit="property-name"

B.1.3 ex:control="start-editing"

B.1.4 ex:control="stop-editing"

B.1.5 ex:role="item-creator"

B.1.6 ex:role="scraper"

B.1.7 ex:role="change-list"

B.1.8 ex:role="submission-button"

B.2 Modified Item Property
List of Figures

1-1 Exhibit of the U.S. Presidents ............................................. 15
1-2 Editing interface in Wikipedia ............................................. 17
1-3 Infobox template for probability distributions ......................... 18
1-4 Infobox for Bernoulli probability distribution ......................... 18
1-5 Bernoulli distribution rendered in browser ............................. 19
1-6 Editing interface in Freebase ............................................... 20
1-7 Data entry form in FileMaker Bento .................................... 21

2-1 Sample Exhibit JSON data file ........................................... 25
2-2 HTML required for a minimal exhibit ................................... 26
2-3 Screenshot of exhibit ...................................................... 27
2-4 HTML for a simple book lens .............................................. 28
2-5 Screenshot of exhibit with book lens ................................... 29

3-1 Exhibit with edit lens active .............................................. 32
3-2 Book edit lens ............................................................... 33
3-3 Example data graph of family relationships ............................ 34
3-4 Compound lenses for editing birth city of step parents ............. 34
3-5 Screenshot of compound lenses .......................................... 35
3-6 Screenshot of an item creation form .................................... 36
3-7 HTML snippet of item submission element ............................ 36
3-8 HTML snippet for web scraper widget .................................. 38
3-9 Screenshot of scraping operation in progress ......................... 38
3-10 HTML snippet from Amazon.com book page .......................... 39
3-11 Screenshot of change list widget .......................................... 41
3-12 Screenshot of a successful submission ...................................... 43
3-13 Subject-Predicate-Object representation of sample Exhibit database . 44
3-14 Exhibit with modified facet in use .......................................... 47
3-15 Structure of a JSON change object ......................................... 48

4-1 Screenshot of exhibit tracking exhibit before modifications ........... 50
4-2 Screenshot of exhibit with item submission form .......................... 52
4-3 Screenshots of Dorm Sale ....................................................... 55
4-4 Screenshots of the knit project ................................................ 55
4-5 Screenshots of my gifs // ur gifs // our gifs ............................... 56
4-6 Screenshots of The Wang Clan ................................................ 57
List of Tables

4.1  Statistics of exhibits made in study  . . . . . . . . . . . .  58
Chapter 1

Introduction

From inception, the world wide web’s adoption has been driven by the personal expression it empowers. Anyone could build an online space for interests and hobbies, to whatever depth and format spare time afforded, simply by editing and uploading text files. Many of these sites presented information important to the author in an organized fashion, breaking down collections of items into categories, and relating them with hyperlinks. The freedom to cheaply publish structured, personally-organized information was key to the popularization of the web.

Time and the web’s growth presented these hobbyists with an information scaling problem. A website that began with fifty items could quickly grow into thousands, straining the navigation and visualization aides the author built into the site. Technically-inclined web developers solved this problem with server-side programming, building sophisticated search and browsing interfaces, powered by relational databases. Programs that performed this function are known as Content Management Systems (CMS) [3], and many were built to accommodate a variety of domain-specific data publishing needs. These solutions accommodated growing volumes of data, richer user interfaces, and more complicated data authorship and publication models, but their technical and administrative complexity was grossly inappropriate for a hobbyist website author. Smaller websites typically offered only a single perspective of the data, without any searching or filtering, and adding new items or views
to hand-written HTML pages was a tedious and error-prone process for the author.

There is a clear need for software that empowers online data-exploration interfaces, while keeping the approachability that drove early web page authorship. Exhibit [9] is a novel web-publishing framework designed for that exact purpose. It runs entirely in the browser, and reinvents the traditional CMS model in two key ways. Data is represented as sets of key value pairs, stored in extensible and human-editable formats such as JSON [4] files or Google Spreadsheets\(^1\). Further, Exhibit users define the presentation of their data within a single HTML document, using regular HTML tags with Exhibit-specific structure and attributes. With this powerful combination, authors can employ Exhibit to make rich, interactive web sites without learning a programming language or configuring a server-side web application.

Exhibit has been used in hundreds of published web sites, with some sites containing over a thousand items. But Exhibit’s success highlights another information-scaling problem – the human cost of maintaining and growing the informational content of a site. If a website visitor finds an omission in a web site, the standard recourse is to email the author with the missing information. If the visitor is unaware of what information is required to describe a missing item, the author must begin a conversation to elucidate it; once the information is gathered, it must be manually entered into the Exhibit database. All the steps in this process are time-intensive and inefficient.

We need look no further than blog commenting to see how the community process has become central to web authorship. That Exhibit does not offer a way to describe changes as first class data is a disappointing barrier between author and visitor. This thesis is primarily concerned with an investigation of how to break that barrier down. The proposed way to do this is by building a set of tools for website authors to incorporate into their exhibits; these tools aim to be simple to use, while allowing

\(^1\)http://docs.google.com/
authors to create powerful data-editing interfaces specific to their site's content and presentation. An author should be able to create item submission and editing forms in under an hour, without needing to confront programming languages or setting up database. This thesis is primarily concerned with the problem of presenting mutable data in the context of a lightweight content publishing framework.

1.1 Related Work

When authors create information displays of mutable data, they must decide how their data is stored, how the data is mapped to a representation that viewers can edit, and how the viewers' edits are merged back into the stored data. This is a complicated process, that has long been of academic interest[11]. In the context of web pages, the author must map the set of data to an HTML representation with a set of associated editing widgets. As background for this thesis, we will review prior efforts at making this mapping process more author friendly.
1.1.1 Exhibit

The official release of Exhibit does not support data mutability, but it will serve as a point of comparison for the other projects in this section. With Exhibit, authors use an HTML mini-language to specify the creation of interactive widgets, and the mapping of database items to rendered HTML. See Figure 1-1 for an example of what Exhibit can be used to accomplish: the boxes in the left column let the viewer select which presidents to show by their attributes, while the map and the timeline represent the selected presidents chronologically, pictorially, and geographically. In this exhibit, filtering, browsing, and visualization are intertwined to present a single cohesive view of the U.S. Presidents. The text search widget in the left column was added with the html tag `<div ex:role="facet" ex:facetClass="TextSearch" />`. Similarly, the two different presentations of Ronald Reagan are specified as HTML blocks mixed with a small number of Exhibit-specific attributes. Only two attributes are needed to create the hyperlinked title shown in the lenses:

```html
<a ex:href-content=".url"><span ex:content=".label" /></a>
```

As will be covered in more detail in later chapters, Exhibit provides a rich data visualization toolset designed for use by web page authors familiar with HTML.

1.1.2 Wikipedia

Wikipedia\(^2\) [16] is the premier online encyclopedia, over two million articles strong. Each article is freely editable by any of its over five million users, showing that an open data publishing model can scale globally. Wikipedia is an incredible success, yet it relies on a very rudimentary data model. Wikipedia articles are specified through a compact, readable markup language called WikiText, which allows users to easily change both the informational content and formatting of an article. WikiText supports a limited specification of structured data in the textual articles, through domain-specific templates called infoboxes. Infoboxes map a list of name-value pairs

\(^2\)http://www.wikipedia.org
Figure 1-2: Editing interface in Wikipedia

to an HTML form. Figure 1-3, Figure 1-4, and Figure 1-5 show a sample infobox template, specific infobox, and rendered infobox respectively. Some templates even conform to an appropriate micro-format [6] specification, showing that semantic data can be encoded in non-semantic formats.

