User Interface Handles for Web Objects

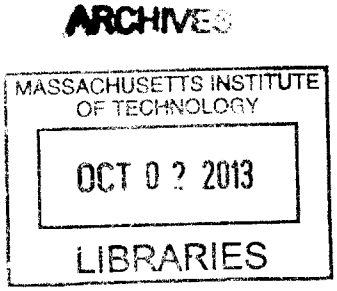
by

Hubert Pham

S.B., Physics
Massachusetts Institute of Technology (2005)

S.B., Electrical Engineering and Computer Science
Massachusetts Institute of Technology (2005)

M.Eng., Electrical Engineering and Computer Science
Massachusetts Institute of Technology (2005)



Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2013

Author .....
Department of Electrical Engineering and Computer Science
August 30, 2013

Certified by .....
Stephen A. Ward
Professor
Thesis Supervisor

Certified by .....
Robert C. Miller
Professor
Thesis Supervisor

Accepted by .....
Leslie A. Kolodziejski
Professor
Chairman, Department Committee on Graduate Students

User Interface Handles for Web Objects

by

Hubert Pham

Submitted to the Department of Electrical Engineering and Computer Science
on August 30, 2013, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

On the desktop, users are accustomed to having visible handles to objects that they can organize, share, and manipulate. Web applications today feature many loosely defined classes of such objects, like flight itineraries, products for sale, people, recipes, and restaurants, but there are no interoperable handles for these high-level semantic objects. On the web, users need visible handles that can represent an evolving set of semantically rich objects. Such handles would enable a simple, direct, and consistent interface for data representation and transfer.

This thesis proposes Clui, a platform for exploring a new data type, called a Webit, that provides uniform handles to objects. Users drag and drop Webits between sites to transfer data, auto-fill search forms, map associated locations, or share Webits with others. While Clui offers a developer API to add Webit support to web sites, Clui plugins allow users to use Webits immediately. Plugins create Webits by extracting semantic data from existing web pages, and they augment sites with drag and drop targets that accept and interpret Webits, all without requiring the cooperation of site developers.

Contributions of this thesis include design principles, derived from experimentation, that guide the functionality and behavior of handles for web objects; a system design that provides an adoption path for such handles; and a scalable approach for realizing handles that enforce access controls. To evaluate the usability of Webits, we conducted two in-laboratory studies and collected qualitative observations and feedback. The results suggest that the system is usable and effective in improving user efficiency. While using the system, participants expressed enthusiasm and delight, and believed that Webits would be useful for their daily web activities.

Thesis Supervisor: Stephen A. Ward
Title: Professor

Thesis Supervisor: Robert C. Miller
Title: Professor

Acknowledgments

I am indebted to my advisor, Steve Ward, for his encouragement, humor, and friendship. As I reflect upon the time I have spent with Steve, I now more clearly see and appreciate just how expertly he has guided me. As I once said, I could not have asked for a better teacher.

To Rob Miller, I also owe my deepest gratitude. He inspired me to reach further than what I believed to be feasible. Along the way, he taught me the fascinating art of user interface design and instilled in me the value of solid engineering. I am proud to call Rob my co-advisor.

To Larry Rudolph, who serves as a reader for this thesis, I am deeply grateful for his tutelage. It was a brief but prescient comment he made that helped ignite the work that would become this thesis. Larry's perspective never fails to stretch my thinking, fuel my curiosity, and ultimately yield insight.

I am enormously thankful to Justin Mazzola Paluska for his longtime friendship. I have so much fun working with Justin, and I learn a lot from him as well. I am fortunate for having made this journey with Justin, and I will sorely miss the special partnership we have in our work.

I resoundingly thank Max Goldman for his generous help with the user study design presented in this thesis, along with his advice on conducting the study sessions. As well, I thank Stephanie Yu, whose design and prototyping talents benefit the work described herein.

The User Interface Design group exudes MIT's motto, *mens et manus*. I cherish the opportunity to learn from and work with this team of stars. I am especially grateful to the following for both their friendship and helpful feedback on this work: Chen-Hsiang (Jones) Yu, Katrina Panovich, Michael Bernstein, Adam Marcus, Joel Brandt, Juho Kim, Elena Glassman, and Geza Kovacs. I also thank Cree Bruins and Maria Rebelo for their support.

A journey is made easier with mentors who know the road ahead. I am fortunate to have the care and guidance of Chris Terman, Daniel Jackson, Ricardo

Jenez, Mark Silis, Suzana Lisanti, Theresa Regan, Steve Deasy, Craig Newell, Ayida Mthembu, Karen Donoghue, Steve Heller, and Bill Stasior. The journey is also more fun with the company of friends on a similar quest. I have enjoyed the trek alongside Eric Jonas, Jim Psota, Philip Guo, Jason Waterman, Manas Mittal, Tilke Judd, Ramesh Chandra, Eugene Wu, and others I regrettably fail to name.

I owe much to my family. I am indebted to my parents, Loi Pham and Huan Pham, for their sacrifices and enduring support. I am also fortunate to join the family of Yelena and Zinovy Barch, who provide endless encouragement. I thank my siblings, Justin Pham, Rena Barch, and Jonathan Rourke, for expanding my sense of wonder. Finally, to Masha, thank you for your love and energy, and for seeing to it that I join you at the finish line. I am grateful to you, and I am looking forward to our first step together in the next adventure ahead.

This work was sponsored by the T-Party Project, a joint research program between MIT and Quanta Computer Inc., Taiwan. Portions of this thesis were previously published at ACM UIST [60].

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

The web has expanded the set of objects with which users interact, like representations of people, apartments, locations, flight itineraries, and products. Yet the web lacks a standardized type and handle for such semantically rich objects. That deficiency invites interface inconsistency across different sites, as each site must implement its own handles. In addition, transferring information between web applications is difficult, as handles and data are not generally interoperable.

This thesis explores the design and behavior of standardized handles that represent semantically rich web objects. The handle is visual, so users can select and manipulate it, and it bundles an interoperable, machine-readable description of the resource it represents. A user drags and drops such handles between web applications to transfer data. This thesis evaluates the hypothesis that standardized handles on the web improve interface consistency and data interoperability, thereby making users more efficient.

1.1 Motivation

In recent years, the usage and popularity of browser-based web applications have begun to rival that of traditional, offline desktop applications. While the user's computing experience once centered around the desktop and its applications, today users find web applications both more convenient, as they require neither in-

stallation nor user maintenance, and more powerful, given their inherent online nature. For example, using a browser, users can shop for and purchase goods, keep in touch with friends, get directions to places, discover and listen to new music, and collaboratively edit documents with others in real time.

While the web is undeniably successful, it still suffers from limitations in interface consistency, data transfer, and data interoperability. To accommodate such application breadth, the web offers developers only primitive interface elements, like forms, text, images, and media objects. One consequence is that sites that require richer user interface elements must construct their own custom implementations, leading to interface inconsistency across different sites. For example, Google Docs features document handles in its document list, which differ from and are not interoperable with Microsoft Office 365's Sharepoint document handles. Similarly, representations of people are unique to each social media site, e.g., Facebook and Google+, making it difficult to share, transfer, and compare people between different sites. With regards to data transfer and interoperability, user data tend to live in silos across the web, rendering them inoperable across vendors. For instance, documents stored on Dropbox are inaccessible to Google Drive and vice versa. Because the two products compete to provide storage for end-users, there is little incentive for them to interoperate. As a result, users must discover, learn, and remember the intricacies and limitations of each site they visit.

Even as its prominence diminishes, the desktop environment shines in interface consistency and interoperability. The operating system's window toolkit typically provides both a rich set of interface components and design guidelines that help achieve consistency across different applications. The file system, a central component in the desktop metaphor, enables interoperability in that different applications can operate on a given file. In fact, the desktop is so effective at interoperability that many users resort to using the desktop area to overcome limitations on the web for transferring data between sites [50]. Despite the web's popularity, users still report needing the desktop, even if only primarily as an intermediary target to save downloaded files before uploading them to a different site.

Unfortunately, the desktop and the web continue to evolve separately. While the web has been successful without much influence from the desktop, there are lessons from the desktop that can motivate solutions to some of the web's problems. At the same time, the desktop, while still mostly oblivious to the web, could regain relevance by fulfilling the need to support web-based workflows. There is a great opportunity to improve the web's usability and interoperability by reconsidering the desktop's role in a web-centric world and carefully applying some of its successful concepts to the web.

1.2 Visual Handles

A cornerstone of the desktop metaphor is its use of visual handles for objects. The desktop metaphor not only popularized the general use of icons to intuitively convey meaning but also pioneered the use of icons as handles to objects. These handles derive their power by 1) using icons to represent objects that are comprehensible and relevant to the user, like files, folders, and trash cans, and 2) acting as user-manipulable proxies for those objects. For example, a user may delete a document by dragging a handle representing that document to a handle representing a trash can.

The desktop metaphor is intuitive in part because its designers chose handles that match important objects in the user's everyday work. On the Xerox Alto and Star [48, 69], the first systems to popularize graphical user interfaces and the desktop metaphor, those handles represent digital objects that bear a strong resemblance to the physical objects present in an office of the late 1970s. Such objects include documents, folders, file system cabinets, paper trays for in- and outboxes, trash cans, printers, and desktop utilities like calculators, clocks, and dictionaries (Figure 1.1). Subsequent popular desktop systems, e.g., Microsoft Windows and the Apple Macintosh, adopted these handles, while occasionally introducing new ones, e.g., to represent local networks and remote storage shares, as new resources became commonplace in the consumer PC market.

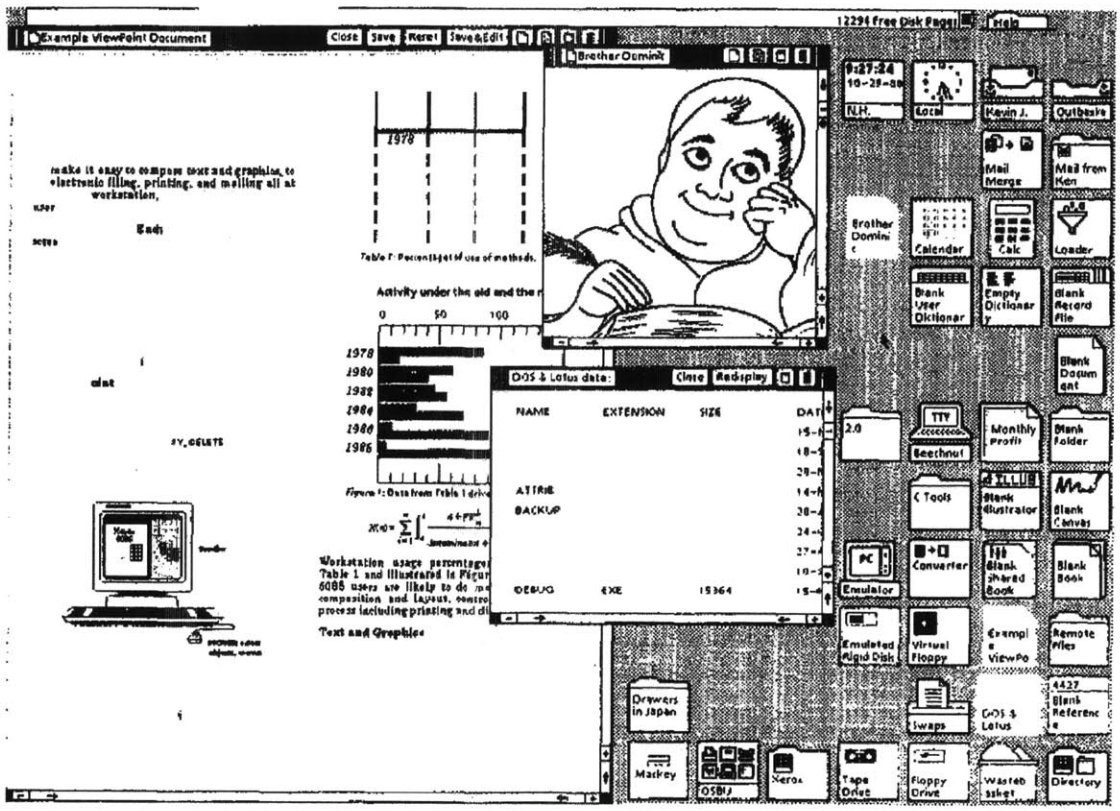


Figure 1.1: The Xerox Star desktop. Image [48] © 1989 IEEE.

As the web platform matured and web applications proliferated, the set of objects considered important to users expanded. For example, the web enables users to interact with representations of many real world objects, like products for sale, apartments for rent, friends, messages, hotels, flights, restaurants, in addition to files and documents. While handles on the desktop have enjoyed success in the desktop environment, that set of handles has not evolved to accommodate the expanding universe of objects that users encounter on the web.

Meanwhile, the web has developed its own handles, suggesting that handles are relevant beyond the desktop. URIs are a ubiquitous example, typically serving as handles for web pages. Browser bookmarks and links, in turn, are common handles for URIs. To bridge the web and the physical environment, the Cooltown project [51] explores physical handles for URIs, using mobile devices to both ingest URIs from and push URIs to physical objects equipped with computational ability. Beyond URIs, certain sites develop custom handles that are featured prominently

Figure 1.2: Google Docs features draggable handles for documents to aid with document organization.

and are directly manipulable. For instance, Google Docs (Figure 1.2) and Google+ (Figure 1.3) feature handles for documents and people, respectively, both of which can be dragged and dropped to organize those objects. Other sites develop handles that are more subtle, without support for direct manipulation. For example, YouTube.com and Vimeo.com both feature videos, playlists that reference those videos, and representations of the users that create and share those videos and playlists. However, users manipulate those objects by navigating menus, clicking links and buttons, and filling forms.

Regardless of how handles manifest themselves on the web, each site that needs them must develop its own implementation using the web's primitive types. Custom implementations at each site invite interface inconsistency across the web and thwart interoperability, an example of which is shown in Figure 1.4. Users must learn the usage and idioms of each site's handles, and the ability to use a given handle with multiple sites requires cooperation between sites that may be beyond reach. This is unfortunate because as the desktop demonstrated, handles are most

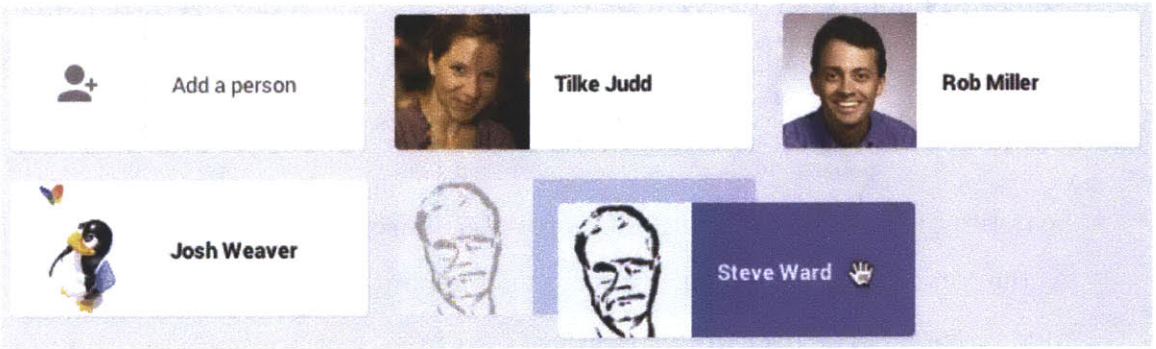


Figure 1.3: Google+ uses draggable handles to represent people.

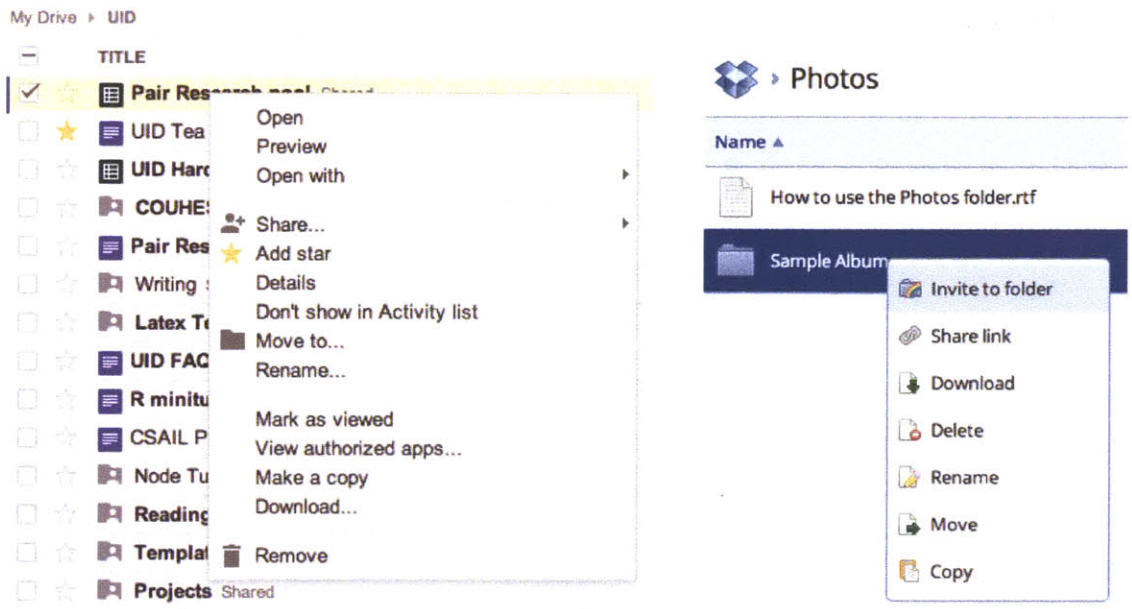


Figure 1.4: Dropbox and Google Drive both provide cloud storage, but their handles have different interfaces and are not interoperable.

powerful when they are intuitively understood by users and interoperate across applications.

1.3 Handles for Web Objects

As a step towards the goal of improving the web's usability, this dissertation investigates standardized handles to objects on the web. It presents a system that adds support for visible handles that 1) consistently represent objects that are understandable to the user, 2) capture semantics for those objects, and 3) bundle those semantics and other relevant object details into a standardized, machine-interpretable description. Users drag and drop these handles to transfer objects from site to site. A range of interactions and workflows arises:

- Searching for a place to live, a user browses apartment listings on Craigslist. At the top of each apartment listing, a visible handle representing that apartment appears. The user drags handles of promising apartments to her workspace area (Figure 1.5) to keep track of them. Or, she may drag and drop

those apartments onto Google Spreadsheets, which interprets the bundled data and generates a spreadsheet row for each apartment, along with the relevant columns (Figure 1.6). She drags handles to Google Maps, which displays the locations of the apartments by interpreting the embedded location data (Figure 1.7). She advertises for potential roommates by dragging handles of the apartments to her social network, e.g., Twitter, to broadcast the apartments under consideration (Figure 1.8); her followers see the same handles (Figure 1.9) and may interact with them in same manner. She can also contact the landlord for a given apartment by dropping the handle in Gmail, and she can also send that handle in the body of the message to provide context (Figure 1.10).

- A user looking for flights on AA.com drags and drops a prospective itinerary to other sites, e.g., kayak.com, hotels.com, and alamo.com, to search for competing flights and related services. The user then shares that handle to ensure that friends are on the same itinerary. He also drops that handle on various sites to obtain weather predictions and cultural events for the travel dates.

- A user can drag and drop recipes, which embed ingredients and their required quantities, to his online to-do list which also functions as a grocery list. Alternatively, he might instead drop a set of recipes on an online grocery site, like peapod.com, to add the ingredients to his cart for delivery.

- A handle could represent a live shopping basket for products, which contains other handles that represent products. A user could drag different products from different vendors into the basket, use Google Shopping to update the basket with cheaper alternatives, and share that basket with others to collaboratively shop. Similar scenarios are possible with using handles to directly create shareable playlists that contain videos.

- In an online conference management system, a conference program committee chair could drag paper submissions into groups of peer reviewers to as-

sign papers to those people. The chair might also drag a group containing all program committee members to a web service that generates a list of names, photos, and affiliations for inclusion in the conference web site.

One significant challenge for realizing the scenarios above is interoperability. Specifically, the workflows in those scenarios depend crucially on the ability to capture object semantics in a manner that is interpretable by a wide range of non-cooperating sites. The semantic web [15] is paving the road towards that reality.

1.4 An Interface for the Semantic Web

Semantic web technologies encourage common, standardized data formats. Using the semantic web, systems can 1) encode both the descriptions and semantics of objects, and 2) parse and interpret such descriptions. For example, the semantic web provides a way to encode a machine-interpretable description of a person, including that person's name, contact information, home and work addresses, along with a list of friends, represented as links to other descriptions. Other processes can interpret these descriptions and perform useful computations or actions. Descriptions consist of properties that are uniquely typed using URIs, allowing for an open-ended set of properties that is potentially interoperable across the web.

Several applications today employ the semantic web. For example, backend enterprise systems like Oracle's Database Semantic Technologies [13] model, store, query, and perform inference against complex relationships between objects. On the web, to improve search quality, search engines parse and index semantic web descriptions encoded in RDFa [20] or microdata [46]. Some sites with large data sets, e.g., data.gov [3] and datahub.io [4], export their data encoded using semantic web standards, allowing consumers to use generic semantic web tools like SPARQL [61] to query, analyze and process the data.

However, the semantic web faces several challenges to widespread adoption. The first is bootstrapping. For the semantic web to offer value, sites must produce data encoded in one of the prescribed standards, but sufficient user demand

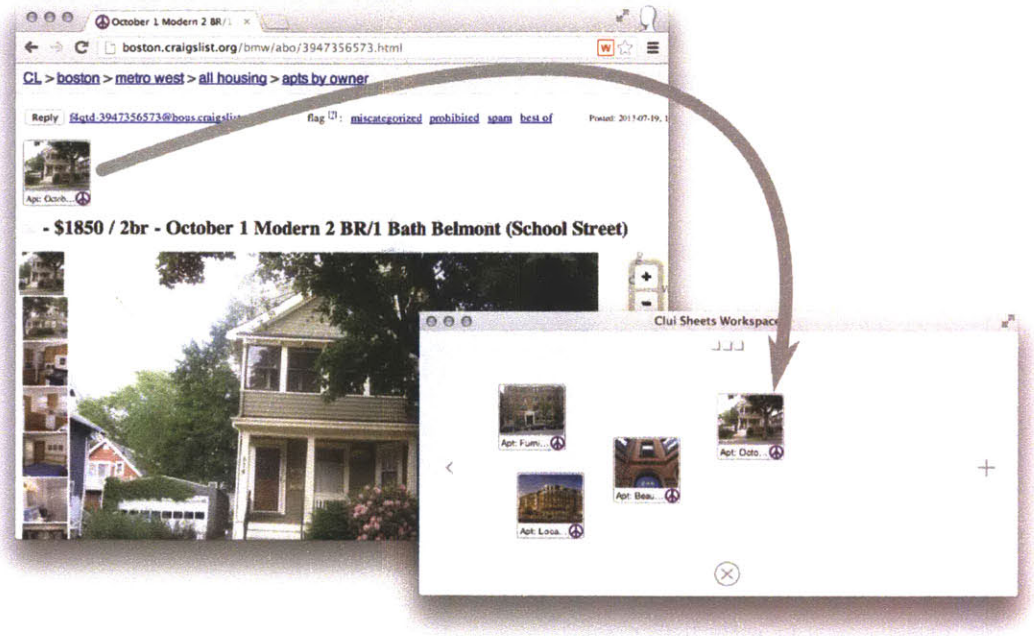


Figure 1.5: A handle appears on craigslist.org that represents the displayed apartment. It embeds a machine-readable description of that apartment and its features. The user may drag the handle to the workspace or other sites to transfer the embedded data.

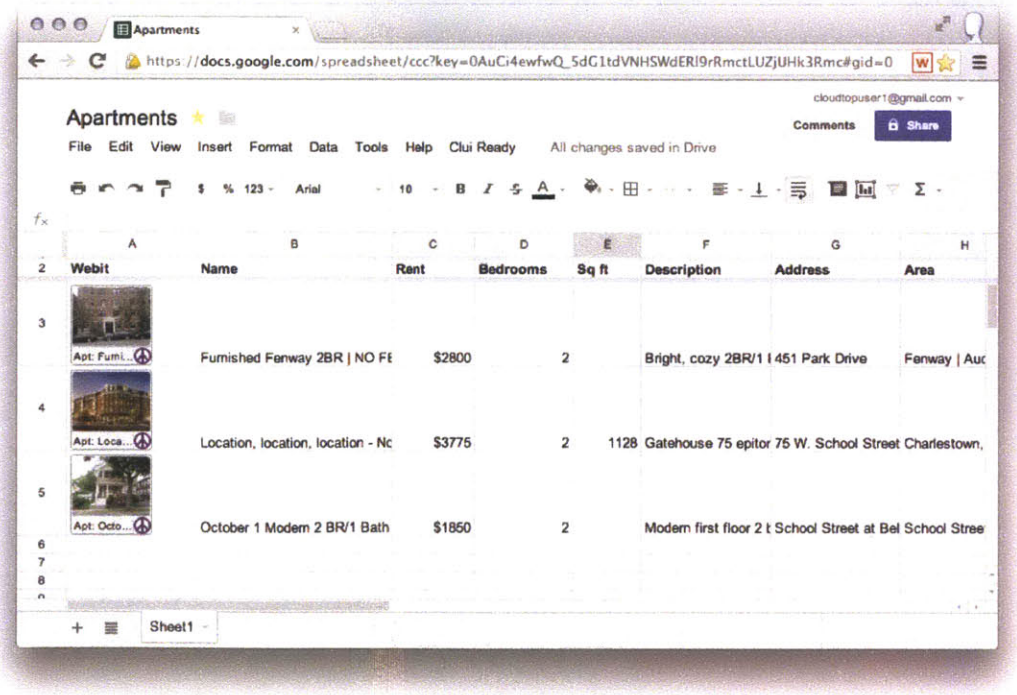


Figure 1.6: Dragging a handle, such as one for an apartment, into Google Spreadsheets automatically pastes the bundled data in a new spreadsheet row.

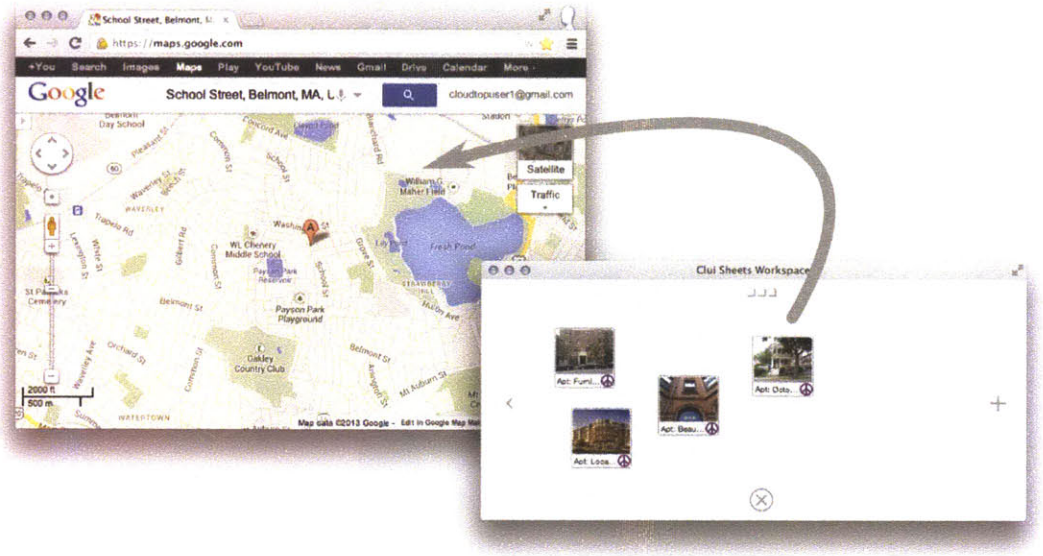


Figure 1.7: Dragging a handle into Google Maps displays a map of any location data embedded within the handle.

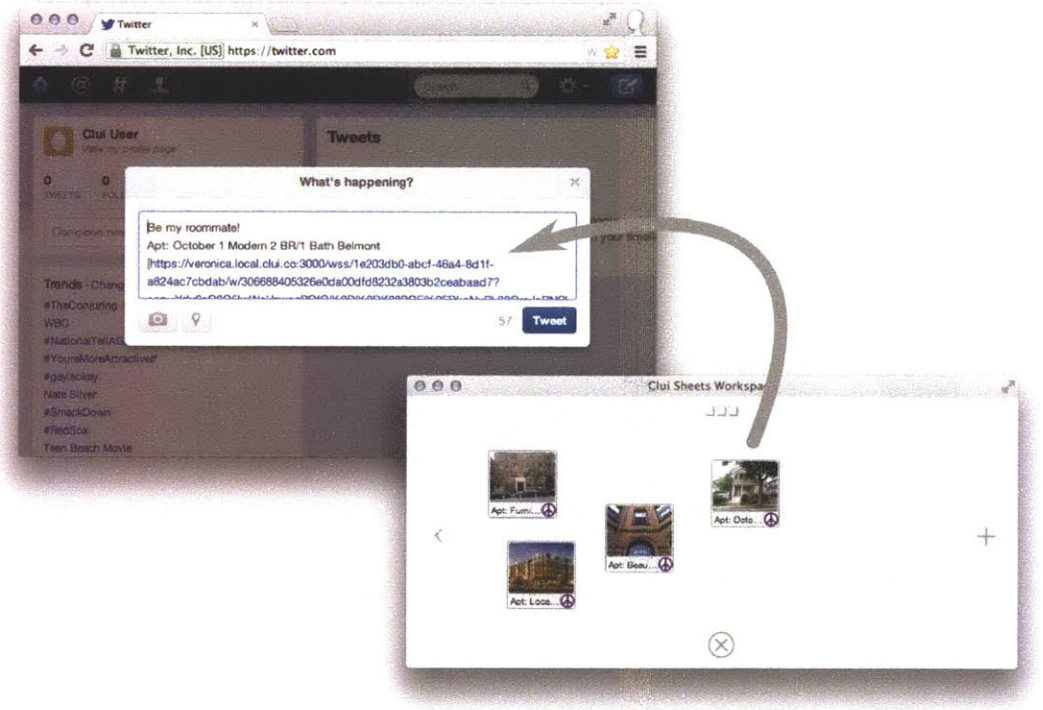


Figure 1.8: Dragging a handle into a plain-text box pastes a textual representation and link to that handle. A textual representation enables existing web applications to persist the handles in their databases.

