A Widget Library for Creating Policy-Aware Semantic Web Applications

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
AUG 2 4 2010
LIBRARIES

by

James Dylan Hollenbach

S.B., Massachusetts Institute of Technology (2009)

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

Master of Engineering in Computer Science and Engineering

at the

ARCHIVES

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2010

© Massachusetts Institute of Technology 2010. All rights reserved.

Author ....
Department of Electrical Engineering and Computer Science
May 21, 2010

Certified by ....
Tim Berners-Lee
3COM Founders Professor of Engineering
Thesis Supervisor

Accepted by ....
Dr. Christopher J. Terman
Chairman, Department Committee on Graduate Theses

A Widget Library for Creating Policy-Aware Semantic Web Applications

by

James Dylan Hollenbach

Submitted to the Department of Electrical Engineering and Computer Science
on May 21, 2010, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

Abstract

In order to truly reap the benefits of the Semantic Web, there must be adequate tools for writing Web applications that aggregate, view, and edit the widely varying data the Semantic Web makes available. As a step toward this goal, I introduce a Javascript widget library for creating Web applications that can both read from and write to the Semantic Web. In addition to providing widgets that perform editing operations, access control rules for user-generated content are supported using FOAF+SSL, a decentralized authentication technique, allowing for users to independently manage the restrictions placed on their data.

I demonstrate this functionality with two examples: an aggregator application for exploring information about musicians from multiple data stores, and a universal annotation widget that allows users to make public and private comments about any resource on the Semantic Web.

Thesis Supervisor: Tim Berners-Lee
Title: 3COM Founders Professor of Engineering

Acknowledgments

I would like to thank my advisor, Tim Berners-Lee, for the five years of guidance he has provided me as an undergraduate and then as a graduate student. He has always encouraged me to think big about my work, and remained unwaveringly supportive throughout the course of my time as his student.

It was a great experience to work with the students, faculty, and staff of MIT's Decentralized Information Group. I would like to thank Matt Cherian, Ian Jacobi, Joe Presbrey, Oshani Seneviratne, Fuming Shih, Jose Soltren, Ralph Swick, Amy van der Hiel, Sam Wang, and K. Krasnow Waterman for the many discussions and moral support. I also will single out Lalana Kagal, who provided a steady stream of constructive criticism and useful commentary throughout the course of my research.

Special thanks go to my friends from outside of lab, who supported me throughout the stress of finishing my final year at MIT. Particular thanks go to Tyler Hutchison, who was very good at pretending to understand when I came downstairs at 2 AM to announce that I had figured out a bug, and my girlfriend, Alison, who had the foresight to live in California this year in case those moments came at 4 AM. Thanks also go to Violetta Wolf, who bought me a pizza one time.

Finally, I'd like to thank my parents, who, in addition to their unconditional love and support, let me use the computer enough that I made my first Website at age 9 in 1996, and my sister, who gave me no choice but to make that first Website when she beat me to the punch days earlier. The rest is history.

This work was supported by a National Science Foundation Cybertrust Grant and an Air Force Office of Scientific Research Grant.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The primary purpose of the Semantic Web is to model and link data using similar mechanisms to those that are currently used for documents on the Web. The Semantic Web uses these mechanisms to provide a machine-readable network of concepts that can easily be shared between applications. By design, any kind of structured data can be published on the Semantic Web: social networks, protein structures, and constitutional law could all potentially be mentioned within a few short hops of one another. The broad vision of the Semantic Web is that one day all of the data on the Web, whether it be medical records, social networks, or encyclopedia articles, will be as well-structured as the vast amount of hyperlinks between documents on the Web today.

If we are to truly benefit from the data available on the Semantic Web, we must first overcome two major obstacles:

1. **Semantic Web data is hard to visualize.** While there exist certain interface techniques such as property tables and faceted browsing for generically displaying data, neither of these options necessarily provides an optimal solution to a given interface problem. A scientist, for example, may wish to provide methods for searching and displaying brief descriptions of proteins based on certain fixed properties, or a Web hacker may wish to write a mashup for exploring and listening to new musicians. To develop truly powerful applications on the Semantic

Web, developers need tools that abstract out many if not all of the gritty details of working with raw, machine-readable data, while still allowing developers to freely create interfaces that are best suited to their working domain.

2. **Not all Semantic Web data is open data.** The fact that the structure of the Semantic Web allows for data to be shared between applications does not imply that all data *must* be shared between applications. More often than not, developers need to ensure that the data entrusted to them by their end-users is protected in a way that respects the end-users' demands–for example ensuring that a user's bank statements are kept private, or that their family photos are only viewable to family and friends. However, since the Semantic Web fundamentally aims to support the sharing of information between applications, it seems only logical that any policy system for sharing data on the Semantic Web should work within the Semantic Web as well.

The widget library developed over the course of this thesis takes steps toward resolving both of these issues. The library provides an easily deployed set of Javascript widgets that read and write Semantic Web data, allowing developers to pull in and manipulate data from multiple remote sources on the fly, exactly as the Semantic Web was designed to function. Additionally, the library supports the use of access control rules for user-generated data, providing information security when dealing with data in such an open environment.