Efforts have been made to extract structured data from plaintext Wikipedia articles to power sophisticated queries and to connect Wikipedia with other semantic data sets, though this approach has many limitations [1, 10, 14, 17]. Others propose semantic extensions to Wikipedia, embedding type declarations into the data of articles via textual tags [15]. But editing structured data through plaintext (see Figure 1-2) is fragile and affords no editing feedback beyond toggling the article’s editing and viewing modes [5]. Authors seeking to construct interactive displays within their articles are limited to sortable tables, with no support for other widgets. The WikiText system powers the world’s largest global encyclopedia, but it is a text-only template system that falls short of Exhibit’s data model and interactivity.
Figure 1-3: Infobox template for probability distributions

```markdown
{{Probability distribution
    name =Bernoulli
    type =mass
    pdf_image =!
    cdf_image =!
    parameters =<math>1>p>0, p\in\mathbb{R}</math>
    support =<math>k\in\{0,1\}</math>
    pdf =<math>
        \begin{matrix}
            q=(1-p) & \mbox{for }k=0
            p & \mbox{for }k=1
        \end{matrix}
    </math>
    cdf =<math>
        \begin{matrix}
            0 & \mbox{for }k<0
            q & \mbox{for }0\leq k<1
            1 & \mbox{for }k\geq 1
        \end{matrix}
    </math>

Figure 1-4: Infobox for Bernoulli probability distribution
```
1.1.3 Freebase

Freebase [2] is a recently introduced service from Metaweb Technologies\(^3\) that like Wikipedia, aims to create “an open shared database of the world’s knowledge.” In contrast to Wikipedia’s system of articles and templates, Freebase organizes its contents with an ontology of topics, types, and schemas. A topic represents a subject of interest with multiple types associated with it that all users can add and remove. Types are defined as an enumeration of properties, each a data slot to be be filled for every topic that belongs to the type. Properties can be primitive types, such as numbers, text, and dates, or they can point to instances of other types as well. Rich javascript editors are provided to edit both type schema and topic values, and present the constraints of Freebase’s topic-centric database model in a friendly, discoverable manner (see Figure 1-6).

Freebase’s editing process is possible because of its structured, schema-based models. However, there is little flexibility or interaction in the data’s presentation, as a

\(^3\)http://www.metaweb.com
given topic consists of a brief descriptive passage and the topic's structured data representation. Freebase intends for third party applications to fill this gap, and provides tools and libraries so third party applications can read and write the Freebase database. However, Freebase requires application authors to use Javascript to create interactive applications, or to load or store data to the Freebase database. Freebase provides a compelling and open platform for interoperable semantic applications, but does not try to shield page authors from the technical complexity of web application development.

1.1.4 FileMaker

FileMaker\(^4\) is an end-user targeted database program with an over twenty-year history, and is currently developed and distributed by FileMaker, Inc., a subsidiary of Apple, Inc. Originally created as a personal database, FileMaker has since gained web-publishing functionality with the addition of FileMaker Server. Similar to Freebas-

\(^4\)[http://www.filemaker.com]
base, users can build data models consisting of primitive types such as text, numbers, media, addresses, dates, and durations, each with its own graphical widget used to display and edit it. Note in Figure 1-7, a screenshot of FileMaker Bento in action, how the larger project editing form contains within it editable subforms for project tasks, important files, and attributes. These are all FileMaker primitive data types, as FileMaker does not support custom or nested data types, nor can user-created editing interfaces be provided for the packaged types. Though the built-in types are attractive and cover a wide range of possible uses, they limit what can be accomplished with FileMaker.

Exhibit and FileMaker provide tools for individuals and small groups to model and present their data, but the form of each tool’s intended presentation differs greatly. An exhibit is a web site that publishes a collection of data with rich filtering and searching constructs, intended for widespread viewing across the web. FileMaker is a local desktop application for data storage and retrieval. Integrating FileMaker with external data sources and applications is possible, but requires considerable technical ability. FileMaker doesn’t make publishing data to the web easy either, as users must purchase an expensive FileMaker Server program and host it on their own server. The option of data publication is intended for corporations with budgets and dedicated
web developer and IT support staff, and is clearly infeasible for hobbyist purposes. Though FileMaker shows that data-editing forms can be assembled graphically via a rich, interactive interface, it also shows the dangers of a proprietary data model built without concern for extension, web publication, or interoperability.

1.2 Thesis Contributions

The proposed thesis is an investigation of how to integrate user-generated data and dynamic data-editing tools into the Exhibit framework. The goal is to support two new modes of data management: submission, where users can submit changes and additions to an existing exhibit, and curation, where the exhibit author can securely review those changes and choose which to accept. These two modes are complementary, and can freely be mixed and matched within a single exhibit. The thesis will explore the dual aims of providing simple primitives upon which these modes can be constructed, while minimizing the imposed administrative and technical burden.

The research will also provide a significant contribution to the Exhibit project, a valuable feature for the Exhibit user-base, and a better understanding of how to successfully integrate mutable data sources into lightweight publishing mechanisms.

1.3 Outline

This first chapter has been an introduction to this thesis’ goal of empowering zero-configuration customizable data-editing interfaces, along with an overview of relevant online interfaces and data models. The second chapter describes Exhibit and the features which make it a compelling base upon which to build a data collection and editing extension. The third chapter details how such an extension was implemented, and the design decisions made. The fourth chapter explains the results of the thesis, by analyzing how well the implemented extension suits the overarching Exhibit frame-
work, and the extension’s success in powering lightweight data editing interfaces. The fifth chapter provides a summary and explores directions for future work.
Chapter 2

Overview of Exhibit

Exhibit [9] is a unique testbed to explore representations of mutable structured data, because of its unorthodox strategies for empowering user-driven data publishing, and its base of active users. Webpage authors can use Exhibit to create and customize interactive presentations of structured data, while employing data filtering and selection tools.

2.1 Minimal Setup and Startup

Novice authors can easily get started with Exhibit, as there is little required configuration, and a rudimentary item view can be added to a web page with a few lines of HTML. Consider building a minimal exhibit to show off a small collection of books.

```json
{
  items: [{
    label: "Great Expectations",
    author: "Charles Dickens",
    year: "1861"
  },
  {
    label: "Tom Sawyer",
    author: "Mark Twain",
    year: "1876",
    url: "http://en.wikipedia.org/wiki/Tom_Sawyer"
  }]
}
```

Figure 2-1: Sample Exhibit JSON data file
Figure 2-2: HTML required for a minimal exhibit

Figure 2-1 is the data file that encodes the data to be presented, and Figure 2-2 is the HTML page that presents that data. Figure 2-3 shows how that exhibit appears in a web browser. All that's required to include the Exhibit framework is the single script tag in the document head. Though this exhibit is threadbare and uses Exhibit's default item view, it contains an advanced sorting widget that can sort the items by any combination of the items' attributes. The Javascript to implement this feature alone is likely too complex for amateur web authors, as building a multi-attribute sorting control requires extensive manipulation of the page's DOM. Even for experienced web developers, getting this for free is a great boon.