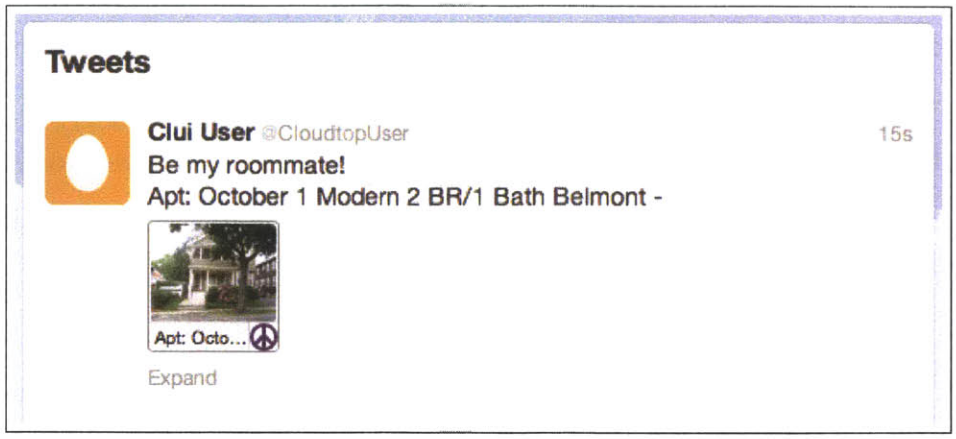


Figure 1.9: Even though services might only store a textual representation of a given handle, users still see a visual handle.

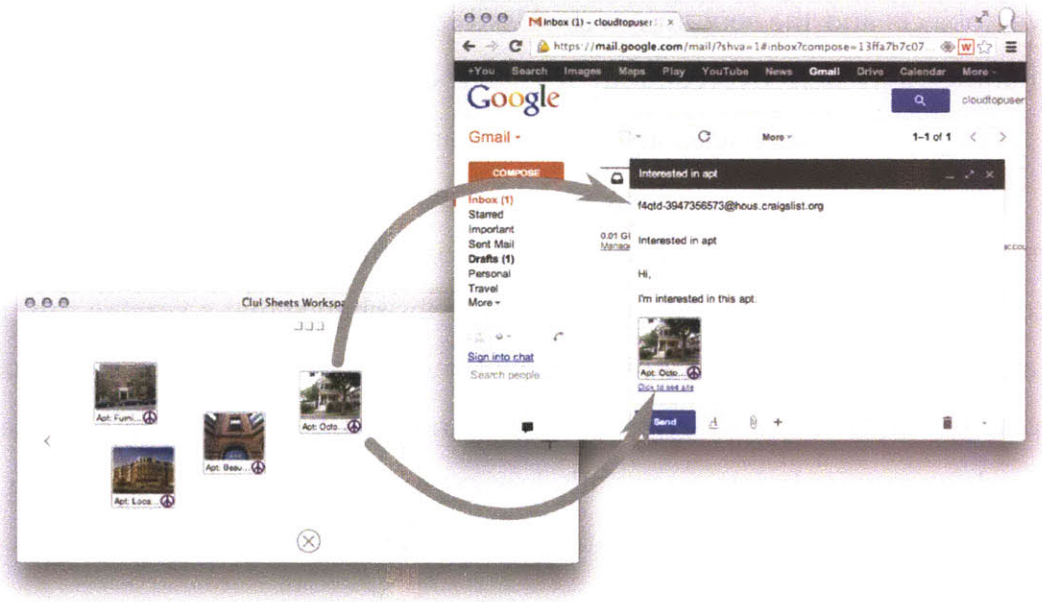


Figure 1.10: Dragging a handle to Gmail's To/Cc/Bcc fields pastes an email address embedded in the handle, if any, while dragging it to the rich-text body pastes the visual handle inline.

must also be present to justify the necessary development and support costs. User demand, however, remains stunted until there are both enough compelling use cases and participation from sites. Another challenge lies in the lack of simple user interfaces for the semantic web. While prototype tools, like PiggyBank [47], exist to help users organize semantic web descriptions, semantic web tools largely require expert users familiar with sophisticated concepts like XML, Resource Description Framework (RDF) [52], graphical models, ontologies, and so on. Without a simple, intuitive user interface, the semantic web's audience is limited. Finally, a significant challenge lies in the standardization of vocabularies and ontologies. The semantic web prescribes a standardized framework for describing objects and their semantic constraints, but it does not standardize the description standards, e.g., the specific names and types of important properties for a product. As a result, competing standards may arise to describe the same concept. This is desirable for encouraging experimentation and openness, but a challenge for interoperability.

Fortunately, there is a symbiotic relationship between user interface handles for web objects and the semantic web. Used together, semantic web technologies and handles may help overcome their shared challenge of interoperability. Handles and their use cases may also drive demand for greater semantic web adoption. Specifically, handles can achieve interoperability by bundling semantic web descriptions, while providing a user-friendly interface that shields the user from the complexities of semantic web. To increase the chance that a service can interpret a handle's bundled data, a handle may embed many descriptions, each using competing standards, to describe the same underlying object. Doing so enables standards experimentation, eventually leading to de facto standards, without sacrificing interoperability in the meantime.

1.5 Adoption Strategy for Web Handles

This thesis proposes the *Webit*, a visual handle to an object on the web. A Webit attaches to a user-visible interface element, typically an icon, and bundles machine-

interpretable, semantic web descriptions of the resource it represents. Unlike conventional handles on the desktop and the web, Webits are general purpose and may represent resources from an open-ended universe. Users interact directly with Webits by dragging and dropping them across sites to transfer information.

In addition to improving usability and user efficiency, the thesis also advances the view that visual handles like Webits serve as a natural user interface for semantic web descriptions. In other words, handles serve as one vehicle to promote end-user adoption of the semantic web. To be effective, Webits must first enable and demonstrate compelling use cases to end-users. Achieving that relies on widespread Webit support on existing sites.

This thesis presents a system, called *Clui*, that brings Webit support to web pages. To encourage rapid adoption and minimize development costs, Clui provides an open-source library and API that web developers may use to add direct support for Webits in their web application. Even so, site operators are unlikely to add Webit support until incentivized by competitive pressure and user demand.

To bootstrap and generate user demand today, Clui also allows any developer to add Webits to any web site. Developers contribute Clui *plugins* that execute in the browser and scrape content from known web pages, generate the appropriate semantic web descriptions, attach those descriptions to new Webits, and using Clui's API, insert those Webits directly onto the page. Plugins may also augment existing pages with computation that interprets dropped Webits and implements useful actions. Developing plugins for a given site requires no cooperation or support from that site, enabling handles to quickly evolve independently of site operators. In addition, Clui automatically generates Webits for primitive resources like HTML snippets, links, and images.

In addition to bootstrapping Webits, Clui provides system support for Webits that represent changing data, e.g., a shopping cart, the current weather, or the current price of a stock, and for sharing Webits between websites and users. Clui supports an access control system to enforce permissions that specify who may see or change certain parts of Webits. Clui uses a capability-based scheme for scalabil-

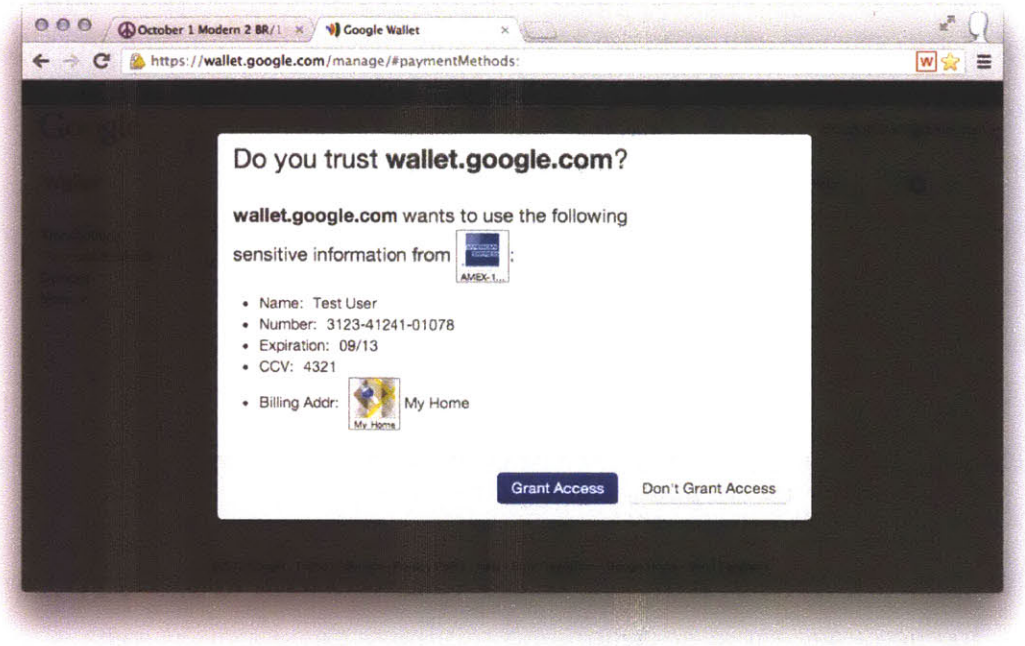


Figure 1.11: Clui must sometimes warn the user when websites request access to sensitive data bundled in Webits. Clui requires user confirmation before delivering such data.

ity and usability. A capability confers privileges to the holder of that capability.

Clui strives to protect the user from unknowingly sharing Webits with bundled sensitive information. Developers should generally avoid bundling in Webits sensitive information, e.g., a user's social security number or passwords. However, when such data must be bundled, Clui automatically filters that data before the user shares the Webit, without user interruption. Only when a site requires access to the stripped data in order to function will Clui prompt for confirmation (Figure 1.11).

As handles become popular on the web, the desktop environment might evolve to support and manage those handles, bridging the gap between the web and the desktop. Clui takes a step towards unifying the desktop and web experience by enabling experimentation with workspaces that operate natively on Webits. This thesis presents one such workspace, called *Sheets*, that explores a notebook metaphor for supporting web-based workflows. Sheets provides a lightweight, spatial area to hold Webits that are important in the user's current task.

1.6 Contributions

This dissertation explores the hypothesis that standardized user interface handles to web objects:

- promote data interoperability and a consistent interface for transferring information between websites;
- foster new interactions not yet possible with the current web or desktop; and,
- and thus enhance user efficiency and delight in web-based workflows.

This thesis verifies the first two claims by illustration. It presents evaluation on the user efficiency and delight claim by reporting on observations and user feedback collected from two qualitative, in-laboratory studies. Study participants expressed enthusiasm towards Clui and believed that Webits would be useful to them in their daily work.

Contributions of this thesis include:

- the Webit, a new data type that 1) provides a visual, user-friendly handle to web objects, and 2) bundles machine-interpretable descriptions of those objects;
- a set of principles that guide the design of handles;
- demonstrations of the practicality of handles on the web;
- a system design, Clui, that provides an implementation and adoption path for handles;
- a scalable approach for sharing handles that enforce access controls on their bundled data; and,
- an initial prototype of a Webit-aware workspace that explores a new metaphor for supporting web-based tasks.

1.7 Thesis Outline

This thesis draws upon many existing ideas, such as structured data extraction approaches, the desktop interface, and usable security principles. The next chapter summarizes the related work. The core of the thesis is Chapters 3 and 4, which discuss and motivate the design of Webits and Clui. For Webits to have impact, it is important to demonstrate an implementation approach and discuss related challenges. Chapter 5 describes the browser support in Clui that enables bootstrapping Webits onto existing web pages, and Chapter 6 discusses the server-side components to support Webits with mutable data. Chapter 7 details the results of several evaluation approaches, which include user studies we conducted to evaluate the usability of Webits. Finally, Chapter 8 concludes.

Chapter 2

Related Work

Several areas of previous work inspire Clui. One such area is the use of information scraps, or loose pieces of information relevant to the user. As users spend more time on the web seeking and posting information, the need for tools that manage information scraps increases. This chapter begins with an overview of such tools and related studies. To bootstrap the use of Webits, Clui must identify and extract structured data from existing web pages. That structured data is then encoded using semantic web standards to achieve interoperability across sites. This chapter discusses prior work that pioneer techniques to capture structured data, along with relevant semantic web projects. As the desktop environment also inspires Webits and the Clui workspace, the chapter continues with an overview of studies conducted on desktop usage, along with projects that expand and depart from the desktop metaphor. Finally, a user may specify access permissions when sharing Webits, which may contain sensitive information. Clui borrows and adapts concepts from capability operating systems and usable security approaches. Related work concerning those topics are discussed last.

2.1 Information Scraps

Users clip and store loose information scraps, both in the physical world and digitally. Bernstein et al. studied the use and life cycle of such scraps [26], like notes

saved in a text file, a to-do item on a Post-it note, a phone number written on scratch paper, or a confirmation number saved in an email message sent to oneself. Information scraps play several roles, such as providing temporary storage, a reminding mechanism, an archive, or a catch-all for information that does not fit in existing tools. Bernstein et al. suggest design principles for information management tools, such as lightweight capture, flexibility in content and representation, flexibility in information use and organization, and visibility. As Webits can be viewed as encapsulated information scraps, those principles guide the design of Webits and the associated workspaces for storing and organizing Webits.

Many existing web clipping and note taking packages aim to help users collect and manage heterogeneous information snippets, like text, images, and web links. Some are general purpose, such as Evernote [5] and Microsoft OneNote [10], while others are optimized for specific kinds of data, like Zotero [18] for bibliographic references. Van Kleek et al. [73] reported that users typically need to record quick and terse notes. They developed list.it, a tool that specializes in rapid capture and retrieval of short, textual notes.

An important goal for the projects above is to provide a digital home for information scraps, with emphasis on long-term archival and retrieval. Clui differs by focusing on the use of handles to rich objects, rather than primitive data types. It also differs by enabling information transfer between sites rather than data organization and archival.

2.2 Structured Data

Detecting, extracting, and parsing structured data from existing content is a key challenge for many domains, include Webits. Below are systems that address that challenge in a variety of contexts.

2.2.1 Programmatic Extraction

Clui's plugin system is similar to projects that detect structured data from the clipboard and documents, like Citrine [70], Microsoft's Live Clipboard [9], and Apple Data Detectors [59].

Citrine detects structured data copied to the clipboard, such as contact information, calendar appointments, and bibliographic citations. Once copied, users may paste such structured data into forms. In addition, users may also train the system to map certain fields to columns in Microsoft Excel, which inspires some of Clui's workflows.

Live Clipboard is similar to Citrine in that it operates on structured clipboard data. Live Clipboard exports the general idea to the web, where users may copy and paste semantic objects, like calendar entries or addresses, across different sites. Sites, however, must explicitly implement support for Live Clipboard, and thus the system faces challenges in bootstrapping.

Apple Data Detectors is a system that detects structured or semantic information, like email addresses and dates, in documents and displays a popup menu with actions to operate on that data. Clui's plugin system is inspired by Apple's use of detectors, or grammars that detect structured data in a document, and executable action recipes, which operate on the detected data.

2.2.2 Structure From User Input

Jourknow [72] targets content that the end user creates. It combines the efficiency of lightweight, free-form text entry with the benefits of structured data for information retrieval. Users enter text snippets using tags, a pidgin grammar, or Notation3 [25], which Jourknow parses to derive actors, locations, and structure. In addition, to support re-finding, Jourknow captures the user's environmental context, e.g., the programs running, the user's location, and the presence of nearby people, as the user creates snippets. Jourknow's techniques for generating structured, semantic data from free-form text may aid Webit creation by end users.

2.2.3 Visual Extraction Techniques and Automation

One common technique for generating structured content from the web is scraping a page's Document Object Model (DOM) structure. For instance, Greasemonkey [7] is a general purpose tool that enables users to install JavaScript code to scrape and alter specific web pages. Clui's plugin system take a similar approach, and like Greasemonkey, it assumes plugin developers to be expert programmers. However, projects that empower end users to capture structured content from the web typically need to provide higher-level approaches, such as the use of simple languages or direct manipulation, to semi-automate the process of data extraction.

Chickenfoot [27] enables end users to customize and automate web pages without needing to examine DOM structures or source code. Instead, users identify and manipulate page components using keyword pattern matching against the rendered user interface rather than against identifiers embedded in the source code of the page. Chickenfoot's technique for selecting and operating on page components may be an appropriate approach for users who wish to create Clui plugins.

Dontcheva et al.'s work [33, 34] explores visual techniques that enable users to extract and relate structured content across different sites to summarize and collect information. Users select web page elements to create extraction patterns, e.g., on yelp.com for restaurant data, and associate those patterns with similar content on other sites, e.g., restaurants.com, to automatically apply extraction to different sites. Users also create visual card templates that summarize the extracted data. When users search for content, the system finds matching results on all sites for which it has extraction rules and automatically generates cards for each resulting entity. Dontcheva's techniques for relating structured content across sites would enable users to merge related Webits found across the web. While cards are similar to Webits, Webits also act as handles that may be dragged to the web.

Fujima et al. [41] explore tools that enable users to construct new interfaces that bridge data between page elements clipped from different sites. Users create custom interfaces by selecting and importing form elements from existing web

pages, along with the corresponding page elements holding the form results. Users may connect the results of one form to the inputs of another using spreadsheet-like formulas to create a custom, reusable data pipeline.

Vegemite [55] explores techniques to import web data into spreadsheets via direct manipulation, and by observing user action, to automatically generate reusable scripts that operate on spreadsheet rows. Like Clui plugins that parse dropped Webits, Vegemite scripts may also perform actions, e.g., clicking on links or entering values into forms, based on values in table rows. Techniques reported in Vegemite are useful for automatically mapping a Webit's bundled data fields to spreadsheet columns.

Yahoo! Pipes [17] is a web application that allows users to construct data pipelines. The application operates on existing, machine-readable feed content found across the web. Users build pipelines that combine, sort, filter, and translate feed data, using a direct-manipulation interface to instantiate and connect operators rather than using formulas.

d.mix [43] enables developers to obtain code snippets for some functionality by directly selecting live, representative elements on an existing web page, rather than by browsing and massaging source-code examples. Users may compose, configure, and edit clipped elements from an existing web application to create a new application hosted on a wiki.

The direct-manipulation approach of scraping-by-clipping, as proposed in the projects above, would improve the development process for Clui plugin authors. Clui differs in focus from some of those projects by supporting an on-demand dataflow via drag and drop, rather than requiring the user to invest time in building data pipelines that automate tasks.

2.2.4 Semantic Web

The Haystack project [19, 49] stores, retrieves, and displays semi-structured data, towards the goal of personal information and knowledge management. It ingests

and aggregates data from multiple information sources, and generates RDF to represent information objects, along with their attributes and relationships. Haystack strives for flexibility, both in its data modeling and user interface. It avoids imposing pre-defined data schemata and presents a versatile user interface that allows users to browse and navigate relationships between their personal data.

Piggy Bank [47] scrapes pages open in the user's browser and generates structured data. Piggy Bank's aim is to enable users to subsequently browse and query that data within the browser via a Piggy Bank-generated web page. Clui focuses less on allowing users to inspect structured data and more on the interactions for transferring structured data from site to site. Since Piggy Bank is also implemented as a browser extension, it would be fruitful to leverage Piggy Bank as a library within Clui.

Fundamentally, scraping DOM nodes intended to hold values for human consumption is brittle, as sites change and evolve. As such, microformat standards [46] encourage site authors to embed invisible semantic data on their pages. Such data is crawled, indexed, and processed by some search engines and thus may improve relevance in search results. Clui may automatically generate Webits on sites with microdata without requiring plugins for those pages.

2.3 The Desktop and Beyond

A few common themes emerge from many studies on the traditional desktop, e.g., by Malone in 1983 [56], Barreau and Nardi in 1995 [23], Ravasio et al. in 2004 [62], and Katifori et al. in 2008 [50]. Barreau and Nardi first observed that users use the desktop to store information that is either ephemeral, working, or archival. The studies above agree that the desktop is commonly used for temporary storage, such as a staging area for downloads or uploads, and as a device for reminding users of impending tasks. They also agree that archiving resources to tidy the desktop is arduous because doing so involves classifying each resource, a cognitively difficult task. As such, users tend to put off archiving, leading to desktop

clutter. All studies agree that users naturally arrange desktop icons in meaningful, spatial arrangements, such as clustering similarly themed resources together, to aid efficient retrieval. These studies suggest that Clui's initial workspace prototype should borrow from the spatial elements of the desktop.

Many projects expand, alter, or depart from the traditional desktop. For example, Rooms [44] explores the use of virtual desktops to organize user tasks, leveraging a metaphor of rooms connected by doors as a navigation aid. MessyDesk [37] enlarges the desktop into a projected, shareable bulletin-board, with the goal of aiding user-recall by encouraging users to build context for their data on the desktop. The Sharing Palette [74] and Contact Map [76] build on the notion of desktop-supported data sharing by representing people as first-class objects. Presto [35] replaces the location-centric, hierarchical file-system model with meaningful, user-set attributes for document management. Lifestreams [40] and Timescape [64] explore the use of time as a metaphor for retrieving context. Browser-only operating systems like Chromium OS [2] eschew the desktop in favor of just web browser windows.

While the studies and projects above should certainly inform the design of Webit-aware workspaces, we believe that it is important to consider the priorities relevant to web-centric workflows. For example, in such workflows, data not only come from the web but also ultimately return there, e.g., by posting that data to blogs, tweets, or services that curate and organize content. This suggests that local workspaces for Webits might need to prioritize management of temporary data rather than long-term retrieval.

2.4 Web Authorization Protocols

Cross-site authorization protocols, such as OAuth 2.0 [12], provide mechanisms to enable users to share data between sites. Such protocols rely on site operators to set up pathways between each other before users can transfer data. As such, inter-vendor pathways are likely to be limited in number and driven by business

incentives. From the user's perspective, the use of these protocols suffers drawbacks in visibility. For example, when users are prompted to share their data with some external service, they must typically agree to share whole classes of the data, such as all their contacts, photos, or emails, rather than specific items. Once they agree, there is usually little visibility to alert the user when data is transferred, as it occurs out-of-band, directly between the vendors' systems. In contrast, Clui enables a vendor-neutral approach in which users have fine-grained control of the data they wish to share by direct manipulation.

2.5 Capabilities

A capability is an unforgeable token that confers access privileges to the holder of that capability. Capability-based systems have long been explored, from the early days of timesharing systems [32], to operating systems in the past few decades, e.g., KeyKOS [8] and its successor EROS [67], to contemporary designs involving access control on information flows across distributed components, like Asbestos [36]. Capabilities continue to manifest today in more conventional systems, e.g., as file descriptors on POSIX-like operating systems.

For efficiency, capabilities in typical systems are identifiers that index into a table of data structures that hold pointers to protected resources and the associated privileges. The table is typically stored in privileged memory, carefully controlled by the kernel or a trusted component.

While Clui can accommodate any reasonable capability scheme, the reference implementation described in Section 6.2 embeds the description of access privileges in the capability itself, using encryption to achieve tamper-resistance. The approach is inspired by CapaFS [63], a distributed file system, which includes the access control policy in the capabilities that it creates. Embedding the access policy in the capability eliminates the need to maintain a web-scale table of data structures and principals on the server.

2.6 Usable Security

As Webits are easily shareable and may contain sensitive information, usable security is an important but challenging goal to address. In her thesis, Whitten [77] summarizes a few of the challenges, such as *the secondary goal property, the hidden failure property,* and *the barn door property.* The secondary goal property states that security is often a secondary goal to the user's primary task, so any expectation that the user will be motivated to learn and manage security settings is unrealistic. The hidden failure property describes the challenges of accurately conveying to the user the complex state of security configuration, and that furthermore, the configuration is unlikely to match the user's preferences anyway. The barn door property stresses the need to avoid high-cost, irreversible mistakes, such as accidental information leakage.

Yee [79, 80], along with Smetters and Grinter [68], propose design principles and strategies for addressing usable security challenges. For example, they argue for the principle of implicit security, or tightly integrating the security goals with the workflow of the user's main task when possible. Implicit security improves the system's ability to deduce the user's security-related intent from her actions.

Yee further stresses the advantages of designing interfaces to support security by designation, in which the user simultaneously designates the desired action and conveys the authority to perform that action. The alternative, security by admonition, describes systems that interrupt the user and demand attention to potential security problems. Security by admonition typically applies to systems that must statically anticipate all allowable actions the user may need, across a wide set of situations, and so such systems must guess the appropriate times to warn the user. Even though security by admonition is an inferior approach, Yee suggests that it can be an appropriate recourse when security by designation is infeasible.

When warning the user is necessary, Bravo-Lillo et al. [28] offer guidelines toward the design of warning dialog boxes, and experimental results that discourage the use of dialog boxes that offer detailed explanations and options, even for

sophisticated users. Their guidelines encourage visual consistency in layout; comprehensiveness, conciseness, and accuracy in describing risks; and the availability of relevant contextual information.

While implicit security is ideal when possible, a complementary approach is to reduce or eliminate the need for users to comprehend security issues. Systems that take this approach automate or simplify security decisions into coarse-grain options. For example, some approaches involving key management in the context of secure email, like Enigma [30], minimize user interaction as much as possible. Garfinkel [42] argues that email-based identification and authentication is more usable, as it avoids the complexities associated with approaches based on public key infrastructure. WindowBox [21] takes the coarse-grain security approach, presenting virtual desktops with different predefined security policies.

The design of Webits and its associated security mechanism borrow from the ideas summarized above. Clui's primary approach is to avoid situations in which security issues might arise, e.g., by generally discouraging the inclusion of sensitive information in Webits. When a Webit bundles sensitive information, Clui strips that content before users can share it, taking the approach of reducing the need for users to comprehend security issues and automatically applying defaults that respect the barn door property. While such an approach might sometimes violate a user's intention to share sensitive content, detection and recovery is at least possible and even potentially simple, e.g., using reactive security techniques [58].

As a last resort, Clui shows a warning dialog box when a Webit-aware site requests sensitive content from a dropped Webit. The design of the dialog box strives to meet some of Bravo-Lillo's guidelines. Even so, researchers generally agree that warning dialog boxes ultimately train the user to always take the affirmative action to proceed, without investing time to fully understanding the associated risks. While Clui's warning dialog box may be no exception, its design differs from conventional dialogs in ways that may slow the user's insensitivity to warnings. Webits offer general utility even without sensitive information, so the display of warning dialogs should be rare. However, when the system must dis-

play a warning, Clui strives to maximize communication by displaying the actual sensitive values, which users may recognize as private and thus heed the warning.

Chapter 3

The Design of Everyday Webits

The key idea behind Webits is that they 1) represent resources relevant to the user as visual, user-manipulable handles, and 2) bundle machine-interpretable descriptions of the resources they represent. For example, Webits can provide handles to people and bundle the names, contact information, and addresses of those people. A Webit may reference other Webits. For instance, a flight itinerary Webit might reference Webits representing the set of passengers, flight segments, and the origin and destination cities. Similarly, a Webit that represents a trip might reference flight itinerary Webits as well as Webits for hotel and car reservations. By leveraging existing semantic web technologies to express and interpret object descriptions, the set of objects that Webits may represent is unbounded.

This chapter first elaborates on the user interface of Webits and then discusses the general types of applications that Webits improve. Following that are proposed design principles that govern the functionality of Webits. The chapter concludes with a discussion on additional design considerations.

3.1 User Interface

Users interact with Webits, which appear as icons with textual labels, in a number of ways. First, a Webit is associated with some URI, typically a web page. For example, a Webit that represents a restaurant might associate with the restaurant's

homepage. Users open a Webit's associated web page in a browser tab by clicking on that Webit, much like a traditional hyperlink. Users drag and drop a Webit to transfer its bundled information. The resulting drop behavior depends on the drop target type and may be customized, as discussed below. One special drop target is the Clui workspace, which appears in a separate window and provides a surface for holding Webits, such as ones relevant to the user's current task. Workspaces are pluggable; this thesis describes a simple one, called Sheets. Sheets features a chronological notebook metaphor, without collections or hierarchy. Finally, users may inspect a Webit's bundled content through a pop-up context menu. Below are illustrations of Clui's interface features:

Discovery While Webits may appear as distinctive icons on existing web pages, they may also be embedded in existing elements like images. An indicator in the browser's address bar helps the user discover Webits. When Webits are present on the current page, the indicator appears lit. Clicking on the indicator visually highlights the elements on the web page that contain Webits.