Wherever possible, the widget library has been developed with the goal of being accessible to Web developers with little to no Semantic Web experience. Layout and templating rely entirely on the features provided by HTML, CSS, and Javascript. The library is meant to be used by any Web developer to add Semantic Web capabilities to their pages without providing any indication to their end-users as to what is going on under the hood.

These capabilities are demonstrated by way of two examples. The first, a universal annotation system for describing any resource on the Semantic Web, allows developers to specify the set of access control rules under which users may post shared comments, allowing for annotations to be shared between individuals, groups, or the entire world. The second, an application for viewing information about musicians and their recording sessions with John Peel, aggregates data from multiple sources into a unified, highly-customized user interface.

1.1 Overview

Chapter 2 provides an overview of related work in viewing and editing structured data. The remainder of this thesis focuses on describing and discussing the main body of work. Chapter 3 describes the pre-existing tools that play a role in the library's functionality. In Chapter 4, a detailed description of the widget library's architecture and implementation is provided. Chapter 5 follows with a set of working examples using the library. Chapter 6 discusses the tradeoffs made in the development of the system and investigates possible future paths for expanding the library. Chapter 7 provides a summary of the contributions made with this work.

As the main body of work for this thesis is the source code of the widget library, there is demonstrative code throughout the remaining chapters. Links to the widget library's source code, the examples used in Chapter 5, many other examples for each widget, and documentation are located in Appendix A.

Chapter 2

Related Work

This chapter provides an overview of related projects in the field of viewing and editing Semantic Web data, as well as more recent efforts focusing on integrating the Semantic Web with traditional Web applications. While this review is by no means exhaustive, it will provide an understanding of the strengths and weaknesses of some popular methods for working with structured data on the Web.

2.1 Browsing the Semantic Web

Several different approaches have been taken to displaying Semantic Web data, but the two most predominant approaches have been faceted browsing and graph browsing. Both approaches work well because of their complementary generality: whereas faceted browsing lends itself well to efficiently exploring a predetermined set of structured data, graph browsing allows users to openly explore the Semantic Web on the fly.

2.1.1 Faceted Browsing

Faceted browsing is a popular technique for browsing a set of objects that have a high number of properties in common. For example, a data set containing information about the past presidents of the United States might contain information on each

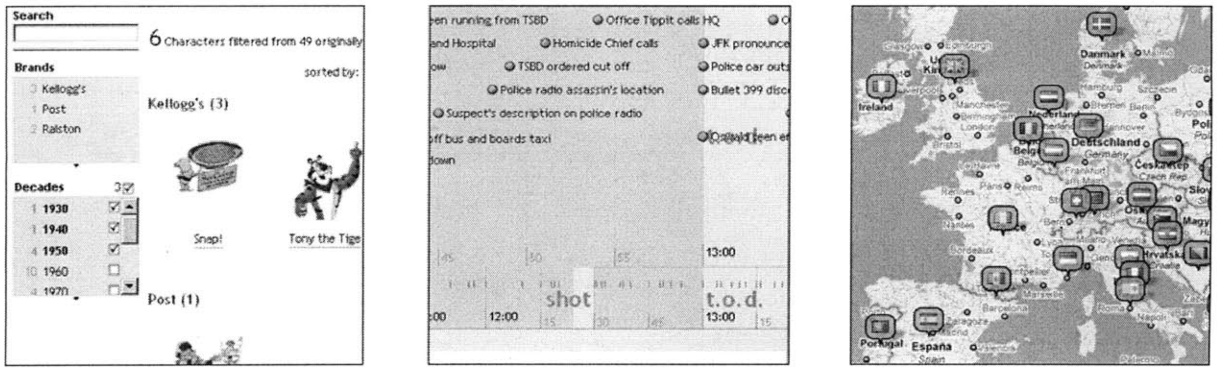


Figure 2-1: Examples created with Exhibit. Left, an example of a set of facets for choosing which data to display. Center, a timeline of the day of JFK's assassination. Right, a map of the flags of the world.

president's political party, religion, age when elected, and year of birth.[1] Faceted browsing relies on these common features to quickly generate interfaces for filtering the data. Each of these filters is called a facet. A facet for a president's political party could, for example, feature the properties "Democrat", "Republican", and "Whig", among others.

Exhibit [12] is perhaps the most popular library for creating faceted browsing interfaces for structured data. Exhibit allows developers to create interfaces out of small structured datasets, including maps for spatial data and timelines for temporal data, as in Figure 2-1. By design, Exhibit is meant to give people with little to no programming experience the opportunity to create highly attractive and usable interfaces. Exhibit marks an important deviation from previous attempts at creating Semantic Web interfaces in that it focuses heavily on user interaction rather than emphasizing the structure of the data; indeed, much of Exhibit's success can be attributed to the usefulness of a good faceted browsing interface, rather than any particularly useful aspect of globally linked data (Exhibit can, for example, provide visualizations using data from a CSV file).