2.2 Transparent, Extensible Data Representation

Exhibit's data representations are open, flexible, and self-documenting. By way of example, consider the JSON representation in Figure 2-1 – to add new items, or new attributes to existing items, the author only needs to edit a single text file. Another common representation is to store Exhibit data in Google Documents spreadsheets. Using the familiar tabular model where rows in a spreadsheet correspond to items, and columns correspond to attributes, managing an exhibit's data is like editing a spreadsheet. Even if automated tools modify an exhibit's schema or contents, it remains understandable and still easily editable – a new item attribute becomes a new column, and a new item becomes a new row.
Adding new items or attributes to relational databases such as MySQL, however, requires programming languages or command line tools. A representative example can be found in Django\(^1\), a Python web development framework, which solves this problem by providing a graphical data editor. While Django’s admin module is a powerful and useful tool, it solves a problem of complexity by adding more, as Django is a full stack web application framework intended for large corporate web applications. It only works when installed into a system with a compatible Python version and a configured database, and requires user effort to maintain coherence between database changes and codebase changes. By contrast, Exhibit’s storage of data in lightweight, human-editable formats means that hobbyist authors are not reliant on complicated tools or programming languages. Authors can take direct ownership of their database’s structure and contents, even as automated tools modify

\(^1\)http://www.djangoproject.com/
2.3 HTML Template System

Exhibit authors create lenses to specify how the exhibit’s items should be mapped into user-presentable HTML. The lenses are written in an HTML mini-language, with exhibit-specific attributes encode rules guiding their interpretation. Figure 2-4 gives a rudimentary lens to display the book items from Figure 2-1.

This lens is no more complicated than a "find-and-replace" insertion of item attributes into the tags marked with \texttt{ex:content}. Figure 2-5 shows the lens in action. The lens system also supports more powerful functionality, as well, such as inserting item attributes into HTML attributes, or conditional inclusion. Consider the following addition to the book lens:

\begin{verbatim}
<div ex:if-exists=".url">
  <a ex:href-content=".url">More Information</a>
</div>
\end{verbatim}

These three lines extend the lens to include a link to the item’s \texttt{url} attribute, only if such an attribute exists. This brief example contains all of the lens syntax in wide use by the Exhibit community, as authors create intricate lenses by chaining these few simple building blocks together. This composability lets novice authors create simple lenses quickly, then iteratively grow their exhibit by copying more advanced features from tutorials and existing exhibits. Exploratory, example-driven development breaks
down the required learning into manageable chunks, as authors can refine each part of their exhibit until they achieve the desired result.
Chapter 3

Data-Editing Tools in Exhibit

Building sufficiently powerful data-editing tools into a framework like Exhibit is a challenging task. Briefly, the aims of these tools are to:

- Strike a balance between utility and simplicity, accommodating the range of needs that Exhibit authors have, while not burdening them with excessive or extraneous features.

- Power a data management workflow that meets the expectations of both exhibit authors and visitors, within the constraints of an in-browser Javascript application.

- Be consistent with the Exhibit ecosystem, so users can leverage their accumulated Exhibit expertise while learning to use the new extension.

These constraints are limiting, but they mould the shape of their solution. This chapter examines the difficulties that the task presented, and how they were resolved. The chapter’s first half details the graphical data-editing tools that have been added to Exhibit, and the language that exhibit authors use to add them to their exhibits. The second section concludes with the changes made to the Exhibit database.
3.1 Graphical Data Management

3.1.1 Editable Lenses

In Chapter Two, Exhibit’s lens system was provided as an easy way for authors to present their data. Extending that system with tools for data-editing is the key concern of this thesis. The introduced solution is the edit lens – while normal lenses provide the static views of exhibit items, edit lenses contain data entry widgets for modifying item attributes. Figure 3-1 is a book collection exhibit with an active edit lens at the top of the screen. The edit lens adopts the structure of a web form, with labels describing input fields and drop-down menus. There is a hyperlink to replace the edit lens with a normal lens, while in the normal lens directly below the edit lens, the edit hyperlink plays the opposite role. When a value in either the text fields or the drop-down box is changed, the change propagates to the exhibit’s database and the graphical elements are updated to reflect the new value of the item.

The syntax used to create edit lenses has much in common with the normal lens
Figure 3-2 lists the HTML required to create a simple edit lens, which is very similar to the normal lens shown in Figure 2-4. The only significant difference is that the div elements with the ex:content attribute have been replaced with input elements with a ex:edit attribute. (The value of the ex:role attribute has been changed from lens to edit-lens.) The interesting difference between the two lens syntaxes is that the attribute names in the ex:edit values are not prefixed with a period. This is because the ex:edit tag does not accept the Exhibit expression syntax\(^1\). Rather, only an item’s immediate properties can be edited. Adding edibility to Exhibit’s full expression system was too complicated, and the lack of support for editing a property with multiple values limits the utility of the feature\(^2\). However, editors for extended expressions can be simulated by chaining standard Exhibit expressions with a custom edit lens, demonstrating how orthogonal features can be combined in powerful ways.

Consider an example expression from the Exhibit documentation that refers to the subject’s parents-in-law: `.spouseOf!parentOf`. An exhibit author wants to extend this by allowing visitors to change the city that the parent in law was born in: `.spouseOf!parentOf.bornIn`. Evaluating this expression on John in the data graph of Figure 3-3 results in `{Boston, Seattle}`. The solution is to use the expression `.spouseOf!parentOf` to identify the subjects of the expression’s evaluation, then to use the edit expression bornIn to identify the attribute on Fred and Sally that

\(^1\)http://simile.mit.edu/wiki/Exhibit/Expressions

\(^2\)The lack of multi-valued editing, and the restrictions placed upon edit expressions, are both identified as serious shortcomings.
should be made editable. Figure 3-4 gives the HTML to achieve this, and Figure 3-5 is a screenshot of the lens in action.

(Note in Figure 3-4 that there are input elements with the ex:edit tag, but the lens is not an edit lens. Exhibit authors can add static content to their editing lenses alongside the editable content, as was demonstrated in Figure 3-2. This goes both ways; editing fields can also be added to normal lenses.)
3.1.2 Item Creation Form

On the Exhibit mailing list, requests have been made for data entry forms that authors and visitors can use to submit new items to exhibits. Online data submission forms are plentiful, and a number of online services let users create, distribute, and monitor data collection forms. Google Documents recently introduced support for data submission forms that spreadsheet owners can embed on their website or blog. With the edit lens feature in place, item creation forms were only a small step away.

It is not complicated for an exhibit author to add an item creation form to their exhibit. When an author adds an ex:role="item-creator" attribute to an element, clicking that element triggers the generation of an item creation form in the center of the web page. The form consists of the edit lens along with two buttons, one to cancel.

---

3 One such example: "I wonder if anybody has attempted the following: given an Exhibit item, programmatically construct an HTML form to enter such items".


5 http://googledocs.blogspot.com/2008/05/embed-your-forms.html
the submission process, and the other to complete it, closing the form. Figure 3-6 contains a screenshot of a live item creation form; the attributes of the item being created can be seen in the owner facet, and the changes box on the right.

There are a number of options that can be configured as an HTML snippet copied from a published exhibit demonstrates in Figure 3-7. Most of the options are cosmetic ones that let the user customize the text of buttons and alert messages, but one determines the mode of operation of the creation form. When ex:automaticallySubmit is set to true, adding a new item via the form also immediately submits the item to
the external service that collects submissions. In the absence of automatic submission, changes made to the database are queued up and submitted en masse when the exhibit visitor clicks the submission button. The item creation form is unique among the data editing widgets in that offers the option to immediately submit its changes to the external service. This option ensures that exhibits can be built with a quick and mostly stateless item submission process.

### 3.1.3 Web Scraper

The web scraper widget is an interesting example of how data, even in small amounts, can be leveraged to facilitate the collection of further data. The role of the web scraper widget is to try to extract a new item from the contents of either a web page or a snippet of plaintext. A visitor to an exhibit would activate the scraper by entering the url or text they want to scrape into an input element, then clicking on a hyperlink or button. If the input is a url, the contents at that url are retrieved by using an external scraping service, and the DOM structure of the retrieved page is converted into plaintext. Once the plaintext has been retrieved via the external service, the scraper extracts attributes from the text using the existing contents of the database as a guide. An new item creation form is made, populated with the scraped attributes.