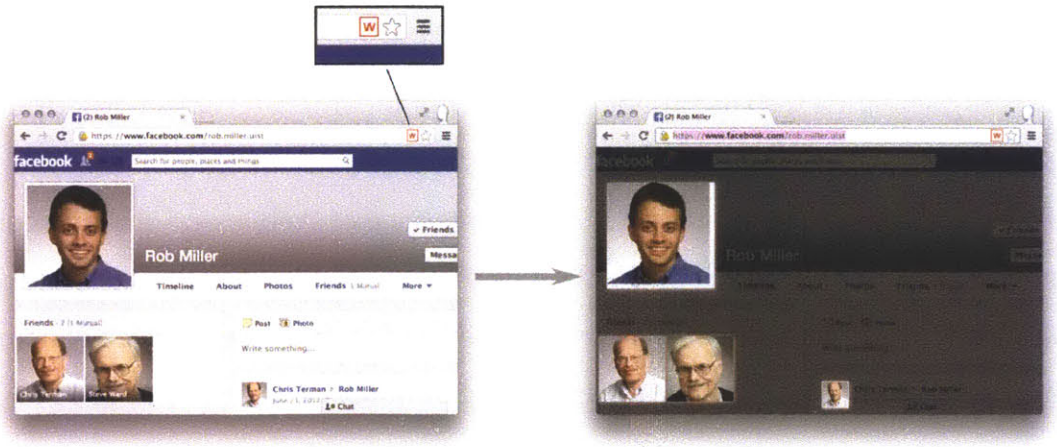


Figure 3.1: Discovering Webits.

Traditionally, discovering drop targets and the associated behavior of dropping an object on those targets is difficult. Clui improves discoverability by displaying tooltips that preview the drop behavior as the user drags a Webit over targets.

Default Drop Behaviors To aid predictability, Webits have a consistent, default behavior for every type of drop target on which they may be dropped. In general, dragging a Webit to a web page transfers that Webit's bundled information in its entirety, without loss of information. For example, when dragging a Webit to a rich-text input box, e.g., Gmail's message composition window, Clui pastes the handle with the bundled information embedded, so that it may be dragged by other users, e.g., the email recipients.

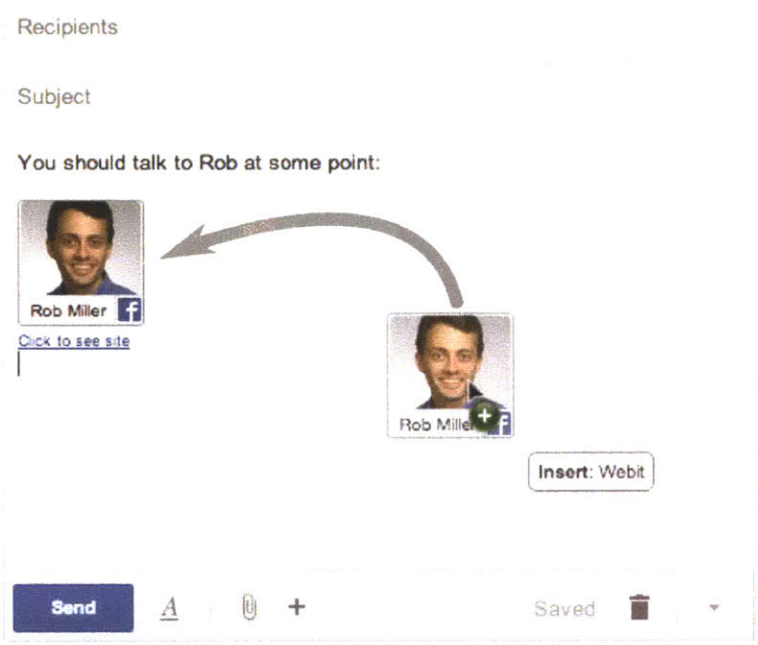


Figure 3.2: A Webit in Gmail.

When it is not possible to represent Webits as icons, e.g., when pasting a Webit into a plain-text input box, Clui pastes a short description of the Webit and a globally dereferenceable link to that Webit.

Figure 3.3: A Webit pasted in a plain-text input box.

When the data is rendered later, e.g., as a tweet, Clui restores the icon version of the Webit (Figure 3.4) by dereferencing the Webit's link. Those without Clui installed see the link, which when clicked, navigates the browser to a page that describes the Webit.

Figure 3.4: A Webit rendered in Twitter.

Customizable Drop Behavior Websites or Clui plugins may override or customize the default drop behavior. For example, when dragging a Webit representing a person onto the "To" field of an email composition form, Clui pastes the person's email address rather than the Webit's link. As mentioned earlier, tooltips preview the resulting drop behavior to inform the user of non-standard behavior.

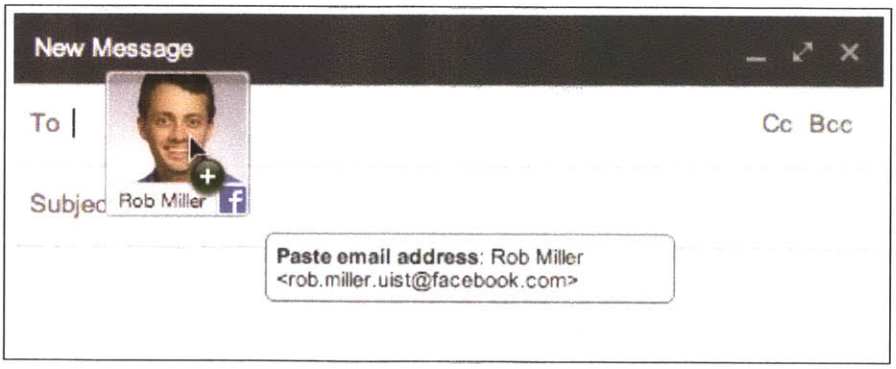


Figure 3.5: A Webit's drop behavior may be customized.

Bundled Data Inspection A Webit combines many properties that describe the represented object. Users may inspect those properties using an inspector, available from a context menu item. Users may also drag a specific property value to

paste it in a form field. For example, the user may paste a friend's homepage URI by dragging the "Homepage" field in the inspector.

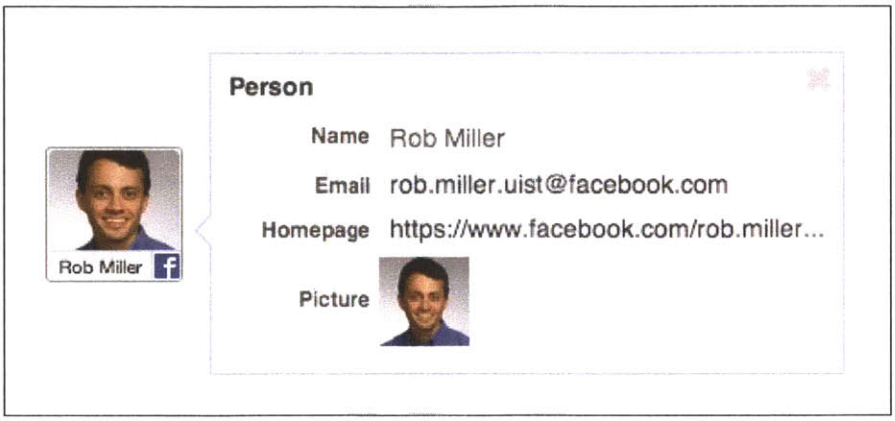


Figure 3.6: A Webit's bundled metadata

Sensitive Information Redaction While discouraged, Webits may contain sensitive information. As described later in this chapter, bundled properties may be marked as sensitive. Those properties are automatically stripped from a Webit before the user can transfer that Webit.

When the user drops a Webit with redacted properties on a site that 1) is specifically designed or augmented to interpret Webits, and 2) requires access to sensitive properties, the site must inform the user and provide an affordance, such as a button, that allows the user to review and grant access (Figure 3.7). When the user invokes the affordance, Clui displays a modal confirmation dialog on the page (Figure 3.8). The dialog displays all the properties the site requests, along with the current values that would be shared.

Dragging a Webit with redacted properties to a generic input form element causes Clui to display a non-modal information bar (Figure 3.9) to alert the user that some of the Webit's bundled information is stripped. The user may customize what data is shared by clicking the button in the information bar, which reveals a modal confirmation box. Because the site does not request specific access to properties, the dialog box shown differs from the one mentioned above by providing controls to specify precisely which properties to share (Figure 3.10). A user may

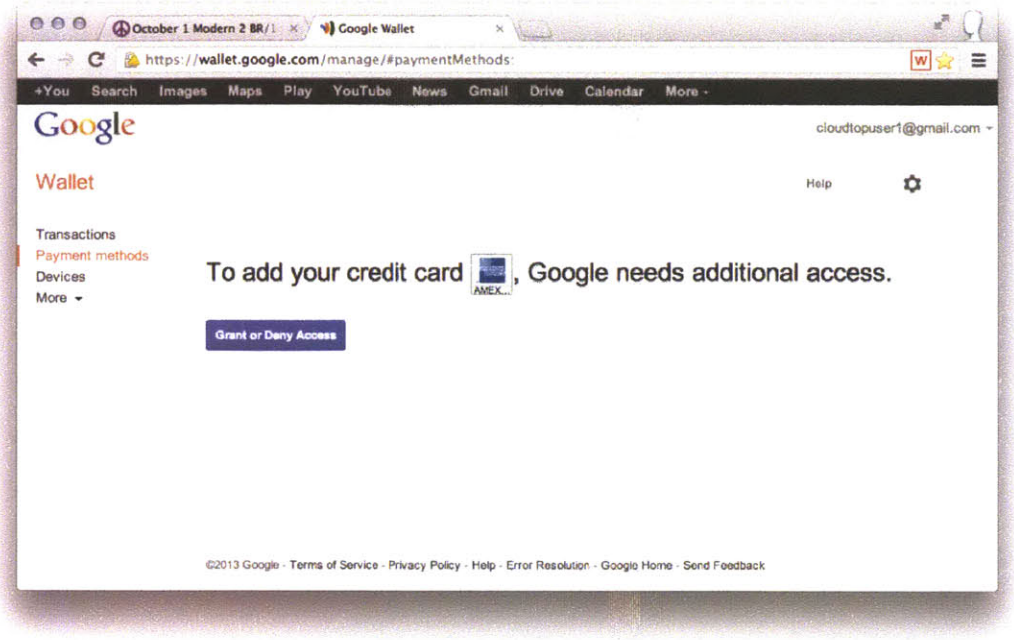


Figure 3.7: A site informs the user that it needs access to sensitive data.

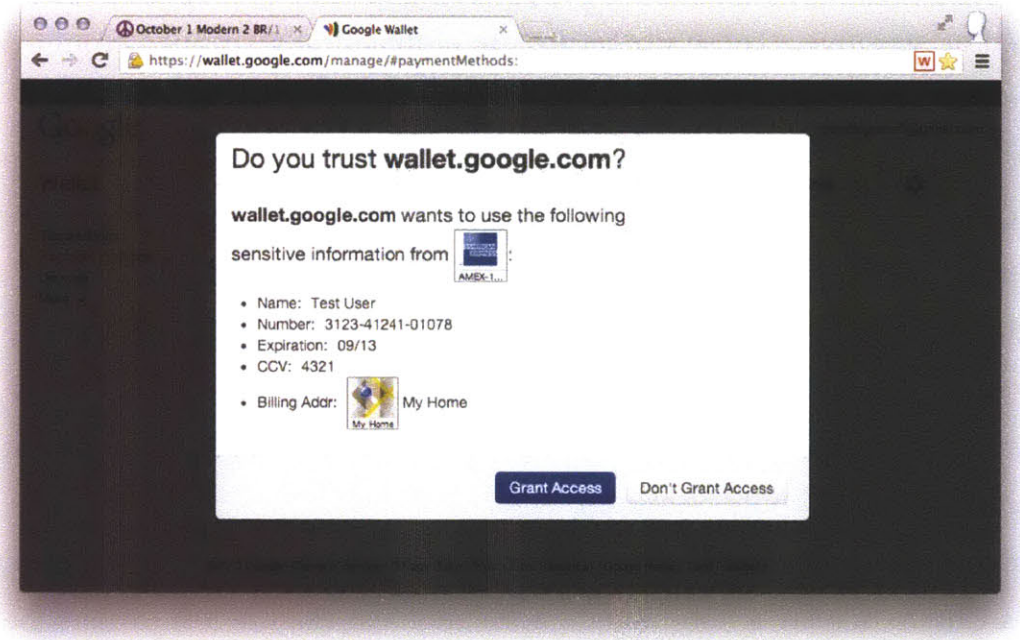


Figure 3.8: A confirmation dialog for sensitive data access.

directly invoke the dialog box in Figure 3.10 by holding a modifier key while dragging the Webit to customize its permissions, even when that Webit contains no sensitive content.

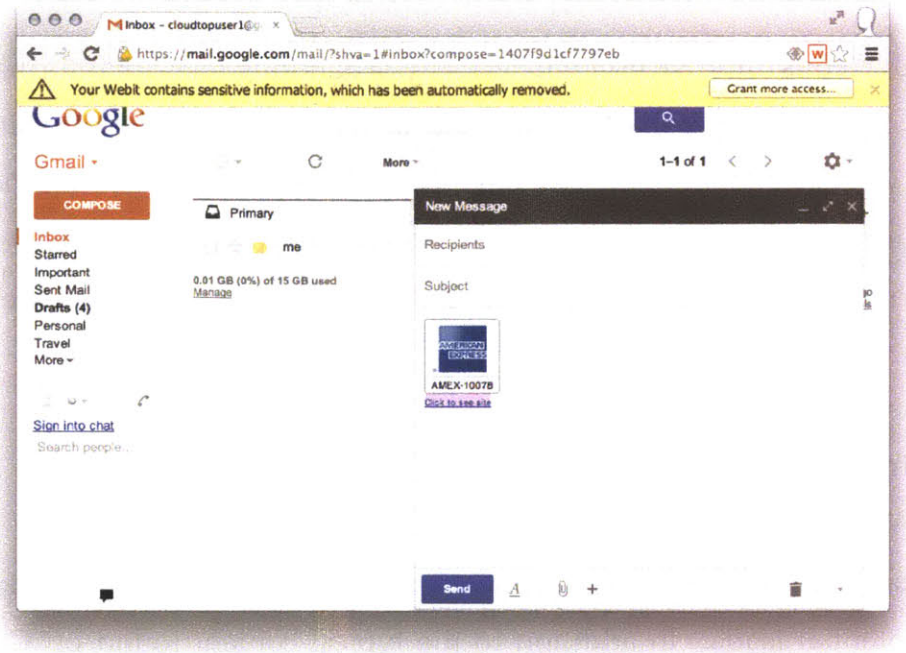


Figure 3.9: When dropping a Webit with redacted properties in a form element, the browser informs the user with a non-modal information bar.

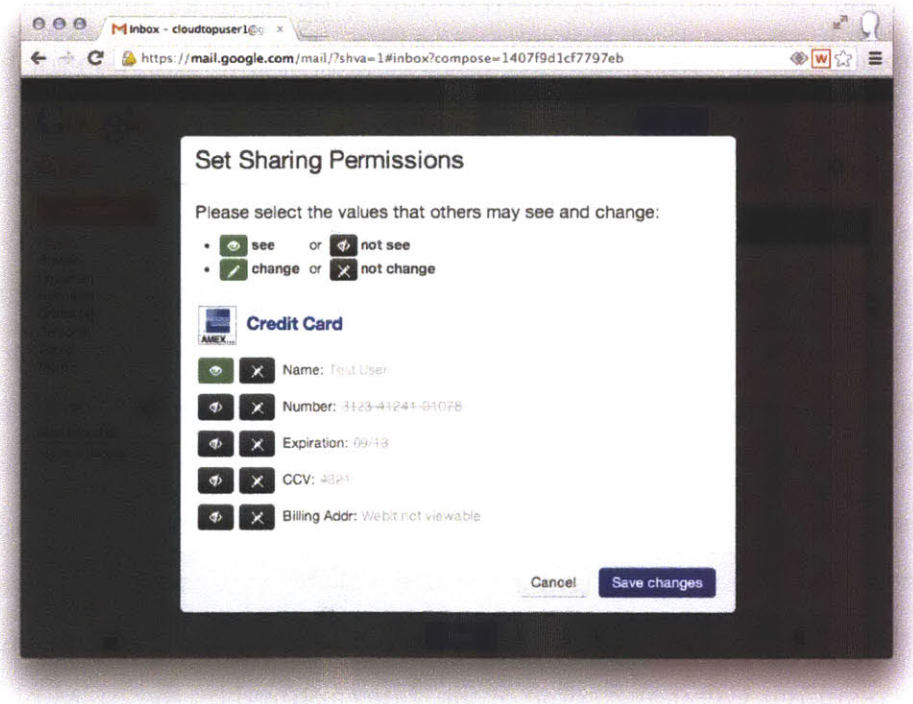


Figure 3.10: Users may specify precisely what to share when dropping Webits with sensitive information onto form elements.

3.1.1 Application Scenarios

In general, Webits improve user efficiency by 1) bundling relevant information into a handle, 2) maintaining that handle intact as it is shared from site to site, and 3) enabling web sites to automatically take appropriate action based on the Webit's bundled information when dropped in various contexts. There are several kinds of situations that leverage the efficiency that Webits afford, illustrated below with scenarios.

Retrofit Integration of Webits Web applications or Clui plugins may add support for Webits to augment an application's core functionality.

For example, Jack is looking for a new apartment. He searches sites like Craigslist for potential leads. In addition to the typical content found on Craigslist's real estate pages, a Clui plugin generates a Webit on each page that represents the associated apartment, capturing metadata like the cost of the monthly rent, the location of the apartment, the landlord's contact information, and the description of the property. Jack drags those Webits to Google Maps, which, with the help of an installed Google Maps plugin that parses dropped Webits for locations to search, maps the locations of the apartments he likes. When he chooses the apartment in the best location, Jack composes an email to the landlord. Rather than typing the address, Jack drops the apartment Webit onto the "To" field; a Clui plugin parses the Webit for contact information and pastes the email address of the landlord.

As another example, Sarah organizes a weekly academic reading group. She needs to select papers from the ACM Digital Library (DL) and notify her group. She opens the ACM DL page for potential papers. In addition to the standard content on the ACM DL, a Clui plugin generates a Webit to represent that paper, embedding the bibliographic information, abstract, PDF link, and so on. Figure 3.11 shows the resulting Webit and some of the data captured by the ACM plugin.

Webit Communication Conduits Web applications that serve as communication channels can automatically transmit Webits without site-specific plugins.

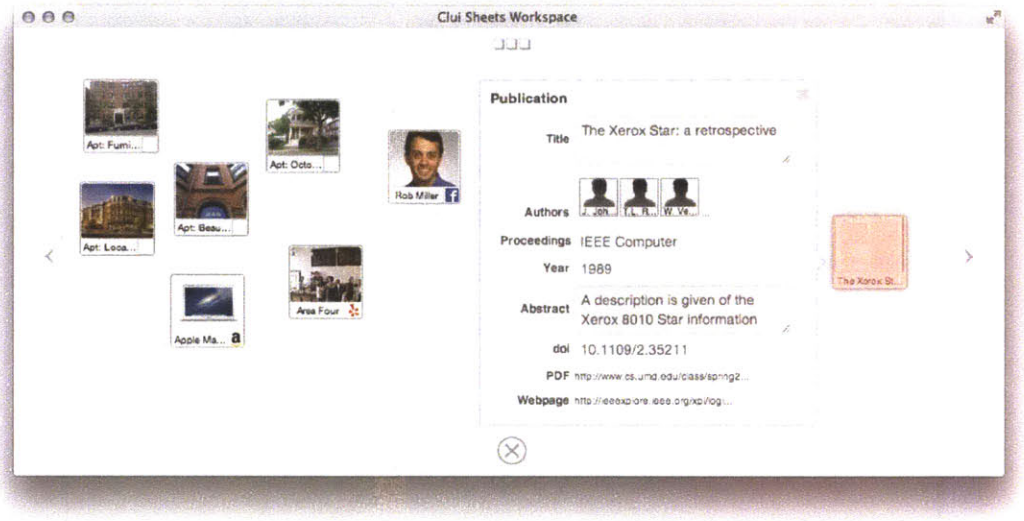


Figure 3.11: A publication Webit's bundled metadata on the Sheets workspace.

For example, Jack, who has found an apartment, now needs to find roommates. He advertises this fact by sharing the apartment Webit with his friends over Twitter. While the Twitter input box for tweets only accepts plain text, Clui pastes enough context such that those who view the tweets with Clui installed will see an inline, draggable Webit. His friends can interact with the Webit in the same ways that Jack can.

Back to Sarah, having selected the paper for discussion, she opens Gmail to send the paper details to her group. Sarah may drag the Webit that represents the paper directly into the message body to share the paper. To provide context, she may also paste a copy of the abstract by dragging the "Abstract" item from the inspector into the message body. Her recipients see both the abstract as plain text and the Webit representing the paper.

Form Filling With the help of plugins, Webits can also efficiently fill out existing web forms.

For instance, Mary is shopping for flights. She first visits AA.com and enters the origin and destination airports, along with the relevant dates. After perusing

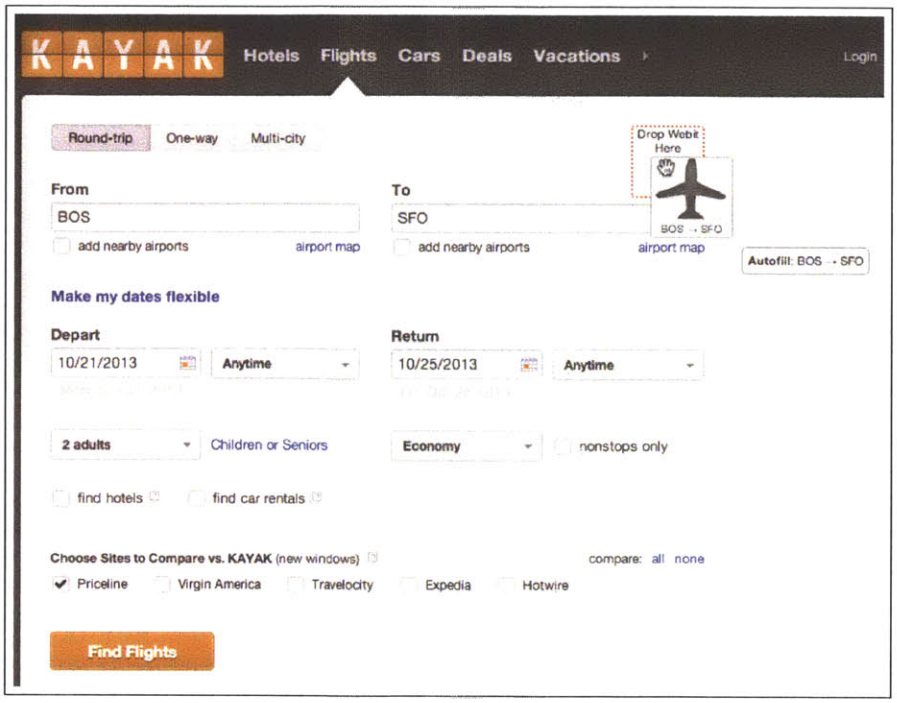


Figure 3.12: Webits help auto-fill forms.

several options, she finds an itinerary that is potentially satisfactory, but she still wants to comparison shop in case there are cheaper alternatives. The AA.com plugin generates a Webit that represents her candidate itinerary.

Mary opens kayak.com to find alternative flights. Rather than re-enter the airports and dates, she drags the Webit from AA.com into the kayak.com form, which auto-fills the relevant fields with the help of a kayak.com Webit plugin (Figure 3.12). In contrast to browser auto-fill features that fill already-visited forms with values cached from previous submissions, Webits enable users to auto-fill new values or fill values on new forms.

Native Webit Applications Previous scenarios illustrate the use of plugins to enable existing pages to create and process Webits. The following scenario illustrates an example of a web application that natively operates on Webits.

Jim is shopping for camera equipment to start a new photography business with a business partner. As Jim shops various vendors, e.g., amazon.com and new-

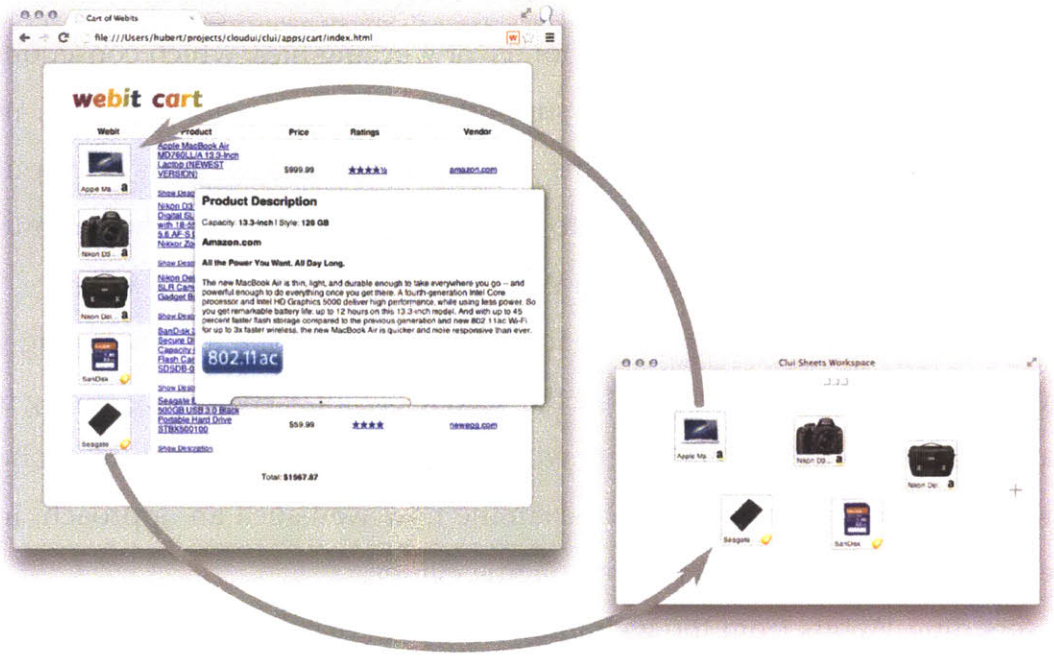


Figure 3.13: The "Webit Cart" is an example of a web application that understands Webits natively. It sorts product Webits based on various parameters and displays additional product details that are bundled within the Webits. Users drag Webits in and out of the cart.

egg.com, he drags Webits of products, produced either natively or with the help of Clui plugins, from these vendors into his Clui workspace. To share his selections with his partner, Jim uses a shared shopping cart service, which is independent of any vendor. The shopping cart natively understands Webits, so when he drags Webits to the page, it displays and sorts relevant parameters for each product (Figure 3.13). The cart may provide a service that finds the cheapest vendor for each product and facilitates the purchase of those products across different vendors.

3.1.2 The Sheets Workspace

The Sheets workspace (Figure 3.11) provides an area to collect Webits, analogous to a desktop surface for files and folders. Users also view and change a Webit's bundled content using the Sheets inspector utility. Sheets aims to retain the benefits of the desktop, such as quick, spatial access to temporary or working resources. However, Sheets also departs from the desktop metaphor in significant ways.

One potential problem with the desktop is clutter management. Some users use the desktop as a pastebin for temporary or working files, which consequently becomes cluttered [23, 50]. Without support for managing old resources, users must occasionally garbage collect items—a difficult task, which users often put off [56]. A desktop simply augmented to support Webits would likely continue to suffer from clutter when used to keep Webits visible and handy.

Web providers already offer permanent homes for storing and archiving user resources, so Sheets can de-emphasize the need to organize Webits, e.g., using folders and groups. Sheets experiments with a chronological notebook metaphor—without collections or hierarchy—to examine how well such an approach, in its extreme simplicity, would serve in managing clutter.

Users drag Webits from the web into the current Sheets page and arrange the icons spatially, much like the traditional desktop. When the current page becomes cluttered, users create a new page to start another blank sheet. Sheets keeps all pages ordered by time, much like a physical notebook.

We hypothesize that the notebook metaphor offers several advantages. Because Sheets is intended to hold temporary working resources, by the time the page is cluttered, many of those resources may no longer be relevant. Rather than expend effort to clean up, conjuring a fresh page is quick and allows the user to postpone garbage collection. Because resources are generally temporary and ultimately return to the web through the course of the user's task, e.g., for sharing or archiving, we speculate that organizing or deleting resources is largely unnecessary.

3.2 Webit Principles

Several design principles, derived from desirable usage scenarios, guide the behavior and features of Webits. Those principles are summarized as:

Bundling A Webit is a bundle of information relating to a semantically meaningful concept or resource and attaches to a visual icon. Bundling interpretable semantic information enables the user to efficiently transfer data between

sites, while the visual icon allows users to quickly identify these semantic objects.

Identity A Webit has an identity. Users and systems can thus determine whether two Webits represent the same underlying object.

Typing Because Webits represent meaningful resources, a Webit has a natural notion of type. A Webit's type helps systems infer the appropriate set of actions to take on that Webit.

Liveness The data encapsulated in a Webit can change over time. Data mutability allows Webits to represent objects whose properties change, e.g., the current weather.

Access A Webit is shareable by sharing its identity and thus its bundled content, providing a simple way to distribute Webits.