Without a doubt, the faceted browsing paradigm is a powerful one. However, many Web applications, Semantic Web or not, do not require such an interface, and cannot make use of Exhibit–particularly those that require complex interaction

[1]An example of this functionality implemented in Exhibit can be found at `http://www.simile-widgets.org/exhibit/examples/presidents/presidents.html`

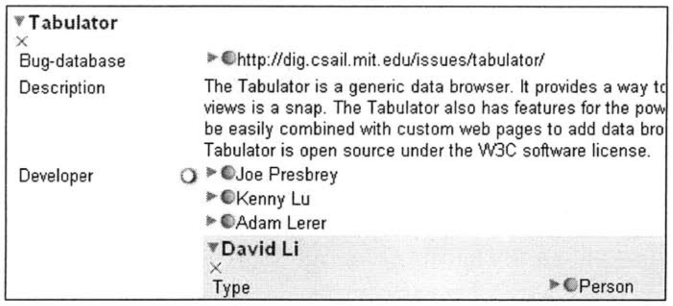
between components, as would normally be accomplished through Javascript. Additionally, faceted browsing is primarily geared toward drilling down a set of results in a pre-determined data set, and often does not make sense in applications where users wish to perform tasks such as exploring a live social graph.

While libraries such as Exhibit give much greater power for creating interfaces to those with little or no programming skill, they are not particularly geared toward developers who are trying to create a unique user experience in their problem domain. Though Exhibit and others focus on providing an excellent interface that works right out of the box, there is still a necessity for more broadly customizable and extensible interface libraries that can function with the Semantic Web.

2.1.2 Graph Browsing

Another common method for looking at Semantic Web data is graph browsing, as exemplified by Tabulator [6]. Other examples not discussed here include sig.ma [27] and the Disco Hyperdata Browser [7], which generally draw inspiration from Tabulator. In graph browsing interfaces, resources are displayed in the form of a property table as shown in Figure 2-2. Users can navigate between resources by clicking on the properties listed in the property table. New resources are requested as the user requests to view them, and the data loaded is aggregated into a single local store. As the user browses through more and more data, they may cause documents containing additional information about previously viewed resources to be loaded. This new information is aggregated and included in the respective resources' property tables. Additionally, Tabulator allows users to edit data if the server sends a special response indicating that a given document is editable. In this way, graph browsing interfaces allow users to actively browse the Semantic Web, discovering and creating new data on the fly.

Graph browsing is advantageous in that it allows users to actively browse the Semantic Web, but the property table interface that accompanies such browsers becomes overloaded. Certain resources, such as a description of The Beatles, can quickly grow to the point where the property table spans several screens of text. Tabulator



Tabulator	
×	
Bug-database	http://dig.csail.mit.edu/issues/tabulator/
Description	The Tabulator is a generic data browser. It provides a way tc views is a snap. The Tabulator also has features for the pow be easily combined with custom web pages to add data bro Tabulator is open source under the W3C software license.
Developer	Joe Presbrey
	Kenny Lu
	Adam Lerer
	David Li
	×
	Type Person

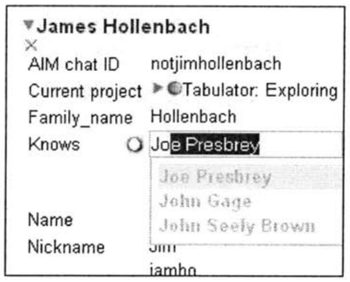
Figure 2-2: A property table in Tabulator. The subject of the table is the Tabulator project itself. Each predicate is displayed on the left side of the table (Bug-database, Description, Developer). Each node with a circle and triangle on the left represents another resource which can be expanded in place for further investigation.

attempts to mitigate this problem by introducing "panes", small interface widgets that are displayed when certain pieces of data are available for a resource: a "social" pane is displayed when viewing a person, and a "justification" pane is displayed when viewing the output of a reasoner, for instance. Additionally, Tabulator provides "views" which allow the user to jump from a query generated during graph browsing to maps, data tables, and timelines, allowing for the data to be explored in more intuitive ways. While many of these panes and views are highly indicative of the sorts of applications that developers should be able to create with the Semantic Web, they are somewhat out of place as parts within a larger application, and the computational overhead for choosing which panes to use can weigh down the application. Even so, Tabulator's back-end library provides many useful functions for manipulating RDF in Javascript. Section 3.1 discusses why the Tabulator Library's established ability for creating panes made it the clear choice for back-end data handling for the widget library developed in this thesis.

2.2 Editing

2.2.1 SPARUL and Live Updates

Though it is still going through the standardization process, SPARQL Update (SPARUL) [24] has risen to be the de facto standard in Semantic Web data authoring. SPARUL pro-



James Hollenbach

AIM chat ID	notjimhollenbach
Current project	Tabulator: Exploring
Family_name	Hollenbach
Knows	Joe Presbrey
	Joe Presbrey
	John Gage
Name	John Seely Brown
Nickname	Jim
	jambo

Figure 2-3: Editing data in the Tabulator Firefox extension.

vides an extension to the SPARQL query language that allows for data to be dynamically inserted into and deleted from the triple store exposed by a SPARQL endpoint. Despite its utility, the support provided for SPARUL in client-side Web development libraries remains relatively lean. The Tabulator Extension [5] actively supports authoring via SPARUL when viewing data in an outline, as shown in Figure 2-3, but the fact that SPARUL is not yet normalized across all of the major server-side libraries has limited the utility of such editing. Nevertheless, the prevalence of some level of SPARUL in server-side SPARQL implementations and its ongoing standardization process makes it the clear choice for Semantic Web authoring.