The syntax for adding a web scraper widget is very similar to the syntax for the item creation widget. The key difference is that when providing the options for the scraping widget, the element id of a text field in the page must be given. When the visitor clicks the scraping widget, the contents of the text field are provided as input to the scraping module. By default, the scraping module uses a simple heuristic to test whether the input is a url or plaintext. Figure 3-8 lists an HTML snippet that contains a scraping link and its corresponding text field.

---

6 This external service is intended to be a Google Spreadsheet, though an exhibit could be configured to submit its data to an alternate service.

7 The purpose of using an external service to download the contents of the web page is to circumvent the browser’s security restrictions.
Scraping is a minimal-cost operation for users to perform, as they only need to paste text into a text field and click a button. The time spent waiting for the url to be scraped is almost always less than a second, barring unusual network conditions, and the time spent could be reduced further by starting the retrieval operation in the background once the url is entered into the text field. This was not implemented due to the expectation that scraping is not a high-value feature in its current incarnation.\textsuperscript{8}

To see a scraping operation in action, consider Figure 3-9. Scraping the product

\textsuperscript{8}It is well known that even fractional seconds of interface latency can significantly hurt uptake of a service or feature. At a 2006 conference Google Vice President Marissa Mayer shared Google findings that indicated a half second delay caused as much as a 20% drop in user traffic\cite{13}.
Figure 3-10: HTML snippet from Amazon.com book page

page for a book on Amazon retrieved five attribute values, though unfortunately, all but one were wrong. The extraction strategy is to match existing attribute values in the database to the scraped page’s text, so the book title already in the database is extracted, not the book that’s the topic of the page. Similar reasons explain the other extraction errors. By way of contrast, a simple yet promising strategy is range matching. Exhibit databases often contain numerical or date attributes, whose values often fall within a range. For example, an exhibit of apartment listings will have availability dates within a two month range, prices that fall within a thousand dollar range, square feet that falls within a range of a few hundred, etc. These ranges, along with standard deviations and more complicated statistical measures, can be used pick out numbers from a page and match them to attributes with varying degrees of probability. This worked well when extracting apartment information, which is marked by numerical attributes that fall into narrow ranges.

A more ambitious strategy is to analyze the HTML structure of the document being scraped, and infer meaning and importance from local context. For example, consider Figure 3-10, in which all sorts of information can be recovered from even a
rudimentary scraping. For example, the text within the h1 tag perfectly describes the subject of the page, and the class and id attributes further provide that the attribute being given is the book’s title. Likewise, the author of the book can be captured by noting that the a tag whose contents are the author’s name, are right next to the text “(Author).” While extracting information from web pages is a difficult task that large corporations devote considerable resources, the scraping performed by Exhibit only needs to extract a few attributes that often are listed prominently on the page. Without prototyping and testing, this strategy’s success cannot be known, but there is reason for optimism. Work has also been done with user-annotated data extraction from high value sites [7, 8], and Exhibit could either reuse existing web page annotations, or offer tools for authors to add their own annotations to their exhibits. To further improve the robustness of item extraction, several independent strategies can be combined, then for each attribute, present the best extraction from each strategy. These topics are an interesting direction for further research, and Exhibit has shown itself to be a useful platform for exploration.

3.1.4 Change List

Making the editing process understandable, so users do not lose track of the edits they have made, is a key goal of this work. To this end, an Exhibit component has been built to display the changes made to the exhibit. Figure 3-11 shows the list in action on the right of the screen. The change list widget was initially much more complicated than its current form. It now has four configurable options, which are used to configure cosmetic options such as stock messages and rules for truncating long values. Here is a sample usage:

```html
<div ex:role="change-list"
    ex:placeholderText=""
    ex:maxValueLength="40"
    class="changes" />
```

Initially the change list incorporated author-provided HTML snippets that would be used as templates for the various components in the change list. The templates
would create the frame of the change list, the heading for each changed item, and the
details for each changed attribute. This configurability was removed, as it attracted
little user interest and did not benefit the core use cases that drove development.

While the previous configurability was unnecessary, the reintroduction of a suffi-
ciently pared-down template mechanism might be beneficial. One alternative would
be for exhibit authors to provide html snippets via the standard Exhibit option syntax,
using special strings to signify where the variables need to be inserted into the tem-
plate. In another alternative, exhibit authors would provide the names of Javascript
functions that map the input values into the HTML structures that are rendered on
the page. The second alternative would require Javascript programming ability, but
would be consistent with other schemes for Exhibit customizations, like the styling
of rows and tables in the tabular view element\textsuperscript{9}. Whether this customizability would
be useful to a broad swath of the Exhibit community is unclear.

\textsuperscript{9}http://simile.mit.edu/wiki/Exhibit/2.0/Tabular_View
3.1.5 Submission

In the final step in the editing process, the exhibit visitor submits the changes she has made. The goal of submission is to save the information for each changed item as a row in a Google Spreadsheet, for the exhibit author to review later. When the user clicks the submission button, a JSON-encoded representation of the changes to the exhibit is submitted to an external service. The external service decodes the JSON-encoded changes and adds them to the Google Spreadsheet. If the submission to the service returns successfully (see Figure 3-12), a call to $\texttt{database.fixAllChanges}$ is made. This call cements the current state as permanent, and reverting future changes will return the database to this state. If the call does not successfully complete, the exhibit visitor has the option of retrying the submission, should they suspect the error was transient. The submission button element is specified through the following syntax; the lack of configurable options makes the button easy to add to an exhibit:

\[
\texttt{<input type="button" ex:role="submission-button" value="Submit Changes" />}
\]

The change submission is configured in a $\texttt{link}$ tag in the exhibit’s head element, paralleling how the loading of exhibit databases is configured with link tags. This also decouples submission configuration from the submission button, which proved useful when providing the option for the item submission form to automatically submit changes. Configuration involves identifying the Google worksheet that the changes will be submitted to; this is done by providing the spreadsheet id, along with either the name or index of the worksheet (if neither are provided, a worksheet index of 0 is assumed). By setting the $\texttt{ex:url}$ attribute, the exhibit author can provide an alternate url destination for changes. Even if few exhibit authors find this useful, a pluggable submission service ensures an important degree of flexibility. Were an exhibit author concerned about the privacy or security of relying on a centrally hosted submission service, the author could run her own alternate submission service\(^\text{10}\).

\(^{10}\)The changes are sent as plaintext JSON objects, meaning any language with a JSON library could be used to write a submission service.
3.2 Changes to Exhibit’s Data Model

Items in an exhibit instance are stored within the Exhibit database object, which the Exhibit UI components query to generate their contents. As Exhibit’s database was only designed to handle static data, the database needed modification and extension to support the new demands of data mutability. We will explore the major changes that were required.

3.2.1 Adding an Item-centric Layer

The data in the Exhibit database is represented by a collection of \((subject, predicate, object)\) tuples, inspired by the RDF model [12]. Items are the set of all tuples that share an identifying \(subject\) key. In the book example introduced in Chapter 2, the label attribute is the identifying string for each item. (In Exhibit, the identifying string is always either the \(label\) or the \(id\) attribute of the item.) For each attribute of an item, there is a tuple of the form \((item\_id, attribute\_name, attribute\_value)\) stored in the exhibit database. See Figure 3-13 for a listing of the subject-predicate-object tuples that correspond to Exhibit’s internal representation of the database in Figure 2-1.
This design makes it easy to answer questions such as “What year was Tom Sawyer written in?” and “What books are written by Mark Twain?” The database uses two parallel data structures: one to map subjects to objects, and the other to map objects to subjects. The answers to the two questions are generated in an almost identical fashion, with the only difference being the directionality of the data structure used to answer each question. Tuples were not intended to be removed or modified once inserted, and four new methods were added to support a mutable-item abstractions.