Security A Webit may be shared without allowing others to change it.

Privacy A Webit may be shared without revealing all of its information.

The follow sections elaborate on those principles.

3.2.1 Bundling

A Webit bundles machine-interpretable properties of some semantically meaningful resource into a user-facing, visual handle.

Many programming languages and type systems provide ways to bundle or aggregate related information, e.g., structs and classes in C/C++, and to treat those bundles as a new abstraction or data type. The web and its native interface elements provide no mechanism for data bundling, so each site developer that needs it must craft a custom approach using web primitives. One of the central themes of Webits is to standardized the bundling of related information into a user-meaningful handle.

In the abstract, the bundled information in Webits takes the form of key-value pairs. Keys are properties expressed as strings, e.g., a string indicating an address property or a string indicating a date property. Values may be strings, generic URIs, references to other Webits, or arrays, whose elements may be any valid value type. Other primitive value types, like integers, floats, booleans, are encoded as strings and may be interpreted appropriately based on the associated key.

While there are different ways to encode the key-value properties of Webits, the semantic web's Resource Description Framework (RDF) is a natural fit because of its emphasis on enabling different sites to interpret resource semantics. To express statements, RDF uses triples, consisting of a subject, predicate, and object, where the predicate specifies a relationship between the subject and object. The subject and predicate components must each be unique URIs, though those URIs need not be dereferenceable; the object component may be a URI or a string. For example, to express the statement that *Massachusetts has the postal abbreviation MA*, one suitable RDF triple is (urn:x-states:Massachusetts, http://purl.org/dc/terms/alternative, "MA"). The subject is a Universal Resource Name (URN), a special type of URI, that identifies the state of Massachusetts. The predicate references the concept of *having an alternative title* using a URI that represents that concept. The object is a simple string that indicates the alternative title, or abbreviation, for Massachusetts. One describes multiple properties for some object by writing a set of triples which share a common subject.

In addition to URIs, RDF subjects and objects may also be "blank nodes", which are subjects internal to a given set of triples. Blank nodes are useful for referencing anonymous objects, like collections or other resources that only make sense in the context of some other resource being described. For example, the triples (urn:x-person:Bob, http://xmlns.com/foaf/0.1/knows, _:p1) and (_:p1, http://xmlns.com/foaf/0.1/age, 52) express the fact that Bob knows a person whose age is 52.

A Webit's bundled information is encoded in RDF. The keys are expressed as RDF predicates, and the values are RDF objects. The RDF subject specifies the Webit's unique identifier, discussed further below. In addition to strings and generic

URIs, values may encode references to other Webits. Those Webits are expressed in RDF as URIs, as further described in the next chapter.

3.2.2 Identity

Each Webit has a universally unique identifier (UUID) [53]. A unique identifier associated with each Webit allows the system to distinguish different Webits or recognize a given Webit as the same, even if it is encountered across different sites. From the user's perspective, two Webits are the same if changing the bundled information in one is reflected in the bundled information of the other.

Even so, Webits may also leverage a user's context-dependent notion of identity. For example, suppose two Webits describe a specific, television model available for sale, one created by Amazon.com and the other by BestBuy.com. They each have different identifiers, so the system certainly considers those Webits different from each other. The distinction makes sense if the user views one Webit as a reflection of Amazon's prices and reputation and the other as a reflection of Best Buy's. That said, certain contexts would allow the user to treat those Webits as identical, in the sense that they represent the same underlying real-world object. For example, those television Webits may behave identically, e.g., they trigger a search query, when dropped on Google Shopping, Amazon.com, or BestBuy.com.

3.2.3 Typing

Because Webits represent meaningful resources, there is a natural notion of type that arises. While many typing systems are possible for Webits, we considered static typing and duck typing. With a static-typing approach, each Webit type has a statically-defined set of properties, whereas with a duck-typing approach, Webits are typed by the properties that are present. In practical terms, static-typing requires the Webit creator to find an existing type, with an appropriate set of statically-defined properties, or to define a new type. In contrast, duck typing is more flexible and requires less up-front effort to make Webits. One can simply

mix-and-match globally unique properties, each with well-defined semantics, that collectively best describe a Webit's underlying concept or resource. For example, by adding an address property to a Webit, a site like Google Maps can display the Webit on a map, without requiring knowledge of what the Webit actually represents.

This thesis explores Webits that are duck typed. While the presence of properties defines a Webit's type, Webits may also declare a type name for special interpretation by applications. For example, a Webit that represents a collection of items will have properties appropriate for collection types, but the type name could specify that the Webit functions as a shopping cart. As a result, when the user edits that shopping cart, the system could launch the appropriate editor or application designed for manipulating carts. In a Webit's RDF-encoded data bundle, the type predicate defined in the RDF Schema [29] declares a Webit's type.

3.2.4 Liveness and Access

Webit values may change over time. For example, the values bundled within a Webit that represent the current weather or stock price, as shown in Figure 3.14, may change to reflect changing states. Users may also modify values, e.g., to update entries or correct errors.

Websites and users share a given Webit by communicating its identity, that is, Webits are shared by reference. Updates to a Webit's bundled information are visible to those that know that Webit's identity. A user that wishes to copy the current state of a Webit may clone that Webit. The clone is a distinct Webit, with a new identity, and thus receives no updates associated with the original Webit. In the implementation presented herein, each Webit has a single master copy of its bundled data hosted on a server, which serves as the true state for that Webit. A user that holds a reference to a given Webit may potentially obtain, modify, or clone the associated bundle.

An alternative design for sharing Webits is to share by value rather than by ref-

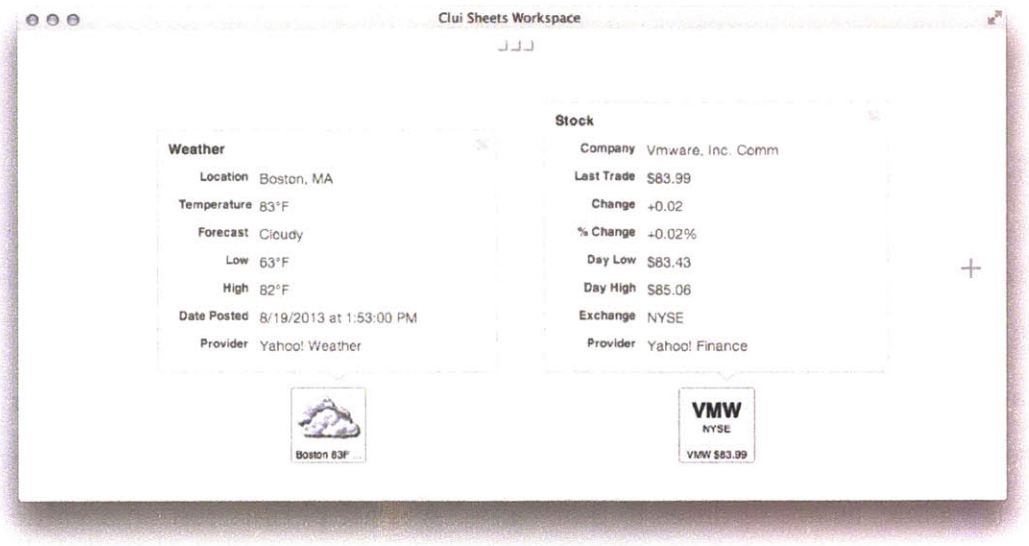


Figure 3.14: Webits may represent data that changes over time, like the current weather or the current price of a stock.

erence, that is, to share a copy of a Webit's bundled data rather than just its identity. Sharing by value, however, imposes several implementation challenges. First, sharing Webits by value using existing services like Gmail or Twitter requires that those services store the Webit's bundled data. Some sites can accommodate such storage, e.g., by leveraging file attachments in Gmail messages, but many sites, like Twitter, impose severe limitations on the kinds or amounts of data they can store and share. Second, if a Webit represents data that changes over time, propagating those changes to all the copies presents a significant implementation challenge. One possible share-by-copy approach that avoids this challenge is to introduce a Webit type that may bundle only immutable data. However, requiring users to distinguish between mutable and immutable Webits, along with their associated semantics, adds complexity to the user interface. Furthermore, immutable Webits do not solve the challenge of storing bundled data on sites that impose storage restrictions.

Sharing by reference, coupled with a single master copy, simplifies data coherence, especially when all systems remain online and connected. For offline or disconnected operation, Webits must implement techniques for resolving conflicts,

such as those described by Coda [66].

3.2.5 Security and Privacy

Security concerns the ability to control how a Webit's bundled content is modified, while privacy deals with restricting what content is revealed to others.

Each property value has an independent read and write permission for privacy and security, respectively. A read permission on a property value indicates that the value is visible, as are updates to that value. A write permission on a value indicates that one may change that value. The presence of properties and the permissions on values are always visible. A Webit is associated with one owner, typically the user or website that creates and maintains storage for the Webit. The owner always retains read and write privileges on all bundled values.

Users and websites may share any Webit for which they have access, and in doing so, they may specify different permissions on property values for each recipient. In other words, different users may hold different levels of permissions for a given Webit. Sharing privilege must remain the same or become reduced, i.e., a user with read-write access to some property value may share that Webit with the same privilege, read-only privilege, or write-only privilege. A write-only permission is useful for situations where a property value holds sensitive data, e.g., a shipping address, that may be changed but not viewed by others. Users or websites may specify permissions each time before sharing a Webit, or they may rely on rules that generate the appropriate permissions, such as *never share my home address with read and write permissions*, which the system automatically enforces.

Several scenarios motivate the scheme described above. In one example, websites that publish Webits representing the current weather or stock price would set most or all properties as read-only. A shared shopping cart might instead have its property values shared with read-write privileges for collaborators, while passive participants may only observe Webit values with read-only access.

3.3 Additional Design Considerations

The sections below discuss additional considerations on the design of Webits.

3.3.1 Predicate Standardization

The effectiveness of Webits depends in part on their interpretability across a wide set of sites. Even though the semantic web and RDF standardize the framework and encoding of semantic information, they do not standardize the vocabularies and semantic constraints for any specific kind of object. Thus, while RDF predicates are URIs and unique, there may be different, competing predicates that describe the same relation or concept. For example, there may be competing standards that define predicates to identify the concept of a physical location's address. This is a challenge to interoperability because a site interpreting a Webit may only recognize one predicate for a given relation, but not the one bundled in that Webit.

There are two ways to overcome such challenges of interoperability. The first is to encourage sites that interpret Webits to accommodate different standards. Another approach is to bundle all known predicates for a given relation in a Webit. Both approaches may be used in concert to increase the chance of successful interpretation, at the cost of added redundancy and complexity. However, those costs may be justified to foster experimentation until de facto standards arise.

3.3.2 Property Visibility

The existence of properties and the permissions on their associated values are always conveyed when Webits are shared. When a value is unintentionally restricted from being shared, recovery is easier when recipients can determine what it is they cannot see. This situation is likely if the user relies on system-enforced rules to automatically restrict what is shared. While sharing the existence of properties may appear to violate privacy, in practice a savvy user will know about the existence of standard properties in various types of Webits.

In general, a user treats Webits as opaque objects, without needing to inspect the bundled properties. When a user needs to view or change a Webit's bundled properties, e.g., to see or change the products in a shopping cart Webit, he may do so using an inspector on the workspace, or he may drop the Webit on a site that interprets and provides a rich interface for manipulating that Webit.

3.3.3 Sensitive Information, Warnings, and Dialogs

In de-emphasizing the need for users to inspect a Webit's bundled content, a challenge arises if Webits contain sensitive information unbeknownst to the user. For example, a trip itinerary Webit that a user receives after booking a flight may contain a reference to a Webit that represents the credit card used for payment. Sharing the itinerary Webit with colleagues may unwittingly reveal sensitive credit card information.

The preferred approach to addressing sensitive information is to avoid creating Webits that contain sensitive data in the first place. For example, rather than embed the actual credit card information, which includes the number, expiration date, security code, and so on, a Webit representing a credit card might contain only non-sensitive information, like the person's name and a friendly identifier for the card, thus acting as a "stand-in". With this approach, the Webit still conveys to the user which credit card it represents, but sensitive information is never transmitted. Additionally, if the user has provisioned her credit card with a given site, e.g., Amazon.com, the site can accept the stand-in Webit and correlate it with the sensitive information already on file, allowing the user to indicate which credit card to use by dropping the appropriate Webit. The user would however need to initially provision sites by manually entering sensitive information, rather than having sites interpret it from Webits.

While avoiding the bundling of sensitive information in Webits is desirable, Webits may still contain sensitive data when it would be inappropriate to use a stand-in, or doing so would defeat the utility of Webits. For instance, a Webit

representing a user's personal contact information, like a home address and phone number, might be useful for finding nearby restaurants on a map via a generic service that does not provision nor retain user data. Using a stand-in Webit would require the user to manually enter her information each time, which defeats the usefulness of Webits. Ultimately, the Webit creator decides what information to include, so the developer must weigh the benefits and risks.

The system must protect the user from accidental sharing of sensitive information. It must strip out sensitive information by default, which prevents irreversible information leakage. In addition, the system warns the user before revealing sensitive data. For instance, while by default the system silently strips sensitive data from Webits, a sophisticated website that detects a stripped Webit may request the system to prompt the user to grant additional access. To minimize user surprise, sites should prompt users, as shown in Figure 3.15, to launch the confirmation dialog, shown in Figure 3.16, rather than display the dialog immediately upon Webit drop. However, sites should accommodate familiar users by invoking the dialog automatically when the user holds a modifier key.

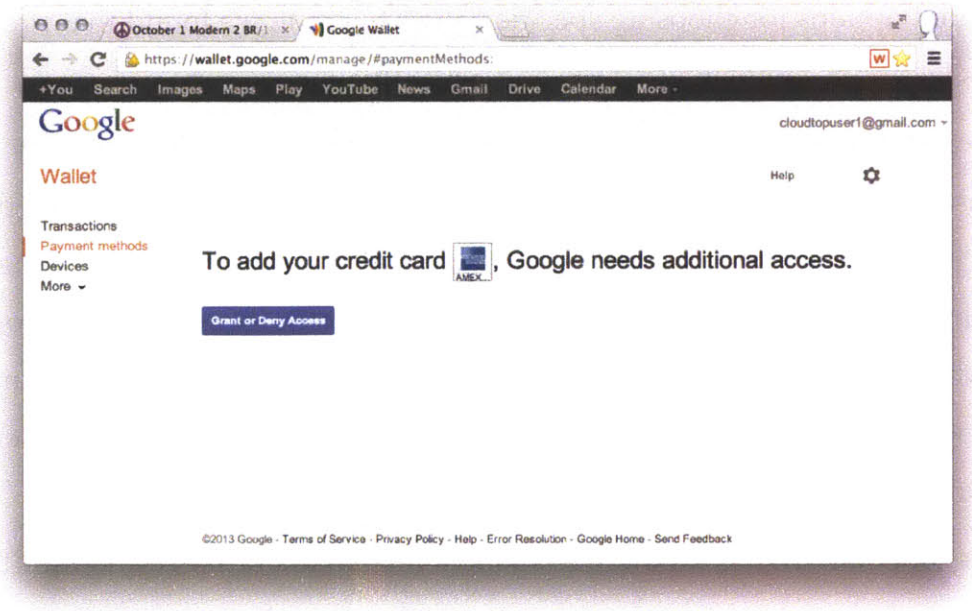


Figure 3.15: A site informs the user it needs access to sensitive data.

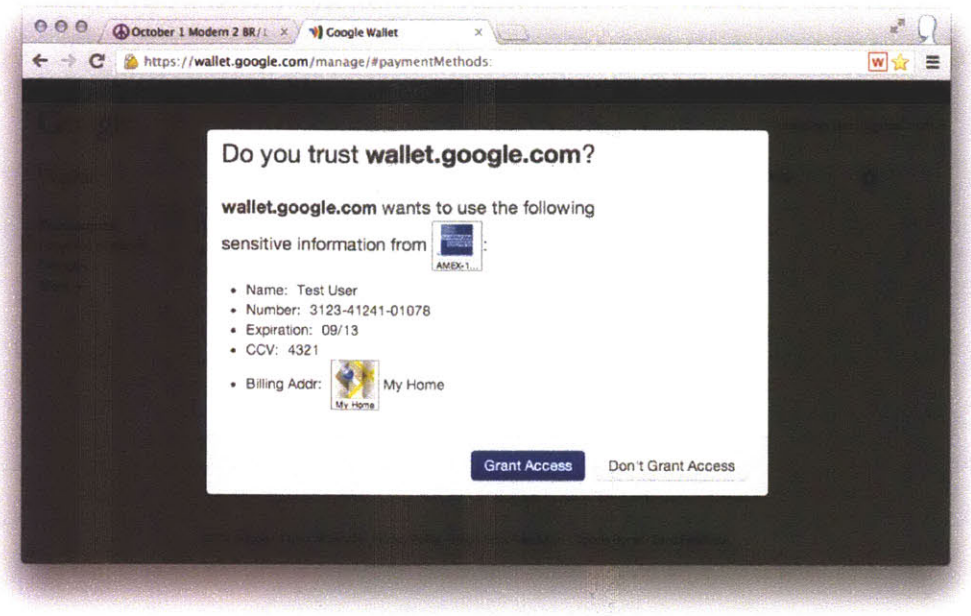


Figure 3.16: A confirmation dialog for sensitive data access.

Figure 3.16 shows a simple dialog box that allows users to grant a web application access to all sensitive values in a given Webit or to grant no access. While we experimented with alternative designs, including ones that included more sophisticated options for choosing specific values to grant, we found that providing options was both confusing to users and generally unnecessary.

The ability to warn users relies on the system knowing which property values are sensitive. The recommended approach is to statically mark sensitive properties. A process that creates a Webit, e.g., a web application, must mark the properties that could contain sensitive values. Note that the system marks properties rather than values. As values may both change and refer to other Webits, marking values would require dereferencing Webits recursively when stripping sensitive information.

As a general design goal, the system must remain unobtrusive and intuitive for common tasks but also accommodate advanced users, who might desire finer control over which values are shared. Advanced users may exercise such control when dropping sensitive Webits on generic form input elements and invoking the fine-

grained access control dialog box from the information bar. Alternatively, users may hold down a modifier key while dragging to automatically launch the dialog, regardless of whether the dragged Webit contains sensitive data. Using that dialog box, a user may fine tune the access control of individual bundled values.

3.3.4 Methods and Webits

Webits do not bundle code methods, which might be called to, e.g., operate on the bundled data or invoke actions with side-effects. Conceptually, methods can be viewed as additional Webit properties whose values are either implementation code or URI pointers to code. However, such a scheme requires additional complexity to verify and trust the code, along with issues associated with managing library dependencies. It is unclear whether the benefits of Webit methods justify the real-world complications of supporting them. Investigating the need for methods is future work.

In the meantime, Webits adhere to a model akin to object-oriented programming using functions that accept the object as a parameter, much like C functions that operate on structs, or Python methods that operate on `self`. Websites, in a position to trust the client-side code that they serve, develop or employ functions that operate on known types of Webits, which the user passes in. In this scheme, library code that operate on Webits may evolve independently of specific Webit types, and sites may vet such code before employing it, rather than need to trust code embedded in Webits.

3.4 Summary

This chapter covered the user interface for Webits, their application classes, and the principles that govern their design and behavior. The next few chapters discuss the underlying system design and implementation of Webits.

Chapter 4

System Design

A key enabler for the adoption of Webits is a system architecture that allows users to use Webits on today's web platform. This chapter discusses the system-level design of Webits and how that design enables Webits to function in the scenarios described previously. It begins with a discussion of design goals and follows with an overview of the system components. The chapter then describes the data structure that underlies Webits.

4.1 Goals

Several goals motivate the system design of Webits. The goals are to:

- optimize for storage scalability and a fair distribution of the storage burden;
- minimize overhead on the user in managing privacy and security;
- use existing mechanisms on the web for sharing Webits; and
- provide reasonable defaults when sharing.

The storage scalability and fair distribution goal states that storage must scale with the number of Webits, rather than the number of users that have access to a given Webit. The goal applies to the implementation of the access and liveness

principles, which describe the behavior that updates to a shared Webit are witnessed by those that have access to the Webit. Clui's approach is to host a single, master copy of a given Webit on a designated server. Those with access to that Webit communicate with the server to make or obtain updates. Because different users may have varying levels of access to a given Webit, the server must enforce appropriate access control according to the user making requests. Furthermore, a user may share any Webit to which he has access and potentially change the access permissions before sharing.

The scalability and fairness goals discourage designs in which a server hosting a Webit must track the associated permissions for each user that has access to that Webit. With such approaches, sharing a Webit with each additional user increases the storage costs. As any user with access to a given Webit may grant others access to that Webit, server costs may be difficult to predict.

The second goal, to minimize overhead on users, implies that system must avoid burdening users with configuration, especially in managing access to Webits. For instance, a system design that requires the user to manually request access to a Webit or permission to share a Webit adds additional steps that impede the user's goal. Such a scheme would also complicate matters for a Webit's owner, who must now approve or deny such requests, or worse, must also interpret and manage the principals for those requesting access. In contrast, a more desirable system avoids interface overheads that impede the user's primary task, while still affording access, privacy, and security.

The third goal, to use web mechanisms for sharing Webits, encourages designs that allow users to share Webits by using existing web applications rather than through a separate, specialized interface. For instance, users must be able to share Webits by embedding them in emails, instant messages, or micro-blogging posts, using the appropriate web application to which users are already accustomed. A related goal is to enable communication services to accept, store, and retrieve Webits without any additional custom code in the storage back end.

Finally, users may acquire and accumulate Webits from different sources, without being fully aware of the contents bundled within each Webit. As some Webits may contain private information, a system that automatically protects the user from accidental sharing of sensitive information is more desirable than requiring the user to inspect and manually filter Webits each time before sharing them.

4.2 System Approach and Architecture

Clui uses capabilities [32, 54] to addresses some of the goals above. A capability confers privileges to the holder of that capability. In the context of Webits, a capability is associated with a specific Webit. A user that possesses a capability for a Webit enjoys the privileges associated with that capability, i.e., the ability to read and/or modify certain bundled values in the Webit. The user may also give copies of capabilities to other users, thus granting others with the same privileges on the associated Webits.

In Clui, a master copy of each Webit is hosted on some designated server. Servers generate capabilities for each of its Webits, and each server may employ its own scheme for generating capabilities. Capabilities must be interpretable by the server that created the capability but remain opaque to others. To retrieve or modify bundled values within Webits, one communicates with the appropriate server hosting that Webit and presents an appropriate capability. Given a capability that the server once created, the server derives an access policy to enforce when performing the requested operation on the Webit.

Depending on the scheme used to generate capabilities, a capability system can achieve scalability and fairness in storage, while imposing minimal overhead on the user. By encoding all necessary information in the capability for the server to determine an access policy for a given Webit, the server needs no additional storage each time that Webit is shared. Instead, each entity that desires access to the Webit is responsible for storing the associated capability. Because capabilities specify the access policy and may be transferred freely, their use eliminates the

requirement that servers track user principals for those who have or desire access to Webits.

As described further below, Clui addresses the goal of allowing users to leverage existing web applications to share Webits by encoding Webits and their associated capabilities as URIs. By exploiting the fact that many sites already support sharing links, Clui enables users to share and store Webits using those sites.

Given an existing capability for a Webit, a server may generate new capabilities with more restrictive access for that Webit. To address the goal of always providing reasonable defaults with regards to access control, the system automatically generates capabilities that allow access to safe, public parts of Webits, and shares these capabilities by default when users share the associated Webits.

This section next discusses the main components of system architecture, the dataflow between those components, and additional considerations concerning the role and provisioning of Webit servers.

4.2.1 Architecture

The three main components to Clui, as illustrated in Figure 4.1, are a browser extension, which provides interface support for Webits; the Webit Sharing Server, which hosts the master copies of Webits; and the Webit Desktop Server, which stores references to Webits the user collects.

Browser Extension The browser extension's main purpose is to 1) provide a user interface for managing Webits, e.g., in a workspace like Sheets, and 2) bootstrap the creation and use of Webits on today's web, where existing sites do not yet offer native support. As further described in Chapter 5, the browser extension creates and renders Webits on specific websites using plugins, handles the mechanics and data transfer associated with drag and drop operations, and provides a workspace area for users to collect and organize Webits, as well as to inspect and modify their bundled content.

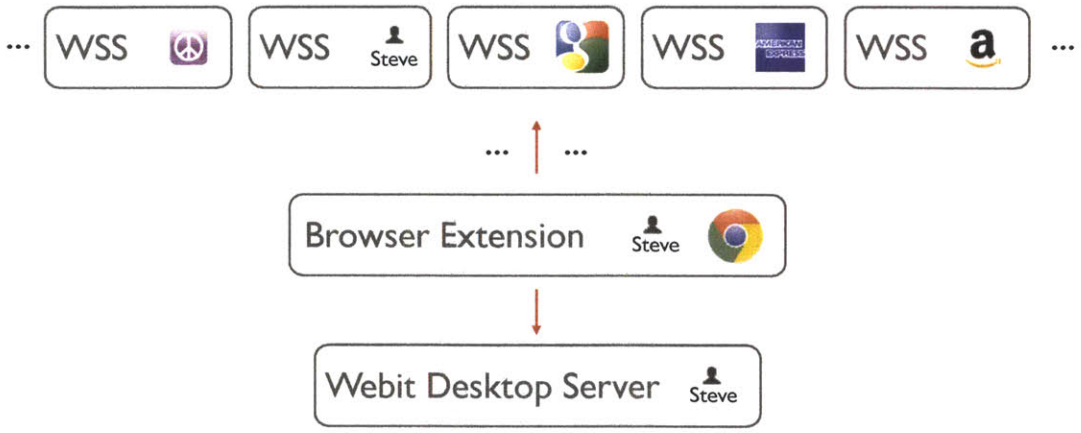


Figure 4.1: The Clui architecture, which consists of the browser extension, the Webit Sharing Server, and the Webit Desktop Server.

Webit Sharing Server The core purpose of the Webit Sharing Server (WSS) is to enable the implementation of the liveness, access, privacy, and security principles. The WSS functions as a designated server to host, serve, and update master copies of Webits. The WSS handles requests to retrieve, update, and delete Webits, as well as requests to create new capabilities from existing ones. Requests on Webits require a suitable capability, which enables the WSS to deduce and enforce the permissions granted to the holder of that capability.

Each user or entity, such as a company or service provider, may run and administer an independent WSS to host and manage Webits that the user or entity owns. The user or entity that exercises administrative control over a given WSS is said to own the Webits hosted there. Figure 4.1 shows servers administered by different entities, each hosting Webits owned by the respective entity.

A given Webit has only one master copy on a designated WSS, though there may be other copies that act as caches. For instance, each web service that creates Webits runs a WSS to host the Webits it creates. A user that gains access to one of those Webits does not store a copy of the Webit in her WSS. Instead, she keeps track of the designated WSS hosting the Webit, the associated capability for that Webit, and optionally a cache of the Webit's bundled information.

Webit Desktop Server The Webit Desktop Server (WDS) serves as a user's personal store for Webit references. A Webit reference, described further below, essentially consists of the Webit's identifier, the address of its WSS, and a capability. In the course of working, a user may encounter on the web many Webits, hosted on various WSS servers. The user may decide to drag some of those Webits and keep them on her workspace. The system thus needs to persist references to those Webits. Each user must have a running instance of the WDS, which acts like a directory and keychain for Webits known to that user. The WDS may also be extended to store workspace metadata, such as the spatial coordinates for each Webit icon on the workspace.

The WDS runs best on a publicly accessible server, as doing so allows users to more easily synchronize their Webit references across different browsers. In theory, a commodity cloud storage service like Dropbox or Google Drive could serve as a WDS.

4.2.2 Workflow

Webits appear on websites either through deliberate programming by site developers or through Clui plugins installed in the user's browser extension. Users may drag Webits from site to site, or they may drag Webits first into their workspace. When dropping a Webit into the workspace, the browser extension stores a reference to that Webit in the WDS, and may cache the Webit's bundled information in local storage. The workspace may either poll the Webit's associated WSS for changes to the bundled values or use a push-based scheme if the WSS supports it. The user may inspect the bundled values in the workspace, and if holding an appropriate capability, may request changes to those values, prompting the workspace to make the appropriate request on the WSS.

When the user drags a Webit to a website, the browser extension transmits both the Webit reference and a cached copy of the bundled values. The reference is essential, as the website requires it to retrieve the bundled content from the ap-

propriate WSS, while the cached copy serves as a performance optimization. The cached copy allows the site to immediately process the bundled values and offer immediate feedback, e.g., in the form of tooltips or other hints.

4.2.3 The Role of the Webit Sharing Server

All Webits must be hosted on some WSS, so that they are shareable. This thesis envisions that each web application that generates Webits will eventually run its own WSS to host the Webits it creates. For example, amazon.com would run a WSS to host Webits that represent products for sale. A user that drags a product Webit from amazon.com to his workspace need only store a reference to that Webit in his WDS. If each site on the web runs a WSS, the user would generally collect, manage, and share Webit references.

While it would appear that the user would have little need to run his own WSS, as all Webits would already be stored elsewhere, there is still value in running a personal WSS. One reason is that the user may wish to exert greater control over the access permissions of Webits, say to more closely monitor the sharing of certain Webits or to exercise revocation of capabilities. Another reason is that Webits may initially appear on the web without an associated WSS. For example, a site may offer a utility that creates Webits for storage on the user's WSS. In addition, some Clui plugins, which create Webits on existing pages, may create Webits that are not yet hosted by any WSS. For these cases, a personal WSS serves as the default host.