Extending existing structured data libraries to support RDF and SPARUL is a relatively straightforward task. For example, despite Exhibit's lack of focus on globally linked data, it is not fair to suggest that making the move from a structured JSON data file, as Exhibit uses, to some more expressive RDF data is necessarily a difficult operation for a skilled developer. Indeed, more recently Exhibit has been extended to implement a locally-editable data store [14], and the move from there to a version that can write out to SPARQL stores on the Web does not seem far-fetched. It seems more fair to say, then, that the real challenge lies in developing the portions of a library that allow users to edit data at the level of complexity permitted on the Semantic Web.

2.2.2 Integration with Non-Semantic Web Applications

Recently, efforts have been made to transfer application data generated on the Semantic Web back to traditional Web applications, as accomplished by the Pushback project [10]. The primary function of Pushback's RDForms language is to generate the correct set of inputs for a legacy service (such as a Twitter update) from RDF data. While the RDForms language is highly effective at describing the intricacies of the operations to be performed on legacy services, the syntax overhead for getting these conversions right is probably too high to include in a library geared solely toward the Semantic Web. Nevertheless, as the Semantic Web grows in popularity, services like Pushback will probably become helpful in interfacing with those services that continue to hold out on publishing data via RDF and SPARQL.

Chapter 3

Tools

This chapter describes the technologies used in the implementation of the widget library in preparation for Chapter 4, which describes the architecture of the widget library itself.

3.1 Javascript RDF Library

The Tabulator RDF Library serves as the back-end data store in the widget library. Originally written in 2005, the Tabulator Library has over time grown to house a stable core of RDF processing functionality. It features RDF/XML and N3 Parsers (an RDF+JSON parser was written for the purposes of this thesis), and a host of convenience functions for querying over the RDF store. Most notably, the Tabulator library implements "smushing", which allows for resources to be equated according to OWL [15] `sameAs` properties, functional properties, and inverse functional properties. This feature proves indispensable when dealing with resources coming from across the Semantic Web, as it allows for data from different sources such as the musician data seen later in Section 5.2 to be merged and reasoned about as a single entity.

Additionally, many of the panes developed with the Tabulator Library in the Tabulator Extension are highly representative of the sort of interface that might be created by a developer using a quality Semantic Web widget library: the panes available are geared toward social networking, blogging, and picture sharing, among many others. This shows that Tabulator's functionality is already able to support the types of interfaces to be supported by the widget library.

But while the back-end functionality of the Tabulator Library is time-tested and relatively stable, Tabulator's panes share very little interface code–instead, they each implement their own user interface features that are too highly tailored to be easily abstracted into a library. This has led to a large amount of bloat and duplicated code in the Tabulator pane library. For this reason, Tabulator made an obvious choice for the back-end functionality of the widget library, but at the same time particularly exemplifies the need for a unifying widget library for interface development.

Finally, it is worth noting why the Tabulator library was chosen over rdfQuery [29], essentially the only other feature-rich Javascript RDF library. While rdfQuery provides a large number of highly useful shortcuts for querying RDF data (In fact, some of rdfQuery's functionality was mimicked on top of the Tabulator library, as will be shown in Section 4.4.2), rdfQuery is primarily geared toward handling RDFa [1] data embedded in a single page. It does not natively provide provenance tracking or smushing, which are key components of developing a library geared towards data aggregation. While the syntactic features of rdfQuery are useful out of the box, it still lacks some of the key features that Tabulator provides for working with the Semantic Web in real time.

3.2 Javascript Development Toolkit

The jQuery [21] Web development toolkit provides an excellent lightweight framework for Javascript development. In particular, the jQuery UI library's syntax for creating new widgets is particularly useful in the context of the Semantic Web widget library. The jQuery UI library provides a base Widget class that can quickly and easily be extended to work directly within the jQuery syntax, providing garbage collection and library functions as built-in features of widgets. In Section 4.4.1, I will show how the jQuery UI widget class was extended to provide a base class for quickly developing Semantic Web widgets.

3.3 RDF Access Control

In order to provide a library that works with real-world applications, it is necessary to support some form of access control. To be secure, access control must be handled by the server sending and receiving data, but it is still helpful to users if the interface they are using has some knowledge of how the authentication process works. Therefore, while this section tends to focus more on server-side issues of RDF access control, these issues had broader implications in the development of the widget library.