- **database.addItem(item):** takes the attributes of item and turns them into a new item in the database. For example, adding Tom Sawyer a database would be done by: database.addItem({label: "Tom Sawyer", author: "Mark Twain", year: 1876}). If no addedOn attribute is present in item, it is set to the current time.

- **database.getItem(item_id):** collects the attributes of a given item into a Javascript object which is returned. For example, the call database.getItem("Tom Sawyer") would return {id: "Tom Sawyer", label: "Tom Sawyer", type: "Item", uri:...}.

Note that the Exhibit

---

11"http://localhost/sostler/books/item#Tom%20Sawyer" – not included inline for sake of space.
generated attributes id, type, and uri are included in the returned object.

- `database.editItem(item_id, attribute.name, new.value)`: sets the value of an existing item’s attribute. If a value for that attribute is already present, it is overwritten. Adding that *Tom Sawyer* was written in English would be done by `database.editItem("Tom Sawyer", "language", "English")`, for example.

- `database.removeItem(item_id)`: removes the item, and all its attributes, permanently from the in-memory Exhibit database. Calling `database.removeItem("Tom Sawyer")` would remove the *Tom Sawyer* item from the exhibit.

One alternate design would be extending the tuple-centric API with calls like `addTuple` and `removeTuple`. The item-centric API was added to better support Exhibit extensions that work with items and attributes, however this functionality could have been layered elsewhere without modifying the core database API.

These data-editing methods use Exhibit’s event notification system to update the user interface when called. Because there is no mechanism to selectively update only a part of an Exhibit component, all of the views and facets in an exhibit must be redrawn after a data editing operation\(^\text{12}\).

### 3.2.2 Tracking Mutable Item State

In order to undo or submit modifications made to an exhibit’s contents, Exhibit was extended to track the state changes that an item can undergo:

- *modified*: items whose attributes have been changed by the user via the item editing interface. An item can be set as modified if the exhibit visitor adds it through an item creation form, or if the visitor uses an editing form to add or

---

\(^{12}\) Redrawing the entire Exhibit whenever an item is edited is an expensive operation that is unpleasant for the visitor. If Exhibit were extended to map each item to its generated interface elements, only the invalidated elements would need to be regenerated when an edit is made. Though this modification would require significant changes to Exhibit’s database and interface, it would improve the responsiveness of editing complex exhibits.
change attributes on the item. Calls to addItem and editItem set the item in question as modified.

More granularity could be employed, by exposing the item types and attributes that have been modified. Exhibit authors could provide a facet for selecting which item types or modified attributes are relevant. For a frequently modified exhibit, with a wide domain of disparate items, this extension would let site users better explore a change set.

- *new:* items created outside of the standard item loading process. New items are a subset of modified items, as new items are always the result of user interaction through the item creation interfaces. One clear extension would be support for a *new* item property, similar to the *modified* property.

- *original values:* when an attribute of an item is first modified, the original value is stored in an ancillary data structure used to record the initial states of items. This has two uses; the first is to track when modifications to an item returns the item to its initial state. When this happens, the item’s modified status is removed. (This is to ensure that if a user modifies an item, then reverts it to its original state, it is not marked as modified.) The second purpose is to undo the modifications made. These values are set as the result of editItem calls.

Additional methods were added to the database API to allow higher-level Exhibit components to interact with the recorded item state:

- `database.collectChangesForItem(item_id):` bundles the changes to the item into a Javascript data structure. This structure can be be rendered as HTML for the user, or submitted to an external service. Fig 3-15 depicts a sample change object.

- `database.collectAllChanges():` a convenience method that collects the changes of all items at once into an array of change objects.
Figure 3-14: Exhibit with modified facet in use

- `database.fixChangesForItem(item_id)`: removes all tracked state for the item, clearing the recorded original values, along with the modified and new flags. This method is called after an item’s changes have been persisted to an external service.

- `database.fixAllChanges()`: a convenience method that fixes changes for all items at once.

With these changes in place, other elements of Exhibit can modify the items in the database and query for item changes.

### 3.3 Summary

This chapter has provided an overview of the graphical data-editing tools that have been added to Exhibit and the internal changes to Exhibit’s data model required to accommodate them. The role these tools can play in Exhibit has been outlined, and
Figure 3-15: Structure of a JSON change object

the syntax exposed to authors has been detailed. The relevant design decisions have been discussed and the limitations of the current implementation have been outlined.

The next chapter will address the question of how well these tools meet the needs of exhibit authors.
Chapter 4

Evaluation

When evaluating software, the starting point must be to ask “Who will use it?” Exhibit is a tool for web site authors with little or no programming experience, who can use copy-and-paste to build their sites from existing examples. When building software for hobbyists, a feature’s conceptual weight and complexity must be weighted against the feature’s benefits when used correctly. A feature must be simple to describe quickly, easy to paste into existing exhibits, and it must “fit” within the surrounding Exhibit ecosystem.

The data-management tools will be evaluated in two ways. First, an overview of adding the tools to a sample exhibit will be given to show how a typical exhibit author would use them. Second, a study was conducted of the tools, where users new to Exhibit made an Exhibit, then added the data-management tools, while under close observation. These sections demonstrate the benefits and costs of the new tools.
4.1 Analysis of adding data management tools to exhibits

This section shows how an exhibit author would add an item submission form to an existing exhibit. The exhibit in question is a listing of other exhibits, containing the author's contact information, a description of the exhibit, statistics about the contents, and any other comments from the exhibit author. By submitting their exhibits, authors can show off their work to the Exhibit community.

4.1.1 Adding the item-creation link

The first step of adding an item submission form is to insert the link that opens the item submission form. This is done by adding a link with an attribute of `ex:role`
with a value of `item-creator`:

```xml
<a ex:role="item-creator"
   ex:itemType="exhibit"
   ex:automaticallySubmit="true"
   ex:submissionMessage="Thanks for submitting your exhibit!"
   ex:cancelButtonText="Cancel"
   ex:createButtonText="Submit Exhibit">
   & Add your own exhibit
</a>
```

The options used to configure the item creation link have been covered in Chapter 3. When the item is submitted, the changes are automatically sent to the external submission service and written to the backing spreadsheet.

### 4.1.2 Creating the item-editing lens

Once the item creation link is added, the contents of the item submission form can be customized. This is done by adding an edit lens. The resulting item lens can be seen in Figure 4-2 (the bottom half looks similar, with two buttons that say Cancel and Submit Exhibit). Here is a partial listing of the edit lens:

```xml
<div ex:role="edit-lens" style="display: none">
  <div class="title">
    <span>Add your Exhibit!</span>
  </div>
  <div class="item-entry-field">
    <div class="field-label">Exhibit name <b>*</b></div>
    <span class="field">
      <input ex:edit="label"></input>
    </span>
  </div>
  ... [etc.]
</div>
```

### 4.1.3 Adding a `rel="exhibit/output"` link

The final step is to provide the exhibit with the configuration details required to submit items to the external submission service. This is done by adding the following
4.1.4 Conclusion

Those three steps were all that was required to add the item submission form to the exhibit. Perhaps more importantly, all three additions could largely be copied into the exhibit, with only minor edits required to customize the features for this specific exhibit. The only significant amount of work required was in customizing the edit lens, which itself could be jump-started by providing a minimal edit lens with just a text field for the `label` attribute. If this were done, a minimal item submission form could be added to an exhibit in minutes, even including the time required to set up the Google Spreadsheet, after which the author can iteratively refine the item.

\footnote{The `ex:spreadsheetKey` value can be copied from the url of the Google Spreadsheet.}
submission form to their liking. Low cost, exploratory development encourages rapid addition of new content and experimentation with style and design.