4.2.4 Provisioning

As a consequence of plugins that may inject Webits unattached to vendor-run WSS, which may not yet exist, users must either run their own personal WSS or subscribe to a service that hosts personal WSS instances. The WSS exports an API over HTTPS. Configuring a server to support HTTPS is complex, so the use of a service is more user-friendly and appropriate for end users.

Once the personal WSS is created, it must be linked with the browser and

workspace environment, so that they may store Webits on that WSS. The browser requires the WSS address and an owner capability, which grants the privileged ability to host new Webits and delete existing ones.

In practice, the user might use a web form to request a personal WSS from a service provider. Upon instantiating a new WSS, the provider provides the user with a provisioning Webit, which contains the address and an owner capability for privileged WSS access. The user drags that Webit to the browser's configuration page to link and establish the connection with the user's personal WSS.

4.3 Anatomy of a Webit

Under the hood and hidden from the user model, Webits consist of a reference and a corresponding payload. A Webit reference, as mentioned above, consists of the necessary information to communicate with the WSS that hosts the master copy of the associated payload, which contains the Webit's bundled data and other metadata. Users remain unaware of the distinction between reference and payload, since the user interface abstracts away this detail.

On the wire, a Webit is encoded as a JSON data structure, an example of which is shown in Listing 4.1. The format consists of a required reference portion and an optional payload section, which functions as a cache. As a performance optimization, the payload may contain multiple Webit bundles, e.g., a Webit along with other Webits that it references.

4.3.1 References

The reference consists of a:

- Webit ID, a UUID, for identity comparisons
- WSS ID, the address of the WSS hosting the Webit, and
- the associated capability.

```
{
  id: "1976357c-bdd9-481a-960b-221bcf38d292",
  wss: "webits.amazon.com/wss",
  version: 1,

  keychain: {
   1976357c-bdd9-481a-960b-221bcf38d292 : <capability>,
  },

  payload: {
    1976357c-bdd9-481a-960b-221bcf38d292 : {
      id: "1976357c-bdd9-481a-960b-221bcf38d292",
      wss: "webits.amazon.com/wss",
      gen: 0,
      ux: {
        icon: {
          label: "Apple Macbook Air MD760LL/A 13.3-Inch Laptop",
          mainImage: "http://ecx.images-amazon.com/images/...",
          typeImage: "http://www.amazon.com/favicon.ico"
        },
        open: "http://www.amazon.com/Apple-MacBook-MD760LL-13-3-Inch-
           VERSION/dp/B00746YPQI/",
        dataTransfer: {}
      },
      content: {
        1976357c-bdd9-481a-960b-221bcf38d292: {                  // subject
          http://www.w3.org/1999/02/22-rdf-syntax-ns#type: [{ // pred
            type: "literal",                                     // obj
            value: "http://purl.org/goodrelations/v1#Offering",
            id: "75fb", read: true, write: false
          }],
          http://purl.org/goodrelations/v1#name: [{
            type: "literal", id: "8337", read: true, write: false,
            value: "Apple Macbook Air MD760LL/A 13.3-Inch Laptop"
          }],
          http://purl.org/goodrelations/v1#description: [{
            type: "literal", id: "4a2d", read: true, write: false,
            value: "The new MacBook Air is thin, light, and durable
               enough ..."
          }],
          http://purl.org/goodrelations/v1#hasCurrencyValue: [{
            type: "literal", id: "369f", read: true, write: false,
            value: "1044.99"
          }],
          ...
        }
      }
    }
  }
}
```

Listing 4.1: An example Webit encoded in JSON. Quotations around keys are omitted for clarity.

In the JSON format, the capability can take the form of a keychain, which maps Webit IDs to capabilities. This enables the WSS to return multiple capabilities when handling multiple Webits. For example, when adding a new Webit, along with all the new Webits that it references, the WSS must in turn generate capabilities for each Webit and return them in the keychain.

The reference portion may also be expressed as a URI, an example of which is shown in Figure 4.2. A URI representation carries several advantages. First, many web applications are already designed to accept and store URIs, thus enabling those applications to automatically store Webit references without modification. Second, there are existing tools that operate on URIs to make them more convenient to handle, such as URI shorteners. By using URI shorteners, one can embed a Webit reference in tweets on Twitter, even though tweets are character limited. Finally, the use of URIs also enables a graceful fallback for users without the Clui extension installed, described next.

In normal operation, a built-in plugin in the Clui browser extension detects the presence of Webit references encoded as URIs, transparently dereferences them to obtain the associated payload, and when possible, replaces the URIs with visual handles. Thus, while sites may only store and initially render URIs for Webit references, users with the Clui browser extension installed never see those URIs but instead see visual Webit handles. Users without the browser extension installed will see a link, which when clicked, will cause the browser to make a request on the WSS hosting the Webit. Rather than return the Webit payload, which might confuse the user, the WSS instead displays a friendly HTML page describing the Webit, with instructions to install Clui. To achieve this, the WSS inspects the HTTP `Accept` header, which indicates which MIME types the requester supports. When making requests to the WSS, the Clui extension specifies `application/json` in the `Accept` header, whereas clicking a link in the browser does not. To return the appropriate response, the WSS inspects the `Accept` header in the HTTP request to deduce whether the requester is the browser or the Clui extension.

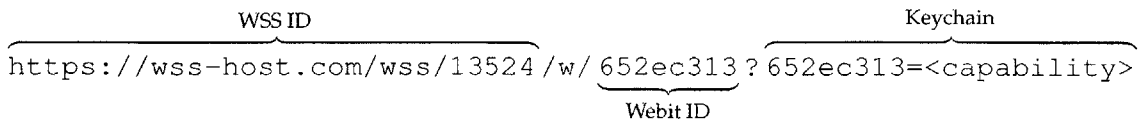


Figure 4.2: The URI encoding scheme for Webit references. Webit IDs are shortened for clarity.

Figure 4.2 illustrates a URI encoding of Webit reference. Because the keychain is sent across the network in requests to the WSS, the protocol must be HTTPS to avoid sending capabilities in the clear. The URI's host, port, and path include the WSS ID and the Webit ID. The keychain is encoded as query parameters. While the user information part of a URI, e.g., the user:pass in https://user:pass@host.com, might serve as a more semantically apt area to embed the keychain, in practice many web applications, like URL shorteners, do not recognize URIs with a user information part.

One consequence of encoding Webits as URIs is that the page that displays those URIs must be served over HTTPS; otherwise, capabilities are delivered in the clear. Fortunately, the adoption of HTTPS for popular sites is growing, but users are still at risk when posting Webits—or any other information, in general—on sites that do not offer HTTPS by default. One stop-gap solution is for the Clui browser extension to check whether a given site employs HTTPS before pasting Webit references on that site and otherwise warn the user.

Finally, embedding the location of the WSS in a Webit reference causes minor complications when the master copy of the Webit payload must migrate to a different WSS. For example, as users may subscribe to service providers to host their personal WSS, users may also change providers and thus need to move their Webits to a different WSS. The system updates the WSS ID portion of existing references by leveraging redirects, such as HTTP's 301 Moved Permanently status. Redirection relies on the original WSS to serve redirect notices for some grace period, so that existing references have time to poll the original server, process the redirect, and update the reference.

4.3.2 Payload

The payload section contains:

- a generation number, in gen, to synchronize updates;
- a section for user experience parameters, in ux, which includes metadata that dictate the rendering and drag-and-drop behavior of Webits; and
- the bundled information, in content, encoded in RDF-JSON [71].

Generation Number The generation number guards against changes to the bundled information based on stale data. When handling a request to change the bundled information, the WSS requires an input generation number, which it compares with the current generation number stored in the master payload. If the numbers do not match, WSS assumes that the requester is operating on stale data and refuses the change request. Otherwise, if the numbers match, the WSS makes the requested update and increments the generation number atomically.

User Experience The UX portion contains metadata that specify a Webit's icon in ux.icon, the canonical page associated with Webit in ux.open, and overrides on the default drag-and-drop behavior in ux.dataTransfer.

The icon component requires a label, a short user-visible string that describes the Webit. Optionally, it accepts a primary image to display, an image to denote the Webit type, and HTML or text to use when an image is not available. The images may either be URIs that refer to an image hosted on the web, or they may be data URIs that encode the image.

A Webit is associated with some web page, the URI of which ux.open specifies. For example, a restaurant Webit may be associated with the web page for that restaurant, a general directory page for that restaurant, or failing those, a map of the restaurant location, e.g., on Google Maps.

Webits have default behaviors when dropped on various targets, as described in the previous chapter. If specified, ux.dataTransfer overrides those defaults. For

example, a Webit that represents a snippet of text, such as a shipment tracking number, might configure ux.dataTransfer so that a drag-and-drop operation pastes the number, rather than a reference to the Webit. ux.dataTransfer is a map between MIME types and the values to paste. The common MIME types, text/html, text/plain, and text/uri-list, represent different drop target types that map to rich-text input fields, plain-text input fields, and whitespace, respectively. A Webit might not override the default drag and drop behavior but instead specify additional MIME types, e.g., application/rdf+xml to leverage sites that natively interpret RDF.

RDF Bundled Content The RDF, encoded in a modified RDF-JSON format, captures the bundled semantic information relating to the Webit. In RDF-JSON, the object portion of an RDF subject-predicate-object triple is encoded as a JSON object, with keys for the RDF object value and object type. The object type may either be uri, literal, or bnode, which correspond to values that are URIs, literal text strings, or blank nodes. The RDF bundled in Webits conforms to the RDF-JSON standard but includes a few additional modifications.

One modification concerns references to other Webits, which are encoded as URIs, in the object portion of RDF triples. To distinguish between Webit references and other URIs, the object type may be set to webit for Webit references. Some care is necessary when creating new Webits that reference other new Webits, as the referenced Webits will not yet have an assigned WSS. In this case, the URIs for the referenced Webits will consist of just the Webit ID. After the WSS stores a referenced Webit, it rewrites references to that Webit into a canonical one that includes the WSS host, path, and capability.

Another modification adds unique IDs to each RDF triple, using a id field. These IDs need only be unique within the set of RDF triples expressed in the Webit. Triple IDs ease the identification of specific triples for various operations, e.g., modifying values.

RDF triples also include a read and write field, indicating whether or not the triple is readable and writable, respectively. Ultimately, the capability dictates the

permissions, but as the capability are opaque, these fields are necessary to aid the user interface. The WSS, able to interpret capabilities, sets these fields when returning the Webit payload. The read field disambiguates between the case of empty values and that of insufficient read permission. The write field enables the user interface to selectively display controls to modify values on the appropriate triples.

Some RDF triples may contain sensitive information, which must not be shared by default, as discussed in Chapter 3. Such triples, identified statically, have a sensitive field set to true. If a Webit has at least one triple with a sensitive field set, that Webit is said to contain sensitive information. When an RDF triple references a Webit that contains sensitive information, that triple must also be set sensitive to avoid leaking the reference. Before sharing Webits, the Clui browser extension strips out predicates with the sensitive field set.

4.4 Summary

The system design of Webits strives to meet several goals, including storage fairness and scalability, minimal overhead on users, and the use of existing web-based mechanisms for sharing Webits. While users perceive a Webit as a single handle to some semantic object, under the hood, a Webit consists of a reference to some payload hosted on a server called the WSS. The Clui browser extension implements the user interface and collects Webit references, which it stores on the WDS. A Webit reference consists of the Webit ID, the address of the WSS, and a capability that grants the holder certain access privileges. The payload consists principally of the Webit's bundled information, encoded in a modified form of RDF-JSON, along with metadata that dictate the Webit's visual appearance and behavior.

The next few chapters further discuss the design and implementation of the Clui browser extension, the WSS, and the WDS.

Chapter 5

Browser Extension Design

This chapter details the design of the Clui browser extension, which targets the Chromium [1] browser. The extension consists of:

- a core component, which provides several APIs that abstract storage, server communication, and security concerns;
- a plugin system, which creates and interprets Webits on existing web pages, as well as generates human-readable descriptions of Webits; and
- a pluggable workspace environment, which provides users with a surface to collect and manage their Webits.

Before discussing each component above, this chapter first provides some background on the Chromium extension system, the HTML5 drag and drop standard [24], and an overview of the dataflow between the components list above.

5.1 Background

Early prototypes of Clui were implemented as a Mozilla Firefox extension. As the Chromium extension system matured, it offered a simpler and more developer-friendly environment, at the cost of some API flexibility as compared to Firefox. This section overviews the Chromium extension framework and the HTML5 drag

and drop standard, two underlying fundamental technologies upon which Clui is built.

5.1.1 Chromium Extension Framework

Chromium extensions are written in JavaScript, along with HTML and CSS for user interface components. Chromium exports an API for extensions that provides access to privileged browser functionality, such as modifying the browser interface, e.g., to create toolbar buttons or popup windows; registering callbacks for various events, such as a new page load; inserting code snippets into web pages currently open in a tab; and passing messages between environments. In addition, extensions may use supported HTML5/CSS3 APIs, such as those for implementing drag and drop, persisting data locally, making AJAX requests, and drawing graphics and animations.

An extension is structured principally around the *background page* and optional *content scripts*. There is one background page for each extension, which loads when the extension loads. The background page, not user-visible, acts as the central component of the extension and enjoys unrestricted access to the Chromium API. Content scripts are JavaScript code snippets that run in the context of a web page open in a tab. A content script may access and modify the DOM of the page it runs in, although the script executes in a sandboxed environment, so as to not interfere with existing JavaScript associated with the page. As a security measure, a content script is unprivileged, in the sense that it has almost no access to the Chromium API. Content scripts may access only API calls that pass messages to the background page, which in turn may carry out privileged operations.

The Chromium API provides no direct access to an open web page's DOM, so if an extension needs access to both the DOM of an open page and privileged parts of the Chromium API, the background page and content scripts must work in tandem. The developer may statically declare which content scripts to load when the browser loads certain URIs. Alternatively, the background page may

load content scripts programmatically, e.g., by registering handlers to call when the browser loads new pages and subsequently injecting the appropriate content script.

The Chromium API offers several approaches to customize or alter the browser interface. One is the *page action*, which manifests as a button with a customizable icon in the browser address bar. The page action is appropriate for operations that operate on the current open page. The background page may programmatically change the icon and register handlers to handle clicks. Another useful approach is to create visible popup windows from the background page. The popup window has full access to the Chromium API, and while separate from the background page, may fully access the background page's JavaScript environment. Clui uses the page action button to indicate the presence of Webits on the page, and it uses a popup window to house the workspace.

5.1.2 HTML5 Drag and Drop

The primary way in which Clui interacts with web pages is through the HTML5 Drag and Drop API. To track and handle drag and drop operations, the API specifies a set of events, namely dragstart, dragenter, dragover, dragleave, and drop. All drag and drop events carry a DataTransfer structure that contains a key-value property list of MIME-typed representations for the item being dragged. For example, the DataTransfer for a text snippet holds a text/html representation containing an HTML string as well as a text/plain one containing the string without markup.

Web pages and browser extensions may add their own custom MIME types to the DataTransfer. During the lifetime of a single drag and drop gesture, all associated drag and drop events share the same DataTransfer object, allowing handlers fired on drag events to populate the DataTransfer with data that is consumed by drop handlers. Note that code handling the dragover event, fired during a drag operation as the cursor hovers some element, may not change the DataTransfer structure. Moreover, it may only inspect the keys, which name the MIME types, but not the

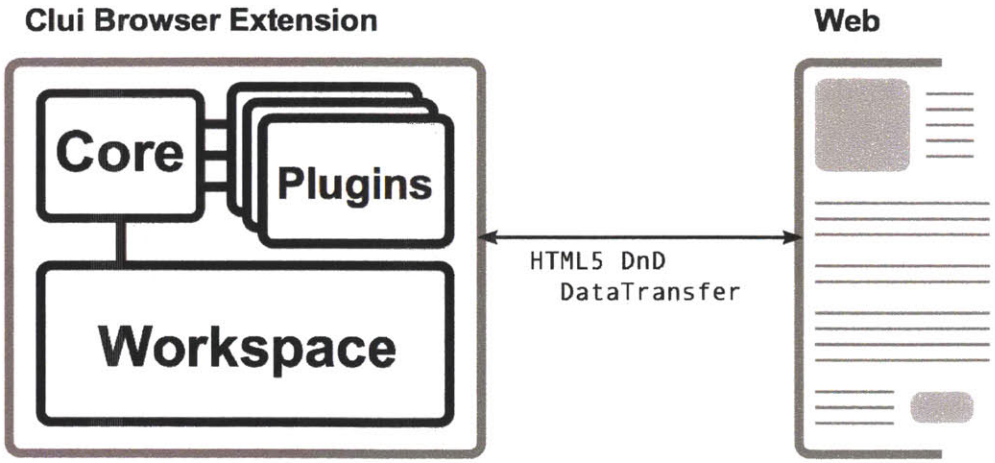


Figure 5.1: The Clui browser architecture.

values. This has important implications for implementing tooltips that indicate and preview the drop behavior.

5.2 Overview of Clui's Operation

As mentioned, the Clui browser extension consists of the core component, the plugins system, and the workspace. Figure 5.1 illustrates the architecture. The core component is implemented as a background page. One of its roles is to manage and load plugins at the appropriate time to scrape and augment web pages to create and use Webits. Plugins are implemented as content scripts. The workspace is implemented in a popup window, which the core can display in response to user command.

Clui responds to several major events, such as when the user navigates to a new page, drags a Webit, drops a Webit, and inspects a Webit using the workspace. Each of these events is briefly discussed below.

The core component registers a handler that is called when the user loads a new web page. The handler determines if there are suitable plugins to execute in the page. If so, the handler programmatically loads them. The plugins scrape the DOM to create Webits or add additional functionality to interpret dropped Webits. Clui includes several system-level plugins, such as one that implements tooltips,

that are always loaded on every page.

When a user drags a Webit, either one embedded in a web page or one on the workspace, Clui augments the DataTransfer structure with several encodings of the Webit. Notable encodings include the strings to paste when Webits are dropped on different kinds of targets, as well as a machine-readable encoding.

When dropping a Webit on a web page, the browser automatically handles the resulting operation depending on the drop target and the contents of the DataTransfer. However, if a plugin is installed on a page, it may override the default behavior with a custom implementation.

On the other hand, dropping a Webit on the workspace persists a reference to that Webit. The workspace handles the initial drop event and calls into the core component to handle the new Webit. If the Webit is not already hosted on some WSS, the core component also persists the Webit payload to the user's personal WSS. The workspace then renders and displays the Webit at the drop location.

Users may inspect a Webit's bundled metadata on the workspace. The workspace calls the core component to generate a human-readable description of the bundled data. The core in turn leverages a suite of plugins that specialize in interpreting Webit types and returning an appropriate representation for user consumption.

The remainder of this chapter discuss in detail the operation of the core, plugins system, and workspace.

5.3 Core Component

While the core component handles navigation and user interface events, it also exports an API for plugins, workspaces, and websites. This section discusses the core's exported API and various services the core provides.

5.3.1 Storage Services

One key component of the core is the storage module, which persists Webits. The storage module consists of a WSS client that interfaces with any given WSS to retrieve or update Webits. With the user's personal WSS, the client may also add or delete Webits. The storage module also includes a WDS client that manages Webit references, such as storing them, retrieving previously acquired capabilities for references, listing all references, and deleting references.

Capabilities are generally required to carry out most storage operations on the WSS, requiring calls to both the WSS and WDS clients. The storage module wraps those clients with a higher-level API to add a new Webit, retrieve a Webit's payload, delete a Webit, and update a Webit's payload values.

In principle, to minimize the need to poll WSS hosts for payload changes, the storage module may employ a push-based scheme with WSS hosts that support it, e.g., using WebSockets [38]. Such WSS servers either optionally elect to bear the costs associated with the additional state necessary for a push-based feature or use a third-party intermediary, like firebase.com. Alternatively, the storage module could implement sophisticated polling and caching schemes [22, 31, 65, 75, 78]. The current implementation does not yet apply these techniques, instead employing a simple approach of fetching updates on-demand, e.g., when the user inspects the bundled content in a Webit.

As mentioned in Chapter 4, the storage service requires configuration when the extension is initially installed. Configuration parameters include the server address and path of the user's personal WSS, an owner capability to make privileged requests that add or remove Webits, and the server address and path of the user's WDS. While users may enter this information manually in the extension's configuration page, users may also drag and drop a configuration Webit with the parameters bundled, furnished from a service provider hosting the user's personal WSS and WDS.

5.3.2 DataTransfer API

The core exports an API call, setupDataTransfer, that initializes an appropriate DataTransfer structure for a Webit when it is dragged. Typically, the workspace or other core services described below make use of this function by first registering a dragstart handler on the DOM element representing the Webit, and in that handler, calling setupDataTransfer.

The contents of a DataTransfer also function as an interface for websites that handle dropped Webits. While a Webit may augment the contents of the DataTransfer, by default, it contains:

- a value for the text/plain MIME type, which is automatically pasted in plain text boxes;
- a value for text/html, which is automatically pasted in rich text editors;
- a URI in text/uri-list, which the browser navigates to when the Webit is dropped on whitespace;
- a JSON-encoded representation of the Webit in application/x-clui-webit, for consumption by plugins or Webit-aware websites;
- a flag in application/x-webit-sensitive, indicating whether the Webit contains sensitive information; and
- a variety of internal-use MIME types for rendering tooltips.

The default value for the text/plain entry is a text string that includes a short, human readable description of the Webit and a URI reference. This scheme allows users to paste Webits in regular forms and persist them on servers that accept only textual content, such as Twitter. However, pasting a description with a Webit reference into search engines may yield undesirable results. In those cases, if the site uses HTML5 semantic markup to indicate that a given input field functions as a search query input box, Clui can automatically remove the reference. For sites that

have not yet adopted semantic markup, site-specific plugins must delete the Webit reference.

The default value of text/html is an HTML rendering of the Webit. The resulting effect is that when the user drags a Webit to a contenteditable element, such as a rich-text editor, she sees the Webit appear in that area.

The value of text/uri-list is the Webit's associated web page URI. For example, a restaurant Webit may specify the homepage for the restaurant as the associated URI. When dropped onto whitespace, the Webit functions like a bookmark and navigates the browser to the specified URI.

A Webit may override the values for text/plain and text/html, and it must specify text/uri-list. In addition, a Webit may specify additional types to include in the DataTransfer. However, it may not override the types described below, which are automatically added by setupDataTransfer and used by Clui to communicate with web pages and other components.

The value in application/x-clui-webit is the JSON-encoded serialization of the Webit, as described in Section 4.3. The principal use for this type is for plugins, workspaces, and Webit-aware websites to parse dropped Webits and take appropriate action. For example, a plugin for Google Maps parses the JSON-encoded Webit in application/x-clui-webit, scans for location metadata, and if found, maps the location. In another example, dragging a Webit from the web to the workspace prompts the workspace to parse the JSON-encoded Webit, persist it if necessary, and render it in the user interface.

The flag in application/x-webit-sensitive indicates whether the dropped Webit contains sensitive information. The core strips sensitive information from Webits, so no sensitive information appears in the JSON-encoding associated with application/x-clui-webit, preventing Webit-aware sites from capturing that data without user permission. Webit-aware sites that wish to access or operate on sensitive Webits may inspect the application/x-webit-sensitive flag and request further access to the sensitive data, as described in the next section.

Tooltips appear during a drag operation, indicating to the user what data will

be pasted to various drop targets. The MIME types for tooltips address the issue that while the MIME types are enumerable during a drag operation, the associated values are not visible until the user drops the Webit. This presents challenges for generating tooltip content while the user hovers the cursor, as the system cannot parse the JSON-encoded Webit in application/x-clui-webit until the user drops the Webit.

Clui's workaround is to mirror the contents of text/plain, text/html, and text/uri-list in the keys of the DataTransfer, as special MIME-types, e.g., application/x-clui-tooltip-plain/<user visible tooltip text>. As keys are enumerable, the tooltip system can scan for the special MIME-types, parse out the mirrored content, and display the appropriate values as users hover.

Some sites need to inspect the bundled content during a drag operation to display tooltips. For example, an apartment Webit may contain a reference to another Webit representing the apartment's location. As the user drags that the apartment Webit over Google Maps, to show the location that Google Maps will display in a tooltip, a plugin for Google Maps must parse the bundled content in the apartment Webit, scan for location Webits, and dereference those Webits. Because the JSON-encoded Webit in application/x-clui-webit is not available until the drop event, setupDataTransfer also mirrors the JSON-encoded Webit in a key of the DataTransfer. The keys in the DataTransfer are case-insensitive, so the JSON-encoding is also Base32-encoded in order to preserve case. Case is especially important in Webit references, which contain capabilities that may be case-sensitive.

5.3.3 API for Plugins and Websites

The core exports several API functions for plugins and websites, described further below. The API is summarized in Table 5.1.

Plugins and Webit-aware websites make calls on the API with the help of a JavaScript library they import. As the library may be running in the context of a web page, it may not have access to the message passing API that content scripts

use to communicate with the background page. Hence, the library communicates with the core using a trampoline. It forwards every call to a content script trampoline, managed by the core, using the HTML5 Web Messaging API [45]. That trampoline in turn forwards calls to the core using Chromium's message passing mechanism for content scripts.

Storage Plugins typically create Webits not yet hosted by any WSS. When the user drags the Webit to her workspace, Clui persists the Webit, assigns it a WSS ID, and generates a capability. Until that happens, the user cannot safely share the Webit, as others will not be able to dereference it.

To allow users to drag a plugin-created Webit from page to page, without needing to first drop it on the workspace, plugins call phost to provisionally persist unhosted Webits before they are made draggable. If a user drags a provisionally hosted Webit, Clui upgrades the Webit to be fully hosted. When the user navigates away from the page, Clui automatically garbage collects all provisionally hosted Webits not upgraded.

In addition, the core exports the ability to retrieve a Webit payload given its reference. It does not export storage operations concerning deleting or updating Webits to minimize security issues involving sites that might manipulate Webits without user knowledge. If the need arises, future versions of Clui may export a more liberal subset of the core's storage API, with access restricted to installed plugins.

Interpretation The core exports a function interpret that returns a human-readable version of a Webit's bundled content. For example, when dropping a product Webit on Google Spreadsheets, a plugin calls interpret to obtain a human-understandable mapping between properties and values in the bundled content. The plugin then automatically adds the appropriate column headers, or detects existing ones that match, and inserts a row containing the bundled values.

Call	Description	Parameters	Return Values
get	Retrieve a Webit	A Webit reference	A Webit
phost	Provisionally host Webits	Webits to host	Hosted Webits
interpret	Return a human-readable description of a Webit's fields	A Webit	A map between human-readable property names and values, and a display order for property names
setTooltipText	Set the tooltip text for the current drag operation	An event object, and the tooltip string	None
requestAccess	Request access to sensitive information in a Webit	A Webit; a list of desired properties and permissions for those properties	A new Webit
getRdf	Extract RDF from the DataTransfer	The DataTransfer object	An RDF-JSON object
getPredicates	Retrieve a set of property-value pairs embedded in the RDF	The RDF, a list of predicates to retrieve, optionally a list of referenced Webits to recurse	An object with property-value pairs
Webit	Webit object constructor. The object provides convenience methods for rendering and RDF manipulation	The content to bundle and ux parameters	A Webit object

Table 5.1: The API for Plugins.

Tooltip Overrides The core renders tooltips, which are shown during a drag operation, using a content script, but plugins and websites may override the default tooltip text using setTooltipText. For example, a plugin for Google Maps displays the text "Show location for ..." as the user drags a Webit over the map or search input box. Plugins and websites typically call setTooltipText in a dragover handler for the appropriate element to set the text. The tooltip text is reset upon a drop.

Privacy and Security Websites and plugins may request additional access to a Webit's bundled information. For example, they may require access to sensitive data or seek write permissions to change the bundled data. Requesting access requires the caller to explicitly specify the desired fields and permissions. The core may then immediately grant access, if the user has previously established appropriate rules allowing access. Otherwise, the core prompts the user to approve access. Once access is granted, the core returns a new Webit, including a new capability and payload, to the requester.

Utilities The core provides various utilities for websites and plugins. A notable one, getPredicates, aids in parsing a Webit's bundled RDF and extracting values from specific predicates. The caller may also request that getPredicates dereference Webit references attached to certain predicates and extract values from those Webits. For example, to handle tooltips when mapping locations, the Google Maps plugin uses getPredicates to scan for location-related predicates directly bundled in a location Webit, as well as for those predicates bundled in a location Webit that may be referenced within, for example, an apartment Webit.

5.3.4 Security Services

In managing privacy and security, the core automatically strips sensitive content from Webits before they are shared and displays dialog boxes that prompt the user to grant additional access when necessary. The core may display one of two different boxes. For sites that request access using requestAccess, the core displays a

warning box that allows the user to see the requested content and confirm or deny access. A dialog that offers fine-grained controls to set permissions on individual properties is necessary when the user drops Webits on a target that does not request access to specific properties, e.g., a generic form element.

Rather than show a pop-up window, the core superimposes a modal dialog in the open web page that requests access, in order to achieve a more seamless user experience. The core displays the modal dialog by inserting an iframe in the web page and rendering the dialog box in that iframe. As the dialog displays sensitive content to the user, the iframe prevents a malicious web page from scraping the contents of the dialog.