A user's identity plays an important role in any decentralized Web application that wishes to make use of user profiles or access control. In order to decide what information should be displayed to a given user requesting data, the server must first know who the user is. Once the user is identified, their identity can be used to perform any of a large number of customizations that are based entirely on Semantic Web data. A user's FOAF profile, for example, may be used to provide context around resources, such as "The WWW Conference, which you attended last week" or, "Joe, your friend from high school". In decentralized systems, there is no central authority that can determine a user's identity and thereby provide personalized features. Instead, protocols must be defined that allow different providers to authenticate users, regardless of where their identity may originate from.

FOAF+SSL [28] uses client certificates to authenticate users . This functionality relies on the placement of a user's public encryption key in both a certificate and a remote RDF file stored on a Web server. Additionally, the certificate is created with the URI for the location of the remote RDF file (called the user's "WebID") stored in an auxiliary field. When a user sends a FOAF+SSL certificate to a server, the server identifies the remote URI, pulls down the public key from that URI, and compares it with the public key in the user's certificate (See Figure 3-1). If the keys match, the user is authenticated as the person associated with the public key-WebID pair described in the certificate. The server can then query itself about the user's access permissions (here, with SPARQL) before finally returning a response to the requesting user.

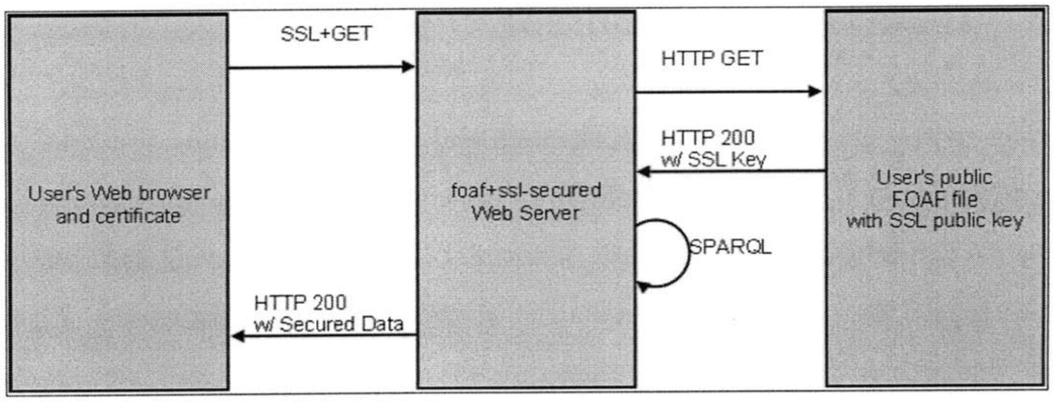

SSL+GET
HTTP GET
HTTP 200
w/ SSL Key
User's Web browser and certificate
foaf+ssl-secured Web Server
User's public FOAF file with SSL public key
SPARQL
HTTP 200
w/ Secured Data

Figure 3-1: Flow of information in a FOAF+SSL Request. The three components are a user (left), the server the user is trying to access (center) and a server holding the user's FOAF file (right).

The advent of FOAF+SSL has proven very useful in the development of RDF-based access control systems. Recent work within the Decentralized Information Group at MIT has led to the development of a full access control implementation on top of Apache [11]. This access control system performs document-level access control using rules written in RDF. The same rules are used in the widget library, but, as will be described in Section 4.5, it was necessary to extend the access control system to work at the named graph level in order to operate more readily with SPARQL endpoints. Interested readers will find a more complete description of the RDF ACL system as well as FOAF+SSL in [11].

3.4 Working with Remote Data

Since the widget library needs to load remote resources from the Semantic Web in a browser's sandboxed Javascript environment, a method for loading remote resources within the browser is required. There are two possible options for this, both of which are used in the widget library: the relatively new Cross-Origin Resource Sharing (CORS) specification [30] and JSON with padding (JSONP) [13].

The CORS specification, though not yet standardized, is largely supported in the latest versions of all of the major Web browsers [20]. It operates by sending additional HTTP headers to the target server, which indicate to the browser and the server that a cross-origin request is being made. The receiving server then has the option of either honoring the request, or rejecting it on the basis of its origin. As it becomes more broadly adopted, the CORS specification will be extremely useful for servers implementing public SPARQL endpoints who wish to allow Semantic Web developers to freely query their data; by receiving direct requests from the user, servers implementing CORS can use any authentication scheme while still sharing data with remote resources.

But for servers that do not yet implement (or do not wish to implement) the CORS specification, there is always the option of using a proxy server to provide a JSONP translation of any data available. JSONP uses a callback function to load data from an external Javascript object: when the script is loaded, the callback function is called with the remote data as an argument. The Talis Metamorph [2] project provides a Web service for converting between different RDF syntaxes, including RDF+JSON. Since JSONP data can be loaded from cross-origin resources in any browser, the use of a single remote Metamorph server allows developers on any domain to request data from any public RDF resource on the Web. However, working with Metamorph has one obvious drawback: since the data is funneled through a proxy server, it is impossible to perform authentication with any third-party data stores that contain secure information.

The combination of CORS and Metamorph allows for most RDF resources on the Web to be loaded remotely. Ideally, future browsers will allow users to better manage their own identity when sending information to remote resources, making CORS a more secure and efficient option than remote JSONP. As the CORS specification moves toward completion, browser developers have already begun considering how such a system and the corresponding shift in demand for identity management would factor into their user interfaces [9].