4.2 User study results

To assess the usability of the introduced extensions, a study was performed in which four participants were asked to create a mutable exhibit on a topic of their choosing. As preparation for the study, the participants came up with a data model for their topic, made a database of five to twenty items, and designed the general layout of the website. (See Appendix A for the preparatory email they were sent.) The studies were conducted one-on-one, and ranged in length from two to three hours. Each study consisted of three parts: designing the site in HTML, adding the standard Exhibit tools to the site, and finally adding an item submission form.

First, the participant copied their data into a Google spreadsheet. The participant then created the site’s layout, including the hand-written HTML for a single Exhibit item. I instructed them in transforming the static HTML into a fully dynamic Exhibit lens. (With all participants, I needed to provide explicit instructions for adding links and images to the lenses.) The edit lens was created similarly. Though I took an active role in the study, I was careful to refrain from suggesting designs except where their initial design was prohibitively complex or where they had trouble understanding how to adapt the Exhibit sample code. I tried to limit my involvement to pasting existing code from the Exhibit documentation or the exhibit we were creating, adding attributes to existing tags, and correcting errors.

At the conclusion of each study, I asked the participant how well they realized their envisioned exhibit, what expectations they had for the data management tools that weren’t met, and what most bothered them about the exhibit as it stood. Despite the large variability in development style, what each participant focused on, and the resulting exhibit, I gained a sense of what was intuitive about the tools, and what
consistently butted against the users’ mental models. First, a short description of each created exhibit will be given, after which will follow general insights gleaned from performing the study.

4.2.1 Dorm Sale

In most dorms, there exists a thriving second-hand economy, mostly conducted by word of mouth and posted flyers. Though MIT has long had a mailing list for reselling items, and services like Facebook Marketplace\(^2\) offer localized marketplaces for college communities, there is still interest in selling locally to friends down the hall. The first participant, a twenty-year-old college student with industry experience in web services, wanted to create an exhibit for dorm residents to sell items, with sorting and filtering controls to assist buyers. The subject cited Craig’s List as a key inspiration for her design. Figure 4-3 shows the exhibit and its submission form. Much of the session was spent on technical configuration issues and in understanding the structure and limitations of Exhibit, with more advanced features left unexplored.

4.2.2 the knit project

Knitting has moved online in a big way. Ravelry\(^3\), a knitting social networking site still in beta, boasts hundreds of thousands of users and a waiting list thousands long. The second participant, an eighteen-year-old computer science student with experience programming wanted to create a site of her own, that she could customize and invite her friends to use. She and her friends would use the site to post pictures of their knitting projects, who the projects were for, the materials and designs, etc. We worked through the core functionality, and from there the participant focused on the cosmetic appearance of the site, finally arriving at the exhibit shown in Figure 4-4.

\(^2\)http://www.facebook.com/marketplace/
\(^3\)http://www.ravelry.com
Figure 4-3: Screenshots of Dorm Sale

Figure 4-4: Screenshots of the knit project
The third participant, a twenty-three-year-old digital artist with experience building web sites but with little programming experience, wanted to organize a collection of animated gifs that he and his friends had accumulated for use in design projects. Exhibit was a good fit for his needs, as he could design his own gallery with tagging, searching, and filtering to manage is growing size. He has hundreds of these gifs, his collection weighing in at eighteen megabytes, but as Exhibit only displays a fixed number of items at a time, he can browse and search the collection without his system grinding to a halt from animating hundreds of gifs at once. We started by spending some time on the Google Spreadsheet used to back the website, using one of the item attributes as a generic tagging system to describe the subject matter of the image (i.e. amazing, fun, religious, etc.), with others to categorize the type of image (text, transparency, etc.). After getting the exhibit working with submission, the participant turned to cosmetic customization, including creating a banner image, figuring out how to use the Exhibit lens system to overlay the labels of items over the images, etc. The final product can be seen in Figure 4-5.
4.2.4  The Wang Clan

The final participant, a twenty-three-year-old quantitative analyst with some experience in web application programming, wanted a site for finding people to share cell phone family plans, for the purpose of saving money. Originally the participant's goal was to make a system that would perform some form of matching algorithm on the users to suggest potential matches for family plans, but we quickly agreed this was too ambitious for a three hour study. This participant was interested in using the advanced functionality of Exhibit's lens system to drive the resulting markup; for example, the `ex:if` control is used to control whether the line "likes text messages" or "does NOT like text messages" is used, and the `ex:if-exists` control determines whether the text "but is willing to switch!" is appended after the area code text. Figure 4-6 shows off this exhibit.

4.2.5  User study analysis

All of the user studies resulted in the successful completion of the designed exhibit. The participants were eager to see their ideas realized, and after a few example exhibits were shown, they saw how their idea could be expressed within Exhibit. As Table 4.1 shows, the participants all created rich exhibits to exploit their problem...
domain. They also had little problem incorporating the data management tools once they had built their Exhibit. In fact, one of the participants expressed surprise that the items in a vanilla Exhibit database could not be modified, unaware that the majority of exhibits store their data in static text files.

<table>
<thead>
<tr>
<th>Exhibit</th>
<th>Items</th>
<th>Properties</th>
<th>Facets</th>
<th>View Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dorm Sale</td>
<td>5</td>
<td>7</td>
<td>2</td>
<td>Tile</td>
</tr>
<tr>
<td>Knitting</td>
<td>12</td>
<td>11</td>
<td>8</td>
<td>Tile</td>
</tr>
<tr>
<td>Gifs</td>
<td>20</td>
<td>5</td>
<td>4</td>
<td>Thumbnail</td>
</tr>
<tr>
<td>Wang Clan</td>
<td>11</td>
<td>11</td>
<td>3</td>
<td>Thumbnail</td>
</tr>
</tbody>
</table>

Table 4.1: Statistics of exhibits made in study

The same few concerns arose with most or all of the participants:

- Care is not taken to ensure that the attributes in a submission can successfully be saved into the spreadsheet. In two of the studies, the first item that was submitted left a mostly empty row in the Google Spreadsheet. This revealed two complimentary flaws: Exhibit cannot correctly read Google Spreadsheets that have column names with spaces in them, and Exhibit’s submission relies on a Google API that does not recognize item attributes with spaces. More generally, Exhibit does not alert the user when a submission operation would partially fail due to a mismatch between an item attribute and the spreadsheet’s column names. Errors in participants’ exhibits manifest as silent data loss during submission.

- The edit lens uses HTML form tags to let visitors edit the values of an item, which is an effective and understandable syntax for Exhibit authors. More work should be done to extend that metaphor by allowing exhibit authors to use hidden inputs to hard-code values into newly created exhibit items, or to specify computed attributes or timestamps. Two of the users, familiar with HTML forms, wanted to specify item timestamps as form input elements – implementation details shape user expectations. Reusing well-known structures
and metaphors lets users carry over built-up knowledge and intuition.

- Exhibit should bundle Javascript editing widgets for more powerful data input. Exhibit authors could use combo-box widget, for example, to let visitors choose between preselected values and provide their own value if needed. A date input widget would make entering dates much more intuitive than manually entering the date string. Two of the participants expressed disappointment that the input options didn’t extend beyond simple HTML forms. One simple solution would be to package some of the better JQuery UI widgets with Exhibit and make them easy for authors to employ.

- The purpose of making existing Exhibit items editable was significantly over-emphasized in the development of the graphical tools, and focused on the wrong editing tasks. None of the participants were interested in any sort of en-masse editing, and did not see the point in queueing changes before submitting them. The exhibits didn’t use any of these features, and an uncontroversial extrapolation is that the item-creator widget should default to submitting newly added items to the Google Spreadsheet, as authors were irritated at adding an additional tag to get what they thought should be default behavior.