Dropping a Webit on a form input element invokes the dialog box with fine-grained controls. The core uses the same iframe mechanism to display the modal dialog. Once the user specifies the desired permissions and closes the dialog, the core requests from the appropriate WSS a new capability that reflects those permissions. If the user originally dropped the Webit into a form element, that element now contains the Webit reference, albeit with the original capability. Hence, upon dismissing the dialog, the core scans form elements and contenteditables for the original reference and rewrites it with the new capability.

5.3.5 Other Services

The core offers several other UI services, notably mechanisms to show tooltips, automatically render detected Webits, and highlight Webits on an open web page.

The tooltips module is a content script that augments the open web page with a handler for the dragover event. The handler inspects the DataTransfer for the appropriate Webit tooltip MIME types, as described above, responds to overrides from plugins, and renders the tooltips using standard HTML elements.

Webits may appear on web pages as either icons, rendered using HTML elements, or as a reference, rendered as a link in an anchor tag. On every open tab, the core scans for both forms. It detects Webit references by iterating through all

anchor tags and evaluating whether the reference URI matches the appropriate pattern. It detects the icon form by scanning for elements that have the data-webit attribute. The value of that attribute is the JSON encoding of the Webit, which includes a cache of the payload. The core replaces detected Webit references with their icon form by first dereferencing those Webits. It then scans for Webits in icon form and attaches drag handlers to set up the DataTransfer. Rendering Webits visually and making them draggable is a common task, so centralizing the process and making it automatic eases development for site-specific plugins.

Users may discover Webits with the aid of the page action button, embedded in the address bar of an open tab. The core changes the icon of a tab's page action button to indicate the presence Webits in that tab, detected in the same way as described above. Clicking on the button toggles the highlighting of Webits on the page (Figure 3.1). To highlight Webits, the core uses a content script to insert a semi-transparent overlay and re-renders the Webits in the overlay.

5.4 Plugin System

Plugins consist of scrapers, augmenters, and interpreters. A scraper creates Webits on existing pages. An augmenter adds code to pages to interpret dropped Webits and perform an appropriate action. An interpreter examines a given Webit and returns a human-readable description of its bundled data.

Scrapers and augmenters serve as a bootstrap mechanism to help Webits gain traction in two ways. First, they add Webit support to existing web pages without relying on the cooperation of the developers for those sites. Second, they serve as a reference implementation for sites to borrow and incorporate. In time, if Webit support becomes widespread, the need for scrapers and augmenters diminishes.

5.4.1 Scrapers and Augmenters

Scrapers and augmenters are implemented as content scripts. In practice, the distinction between a scraper and an augmenter is only a semantic one. Each scraper and augmenter is associated with a URI pattern, such that when the user opens a tab matching that pattern, the core injects the appropriate scrapers and augmenters on the page. Scrapers and augmenters are installed statically in the Clui browser extension. As additional ones are developed and contributed, they are submitted to maintainers of the Clui browser extension, who examine, vet, and add the contributions. Alternative models are possible, e.g., a decentralized one in which plugins are embedded and published in separate, independent Chromium extensions that register with the Clui browser extension at runtime.

Once injected into a web page, the typical operation of scrapers and augmenters is to scan for the known DOM element nodes, and if found, to either create Webits or to attach handlers that interpret dropped Webits. For example, a scraper for a product page on Amazon.com scans for nodes corresponding to the title of the product, its cost, description, ratings, and so on, as inputs to Webit generation. An augmenter for Google Maps scans for the main map container and search box, and attaches handlers that map location data bundled in dropped Webits.

To initially determine the relevant DOM nodes, a plugin developer might use a web debugger to inspect relevant DOM elements and obtain the associated node IDs or XPaths. Once the nodes are determined, scrapers and augmenters may use external libraries to obtain references to those nodes and attach handlers.

However, plugins must wait for elements to load before scraping them or attaching handlers. Operating on static web pages, in which all elements are loaded before the DOM `load` event fires, is straightforward, as plugins simply inspect, augment, or scrape the page once `load` fires. However, many sites load content asynchronously or in response to user action. As an extreme example, when loading certain Gmail or Facebook pages, the browser may execute JavaScript that programmatically builds the document asynchronously. In such cases, the browser

fires the load event once the bare DOM loads, even though the user-facing page has not fully loaded.

While polling is always an option, plugins may effectively handle asynchronously-loaded resources by responding to DOMSubtreeModified events on some parent container, which is fired whenever DOM nodes in that container change. As DOMSubtreeModified may be called frequently due to minor changes, care must be taken to debounce, or filter out spurious events.

Once a scraper finds and scrapes the relevant DOM nodes, it creates a Webit using library code that 1) constructs the RDF bundled data, 2) renders an icon representation in div node, and 3) embeds a JSON-encoded representation of the Webit in the data-webit attribute of the div node. When the plugin inserts the icon representation into the web page, the core automatically detects the presence of the div with the data-webit and makes it draggable, as described above. As such, a scraper essentially need only scrape for content once available, call a library function to construct a Webit, and insert that Webit onto the page.

Plugins that scrape or augment DOM elements are inherently brittle and prone to breaking as sites change. While tools that feature visual techniques for clipping and scraping, such as Dontcheva's work [33, 34], may help ease the development burden of updating plugins, the most resilient approach is for web authors to directly publish semantic data and generate Webits. Alternatively, a more attractive future is for web site authors to embed platform-agnostic microdata within their pages, from which Clui could automatically generate Webits.

5.4.2 Interpreters

The core exports an API call to interpret a Webit and return a human-readable description of the bundled data. To do so, the core relies on interpreter plugins, which scan RDF statements and generate a set of key-value pairs appropriate for human consumption.

Unlike scrapers and augmenters, which are site-specific, interpreters work with

metadata from any source. Each plugin handles a specific type, e.g., people, products, or locations, by scanning for a set of known properties associated with that type. As a Webit is duck typed and thus able to represent different kinds of objects, multiple interpreter plugins may each extract and contribute a description for different aspects of a single Webit. For example, an apartment Webit is interpretable as a real-estate product as well as a location. Each plugin that attempts to interpret a Webit reports a score that rates how well it is able to interpret that Webit. The core composes those descriptions and ranks them by the reported scores.

5.5 Workspace

The workspace provides the interface by which users collect and organize Webits. As the interaction with the core is minimal, Clui workspaces are pluggable to facilitate experimentation with different interface approaches for managing Webits. The workspace may present any interface implementable in HTML, CSS, and JavaScript, and has access to the core API for Webit storage and interpretation, as described above.

The reference workspace, Sheets, presents a notebook metaphor of disposable pages that hold Webits, as described in Section 3.1.2. Each page is a `div` container, arranged side by side to form the notebook. CSS3 transforms and animations provide hardware accelerated 3D effects (Figure 5.2) as users navigate between pages.

In Sheets, as users may arrange Webit icons spatially on each page, much like icons on the traditional desktop, the workspace must also manage and persist the coordinate locations of each Webit. Sheets normally uses local storage to store the coordinates of Webits on each page but may use cloud-based storage to synchronize state across installations.

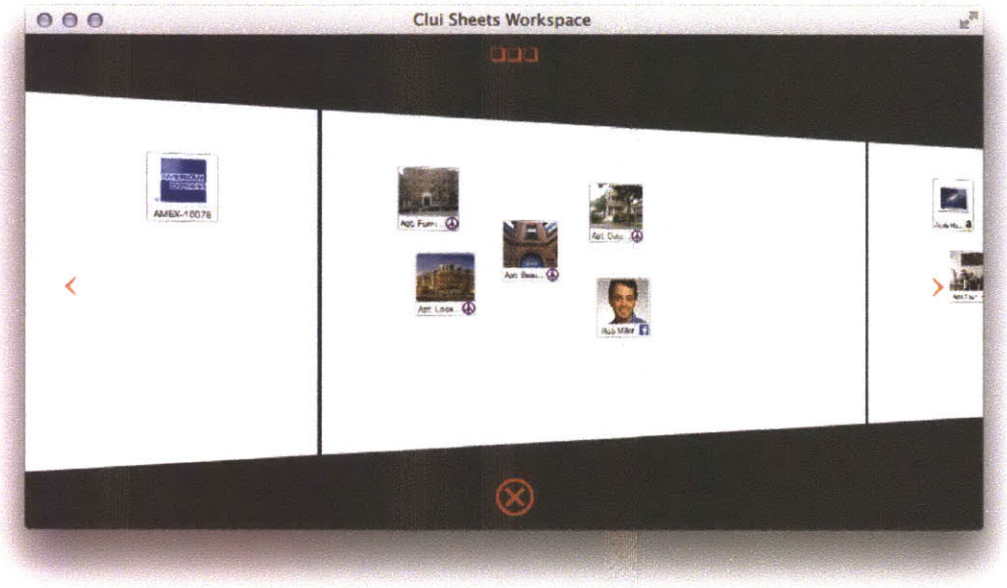


Figure 5.2: 3D effects in the Sheets workspace.

5.6 Summary

The Clui browser extension provides the interface support for Webits. It consists of a core component, plugins, and the workspace user interface. The core provides APIs for plugins, websites, and the workspace to manage Webits, and it centralizes common services. Plugins bootstrap the use of Webits by injecting them into existing web pages, augmenting web pages to perform useful actions with dropped Webits, and interpreting Webits to provide human-readable descriptions. The pluggable workspace offers an interface to hold, organize, and inspect important Webits.

The browser extension described in this chapter constitutes half of the Clui platform. The other half are the server-side components, notably the WSS, which is the topic of the next chapter.

Chapter 6

Webit Server Design

The server support for Webits consists of the Webit Sharing Server (WSS) and Webit Desktop Server (WDS). The WSS hosts master copies of Webit payloads, which may be fetched or modified given the corresponding Webit references and capabilities. The WDS stores references to Webits that the user collects. Web pages or services that create Webits run a WSS to host those Webits. Users may also run a personal WSS to exercise finer control over sharing and access permissions. Each user runs or subscribes to a WDS, which supports the user's workflow by storing Webit references, which notably include capabilities.

This chapter first describes the design of the WSS and its external API. Webit capabilities are at the heart of the WSS, as they dictate the access policies when handling requests. This chapter describes a reference capability scheme that supports access control policies based on both the properties and values bundled in Webits. However, as a capability is opaque and interpretable only by the WSS that creates that capability, each WSS may select any reasonable scheme for capability semantics. Finally, the chapter concludes with a discussion on the WDS.

6.1 Webit Sharing Server

This section focuses on the API that the WSS exports and briefly summarizes some implementation notes.

6.1.1 API

The WSS exports a RESTful API [39] over HTTPS to manage Webits and capabilities. For Webit management, the WSS implements handlers to create, retrieve, update, and delete Webits. Clients may also derive a new capability with fewer access privileges compared to a given capability, or trade multiple capabilities for a single one that represents their union. In adhering to the REST design pattern, URI paths represent resources, like Webits, and HTTP methods represent actions on those resources. Table 6.1 summarizes the API.

A Webit reference, which consists of a Webit ID, WSS ID, and capability, contains the necessary information to communicate with the WSS hosting the Webit. The WSS ID contains the host, port, and path to the WSS, such as wss-hosts.com/wss/17101319/. The protocol is always HTTPS and thus not listed, and the default port is 443. The path identifies the specific WSS instance to query. To support the hosting of multiple WSS instances on a single server, e.g., as a commercial service might do to host separate WSS instances for each subscriber, the path may contain a WSS instance identifier that is internal to the server. In the example above, the instance identifier is 17101319. A resource name, which is either a Webit or a capability utility, is appended to the WSS path. Webit resources are at /w/<webit id>, while the capability utilities are at /policy/. Continuing the example, to operate on a Webit with ID 23283132 hosted on the WSS with ID wss-hosts.com/wss/17101319/, the client makes requests to https://wss-hosts.com/wss/17101319/w/23283132.

The following paragraphs describe the API calls summarized in Table 6.1.

Webit Retrieval Fetching a Webit retrieves the current state of the Webit payload from the WSS host. The request requires a capability that is URL-encoded [57] as key-value pairs in the query string. The keys are Webit IDs, and the values are the corresponding capabilities, e.g., w/23283132?23283132=<some capability>. The scheme supports the transfer of multiple capabilities to optimize the retrieval of multiple Webits that are hosted on the same server in one request. On success, the server returns the JSON-encoded Webit.

URI Resource	Action	Restricted	HTTP Method	Parameters	Returns	Error Codes
w/<webit id>	Retrieve Webit payload	No	GET	Capability	Webit	Bad capability (403) Non-existent (404)
	Modify Webit payload	No	PUT	Webit with updated payload (includes capability)	Webit	Bad input (400) Permission denied (403) Non-existent (404) Stale (412)
	Add Webit	Yes	PUT	Owner capability, Webit	Webit	Bad input (400) Permission denied (403)
	Delete Webit	Yes	DELETE	Owner capability	None	Permission denied (403)
policy/deriver	Generate a new capability from an existing one	No	POST	Existing capability, parameters for new capability	New capability	Bad input (400) Bad capability (403)
policy/combiner	Combine several capabilities into a single one	No	POST	Existing capabilities, conflict resolution	New capability	Bad capabilities (403), Capability conflict (412)

Table 6.1: The WSS API. For calls to URI resource w/<webit id>, the Webit ID is an implicit parameter, derived from the resource location. Webits, as parameters and return values, are JSON-encoded. Restricted calls, intended to serve only the WSS instance owner, are available only to those with an owner capability, which must be embedded in an HTTP header. Hence, while changing a Webit's payload and adding a new Webit both use PUT, the presence of the owner capability disambiguates the intended action.

Webit Modification A request to modify a Webit's payload requires the new, proposed payload values and a capability that allows the requested modifications. Not all fields need be present in the proposed payload, allowing those without access to some fields to still request modifications to accessible fields. Modifying collections is supported, but adding new fields is disallowed. The capability must allow all requested modifications or the operation fails. All requested modifications in a request are committed atomically.

The server must ensure that the requested modifications target the most current payload data. Otherwise, two clients requesting conflicting modifications may yield inconsistent state. The proposed payload includes a generation number, which the server compares against the one on storage. If the generation numbers mismatch, the server returns an error code, leaving it to the client to retrieve an updated copy of the payload, resolve any conflicts, and retry.

Webit Creation A client may add new Webits for hosting on a WSS if the client has privileged access to that WSS. A client has privileged access if it serves an administrative user, e.g., a user who owns or subscribes to the WSS for personal hosting. The client must supply in an HTTP header an owner capability, recognized by the WSS, to identify the client as privileged. While modification and creation operations use the same HTTP method, `PUT`, the server distinguishes the intent based on the presence of the owner capability. Adding a Webit overwrites any Webit with the same Webit ID stored on the WSS.

In general, for every Webit it adds, the server must generate a new capability, embedded in the Webit JSON-encoding that is returned. The capabilities may be full capabilities, which grants unlimited access to the associated Webits. Clients can thus be configured to automatically generate a more restrictive capability that is appropriate for sharing, as described below.

A client may add a new Webit that references other new Webits that also need hosting. A client may add such Webits atomically by sending them all in the `payload` section of the JSON-encoding. Webits that are referenced may have incomplete

references, as their capabilities are still unknown. As the server generates capabilities for referenced Webits, it rewrites the associated references. For example, a new Webit A has an RDF payload that references Webit B, which is not yet hosted, so the client sends both A and B. A's RDF uses an incomplete reference to B, consisting of just B's Webit ID. After the server generates B's capability, it rewrites references to B that are encoded in A's RDF to include the WSS ID and capability.

Webit Deletion Deleting a Webit is also a privileged operation and requires an owner capability. In the future, to support a Webit that migrates to a different WSS, deletion may take an additional parameter that specifies a redirect WSS ID and a requested grace period to serve the redirect.

Capability Generation Any client may request a new, more restricted capability based on an existing one. A more restricted capability means setting an existing permission of read=true or write=true to read=false or write=false. In addition to the existing capability, the parameters include a description of the new, desired capability, which is specific to the capability system employed. The next section describes a reference capability system and provides an example of capability generation.

Capability Combination Any client may request a single capability that represents the union of multiple capabilities for the same Webit. Clients may collect different capabilities for a given Webit over time. For example, a user who requests and is granted privileged access for a public Webit will have multiple capabilities for the same Webit. As capabilities are opaque, without additional context, the client cannot inspect capabilities to determine their privilege characteristics and thus which capability from a set would be most appropriate to use. While it is possible to design Webit references to encode multiple capabilities for a given Webit, doing so becomes unwieldy as a client collects capabilities. The ability to combine multiple capabilities into a single one simplifies bookkeeping and avoids the need for APIs and Webit references that must support multiple capabilities.

6.1.2 Implementation

The prototype WSS implementation uses NodeJS [11] for web request handling and Redis [14] for persistence. NodeJS offers a JavaScript environment to build high-performance web server applications. Redis features a simple, persistent key-value store with support for executing multiple operations as an atomic set. The WSS prototype also includes an implementation of the reference capability system, described next.

6.2 A Reference Capability Scheme

The goals of the reference capability scheme are:

Scalability The storage requirements on a WSS must scale with the number of Webits it hosts, not the number of users that have access to that Webit.

Dynamism Capabilities may express access based on both properties and their values. Values may change over time, along with the user-set policies governing access. A capability must reflect permissions given the current policies and values, even if that capability is created prior to changes in policy and values.

An example of the dynamism goal is a capability that enforces the policy of *no access to personal information*. What a user considers personal information may change over time, e.g., a home address. In addition, a given Webit's field values may certainly change as well and thus may come to hold personal information. A capability that grants no access to personal information enforces that semantic over time, even as Webit values and personal data change.

While similar on the surface, the reference capability system's goals and approach differs from the sensitive field, described in Section 4.3.2. The sensitive field is statically set on properties and is independent of values. Its utility lies in the process of sharing Webits, acting as a conservative heuristic to limit the propagation of fields known to potentially contain sensitive data. The capability system

operates on both properties and values. Once users have access to a given Webit, the capability system aims to enforce policy semantics, even as the specifics of those semantics change over time.

This section next describes the general approach of the reference capability system. It then details the components that make up a capability. Following that is a discussion on algorithms for interpreting and enforcing permissions.

6.2.1 General Approach

To address the scalability goal, a capability in the reference system directly embeds the description of access permissions. As such, a Webit reference, which includes a capability, effectively includes the description of permissions. This approach contrasts with other systems in which a capability is an opaque reference to a description that is stored and maintained on the server. By embedding the description in the capability, a client that maintains interest in some Webit bears the cost of storing the description of its access permissions. Server storage hence scales with the number of Webits it stores rather than with the number of clients that have access to Webits.

Capabilities must be opaque and tamper-resistant to prevent malicious clients from altering the intended description of permissions. In our scheme, the description is called a *policy*. A capability is a policy encrypted with AES-192. The WSS that creates the capability encrypts the desired policy using a guarded secret key, so only that WSS may decrypt the capability and interpret the associated policy. As different policies may have similarities, e.g., a system default policy might apply to many Webits, AES-192 is an apt encryption scheme because it is widely believed to be resistant against known-plaintext attacks. In such attacks, the attacker may deduce the secret key based on knowledge of the ciphertext and plaintext.

```
{
  classes: {
    named: {
      personalInfo: { read: false, write: false },
    },

    default: {
      read: true, write: false
    }
  },

  overrides: {
    http://www.w3.org/2006/vcard/ns#Address: [{
      value: "32 Vassar Street",
      read: true,
      write: false
    }]
  },

  capOverrides: {
    12e78f: {                        // for this triple ID, if the value
      <webitId>: <replacement cap>   // has a reference to this Webit ID,
    }                                // replace its capability with the
  },                                 // replacement capability

  serial: "29af8175bc18ef91",
  version: 1,
  full: false,
  webitId: <webitId>
}
```

Listing 6.1: An example policy, in which personal information is neither readable nor writable, but all other data is readable and not writable. A specific address, 32 Vassar Street, when paired with the http://www.w3.org/2006/vcard/ns#Address property, is readable but not writable, regardless of whether it is considered personal information. Quotations around keys are omitted for clarity.

6.2.2 Policy Components

The goal of our policy scheme is flexibility, but more specifically, to support property value dynamism. The scheme includes several components, described below. A policy is encoded in JSON, an example of which is shown in Listing 6.1.

To calculate the appropriate permissions given a Webit and a policy, the WSS enumerates over the Webit's property-value pairs, and for each, consults the policy to determine whether that pair is readable and whether it is writable.

Data Classes To support of classes of information that may change over time, e.g., personal information, a data class declares specifiers that match Webit property-value pairs belonging to that class. The WSS hosts and maintains data classes and their specifiers. A data class has a name, e.g., personalInfo, and a specifier, or patterns against which to match a given property or value. A policy description embeds the class name, along with the permissions that apply to that class, while the specifier is maintained on the server and may evolve over time. For example, Listing 6.1 shows an example policy which specifies that property-value pairs that match against the specifier for the personalInfo class must not be readable nor writable. Note that the policy does not embed the specifier. To take another example, the WSS for Amazon.com might define data classes to match personalized content within product Webits. It could then issue policies that share the public parts of a product Webit but also selectively issue policies that share a user's purchase history for that product.

Defining a class requires defining a specifier for data patterns that belong to that class. A specifier is a collection of records that match property names and values. Each record contains a rule that matches a property name and a rule that matches values. In principle, rules may be arbitrarily sophisticated, e.g., regular expressions. In the reference capability scheme, rules are simple string patterns that must equal an input string for a match to occur. A Webit property-value pair belongs to a data class if it matches any record in the specifier. Listing 6.2 shows an example specifier for personalInfo, which might exist on some user's personal WSS, encoded as a JSON object where each key-value pair represents a record.

A rule may also be a wildcard to match any input value. For example, a record {someProperty: *} will match all property-value pairs with a property named someProperty, regardless of the value. Similarly, {*: 123-45-6789} matches any value with 123-45-6789, regardless of the property name. The use case is to match known values, e.g., a social security number, regardless of the associated property, or known values in a collection, where the property name is irrelevant.

In addition to a specifier, defining a class also requires specifying default read

and write permissions that apply to that class. A policy may not list all classes known to the WSS because some classes may be defined after the policy is created and circulated. When the WSS receives a policy without all classes enumerated, it must still consider the missing classes when calculating access control permissions. The default read and write permissions for a class must provide a reasonable default to use when that class is missing from a policy.

Default Data Class Some predicate-value pairs will not match any data class, so a catch-all is necessary. As an implicit data class, the *default class* matches all property names or values not matched by named data classes. The policy must embed permissions for the default class under classes.default, as shown in Listing 6.1.

Permission Overrides A policy includes a section to specify permission overrides for certain property-value pairs. Overrides take precedence over the permissions dictated by data classes, allowing users to explicitly mark specific pairs with desired permissions, e.g., using the sharing-permissions dialog box.

Capability Overrides A Webit may reference other Webits, and each of those references will necessarily have some capability embedded. Thus, when sharing a Webit that references others, the recipient obtains the embedded capabilities of the referenced Webits. This may be undesirable, as the embedded capabilities may be powerful and inappropriate to share. For example, a shopping cart Webit hosted on Amazon.com may need a reference to a shipping address Webit. Amazon.com needs a capability containing access to the shipping address Webit in order to obtain all the necessary fields to build a shipping label. However, a user sharing that shopping cart Webit may find it undesirable for the recipient to have full access to the shipping address Webit. Instead, the user might prefer to share the shopping cart with a restricted capability for the shipping address Webit, e.g., one that reveals only the city portion of the address.

A capability override allows the server to return a new capability in place of an

```
personalInfo := {
  "http://www.w3.org/2006/vcard/ns#Address": ["32 Vassar Street"],
  "http://schema.org/CreditCard": "*",
  "*": ["123-45-6789"]
}
```

Listing 6.2: An example class specifier. This specifier matches property-value pairs that either contain the address 32 Vassar Street in the value, any credit card, or the string 123-45-6789 in the value, regardless of property. The value in a key-value pair may either be a string or an array of strings.

existing one. To share a Webit A that restricts the capability for a Webit B referenced within, the system first creates a desired capability for B that would be appropriate for sharing, C_B. Then, the system creates a new capability for A, C_A, that includes an override rule. The rule states that a reference to B must use C_B, overriding the original capability for B. The system packages C_A in the reference to A before sending it to the recipient.

When a client dereferences A using C_A, the server retrieves the payload for A from storage and evaluates each property-value pair. If the pair matches a capability override entry, and the value of that pair contains a Webit reference to B, the server overwrites the capability for that reference using C_B.

In the policy, a capability override specifies the RDF triple to which it applies, along with a Webit ID. The Webit ID ensures that the override applies only to a specific Webit, as values may change over time.

Miscellany A policy includes a variety of other fields. A capability is tied to specific Webit, so embedding the Webit ID allows the server to verify the association between a Webit and a capability. A unique serial number is helpful for the server to revoke capabilities. The full property, when true, indicates that the holder has complete access to the Webit, overriding all other components. When the server stores a new Webit, it creates a policy with full set to true for that Webit, affording the owner full access.

6.2.3 Policy Algorithms

When handling requests on a Webit, the WSS must evaluate an associated policy to determine and enforce the appropriate permissions. This section discusses procedures for performing those calculations.

Fetching & Modification In fetching a Webit's payload, the server evaluates each property-value pair against the following, in the given order for precedence, to determine whether the pair's value is readable and thus can be returned.

- The policy's full value: if set to true, the pair is readable.
- Property-value overrides: if an override exists for the pair, return the read permission dictated by the override.
- Data classes, after merging all known data classes with the data classes listed in the policy: if the pair matches a data class, return the read permission dictated by the data class.
- Default data class: return the read permission specified in the default data class.

In addition, the server also checks each readable pair against the list of capability overrides. If a pair matches, and the value is a Webit reference, the server rewrites the capability.

A pair may match multiple data classes with conflicting permissions. The server takes the conservative approach, in which access denial takes precedence. Users can overcome such situations by using the property-value overrides.

Modifying a Webit payload takes a similar approach, except that the system inspects the writable permission instead. To handle the addition and modification of collections, represented as RDF blank nodes, the server ensures that new RDF subjects are connected by a chain of references to at least one property value in the main set of property-value bundles.

Derivative and Combining Policies Clients may request a new capability C_N with fewer privileges based on an existing one, C_E. The server creates the policy for C_N by first cloning C_E and modifying the clone. For data classes and overrides, the permissions for each is the boolean AND of the desired permissions for C_N and the corresponding permissions in C_E. An AND operation ensures that permissions remain the same or become less permissive. New capability overrides are simply combined and may overwrite those of C_E. The server also generates a new serial number for C_N. Regardless of whether C_E is a full capability, C_N will have its full property set to false.

Combining several capabilities into a single one with the union of their privileges is similar. The server iterates through each of the input capabilities and combines each with the output capability, i.e., in a reduce operation. The permissions for data classes and overrides of two policies are combined with an OR operation, while capability overrides are merged. However, capability overrides may conflict. For example, two capabilities may specify different capability overrides for a given Webit in a given property value. When there are such conflicts, the server returns an error, along with the conflicting capability overrides, and the client must resolve those conflicts, such as by combining the conflicting capabilities. On retry, the client may specify a conflict resolution structure that specifies the capability overrides to use for each conflict situation.

6.3 Webit Desktop Server

The WDS manages Webit references that the user collects. In principle, the WDS is a simple key-value store, where a key is the WSS ID combined with Webit ID, and the value is the associated capability. In practice, clients may want to store different Webit capabilities for different contexts. For example, a client that creates and adds a Webit to a WSS will obtain a full capability for that Webit. However, before sharing that Webit, the client may need to request a new capability with fewer permissions appropriate for sharing. The client may want to cache that capability

URI Resource	Action	HTTP Method	Parameters	Returns	Error Codes
/webits	List all Webits	GET	None	List of wssID + webitID	Permission denied (403)
/webits/<wssID + webitID>	Retrieve the default capability	GET	None	Capability	Permission denied (403) Non-existent (404)
	Set the default capability	PUT	Capability	None	Bad input (400) Permission denied (403)
	Delete default capability	DELETE	None	None	Permission denied (403)
/webits/<wssID + webitID>/named	Retrieve list of named-capability names	GET	None	List of names	Permission denied (403)
/webits/<wssID + webitID>/named/<name>	Retrieve named capability	GET	None	Capability	Permission denied (403) Non-existent (404)
	Set a named capability	PUT	Capability	None	Bad input (400) Permission denied (403)
	Delete a named capability	DELETE	None	None	Permission denied (403)

Table 6.2: The WDS API. All calls are privileged so an owner capability must be embedded in an HTTP header.

rather than need to regenerate it each time.

Table 6.2 shows the WDS API, which like the WSS, also follows the RESTful approach. The WDS supports storing multiple named capabilities. The WDS treats the names as opaque keys, leaving the naming of capabilities up to the client. As the WDS is private to each user, the client must send an owner capability with each request, similar to restricted calls on the WSS. Clients may list Webit references, as well as add, retrieve, or delete capabilities for a given Webit.

6.4 Summary

The server-side components consist of the WSS, which hosts Webit payloads, and the WDS, which houses Webit references that the user collects. The WSS supports an API to add, delete, modify, and retrieve Webit payloads, as well as operations to create and combine capabilities. An operation involving a Webit requires a capability, which conveys the access permissions. While there are many plausible capability schemes, this thesis explores one that minimizes server storage and supports dynamism. The WDS provides a straightforward API to add new references and manage existing ones.