It is important to note that these techniques are only required due to the security measures put into place by browsers. If an application developed with the widget library is used in a privileged environment, such as a Firefox extension or a mobile phone application, the CORS and Metamorph mechanisms will simply never be activated, since these environments will allow the widget library to freely generate cross-origin requests.

Chapter 4

Architecture and Implementation

This chapter describes the architecture and implementation of the widget library. It begins with a simple example and a summary of the common features shared by the provided widgets. Then, a description of several key pre-defined widgets is provided, followed by a description of the inner workings of the library and the functions provided for extending the library.

4.1 Introductory Example

As an introduction to the general functionality of the library, this section dissects a simple program for displaying all of the acquaintances found in a given data set. The example requests data from two different documents describing people and displays all of the "X knows Y" acquaintance relationships found.

The widget library's functionality can be accessed both from HTML and from Javascript. The HTML in Figure 4-1(a) and the Javascript in Figure 4-1(b) can each separately produce the same output, shown in Figure 4-1(c). The remainder of this section steps through each line in these example programs and explains their contributions in producing the output.

```
<html xmlns="http://www.w3.org/1999/xhtml"  xmlns:sw="http://dig.csail.mit.edu/2009/swjs#">
  <head>
    <!--Data sources-->
    <link href="http://web.mit.edu/jambo/www/fred.rdf" rel="sw:data"></link>
    <link href="http://web.mit.edu/jambo/www/jambo.rdf" rel="sw:data"></link>
    <!--Scripts-->
    <script type="text/javascript" src="jquery.js"></script>
    <script type="text/javascript" src="rdfwidgets.js"></script>
  </head>
  <body>
    <p>A list of people where the left knows the right.</p>
    <div id="friendships" sw:widget="triplelist"
                          sw:predicate="foaf:knows" />
  </body>
</html>
```

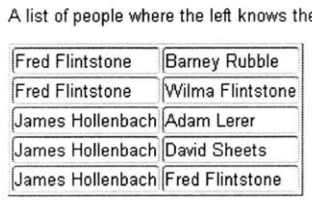
(a)

```
$(document).ready( function() {

    $.rdfwidgets.load("http://web.mit.edu/jambo/www/fred.rdf#fred");
    $.rdfwidgets.load("http://web.mit.edu/jambo/www/jambo.rdf#jambo");
    $("#friendships").triplelist( { predicate: "foaf:knows" } );

});
```

(b)



A list of people where the left knows the right.

Fred Flintstone	Barney Rubble
Fred Flintstone	Wilma Flintstone
James Hollenbach	Adam Lerer
James Hollenbach	David Sheets
James Hollenbach	Fred Flintstone

(c)

Figure 4-1: A simple program written using the widget library. Both program (a), written in HTML, and program (b), written in Javascript using jQuery, can produce the same output, shown in (c). The example in (b) simply requires the same Javascript `script` tags as (a).

4.1.1 Loading Data

First, data must be loaded into the library. This can be accomplished by including data in a `link` element, as on lines 4 and 5 of the HTML example. In this case, two documents are included–a pair of user profiles that use the FOAF ontology. Data sources have the `rel` attribute set to `sw:data`, which indicates that those elements are specifically intended to be used as data sources. Alternatively, the Javascript code allows data to be accessed dynamically using the static library functions defined under the jQuery namespace `$.rdfwidgets`. The `load` function used in lines 3 and 4 of the Javascript example will load any URL containing RDF data.[1] In both the HTML and Javascript cases, the library will attempt AJAX and CORS requests before finally resorting to the Metamorph proxy in order to load the data sources listed. The net result is that the widget library's local data store will combine the data from both sources into a single, unified store.

4.1.2 Selecting a Widget

Now that the data has been loaded, it may be processed and displayed to the user. In this example, the program simply finds the set of triples that match the form `"a knows b"` and displays the matches in a table. To accomplish this, a widget called the `triplelist` is used. The `triplelist` widget simply finds the a of matching triples in the data store, given a set of constraints. To do this from HTML, the `sw:widget` attribute is used, as shown in line 12 of the HTML example. This attribute's value indicates which widget will be used. Any options to be passed into the widget can be passed using the prefix `sw` followed by the option name. In this case, the `predicate` option is set in order to indicate that the widget should display the set of triples with the predicate `foaf:knows`. The Javascript code to invoke the same widget, shown in line 5 of the Javascript example, uses a jQuery selector to choose the set of DOM elements to draw the widget into. Otherwise, it uses the same options as the HTML example, which are provided in an anonymous object.

[1]Currently, parsing and direct loading is supported for RDF+XML, RDF+JSON, and N3. The Metamorph proxy supports RDF+XML, RDF+JSON, N3, and RDFa.