Further, the edit lens concept was not well received by the participants. The Wang Clan author liked customizing the data submission form using HTML, but did not like that his exhibit’s views and facets featured the information in the new item form prior to its submission. He saw the information in the submission form as transient, and thought it should not be confused with the exhibit’s permanent contents. He was also unhappy that default item labels like “Untitled Item 1” were automatically generated, even though they were placeholders meant to be overwritten. The strength of his reactions surprised me, but they showed the importance of his mental model for his exhibit. Similarly,
the Dorm Sale author wanted the sold status of items to be editable, but was unhappy that the same form used for submitting a new item would be used to edit existing ones. For her, once an item was submitted, all but a few fields should be fixed and no longer editable. Both of these reactions demonstrated the need for more author control over the item submission form.

- The participants were interested in letting item submitters change only their submitted items, and restricting them from changing items submitted by others. All but one of the exhibits was built to let visitors of the site share information amongst themselves. An interesting extension would be to add a user-submitted ownership model to Exhibit, so visitors could exercise limited control over the items they submit.

The study’s methodology was designed so that participants would build complete, functioning exhibits without any prior experience with Exhibit. Much of the sessions were spent dealing with the core Exhibit syntax, with the item submission form added only at the end, and as a result, the editing component received less attention and mental focus. Though the participants designed their exhibits to meet a data collection need, the artificial environment of the survey resulted in subjects expressing less ownership over the design and creation of their exhibit. Two of the participants frequently sought approval for design decisions that were solely matters of taste, implying that whether out of detachment, indecision, or anxiety, they were not fully engaged in creating their exhibit.

A few weeks after the study, I took the opportunity to discuss with two participants their thoughts on the study, and on Exhibit in general. Both believed very strongly that edibility deserved a central role in Exhibit, and that from their brief exposure, Exhibit was most useful in providing compelling item visualizations such as the Timeline view and the Google Maps view, item filtering via facets, and in incorporating visitor content. For them, these three features were part of a single
process of content generation and consumption. Two of the participants designed sites with content generated and consumed primarily by visitors, while the other two participants designed sites for a small group to gather and present a shared body of information. The rise of social networks have made user-generated content king, and the roles of author and visitor have blurred.

One of Exhibit’s goals is to provide the interactivity of modern web applications to hobbyist web authors. The study participants were unanimously gratified in seeing their database and website designs come to life in a rich, interactive interface. Their positive reactions validate the worth of this work, and support the assertion that this space deserves further investigation. It is significant that all four participants were able to put together a fully interactive site in a few hours, with little more assistance that would be provided by a web site tutorial or a screen-cast.

Equally clear was that the experience was still technically daunting for the participants. I believe much of the study (e.g. coding the HTML, manipulating and copying images, transferring the site’s files to a web host) was not intuitive or pleasant, not even for those who had previously made websites or written programs. This helps explain why none of the participants have used their sites after the study. One of the participants explained that the process felt overwhelmed by extraneous detail, and that the fun of making the database and laying out the web page was overwhelmed by writing HTML and troubleshooting errors. One conclusion I draw is that even small changes can make a software platform vastly more desirable. Providing clear documentation, with sample code and screenshots, can make tackling new features simple and fun. Likewise, giving clear error messages as soon as possible, that provide links or suggested courses of action, help relieve the frustration of debugging. Finally, I suspect graphical exhibit building tools could help make exhibit authoring discoverable and fun.

4 There is a large software market for graphical website authoring tools, with large industry players like Adobe, Apple, and Microsoft all having released applications in the last two years. While newer products in this space incorporate rich javascript widgets and AJAX effects, they lack Exhibit’s data
Despite the shortcomings and potential refinements identified, the study demonstrated that even novices to Exhibit can build rich sites to showcase mutable data. It highlights flaws in the chosen design, and identifies areas for improvement and future work.
Chapter 5

Conclusion

This thesis describes the motivation, design, and evaluation of data-management tools built on a rich, web-publishing framework. These tools were successfully used to create editable, data-driven web sites, and will become a valuable component in the Exhibit ecosystem.

5.1 Future Work

This work was only a rough start into the questions of lightweight and multiuser data ownership and publication. Opportunities for further research are plentiful. Further advances and refinements could well result in a resurgence of hobbyist driven web application development, where users are freed from the domain-specific assumptions of blogs or the complexity of content management systems. Though many suggestions for improvement were given in the thesis, I highlight these:

- Remove API inconsistencies, include higher quality editing widgets, and provide more thorough documentation and error reporting. Doing so would make using the extension easier to use and more satisfying.

- Provide default editing out of the box. I believe not offering a default edit lens posed too high a barrier to the Exhibit community, as they could not try the
extension without spending significant time understanding how edit lenses and
the submission process worked. A better default setting would be to generate
item editing widgets, and to directly submit edits to the exhibit’s database, in-
stead of queuing them for the user to manually submit. Exhibit authors could
benefit from the extension with a much lower investment of effort.

• Add authentication to the Exhibit data model, so that submitters can update
and remove their submissions. The required authentication functionality is
provided by the Google Spreadsheets API, and would integrate well with the
Exhibit code base. This would let Exhibit authors borrow Google’s authoriza-
tion model for their exhibits, and choose who can edit which items.

• Provide a graphical editing framework. Existing graphical website authoring
tools don’t support the rich, customizable interfaces of Exhibit, nor do they
exploit the rapid advances in web application development techniques over the
last few years. For example, one now common practice is to dynamically load
Javascrip
libra
mark
full website au-
output of which could be used for any purpose imaginable. Rethinking graph-
ical website design tools could make web authorship fun, pervasive, and fresh
for all web authors, experienced and new alike.
Appendix A

Study Preparation Email

The following email was sent to all participants to familiarize them with the task ahead, and to ask them to prepare some preliminary materials for the study. Though none of the participants prepared to the extent suggested in the email, their having read the email and done some work prior to the study likely resulted in more efficient sessions regardless.

Thanks for agreeing to participate in this study! We’ll be testing a new way to manage website submissions, using a new extension to the Exhibit framework[1]. Exhibit is used to create websites that present collections of data, along with ways to filter and visualize that data.

One Exhibit to look at is http://simile.mit.edu/exhibit/examples/presidents/presidents.html -- it presents the US presidents via maps, a spreadsheet, pictures, and short biographies. You can filter the presidents with the filters (called facets) on the right of the page. The input box at the top right of the screen lets you perform a full text search. [2]

Another good example is http://simile.mit.edu/exhibit/examples/CSAIL-PIs/CSAIL-PIs.html, where Exhibit is used to create a faculty directory. Play around with these two examples for a minute or two to get a sense of what Exhibit can do.

You will make an Exhibit of your own! Here’s the three things you need
to do to prepare:

1) Come up with a topic.

An Exhibit is a website that presents a collection of items. You might pick favorite tv shows, music videos, food, knitting patterns. Or maybe books or articles you want to share comments on, or maybe a summaries of blog posts you’ve made.

A good topic is one where each item has a number of attributes that you can use to filter and visualize the items with. You can filter TV series by number of episodes, leading actors, or genre. You can visualize TV shows by a timeline of when they started, or a Google Map of where they were set. You can create a picture gallery of images from the shows.

2) Make a Google Spreadsheet of items

Create a Google Spreadsheet (it’s free, sign up at docs.google.com). Create a column heading for each item attribute. Enter in five items or more, and for each item, fill in as many attributes as you can. If you want to include images, write down the image’s URL in the spreadsheet cell. Finally, share the spreadsheet with me (sbostler@gmail.com).

3) Sketch out how you want your web page to look

Draw it out on paper. Pay special of how you want the items to look, and how you want to position the facets.

A common layout is to place a header at the top of the page, then put the items of your exhibit below that. Then, put a column of item filters to the left or right of your items. That’s how the two examples I gave you did it. When I recently made an exhibit (http://valinor.mit.edu/~sostler/meta/exhibits.html), that’s what I did as well. But be as creative as you like!