With the system design in place, the next chapter discusses various approaches to evaluate Webits in the context of developers and end users.

Chapter 7

Evaluation

This chapter presents evaluation approaches and their results. One approach is to examine Clui's flexibility, or the range of usage scenarios that Clui enables. The ease and efficiency of using Webits in web-based tasks, such as the ones described in earlier chapters, is evaluable by inspection. However, using Webits in those scenarios presumes Webit support via plugins, so this chapter begins with an evaluation of the plugins developed and explored.

User evaluation and feedback is the other approach. Clui was designed iteratively, and we conducted two in-laboratory studies. The first is a pilot study on an early prototype of Clui, called *Vapor*, that inspires Clui's current design and focus. We then conducted a qualitative study on Clui. Both studies are discussed after the evaluation on Clui plugins.

7.1 Developing Plugins

We evaluate Clui's flexibility by reporting on our experience developing 1) scraper plugins to create Webits on existing sites, 2) augmenter plugins to parse dropped Webits, and 3) interpreter plugins to generate human-readable descriptions of bundled data. Table 7.1 lists the scraper and augmenter plugins, while Table 7.2 lists interpreter plugins.

One reflection of Clui's flexibility is that certain built-in services, like tooltip

Plugin	Functionality
Scrapers	
facebook.js	Generate people Webits.
amazon.js	Generate product Webits.
craigslist.js	Generate real estate Webits.
acmdl.js	Generate publication Webits.
newegg.js	Generate product Webits.
reddit.js	Generate social bookmark Webits.
aa.js	Generate flight itinerary Webits.
yelp.js	Generate restaurant Webits.
Augmenters	
gmail.js	Handle dropped Webits on To/Cc/Bcc fields.
kayak.js	Auto-fill form.
gmaps.js	Map location Webits.
twitter.js	Massage tweet text and override tooltips.
google.js	Paste Webit label rather than reference in search box.
amazon.js	Paste Webit label rather than reference in search box.
facebook.js	Paste Webit label rather than reference in search box.
google_spreadsheets.js	Create rows with Webit metadata.
google_wallet.js	Add payment information from Webits.

Table 7.1: Scraper and augmenter plugins.

support and automatic Webit rendering, are actually augmenter plugins. They technically operate just as plugins do, in that they may modify the DOM elements of pages to render tooltips and Webits. These core plugins are classified as part of the core module because 1) they are loaded on every open tab, as they implement generic services that apply to any page, and 2) other plugins need to communicate with the core plugins, e.g., to specify tooltip overrides. Table 7.3 enumerates the core plugins.

7.1.1 Common Themes

Asynchrony Many modern sites, like Gmail and Facebook, load resources asynchronously. Fortunately, waiting for these sites to load before scraping is easy to do. A plugin registers a handler for the DOMSubtreeModified event, which fires when the DOM structure changes. The handler then determines if scraping is necessary, in case the scraping process has already run in the past, and if so, whether the

Interpreter Plugin	Example Metadata
publication.js	Title, Authors, Journal, doi
product.js	Vendor, Price, Ratings
realestate.js	Price, Size, Number of Bedrooms
snippet.js	URL, Snippet Text
person.js	Name, Email, Homepage, Phone Number
itinerary.js	Origin, Destination, Dates, Travelers
social_bookmark.js	Comments, Permalink
location.js	Street Address, Area, City, Country
restaurant.js	Name, Location, Ratings, Hours
credit_card.js	Name, Card Number, Expiration, Billing Address
weather.js	Location, Temperature, Forecast, Timestamp
stock.js	Symbol, Company Name, Price, Exchange

Table 7.2: Interpreter plugins, including the example metadata that they parse.

known, desired DOM nodes are present for scraping. As small changes will fire DOMSubtreeModified, debouncing or coalescing those events reduces wasted effort, which is simple to achieve with libraries like underscore.js [16].

Data Interoperability Some plugins enable experimentation with Webit interoperability across sites. For example, scrapers for amazon.com and newegg.com both create product Webits, either of which can be used wherever product Webits are accepted, such as in a search form. Also, Facebook, Craigslist, and the ACM Digital Library plugins create people Webits with bundled Friend-of-a-Friend [6] descriptions, which Gmail interprets when Webits are dropped on To/Cc/Bcc fields.

Interpreter plugins also demonstrate interoperability. Though Webits may originate from different sources and reference other Webits, interpreters can each interpret the parts of a Webit that it understands. For instance, the product interpreter understands Webits that come from different vendors, like amazon.com and newegg.com. Similarly, the real estate, person, and location plugins parse metadata relevant to each in apartment Webits.

Refactoring to Core Services One natural result of developing different plugins is that common tasks become apparent. Such tasks represent opportunities to abstract away boilerplate into library code, thus simplifying and shortening all

Core Plugin	Functionality
show_webits.js	Display Webits on command (Figure 3.1).
security.js	Display warning and advanced sharing dialog; capability rewriting.
search.js	Remove Webit reference when dropped on search input box.
provenance.js	Capture URI of page containing Webit.
tooltip.js	Tooltip rendering.
render.js	Detect and render Webits.
storage.js	Trampoline for proxying storage-related calls.

Table 7.3: Core plugins.

plugin modules.

One main example is rendering Webits on existing pages and attaching the requisite drag event handlers. After instantiating a Webit and rendering it in a div node, to make the Webit draggable, each scraper plugin once needed to also bind a dragstart handler to prepare the appropriate DataTransfer structure. Grasping the necessary concepts to correctly implement that handler presents an unnecessary barrier for plugin developers. Instead, the core plugin render.js centralizes the rendering and drag handling of all detected Webits. It does so using DOMSubtreeModified to scan for Webits and attaches to them the appropriate drag handlers.

7.1.2 Examples

Appendices A, B, and C show examples of a scraper, augmenter, and interpreter plugin, respectively.

The scraper example in Appendix A creates product Webits from Amazon.com product pages. The scraper looks for content on known DOM nodes, and if found, generates a Webit, provisionally stores it, and inserts it on the page below the product title.

The augmenter example in Appendix B enables users to display a map of a location Webit when dropped in Google Maps. As location data may be bundled either directly in the Webit or in a referenced Webit, the augmenter uses getPredicates to scan for location properties directly bundled and also within referenced Webits at specified predicates. While getPredicates returns the property values asyn-

chronously, as it may need to dereference additional Webits, the call also returns a value immediately. That return value is necessary to render tooltips, the text of which must be determined synchronously in the dragover handler. If the desired properties are found immediately or available in a cache, the return value contains the final result. Otherwise, the return value returns a boolean indicating whether the search is still in-progress.

The interpreter in Appendix C parses people Webits and leans heavily on the generate utility function to produce the appropriate data structure for the core. The main job of the interpreter is to specify the important properties and provide a friendly, human-readable name for those properties. The generate utility function deduces the appropriate type based on the matching RDF predicates it finds, though the types may be overridden, e.g., to indicate that a URI points to an image that should be displayed. Property values may reference Webits, which the core or workspace may lazily dereference and interpret, e.g., upon user action.

7.1.3 Limitations

Plugins may communicate with any arbitrary site, e.g., to invoke API calls on web services or to fetch additional content to aid scraping. For example, the ACM Digital Library plugin scrapes bibliographic data by downloading an EndNote file via an AJAX call, as parsing the EndNote format is trivial. Because plugins run in the context of a trusted browser extension, they are not restricted by the same origin policy, which limits the destination of AJAX calls to that of the host serving the current page.

However, due to the restrictions imposed by the Chromium extension framework at the present, plugins may not separately load a web page in an off-screen context for scraping. Doing so might be useful when a plugin needs to load a separate web page that constructs its content dynamically via JavaScript. In such cases, plugins may download static content, e.g., HTML and JavaScript files, but would not be able to execute them to render and access the resulting DOM nodes.

7.2 Preliminary Study

We designed Clui using an iterative process. An initial prototype, called Vapor, supports the drag and drop of only text, links, and images. A pilot user study on Vapor generated feedback on what kinds of interactions users expect when dragging and dropping web resources. The study suggests that Vapor users need handles to rich data types rather than just primitive ones, informing the current design of Clui.

7.2.1 Vapor Prototype and User Study

Vapor displays a drag-and-drop zone like Sheets (Figure 7.1). Vapor creates primitive Webits that capture the resource, e.g., a text snippet, link, or image, along with provenance metadata like the URL of the item, for images and links, and the URL of the page containing the item.

We conducted an informal pilot study, consisting of seven volunteers within our university computer science laboratory, to obtain a general sense of how users may use Vapor in typical workflows. After demonstrating features of Vapor in a brief tutorial, we asked each participant to compose an email with paper abstracts and titles, gathered from non-adjacent text snippets on ACM Digital Library (DL) pages, along with the URI of the relevant DL pages.

7.2.2 User Feedback

We observed that some participants immediately dragged the abstract and title snippets from the page into Vapor, while others habitually relied on using the operating system clipboard, repeatedly switching back and forth between Gmail and the DL page. Ultimately, subjects who initially used the clipboard realized that they could use Vapor to gather all the information first and proceeded to do so.

Every subject successfully completed the tasks without material intervention or help. In their feedback, participants believed that Vapor would work well for their

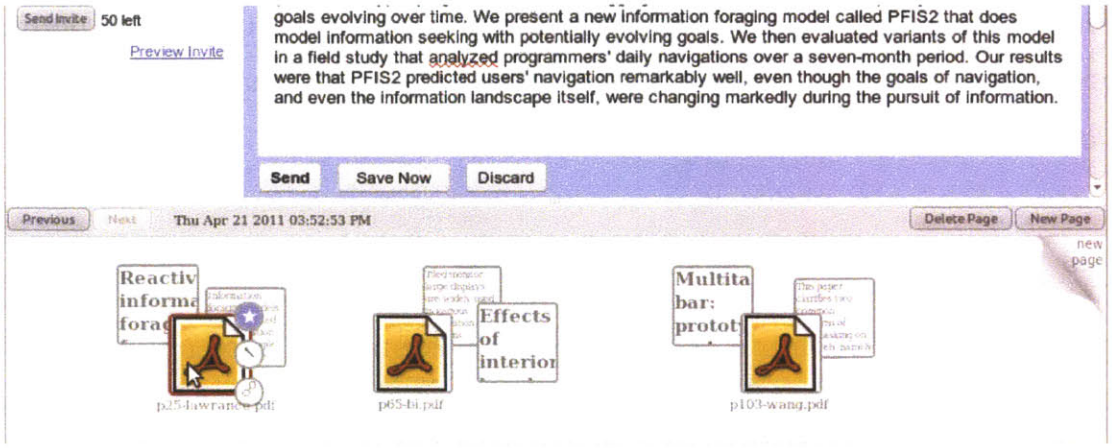


Figure 7.1: Vapor, an early prototype of Clui, features a workspace area like Sheets for primitive Webits.

daily workflows, especially tasks that involved gathering resources first, followed by an aggregation or synthesis process. Participants noted that Vapor alleviated the need to repeatedly context-switch between browser tabs, as a clipboard-based workflow necessitates. Subjects also noted that they especially liked the visible nature of Vapor, and likened it to a powerful cross between a desktop and clipboard.

All subjects noted that the spatial element of Vapor was important, and often clustered related Webits into groups, reinforcing earlier findings [23, 62]. Many participants noted that they enjoyed the "freedom" that Vapor affords, especially for collecting and "quickly organizing" information scraps spatially throughout a task. Subjects approved of the fact that Vapor was part of the browser, instead of being in the area behind the browser like the traditional desktop, with some citing quicker access and constant visibility. However, subjects did resize the visible area of Vapor to make it larger or smaller. Some suggested that they might keep a separate browser window open, dedicated to displaying Vapor full-screen, while others preferred Vapor to be more "integrated" with and customized to the current, active tab.

Vapor often needs to upload resources to the web, e.g., when the user drags an image to a Google Docs document, which introduces variable network delays into user operations. We observed that users expect drag and drop operations to

be instant. Without adequate feedback, some users were confused when nothing appeared to happen immediately. This observation suggests that general visibility is an important requirement to address.

7.2.3 Design of Clui

Vapor's primitive Webits do not capture extensible metadata, so unlike the Clui reading group scenario described in Chapter 3, Vapor users must manually drag individual snippets of text. That observation suggests that users actually need more than just an easier way to collect web snippets. Users also need a way to access and transfer objects with all of the associated metadata bundled, which inspires the current design of Webits as handles for rich, semantic objects.

7.3 User Study

Clui is the result of applying our initial observations from Vapor, followed by several iterations of development. To help evaluate Clui, we conducted an in-laboratory qualitative user study with 10 subjects, where we observed usage and solicited feedback. Specifically, the study design aims to help determine whether users generally understand Webits, whether they find Webits useful and delightful to use, aspects of the system that work well, and opportunities for improvement. This section discusses the design of the study, the solicitation for and demographics of participants, and our resulting observations.

7.3.1 Study Design

Each session in the study consists of an informal introductory interview, a tutorial of Clui, an involved exercise revolving around apartment hunting, a comparatively short exercise concerning a Webit with sensitive information, and finally a debriefing interview. Participants are encouraged to talk aloud, as they carry out tasks associated with the tutorial and exercises.

Introductory Interview We inform participants that they are helping a research team evaluate a new user interface for web-based tasks. To prime subjects and also discover current approaches, we invite subjects to role play and describe how they would carry out apartment hunting related tasks using their current tools. After asking subjects to share the ranking of important factors when considering a potential apartment, we ask subjects how they would:

- find potential apartments in the area, and the general workflow surrounding that search;
- organize and later retrieve descriptions of those properties;
- determine the physical locations of those properties;
- establish contact with the associated landlord or realtor; and
- find potential roommates, assuming the factors important to a given participant conspire to make splitting costs overwhelmingly attractive.

Tutorial We explain the general concept of Webits, focusing on their bundling property, by demonstrating two simple workflows. After each, we invite the subject to mimic the demonstration and ask questions. In the first workflow, subjects observe a product Webit on an amazon.com page describing a laptop for sale. We demonstrate dragging and dropping the Webit on 1) the workspace, 2) the Google Shopping search engine to find alternative vendors with better prices, and 3) a Google Spreadsheets document, where the Webit's bundled data is expanded into a new row. The second workflow demonstrates a Webit that represents a local restaurant that appears on a yelp.com page. We explain and show that because the restaurant Webit represents a real-world location, dropping that Webit on Google Maps displays a map that pinpoints the restaurant.

Apartment Exercise Once the subject is comfortable with the tutorial, he or she begins the apartment hunting exercise, which mirrors the tasks discussed in the introductory interview. We present a live craigslist.org page with apartment listings

and explain that each listing page will display a Webit representing that property. We ask the subject to identify 3–5 appealing properties and suggest that the subject drag those Webits to the workspace for safekeeping.

Once the subject is satisfied with his or her selection of potential properties, we explain the next tasks, which include organizing and sorting the apartments by cost, selecting the best option, obtaining a map of that apartment, finding roommates, and contacting the landlord. We prompt the subject to use Google Spreadsheets to organize Webits, and we collaborate with the subject to select the most appealing option. We ask the subject to display a map of that property. We propose using Twitter to solicit roommates and ask the subject to convey the apartment details to the followers. To contact the landlord, we ask the subject to compose a message in Gmail by 1) specifying the To field, and 2) conveying the desired apartment in the message body.

While we generally guide subjects from task to task, i.e. proposing which service to use based on the available plugins, we strive to prompt subjects to solve high-level goals rather than deliver specific drag-this-to-there instructions.

Sensitive Webit Exercise We ask the subject to use a Webit representing an authentic-looking credit card to provision a Google Wallet account, which serves as an intermediary payment processor. To do so, we role-play a scenario in which we partner with the subject to start a business. The subject's role is to procure business supplies. We explain that using an intermediary service like Google Wallet or PayPal carries both a security and ease advantage, in that users only need to provision those sites with financial information once, rather than with every vendor, leaving vendors to transact with the intermediary.

We provide a credit card Webit on the workspace and show the values bundled within, including the credit card number, expiration, and so on. The subject is asked to provision a Google Wallet account using the Webit. When a credit card Webit is dropped on the Google Wallet account page, a plugin requests access to the sensitive information in the Webit, causing Clui to prompt the user with a

warning dialog box. We silently observe as the subject interacts with the dialog, and immediately afterward, we ask the subject to describe his or her thought process.

Debriefing Interview To debrief, we ask the subject to share:

- Aspects about the Webit experience that the subject liked or found enjoyable;
- Aspects the subject disliked or thought could be improved;
- Other scenarios or tasks in the subject's daily work that Webits might enhance; and
- Overall impressions or final thoughts.

7.3.2 Participants

We solicited for participants using a Craigslist job advertisement to reach a broad user population from the Boston area. To register interest in the study, potential subjects first completed an online form linked from the posting. The form asks subjects to 1) select all browsers, from a list of popular ones, with which they have used, 2) indicate the browser they currently use the most, 3) check all web applications with which they are familiar, and 4) optionally indicate their age bracket. The form serves as an eligibility test to help ensure participants have basic facility using the web.

We invited 10 subjects, 7 female, to the laboratory. Subject ages ranged from 18 to 65. To ensure participant familiarity with the web applications relevant to the study exercises, we selected subjects that reported experience with using Google Chrome as their primary browser, Google Spreadsheets, Google Maps, Twitter, Amazon.com, and Gmail. To minimize distractions associated with using a foreign computing environment, we accommodated each subject's preferred operating system platform. Subjects were compensated with $25 at the conclusion of the session. All sessions completed in less than 1 hour.

Prior to the 10 sessions, we conducted 4 pilot sessions to refine the study design and procedures. Three subjects in the pilot are laboratory colleagues, while the fourth was drawn from one of the Craigslist responders.

7.3.3 Observations and Results

Introductory Interview In asking subjects to describe their process for finding potential apartments using conventional tools, several common themes arose:

- All subjects reported that they would use browser tabs extensively to explore options. To organize or keep track of potential options, most reported that they would use bookmarks. A few said they would email links to themselves. Two subjects indicated they would print hard copies of the listings, handy for scribbling notes.

- All subjects indicated that they would use Google Maps to find the apartment, but one mentioned she would defer to her in-car GPS navigation device for directions.

- Under the assumption that having roommates would be desirable, subjects reported varied approaches to finding roommates. In general, many said they would leverage their social network, e.g., by asking friends to broadcast by word of mouth. Some expressed comfort with using web-based services, like Facebook. Most expressed hesitation in rooming with strangers, so using Craigslist to find roommates was generally undesirable. However, one subject indicated comfort with posting an ad on a physical bulletin board at a local church.

- To contact a landlord or realtor, many indicated a strong preference for either calling or for sending email.

Somewhat surprisingly, nearly all subjects could personally relate to the task of finding apartments. For one subject, the task was less relevant, though that person

indicated that she was a professional landlord for several properties. As such, she expressed no hesitation role-playing as a potential tenant.

Tutorial After explaining that a Webit represents an object and bundles information pertaining to that object, we demonstrated dragging a product Webit from Amazon.com to Google Shopping and then to Google Spreadsheets, to help compare features with competing options. When they witnessed a new spreadsheet row appear with the bundled data in response to dropping a Webit on Google Spreadsheets, some simply acknowledged the action, while others expressed surprise and disbelief, e.g., *"What? Wow—that's amazing!"*, *"Nice, Nice!"*. A few expressed similar sentiments when they saw a map of a restaurant in response to dropping the associated Webit on Google Maps. No subject encountered trouble mimicking the tutorial demonstrations.

Apartment Exercise All subjects selected 3–5 apartment Webits and dragged those to the workspace, Google Spreadsheets, and Google Maps without incident. This is unsurprising, as subjects witnessed similar usage in the tutorial.

When we asked subjects to use Twitter to broadcast 1) their desire for a roommate and 2) the apartment under hypothetical consideration, nearly all users instinctively dragged the apartment Webit to Twitter's tweet composition text input box. Only one user manually typed in a description of the apartment from memory, but after additional prompting, realized that dragging the Webit would be faster and did so. When witnessing a long Webit reference link pasted in the tweet composition box, a few users expressed the concern that the link consumed too many characters of Twitter's 140 character allowance. However, they were convinced that the system was fine when we pointed out Twitter's counter that displays the number of remaining characters, which is computed assuming links are automatically shortened by Twitter's link shortener. One user expressed a different concern, that a long link would be far from ideal when presented to followers. Users generally expressed pleasant surprise when the posted tweet contained a

Webit icon rather than the pasted link.

We asked subjects to change perspectives and pretend to be a friend that is interested in learning more about the apartment just tweeted. All users clicked on the Webit, hoping to view the associated Craigslist apartment page. Unfortunately, at that time, a single click yielded no effect, while a double click achieved the desired effect. When instructed to double click, all users confirmed that the resulting behavior was what was originally expected. We asked subjects how they would find a map of the apartment from the tweet, and all indicated that they would drag the Webit to Google Maps.

Before asking subjects to contact the landlord using Gmail, we told them that the apartment Webit bundled the landlord's contact information. We asked subjects to 1) address a new message to the landlord, and 2) indicate in the message body the apartment under consideration. All users dragged the Webit to the message body without hesitation. All but two users dragged the Webit to the message To: field. Of those, one searched from the email address in the associated row on the Google Spreadsheet tab and used the clipboard to copy and paste the address, while the other required additional prompting. Two dragged the Webit to Subject field and expressed satisfaction when the system pasted a short title for the apartment.

Sensitive Webit Exercise When asked to provision a Google Wallet account with a new credit card, all subjects dragged the provided credit card Webit to Google Wallet. In the pilot sessions, the system immediately displays a complex warning dialog box after dropping the Webit. Pilot participants found this behavior to be disconcerting due to 1) the surprise associated with a sudden dialog box, and 2) the complexity of that box, which offers access controls that are too fine-grain. In the formal study, upon dropping a sensitive Webit, a page that requires access must first inform the user that it needs additional access and provide a button to grant or deny that access. Study participants express little surprise when clicking the button resulted in the display of the warning dialog box.

As mentioned, we silently observed as participants engaged with the website and dialog box. All users successfully granted access, but a two expressed hesitation due to the fear that they were about to release real credit card information that was not theirs.

We asked participants why they thought Google Wallet forced them to click a button after dropping the credit card Webit, when contrasted with Google Maps, which takes action immediately. All mentioned that the financial or sensitive data triggered the extra steps, and most mentioned the value of requiring a confirmation to prevent accidental mistakes or provide the chance to reconsider. One participant said that she was "glad they do that", while another wanted the option to suppress the extra steps for advanced users.

Upon seeing the warning dialog, five users assumed that the dialog box was boilerplate that they had seen many times before, and they granted access immediately. Two users paused and then granted access. Three noticed and appreciated that the dialog displayed the Webit icon and the specific information to be shared. Those three expressed additional confidence that the system was operating on the appropriate card.

Enjoyable Aspects Subjects overwhelmingly appreciate the general simplicity of Webits and the ability to drag and drop Webits. In their own words:

> *"So simple, a lot better than how I work."*
>
> *"I like the simplicity of it, that it takes all the information without me having to type anything. I just drag and drop. That I liked very much."*
>
> *"It seems very user friendly—doesn't take much to train yourself in using it."*
>
> *"Not too much information thrown at you—which is good."*
>
> *"One single drop and that's it ... that's what people would like and that's what it's about."*

Subjects also enjoy the visual nature of Webits and the workspace. Many indicated that they found Webit icons appealing, e.g., *"I like the icon, I'm a visual*

person". One mentioned that while she is generally unlikely to click on links in a Twitter tweet post, an iconic representation would be "more intriguing". Regarding the workspace, many commented that having a separate space from browser tabs to be strongly desirable. They contrasted the Webit workspace against conventional bookmarks and browser tabs, preferring the workspace because it provides a context for holding information outside of the browser, affords freedom to organize objects spatially, and makes more information visible at once. One subject admitted to not knowing how to delete conventional bookmarks and believed that deleting Webits would be easier. Another subject believed that as bookmarks, Webits are more lightweight than the alternative of keeping browser tabs open indefinitely, lamenting that having too many tabs open slows her computer.

While we did not emphasize the feature of multiple workspace sheets, astute subjects indicated that they would find assigning different sheets for different tasks useful. One subject thought that multiple sheets would solve her problem of desktop clutter.

Finally, subjects all agreed that the automatic data bundling and ease of data transfer is useful. Some mentioned that using Webits would help prevent typos associated with manual data entry between sites. Subjects were generally enthusiastic with the ability to drop Webits into Google Spreadsheets and email messages:

> *"The spreadsheet and email is particularly fantastic."*
>
> *"I like that it works with the spreadsheet ... You just drag it in and it's already making your life more organized."*

Opportunities for Improvement In general, subjects desired the ability to customize or select bundled data, better visibility, and greater compatibility with more sites. Regarding customization, subjects appreciate that the system extracts a predetermined set of data. However, they desired the ability to teach the system what data to add or remove, based on the task and context. One subject indicated that removing data is important in order to avoid clutter in spreadsheets, when used

as a destination for holding Webits. On visibility, subjects wanted the ability to inspect the bundled content in Webits, a feature that Clui already provides, though users did not discover it. One subject wished to know where the Webits are stored, particularly ones with sensitive data. That subject also suggested the need for password-protection on sensitive Webits. Many subjects expressed concern on widespread support for Webits, though they believe that Webits would still be useful today even if wider support improved over time. A few suggested Webit support on mobile devices and tablets.

Two subjects wished for greater automation. One suggested that the system should infer his needs based on his social media activity or personal information management tools, e.g., a to-do list, and automatically find and present appropriate Webits for him to consider, e.g., pre-organized in a spreadsheet. Additionally, he wished for the ability to query the system using spoken natural language and manage the results using Webits. Another subject wished to link the contents on a workspace sheet with the contents on a spreadsheet, so that changes in one are reflected in the other.

Other Use Cases Subjects suggested a variety of use cases for Webits, the most popular of which is that of a bookmark replacement. Some subjects believed that with Webits, they would stop emailing links to themselves. Many mentioned coupling Webits with spreadsheets to keep, organize, and annotate important or useful web resources. One suggested that Webits would be useful for general project management. Subjects generally discussed a variety of domains that were important to them.

Many subjects also suggested using Webits to plan travel and for general shopping. Subjects thought that it would helpful to use a spreadsheet to comparison shop different products, job postings, travel activities or itineraries, and so on. Similar to using a recipe Webit to automatically fill a shopping basket with ingredients at an online grocer, one mentioned an analogous scenario with craft products, along with gardening.

Others proposed using Webits to better communicate and share. For example, one subject expressed interest in using Webits to compose blog posts by dropping image and video resources. Others perceived more value in sending Webits, as opposed to links, to friends using email or social networks.

Overall Impressions At the session conclusion, when asked about their overall impressions, subjects expressed enthusiasm:

> *"It's a good thing. It's great. I like it. It makes things so much easier. I think they're on to something."*
>
> *"It's pretty slick."*
>
> *"This is very easy."*
>
> *"If you need a beta tester at some point, I would be interested!"*
>
> *"I think it's really great...I'm excited...Hurry up!"*
>
> *"When's it going to be available? Because I would use it!"*
>
> *"I feel like this is something that is intuitive enough and really be useful for my mom—she can get the drag and drop."*
>
> *"I want it."*

Study Limitations Several factors conspire to temper the results above. One is the novelty effect, namely that subjects with general interest in new technology may be predisposed to responding enthusiastically. As future work, a longitudinal study would better measure user impressions once the novelty wore away. Another factor is politeness, as subjects may feel social pressure to please the investigator. We attempt to minimize this pressure by 1) telling subjects that they would be most helpful when honest, and 2) role-playing as an investigator tasked with producing a report based on user feedback rather than explicitly revealing our role as Clui's system designer.

7.4 Summary

This chapter presents evaluation for Clui and Webits. It evaluates Clui's flexibility and applicability to web applications by reporting on our experience developing plugins. To help evaluate the hypothesis that Clui enhances user efficiency and delight, this chapter presents observations from two laboratory studies. Despite the limitations inherent in the studies, feedback from subjects suggests that users would enjoy using a system like Clui and would find Webits useful in their daily web activities.

Chapter 8

Conclusion

This thesis explores user interface handles to rich objects on the web, inspired by the desktop's successful use of visual handles to represent user-meaningful objects. As the web expands the universe of things users care about, this thesis expands the role of handles to represent an open-ended set of objects. In doing so, it aims to improve interface consistency and data interoperability across the web, along with enhancing user efficiency and delight. Contributions include Webits, which are the handles, and Clui, which is the system design that paves the way towards widespread Webit support and usage.

Webits carry several user interface features. Users drag and drop Webits between sites to transfer information, or they may drop Webits on a workspace area to keep important objects handy. Webit drop behavior depends on the drop target. While guidelines for default behaviors aid predictability, the drop behavior may be customized. To enhance discoverability and visibility, the interface displays tooltips as the user drags a Webit to communicate the resulting drop behavior at the current target.

Several principles describe the design and behavior of Webits. A Webit bundles a machine-readable, semantic description of the object it represents. It carries a notion of type and identity so that users and systems can determine if two Webits are the same. A Webit may represent dynamic data, like the current weather. Users may share Webits, while restricting what others may see or change.

One challenge is interoperability, as Webits depend on the ability to bundle object semantics in a manner that is interpretable by a wide range of non-cooperating sites. While semantic web efforts strive to solve that problem, current usage and demand remain limited to niche domains and applications. To attain traction, Webits must demonstrate value and generate user demand without needing to rely on the cooperation of site operators to invest development effort.