4.1.3 Displaying the Data

Once a widget has been invoked, the library handles the display process internally. All of the native widgets provided, including `triplelist`, will automatically update themselves as new data is loaded. This means that if one of the two documents requested for some reason takes a long time to be served, the widget library will simply display the acquaintances found in the data loaded thus far; optionally, the developer may have a status icon display whether or not all data sources have finished loading by listening for a custom `rdfloaded` event or by using an optional callback argument to the `load` function. Finally, the set of CSS classes provided for each widget is well-defined, allowing for the data table to be styled to taste. In the end, the code provided in Figure 4-1 displays the five pairs of people related by the `foaf:knows` property.

This admittedly simple example serves only as an introduction to the widget library's basic functionality. The following sections describe the default widgets available as well as the potential for extending the widget library in detail.

4.2 Common Widget Features

This section outlines the key features shared between the widgets provided in the widget library. As was explained in Section 4.1 and demonstrated in Figure 4-1, most of the functionality of the widget library can be achieved through equivalent expressions in both HTML and Javascript.

4.2.1 Loading Data

In order to provide any useful output, the widget library must first be supplied with a set of data sources. These data sources are loaded into a unified local store which is then used for displaying information in widgets. The primary methods for importing data are the `load` Javascript function and the `rel="sw:data"` HTML attribute. Using either of these methods will activate the widget library's chain of strategies for loading resources. By default, the library will obey the MIME type sent in the server's response when processing data. If no MIME type is found, the library checks to see if the `type` attribute is set for the source as a backup. This property takes on values describing the MIME type of the data to be loaded, such as `application/rdf+xml` or `text/n3`. If the MIME type cannot be determined, or if the request fails, it is assumed that the resource cannot be loaded without the aid of the Metamorph proxy. Regardless of the outcome, a callback is fired after the request completes, indicating its success or failure.

The loading of data is treated as a live process. Widgets are expected to be able to handle the fact that new data may be loaded asynchronously after the widget is instantiated. Whenever a new data source loads, every widget's `refresh` function is called, allowing for the widget to react to the new data, if necessary. Optionally, however, the `static` option can be used to indicate that a given widget should never be updated when new data is loaded.

4.2.2 Processing Data

Unification and Smushing

Sometimes, Semantic Web documents can use different URIs to describe exactly the same entity–for example, The Beatles are identified with the URI `http://dbtune.org/musicbrainz/resource/artist/b10bbbfc-cf9e-42e0-be17-e2c3e1d2600d` in the MusicBrainz RDF database [16], and the URI `http://dbpedia.org/resource/The_Beatles` on DBPedia [4]. These identical entities can be linked together using a set of properties defined in the OWL ontology using a process called "smushing". The Tabulator Library implements smushing automatically for all data loaded into the widget library's internal store. If two resources are related by the `owl:sameAs` property, then they will be treated as equivalent when data is requested about either resource. For example, if a `resource` widget is created using the MusicBrainz URI above, and the store has equated the MusicBrainz and DBPedia URIs due to the discovery of an `owl:sameAs` relation, then all of the triples known to describe the

DBPedia URI will be displayed in addition to the data describing the MusicBrainz URI. Currently, a developer must explicitly request that `sameAs` relations are followed and loaded into the store. This prevents situations where `sameAs` relationships cause huge amounts of remote data to be requested automatically.

A similar process is followed if two URIs share an `owl:functionalProperty` or `owl:inverseFunctionalProperty`. These properties are specially defined properties that uniquely identify an entity–for example, a user's `foaf:mbox` is meant to be a private E-mail inbox used only by that user. Therefore, if two entities share the same `foaf:mbox` property, then they can be assumed to be the same.

Data Filters

Because the data loaded into the widget library can potentially come from multiple different documents spread across the Semantic Web, not all of which are necessarily trusted, it is important to provide some control over which information is displayed in certain widgets. For example, when showing a user what is supposed to be a summary of their own FOAF profile, it would be in poor judgment to show data originating from a random source that contains arbitrary statements about the user. To help combat this issue, the widget library provides both source-based filtering and triple-based filtering options on each widget, shown in Figure 4-2. When a widget is created, the set of data sources it uses can be specified using either the sources' absolute URIs or an optional plain-text `name` attribute that can be used to quickly refer to datasources. This will cause the widget's processing code to only be provided with triples originating from those sources. Alternatively, a `filter` function can be specified. The `filter` function is called once for each triple that is to be displayed in the widget. If the `filter` function returns false for a given triple, then that triple is not displayed in the widget's final output. These two filtering options provide coarse and fine-grained filtering for widgets.

```
function myFilter( triple ) {

    if( triple.predicate.uri === "http://xmlns.com/foaf/0.1/knows" &&
        triple.object.uri.indexOf( "http://web.mit.edu" ) !== 0 ) {
        return false;  //reject this triple.
    }

    return true; //accept this triple

}

$.rdfwidgets.load("http://web.mit.edu/jambo/www/foaf.rdf#jambo",
                  {name:"jambo"});
$.rdfwidgets.load("http://example.com/someresource.rdf#jambo",
                  {name:"example"});

$("#someelement").resource({
    subject: "http://web.mit.edu/jambo/www/foaf.rdf#jambo",
    filter:  myFilter,
    sources: ["jambo"]});
```