Scan, take a picture, or otherwise digitize your design, and email it to me.

That’s it! If you’re ambitious, you can start writing the HTML for the page, or add a few more items or attributes to your spreadsheet. Email
me any questions you have.

1: http://simile.mit.edu/wiki/Exhibit/For_Authors is the best current source of information on how to make Exhibits.

2: http://simile.mit.edu/wiki/Exhibit/Examples contains many more examples.
Appendix B

Edit Lens Language Reference

The edit lens language extends the normal Exhibit lens language with support for modifiable item representations. This appendix contains a complete list of the language elements that were added, sample usage, and the configurable options that each language element supports. It also describes the modified item property, which is automatically applied to new items, or items that the exhibit user has changed through an edit lens.

B.1 New Language Elements

B.1.1 ex:role="edit-lens"

As the edit lens language is an extension to the lens language, every valid edit lens is also a lens. By extending the regular lens syntax with editing functionality, Exhibit authors can take existing item representations and create editable versions of them. When Exhibit first loads, all items in the exhibit are shown using the appropriate normal lens. When an item is switched into edit mode, then the normal lens is replaced with the edit lens for that item type, should one exist. Edit lenses are also used to show items in the item creation and item scraping forms. Exhibit has been extended to store edit lenses alongside normal lenses, with regular lenses retaining their role as the default representation for an Exhibit item.
Sample Usage

```xml
<div ex:role="edit-lens" ex:itemType="book">
  <div>Title</div>
  <input ex:edit="label"></input>
  <div>Author</div>
  <input ex:edit="author"></input>
  <div>Year</div>
  <input ex:edit="year"></input>
  <a ex:control="stop-editing">stop editing</a>
</div>
```

Options

- **ex:itemType**: Comma-separated list of item types that the given edit lens will be used for. Defaults to item.

**B.1.2 ex:edit="property-name"**

This denotes a form element used to edit a property of an item. When the value of the form element is changed, the Exhibit database is updated with the new value and the Exhibit widgets are redrawn to reflect the new value. The **ex:edit** attribute can be added to three HTML form elements: input, select, and textarea. For input elements, the checkbox, password, and textarea types are supported.

Sample Usage

```xml
<div ex:role="edit-lens">
  <input ex:edit="purpose"></input>
  <textarea ex:edit="desiredFeature"></textarea>
  <select ex:edit="name" ex:options=".name">
    <option>Fred</option>
    <option>Tom</option>
  </select>
</div>
```
Options

- `ex:options="expression"`: Used with `select` elements. The results of evaluating the given expression are included as options of the `select` element. In Sample Usage section, if the expression `.name` evaluated to `{Fred, Joe}`, then the `select` tag's options would be Fred, Tom, and Joe.

**B.1.3 `ex:control="start-editing"`**

Attaching the `ex:control="start-editing"` attribute to an element in a normal lens makes the element into a button, which when clicked, turns on editing mode for the item being shown in the lens. When editing mode is turned on for an item, the normal lens is replaced with an edit lens. Typically, there is a corresponding `stop-editing` control in the edit lens, so editing mode for an item can be toggled off and on.

**Sample Usage**

```html
<div ex:role="lens" ex:itemType="book">
  <div>Title</div>
  <p ex:content=".label"></p>
  <div>Author</div>
  <p ex:content=".author"></p>
  <div>Year</div>
  <p ex:content=".year"></p>
  <div><a ex:control="start-editing">Edit this book</a></div>
</div>
```

**B.1.4 `ex:control="stop-editing"`**

The `ex:control="stop-editing"` attribute is the counterpart of the `ex:control="start-editing"` attribute. When attached to an element in an edit lens, the element can be clicked to replace the shown edit lens with the item's normal lens.
Sample Usage

```html
<div ex:role="edit-lens" ex:itemType="book">
  <div>Title</div>
  <input ex:edit="label"></input>
  <div>Author</div>
  <input ex:edit="author"></input>
  <div>Year</div>
  <input ex:edit="year"></input>
  <div><a ex:control="stop-editing">stop editing</a></div>
</div>
```

B.1.5 ex:role="item-creator"

Clicking an element with ex:role="scraper" triggers a modal item creation form, which contains a create button, a cancel button, and a blank edit lens representing the contents of the new item. Clicking the create button adds the new item to the Exhibit database, whereas clicking the cancel button discards the new item. In both cases, the item creation form is removed.

Sample Usage

```html
<a ex:role="item-creator"
   ex:itemType="Item"
   ex:automaticallySubmit="true"
   ex:submissionMessage="Thanks for submitting your item!"
   ex:cancelButtonText="Cancel"
   ex:createButtonText="Submit Item">
  New Book
</a>
```

Options

- ex:automaticallySubmit: If set to true, then created items are automatically submitted to the external submission service. Defaults to false.

• **ex:createButtonText**: The text of the create button. Defaults to **Create**.

• **ex:itemType**: The given type of the items created by clicking this link. Defaults to **Item**.

• **ex:submissionMessage**: If a value is provided and **ex:automaticallySubmit** is set to **true**, then when an item is created and successfully submitted, an alert with this message is displayed.

**B.1.6  ex:role="scraper"**

Clicking the element with an **ex:role="scraper"** attribute performs a scraping operation. In a scraping operation, a new exhibit item is extracted from a chunk of plaintext, then placed into an item creation form for further editing by the exhibit user. The **scraper** must be provided with the id of a text field, the contents of which are used as the scraping input.

**Sample Usage**

```html
<a ex:role="scraper"
   ex:scraperInput="scraper-input"
   ex:itemType="book">
   Scrape Book From URL:
</a>
<input id="scraper-input"></input>
```

**Options**

• **ex:inputType**: Determines how the input text is interpreted. Can be one of **text**, **url**, and **auto**. If the value is **text**, the scraping algorithm is performed directly on the input text. If the value is **url**, the contents of the input url are downloaded, stripped of HTML tags, then scraped. If the value is **auto**, then the scraping module uses a heuristic to determine whether to treat the input as a url or not. Defaults to auto.
- **ex:itemType**: The item type that scraped items are given. Defaults to **item**.

- **ex:scraperInput**: Specifies the element id of the text field containing the scraper input. Must be provided.

### B.1.7 **ex:role="change-list"**

Displays a list of new and modified items that have been queued for submission, showing the old and new values for each modified property. When the items are submitted, or all items are reverted to their previous values, the change list is cleared.

#### Sample Usage

```
<div ex:role="change-list"
    ex:placeholderText="Edit items by clicking their 'Edit' links"
    ex:maxValueLength="40" />
```

#### Options

- **placeholderText**: If no changes have yet been made, the change list displays this text. Defaults to the empty string.

- **maxValueLength**: When set to an integer, truncates all displayed values to this number of characters.

### B.1.8 **ex:role="submission-button"**

An element marked with an **ex:role="submission-button"** triggers the submission of all queued changes to an exhibit when clicked. After a successful submission, the queued changes are marked as permanent, and the exhibit can again be modified. If there is an error during submission, an error message is shown to the user, and the user can continue editing, or resubmit.

#### Sample Usage

```
<input type="button"
```
B.2 Modified Item Property

A new automatically-set item property was introduced, modified, which is applied to newly created items and items that have been changed through an edit lens. The property can be used in facets to let visitors select either modified or unmodified items. After changes are submitted, all items lose their modified status. Similarly, if a new item is created with the automatic submission setting and is successfully submitted, it is not marked as modified.

B.2.1 Sample Usage

```html
<div ex:role="facet"
    ex:expression=".modified"
    ex:facetLabel="Modified">
</div>
```


Bibliography


symposium on Document engineering, pages 188–197, New York, NY, USA, 2006. ACM.