Clui provides an API and system support for Webits, which notably includes a plugin system that transparently adds Webit support to existing web pages. Plugins scrape data on sites and generate Webits; augment existing sites with the ability to interpret dropped Webits and take useful actions; and interpret Webits, translating the bundled machine-readable descriptions to user-friendly ones. With Clui, Webits may deliver value today, potentially generate demand for wider adoption, and someday incentivize site developers to add Webit support directly.

To evaluate Clui's flexibility, we report on our experiences developing a variety of plugins. To evaluate Webits, we conducted in-laboratory user studies and collected qualitative observations and feedback. Participants expressed general enthusiasm for the system and its simplicity, believed that the system would be useful to them, and indicated that they want to use it.

8.1 Future Directions

This work originated from a general search for new user interface models to support cloud-based workflows. Sensing the diminishing role of the current desktop, we originally set out to design an improved desktop environment that reflects and meets the needs of users in a web-centric environment. Observations on current desktop usage, e.g., by Katifori et al. [50], bolstered by the undeniable success of the desktop, biased our attention towards extending the desktop metaphor rather than exploring a pure browser-only environment. In a future cloud-aware desktop, Webits represent one proposal for the primitives that users manipulate.

One avenue for future work is to resume the investigation of workspace envi-

ronments and metaphors that better fit web-based workflows. This thesis presents Sheets, which represents an early exploration. Sheets replaces a single desktop surface with an infinite roll of paper sheets, akin to an infinite notebook of disposable pages. In web workflows, data tend to ultimately live on servers, so Sheets aims to provide a surface for temporary Webits and eschews organization mechanisms like folders. Other workspace approaches are worth exploring, such as ones that provide users with tools to create, merge, and customize Webits.

In contrast to web sites, the workspace may serve as a neutral area not directly tied to any one provider. One resulting benefit is that it serves as a natural space for exploring standardized interfaces that abstract differences across sites. For example, many commerce sites feature shopping carts, each unique to the given site. Users purchasing a variety of items across different vendors would have to use the carts associated with each vendor. Instead, the workspace could offer a generic shopping cart that accepts product Webits from any vendor, provide automatic suggestions for alternative vendors with lower prices, and assist with checkout by opening browser tabs and pre-filling the appropriate vendors' carts. Users finalize checkout on vendor-specific sites and thus may configure vendor-specific features, like gift wrapping. Users benefit from a unified interface for common web tasks, while sites retain the ability to customize parts of the experience.

Perhaps standardizing interfaces for common web workflows would improve sensemaking activities by centralizing disparate information and adding domain-specific support. The workspace could provide a library of interface templates designed for different kinds of workflows that users instantiate. Templates provide a standardized, vendor-neutral user interface for a specific task, accept Webits as inputs, and provide actions. For example, like the shopping cart scenario above, a template could help users build a travel itinerary by exploring different air and hotel options (Figure 8.1). Coming full circle, using Webits as handles for template instances would enable users to easily collaborate by sharing Webits—further expanding the role of Webits to include workflows and processes.

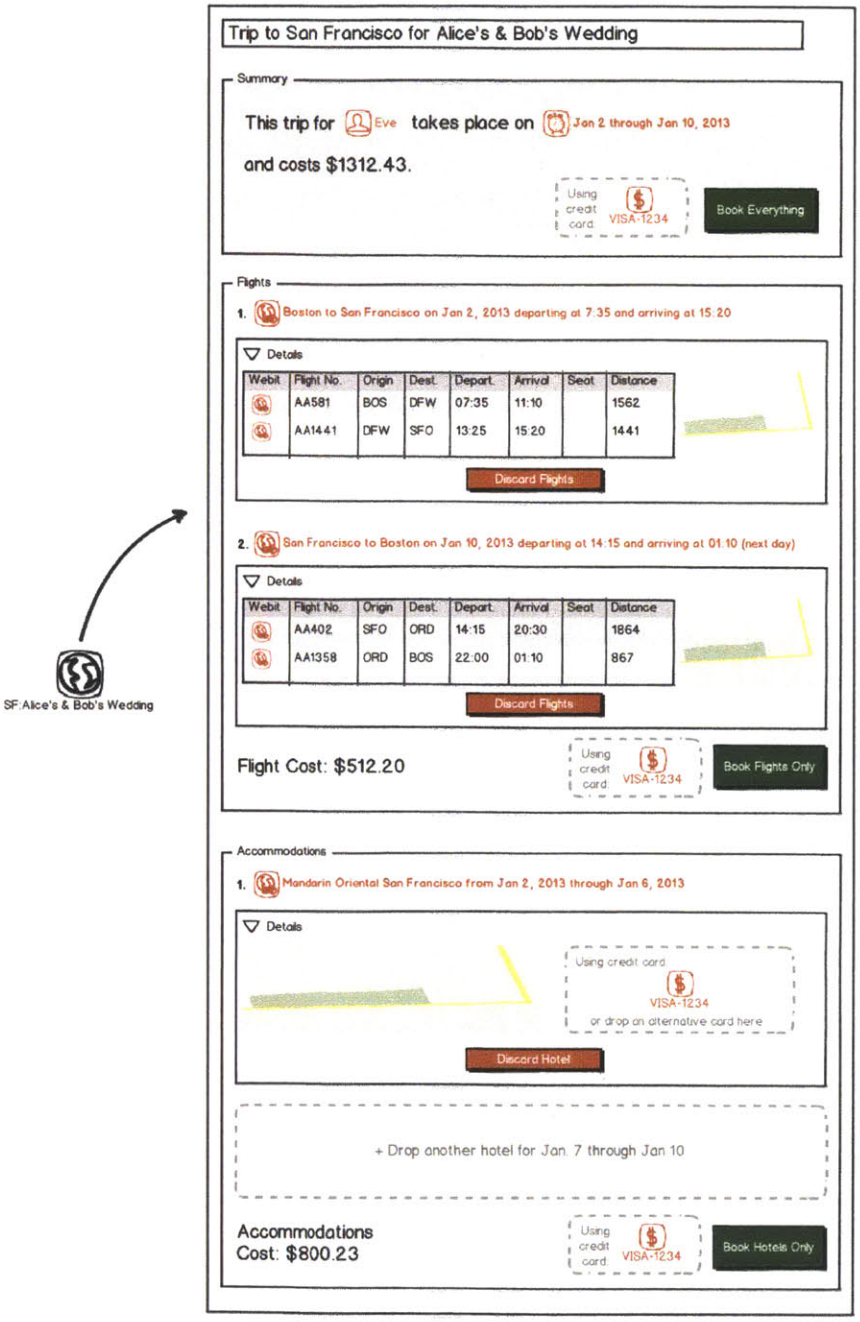


Figure 8.1: A mock-up of an example workspace template that helps users plan travel.

8.2 Concluding Remarks

Graphical user interfaces and the desktop metaphor revolutionized the personal computer, replacing text-based, command-line interfaces with ones that are more broadly accessible and user-friendly. Similarly, the web browser, as the dominant interface for the Internet, makes networked systems even more useful and is displacing traditional applications on the desktop. Meanwhile, smart phones and tablets today are disrupting the dominance of desktop computers altogether. While browsers still serve an important role on mobile devices, phones have repopularized native applications, which feature new interfaces and gestures, in pursuit of providing a richer user experience on small touch-screens. Commercial, mass-produced wearable computing devices are around the corner, and more exotic platforms are no doubt on the horizon awaiting their debut, along with the new interfaces that will power them and enamor users.

In tracing the arc of interface evolution, as new platforms displace old ones, there is a strong tendency to construct new interfaces that better address the features and constraints of emerging platforms. The development of specialized interfaces undeniably advances the user experience, but it is also important to seek and seize opportunities to design interfaces that transcend specific platforms and devices. A few principles and approaches are so successful that they are ubiquitous and timeless, like the use of pictorial icons to convey meaning, buttons as affordances for actions, tooltips to offer lightweight help, and forms to solicit structured input. The handles presented in this thesis are inspired by the success of icons on the desktop, as well as our intuition that humans naturally draw analogy between real objects and the symbolic representations used to represent those objects. That human ability makes us hopeful that the value of user interface handles to rich objects extends beyond the environment on which those handles were developed.

Appendix A

Scraper Plugin Example

```
/**
 * Amazon scraper plugin: content script that scrapes a product page
 * and creates a Webit for that product.
 */

/*global requirejs */

requirejs(
  ["jquery", "underscore", "lib/sha1", "core/webit", "plugins/clui_api"],
  function ($, _, Sha1, Webit, clui) {
    "use strict";

    var console = window.console,
        RATINGS_REGEXP = new RegExp("s_star_(\\d)_(\\d)"),
        FAVICON = "http://www.amazon.com/favicon.ico";

    function scrape() {
      // Does a Webit for this product currently exist?  If so, don't
      // bother scraping.
      if ($("[data-webitized]").length) {
        return;
      }

      // Look for the main product image.  Amazon uses at least two
      // different page formats, hence the scraping complexity.
      var price = $(".priceLarge").text() || $("#price .a-size-large").text(),
          titleDiv = $("#title").add("#btAsinTitle"),
          title = titleDiv.text(),
          productImgUrl = $("#main-image").attr("src"),
          pageUrl = document.location.href,
          descriptionHtml = ($("#productDescription").html() || "").trim(),
          descriptionText = ($("#productDescription").text() || "").trim(),
          reviewsUrl = $("#averageCustomerReviews_feature_div a")
                         .add(".jumpBar a")
                         .last()
                         .attr("href"),
          ratingSpanClasses = $(
            ".jumpBar .swSprite"
          ).add(
            "#averageCustomerReviews_feature_div .swSprite"
          ).attr("class") || "",

          rating = _.reduce(
            ratingSpanClasses.split(" "),
```

```
45              function (memo, cssClass) {
46                if (memo) {
47                  return memo;
48                }
49
50                var match = RATINGS_REGEXP.exec(cssClass);
51
52                if (match !== null) {
53                  var main = match[1],
54                      decimal = match[2];
55
56                  return parseFloat(main + "." + decimal);
57                }
58
59                return null;
60              },
61              null
62            );
63
64        if (!title || !productImgUrl) {
65          return;
66        }
67
68        // Create and insert Webit
69
70        var webit = Webit.create({
71          id: Sha1.hash(pageUrl),
72          ux: {
73            open: pageUrl,
74            icon: {
75              mainImage: productImgUrl,
76              label: title,
77              typeImage: FAVICON,
78              text: title
79            }
80          },
81          content: {
82            "http://www.w3.org/1999/02/22-rdf-syntax-ns#type": {
83              type: "literal",
84              value: "http://purl.org/goodrelations/v1#Offering"
85            },
86            "http://purl.org/goodrelations/v1#name": {
87              type: "literal", value: title
88            },
89            "http://purl.org/goodrelations/v1#hasCurrencyValue": {
90              type: "literal", value: price
91            },
92            "http://schema.org/image": {
93              type: "uri", value: productImgUrl
94            },
95            "http://schema.org/url": {
96              type: "uri", value: pageUrl
97            },
98            "http://purl.org/goodrelations/v1#BusinessEntity": {
99              type: "uri", value: "Amazon.com"
100           },
101           "http://schema.org/ratingValue": {
102             type: "literal", value: rating
103           },
104           "http://schema.org/review": {
105             type: "bnode", value: {
106               "http://schema.org/url": {
107                 type: "url", value: reviewsUrl
108               },
109               "http://www.w3.org/1999/02/22-rdf-syntax-ns#type": {
110                 type: "literal", value: "http://schema.org/Review"
111               }
112             }
```

```
113             },
114             "http://purl.org/goodrelations/v1#description": {
115               type: "literal", value: descriptionText
116             },
117             "http://clui.co/relations/v1#descriptionRich": {
118               type: "literal", value: descriptionHtml
119             }
120           }
121         });
122
123         // Provisionally host Webit
124         clui.phost(
125           webit,
126           function (err, webit) {
127             // Add the Webit icon below the product title:
128             var webitDiv = $(webit.render()).css("margin", "15px");
129             titleDiv.parent().append(webitDiv);
130           }
131         );
132       }
133
134       // After the page loads, attempt to scrape:
135       $(function () {
136         scrape();
137
138         document.addEventListener(
139           "DOMSubtreeModified",
140           _.debounce(scrape, 250),
141           false // bubble phase
142         );
143       });
144     }
145   );
```

Appendix B

Augmenter Plugin Example

```
/**
 * Google Maps plugin that maps location Webits.
 */

/*global requirejs */

requirejs(
  ["underscore", "plugins/clui_api"],
  function (_, clui) {
    var SO = "http://schema.org/PostalAddress",
        VOCAB = "http://vocab.org/places/schema.html",
        done = false,
        cache = {},
        getRdf = clui.utils.getRdf;

    /**
     * Parse RDF predicates and extract location information.
     *
     * @param {Object} predicates an object that maps detected
     *  location predicates to values.
     */
    function parsePreds(predicates) {
      var location = "",
          address = predicates[SO + "#streetAddress"],
          area = predicates[VOCAB + "#District"],
          city = predicates[SO + "#addressLocality"],
          state = predicates[SO + "#addressRegion"],
          country = predicates[SO + "#addressCountry"];

      if (address) {
        location += address;
      }

      // If an address exists, just use the address and city:
      if (!address && area) {
        if (location) {
          location = location + ", ";
        }
        location += area;
      }

      if (city) {
        if (location) {
          location = location + ", ";
```

```
45          }
46          location = location + city;
47        }
48
49        if (state) {
50          if (location) {
51            location = location + ", ";
52          }
53          location = location + state;
54        }
55
56        if (country) {
57          if (location) {
58            location = location + ", ";
59          }
60          location = location + country;
61        }
62
63        return location;
64      }
65
66      /**
67       * Enable elements to accept Webits for mapping.
68       *
69       * @param {DOM} element element to add drop and dragover listeners
70       * @param {DOM} input the search input box
71       * @param {DOM} go the search button (to invoke searches)
72       */
73      function enableDrop(element, input, go) {
74        element.addEventListener(
75          "drop",
76          function(event) {
77            var dt = event.dataTransfer,
78                rdf = getRdf(dt);
79
80            if (!rdf) {
81              return;
82            }
83
84            var locationPending = clui.utils.getPredicates(
85              rdf,
86              [SO + "#streetAddress", SO + "#addressLocality",
87               SO + "#addressRegion", SO + "#addressCountry",  VOCAB + "#District"],
88              ["http://schema.org/location", "http://schema.org/address"],
89              cache,
90              function donePredicateFetch(record) {
91                if (!record) {
92                  return;
93                }
94
95                var location = parsePreds(record);
96                input.value = location;
97
98                // Automatically click the Search button:
99                go.click();
100             }
101           );
102
103           if (locationPending) {
104             if (event.preventDefault) {
105               event.preventDefault();
106             }
107
108             if (event.stopPropagation) {
109               event.stopPropagation();
110             }
111           }
112         }
```

```
113         );
114
115         element.addEventListener(
116           "dragover",
117           function(event) {
118             var dt = event.dataTransfer,
119                 rdf = getRdf(dt);
120
121             if (!rdf) {
122               return;
123             }
124
125             var locationRecord = clui.utils.getPredicates(
126               rdf,
127               [SO + "#streetAddress", SO + "#addressLocality",
128                SO + "#addressRegion", SO + "#addressCountry", VOCAB + "#District"],
129               ["http://schema.org/location", "http://schema.org/address"],
130               cache
131             );
132
133             // locationRecord will either have:
134             //  1) the location string if location data is directly
135             //     embedded in the RDF
136             //  2) true, if a referenced Webit has location data
137             //  3) falsy, otherwise
138
139             if (!locationRecord) {
140               return;
141             }
142
143             event.preventDefault();
144             clui.setTooltipText(
145               event, "Show map for",
146               (locationRecord === true) ? "location" : parsePreds(locationRecord)
147             );
148           }
149         );
150       }
151
152       /**
153        * Enable certain areas of the map (search box and map) to
154        * accept Webits.
155        */
156       function elaborate() {
157         var input = document.getElementById("gbqfq"),
158             mainMap = document.getElementById("main_map"),
159             go = document.getElementById("gbqfb");
160
161         if (!input) {
162           return;
163         }
164
165         enableDrop(input, input, go);
166         enableDrop(mainMap, input, go);
167         done = true;
168       }
169
170       document.addEventListener(
171         "DOMSubtreeModified",
172         _.debounce(
173           function () {
174             if (!done) {
175               elaborate();
176             }
177           }, 250)
178       );
179     }
180   );
```

Appendix C

Interpreter Plugin Example

```
/**
 * Interpreter Plugin for people Webits
 */

/*global define */

define(function (require) {
  "use strict";

  var interpUtils = require("./utils"),
      FOAF = "http://xmlns.com/foaf/0.1/",
      name = "Person";

  return {
    name: name,

    /**
     * Interpret a Webit's bundled metadata.
     *
     * @param {Webit} webit The webit
     * @param {Function} cb Called with interpreted data:
     *  { subject : <record> }
     *
     *  where <record> is:
     *
     *  {
     *    data: [{
     *      dataTransfer: [[mime, value], ...],
     *      label: display label,
     *      value: display value
     *    }, ... ],
     *
     *    satisfaction: <a self-accessed score between 0 and 1.0 on
     *      how appropriate this plugin is for the the given Webit>,
     *
     *    name: <name of this group of data>
     *  }
     *
     *  or called cb(false) if data is missing or uninterpretable.
     */
    interpret: function (webit, cb) {
      interpUtils.generate(
        name,
        webit,
```

```
45          [
46            { pred: FOAF + "name", label: "Name" },
47            { pred: FOAF + "mbox", label: "Email" },
48            { pred: FOAF + "phone", label: "Phone" },
49            { pred: FOAF + "homepage", label: "Homepage" },
50            // The 'type' property below tells generate() to interpret
51            // the URI value as a reference to an image, so that the image
52            // is displayed instead of a URI.
53            { pred: FOAF + "depiction", label: "Picture", type: "image" },
54            { pred: FOAF + "based_near", label: "Near" }
55          ],
56          cb
57        );
58      }
59    };
60  });
```

Bibliography

[1] Chromium - the chromium projects. http://www.chromium.org/Home.

[2] Chromium OS. http://www.chromium.org/chromium-os.

[3] Data.gov. http://www.data.gov/.

[4] The datahub. http://datahub.io/.

[5] Evernote. http://www.evernote.com/.

[6] The friend of a friend project. http://www.foaf-project.org/.

[7] Greasemonkey. https://addons.mozilla.org/addon/greasemonkey/.

[8] KeyKOS home page. http://www.cis.upenn.edu/~KeyKOS/.

[9] Live clipboard - wiring the web. http://liveclipboard.org/.

[10] Microsoft OneNote 2010. http://office.microsoft.com/en-us/onenote/.

[11] node.js. http://nodejs.org/.

[12] OAuth 2.0. http://oauth.net/2/.

[13] Oracle spatial and graph - RDF semantic graph. http://www.oracle.com/technetwork/database-options/spatialandgraph/overview/rdfsemantic-graph-1902016.html.

[14] Redis. http://redis.io/.

[15] Semantic web - W3C. http://www.w3.org/standards/semanticweb/.

[16] Underscore.js. http://underscorejs.org/.

[17] Yahoo pipes: Rewire the web. http://pipes.yahoo.com/pipes/.

[18] Zotero. http://www.zotero.org/.

[19] Eytan Adar, David Karger, and Lynn Andrea Stein. Haystack: per-user information environments. In *ACM CIKM*, 1999.

[20] Ben Adida, Ivan Herman, Manu Sporny, and Mark Birbeck. RDFa 1.1 primer. http://www.w3.org/TR/xhtml-rdfa-primer/.

[21] D. Balfanz and D. R. Simon. WindowBox: a simple security model for the connected desktop. In *USENIX Windows Systems Symposium*, 2000.

[22] G. Barish and K. Obraczke. World wide web caching: trends and techniques. *IEEE Communications Magazine*, 2000.

[23] Deborah Barreau and Bonnie A Nardi. Finding and reminding: file organization from the desktop. *ACM SIGCHI Bulletin*, July 1995.

[24] Robin Berjon, Travis Leithead, Erika Doyle Navara, Edward O'Connor, and Silvia Pfeiffer. HTML5 drag and drop. http://www.w3.org/TR/html5/dnd.html.

[25] Tim Berners-Lee and Dan Connolly. Notation3 (n3): A readable RDF syntax. http://www.w3.org/TeamSubmission/n3/.

[26] Michael Bernstein, Max Van Kleek, David Karger, and mc schraefel. Information scraps: How and why information eludes our personal information management tools. *ACM TOIS*, 2008.

[27] Michael Bolin, Matthew Webber, Philip Rha, Tom Wilson, and Robert C. Miller. Automation and customization of rendered web pages. In *ACM UIST*, 2005.

[28] Cristian Bravo-Lillo, Lorrie Faith Cranor, Julie Downs, Saranga Komanduri, and Manya Sleeper. Improving computer security dialogs. In *Human-Computer Interaction INTERACT 2011*, number 6949 in Lecture Notes in Computer Science, pages 18–35. Springer Berlin Heidelberg, January 2011.

[29] Dan Brickley and R.V. Guha. RDF vocabulary description language 1.0: RDF schema. http://www.w3.org/TR/rdf-schema/.

[30] Ian Brown and C. R. Snow. A proxy approach to e-mail security. *Software: Practice and Experience*, 1999.

[31] K. Selcuk Candan, Wen-Syan Li, Qiong Luo, Wang-Pin Hsiung, and Divyakant Agrawal. Enabling dynamic content caching for database-driven web sites. *ACM SIGMOD*, May 2001.

[32] Jack B. Dennis and Earl C. Van Horn. Programming semantics for multiprogrammed computations. *Communications of the ACM*, March 1966.

[33] Mira Dontcheva, Steven M. Drucker, David Salesin, and Michael F. Cohen. Relations, cards, and search templates: user-guided web data integration and layout. In *ACM UIST*, 2007.

[34] Mira Dontcheva, Steven M. Drucker, Geraldine Wade, David Salesin, and Michael F. Cohen. Summarizing personal web browsing sessions. In *ACM UIST*, 2006.

[35] Paul Dourish, W. Keith Edwards, Anthony LaMarca, and Michael Salisbury. Presto: an experimental architecture for fluid interactive document spaces. *ACM TOCHI*, June 1999.

[36] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazires, Frans Kaashoek, and Robert Morris. Labels and event processes in the asbestos operating system. In *ACM SOSP*, 2005.

[37] Adam Fass, Jodi Forlizzi, and Randy Pausch. MessyDesk and MessyBoard: two designs inspired by the goal of improving human memory. In *ACM DIS*, 2002.

[38] Ian Fette and Alexey Melnikov. The WebSocket protocol. http://tools.ietf.org/html/rfc6455.

[39] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, Irvine, 2000.

[40] Eric Freeman and David Gelernter. Lifestreams: a storage model for personal data. *ACM SIGMOD Record*, 1996.

[41] Jun Fujima, Aran Lunzer, Kasper Hornbk, and Yuzuru Tanaka. Clip, connect, clone: combining application elements to build custom interfaces for information access. In *ACM UIST*, 2004.

[42] Simson Garfinkel. Email-based identification and authentication: an alternative to PKI? *IEEE Security Privacy*, 2003.

[43] Bjoern Hartmann, Leslie Wu, Kevin Collins, and Scott R. Klemmer. Programming by a sample: rapidly creating web applications with d.mix. In *ACM UIST*, 2007.

[44] D. Austin Henderson, Jr. and Stuart Card. Rooms: the use of multiple virtual workspaces to reduce space contention in a window-based graphical user interface. *ACM Transactions on Graphics*, July 1986.

[45] Ian Hickson. HTML5 web messaging. http://www.w3.org/TR/webmessaging/.

[46] Ian Hickson. Microdata HTML standard. http://www.whatwg.org/specs/web-apps/current-work/multipage/microdata.html#microdata.

[47] David Huynh, Stefano Mazzocchi, and David Karger. Piggy bank: Experience the semantic web inside your web browser. *Web Semantics: Science, Services and Agents on the World Wide Web*, March 2007.

[48] J. Johnson, T.L. Roberts, W. Verplank, D.C. Smith, C.H. Irby, M. Beard, and K. Mackey. The xerox star: a retrospective. *IEEE Computer*, 1989.

[49] David Karger. Haystack: Per-user information environments based on semistructured data. In *Beyond the Desktop Metaphor: Designing Integrated Digital Work Environments*, pages 49–99. MIT Press, 2007.

[50] Akrivi Katifori, George Lepouras, Alan Dix, and Azrina Kamaruddin. Evaluating the significance of the desktop area in everyday computer use. In *ACHI*, 2008.

[51] Tim Kindberg, John Barton, Jeff Morgan, Gene Becker, Debbie Caswell, Philippe Debaty, Gita Gopal, Marcos Frid, Venky Krishnan, Howard Morris, John Schettino, Bill Serra, and Mirjana Spasojevic. People, places, things: Web presence for the real world. *Mobile Networks and Applications*, October 2002.

[52] Graham Klyne and Jeremy Carroll. Resource description framework (RDF): concepts and abstract syntax. http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/.

[53] P Leach, M Mealling, and R Salz. IETF RFC 4122: A universally unique IDentifier (UUID) URN namespace. http://www.ietf.org/rfc/rfc4122.txt.

[54] Henry M. Levy. *Capability-Based Computer Systems*. Butterworth-Heinemann, Newton, MA, USA, 1984.

[55] James Lin, Jeffrey Wong, Jeffrey Nichols, Allen Cypher, and Tessa A. Lau. End-user programming of mashups with vegemite. In *ACM IUI*, 2009.

[56] Thomas W Malone. How do people organize their desks?: Implications for the design of office information systems. *ACM TOIS*, 1983.

[57] Larry Masinter, Tim Berners-Lee, and Roy T. Fielding. Uniform resource identifier (URI): generic syntax. http://tools.ietf.org/html/rfc3986.

[58] Michelle L. Mazurek, Peter F. Klemperer, Richard Shay, Hassan Takabi, Lujo Bauer, and Lorrie Faith Cranor. Exploring reactive access control. In *ACM CHI*, 2011.

[59] Bonnie A. Nardi, James R. Miller, and David J. Wright. Collaborative, programmable intelligent agents. *Communications of the ACM*, March 1998.

[60] Hubert Pham, Justin Mazzola Paluska, Rob Miller, and Steve Ward. Clui: a platform for handles to rich objects. In *ACM UIST*, 2012.

[61] Eric Prud'hommeaux and Andy Seaborne. SPARQL query language for RDF. http://www.w3.org/TR/rdf-sparql-query/.

[62] Pamela Ravasio, Sissel Guttormsen Schr, and Helmut Krueger. In pursuit of desktop evolution: User problems and practices with modern desktop systems. *ACM TOCHI*, June 2004.

[63] Jude T. Regan and Christian D. Jensen. Capability file names: separating authorisation from user management in an internet file system. In *USENIX Security*, 2001.

[64] Jun Rekimoto. Time-machine computing: a time-centric approach for the information environment. In *ACM UIST*, 1999.

[65] P. Rodriguez, E.W. Biersack, and K.W. Ross. Automated delivery of web documents through a caching infrastructure. In *Euromicro Conference*, 2003.

[66] Mahadev Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel, and David C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, April 1990.

[67] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS: a fast capability system. In *ACM SOSP*, 1999.

[68] D. K. Smetters and R. E. Grinter. Moving from the design of usable security technologies to the design of useful secure applications. In *New Security Paradigms Workshop (NSPW)*, 2002.

[69] David Canfield Smith, Charles Irby, Ralph Kimball, and Eric Harslem. The star user interface: an overview. In *AFIPS National Computer Conference*, 1982.

[70] Jeffrey Stylos, Brad A. Myers, and Andrew Faulring. Citrine: providing intelligent copy-and-paste. In *ACM UIST*, 2004.

[71] Talis. RDF JSON. http://docs.api.talis.com/platform-api/output-types/rdf-json.

[72] Max Van Kleek, Michael Bernstein, David R. Karger, and mc schraefel. Gui — phooey!: the case for text input. In *ACM UIST*, 2007.

[73] Max Van Kleek, Michael Bernstein, Katrina Panovich, Gregory G Vargas, David R Karger, and mc schraefel. Note to self: examining personal information keeping in a lightweight note-taking tool. In *ACM CHI*, 2009.

[74] Stephen Voida, W. Keith Edwards, Mark W. Newman, Rebecca E. Grinter, and Nicolas Ducheneaut. Share and share alike: exploring the user interface affordances of file sharing. In *ACM CHI*, 2006.

[75] Jia Wang. A survey of web caching schemes for the internet. *ACM SIGCOMM Computer Communication Review*, October 1999.

[76] Steve Whittaker, Quentin Jones, Bonnie Nardi, Mike Creech, Loren Terveen, Ellen Isaacs, and John Hainsworth. ContactMap: organizing communication in a social desktop. *ACM TOCHI*, 2004.

[77] Alma Whitten. *Making Security Usable*. PhD thesis, Carnegie Mellon University, 2004.

[78] Jianliang Xu, Jiangchuan Liu, Bo Li, and Xiaohua Jia. Caching and prefetching for web content distribution. *Computing in Science Engineering*, 2004.

[79] Ka-Ping Yee. User interaction design for secure systems. In *ICICS*, 2002.

[80] Ka-Ping Yee. Guidelines and strategies for secure interaction design. In *Security and Usability: Designing Secure Systems that People Can Use*, pages 247–273. O'Reilly Media, Inc., July 2008.