Figure 4-2: A demonstration of the various filtering options available in the widget library. A function for filtering triples, `myFilter` is used. This example rejects any triple that describes an acquantaince who does not have a URI from within the MIT Web space, which might be useful in an application geared towards the MIT community. A named source, `jambo`, is used as the only data source, so data coming from `example.com` will not be used in display.

```
<link rel="sw:endpoint"
      href="http://jambo.xvm.mit.edu/sparql"></link>
<link rel="sw:data"
      href="http://www.example.com/foo"></link>
<link rel="sw:data"
      href="http://www.example.org/bar"
      sw:endpoint="http://www.example.org/sparql"></link>
```

Figure 4-3: Defining endpoints for different sources. Here, `jambo.xvm.mit.edu/sparql` is defined as the default endpoint. The data source on `example.com` has no endpoint defined, so data edited from that source will be written to `jambo.xvm.mit.edu/sparql`. However, the data source on `example.org` has its own endpoint defined, so any data edited that originates from `example.org` will instead be written to the `example.org/sparql` endpoint.

4.2.3 Modifying Data

Endpoints

When a widget creates new data, it must identify a SPARQL endpoint to send that data to. Different data sources may need to write back updates to different locations. For this reason, the widget library supports the definition of both a global default SPARQL endpoint and per-source SPARQL endpoints for the creation of new data. The default endpoint is defined either with a `link` element with the attribute `rel="sw:endpoint"`, or by calling the widget library's `setDefaultEndpoint` function (See the example in Figure 4-3. Per-source endpoints are set by either placing a `sw:endpoint="someuri"` attribute inside of a data source `link` element, or by specifying the endpoint when calling the `load` function from Javascript. Alternatively, some services can define the endpoint to be used by responding with the HTTP header `MS-Author-Via: SPARQL`. This response header indicates that the resource can be written to directly using SPARUL, and the widget library will use this response as its default behavior unless the developer specifically overrides it.

Some SPARQL servers, such as ARC2, require the use of a named graph when inserting new data into the endpoint. For this reason, widgets that create new resources support a `graph` parameter for indicating the name of the graph that newly created data should be inserted into. If named graphs are required, but the data created should be inserted into a new named graph, the `usenamedgraphs` option for a widget can be set to `true`.

User Identity

While no mechanism for authentication is provided directly within the widget library, it can at times be helpful to have a global URI defined as the active user for the application. This can be set either using the `rel="sw:user"` attribute in a link header or by calling the `setUser` library function. While this system does not on its own provide any form of security, setting a user attribute can be helpful when defining widgets that are meant to display user-specific information.

4.2.4 Access Control Data

Currently, the two widgets that are capable of creating entirely new instances–`createinstance` and `annotator`–accept an optional `acl` parameter that points to an access control rule to be enforced on the named graph storing the data to be created. The access control rules used are defined using the ontology and rule system defined in [11]. A pair of access control rules using this ontology are shown in Figure 4-4. The rule `rule1` grants `read`, `write` and `control` access to a file for a specific user (`control` is an access mode that allows the user to modify the access control rules for a given graph). The rule `rule2` grants `read` access to a class of users–in this case, a special class that encompasses all authenticated users. A server wishing to enforce these rules can use FOAF+SSL to identify a user's URI and then match that user's URI with the access rules for a given file or graph to determine the level of access to grant the user.

The named graph RDF Access Control implementation optionally accepts an `acl` parameter as a query parameter in an HTTP POST, which contains the URI of a valid RDF ACL to be used with a SPARQL INSERT query. If the query generates a new named graph on the server, the access control system will use the rules found at the ACL URI provided in the `acl` parameter to control access to the new resources.

```
@prefix acl: <http://www.w3.org/ns/auth/acl#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .

<#rule1> a acl:Authorization;
         acl:agent <http://web.mit.edu/jambo/www/foaf.rdf#jambo>;
         acl:accessTo <http://example.com/somegraph>;
         acl:mode acl:Read;
         acl:mode acl:Write;
         acl:mode acl:Control .

<#rule2> a acl:Authorization;
         acl:agentClass <http://xmlns.com/foaf/0.1/Agent>;
         acl:accessTo <http://example.com/somegraph>;
         acl:mode acl:Read .
```

Figure 4-4: A pair of rules written with the ACL Ontology.

Rule Name	*Permissions Granted*
permanent	Readable by any user, cannot be edited.
private-editable	Readable and writeable by an owner and a group or other user of the owner's choosing.
private-readonly	Readable and writeable by an owner. Readable for a group or other user of the owner's choosing.
public-editable	Readable and writeable by all.
public-readonly	Readable and writeable by an owner. Readable for all others.

Table 4.1: The set of predefined ACL rules provided with the widget library.

The `acl` parameter in the widget library is used to define the value of the `acl` parameter to be sent to a SPARQL endpoint when new data is created. In addition to supporting a static URI, the `acl` parameter can contain an object selecting one of the library's predefined access control rules. The full list of these access control rules is provided in Table 4.1. These predefined access control rules provide common rules for sharing data, such as publicly editable wiki permissions and private user-to-user read-only sharing. Certain policies require additional information in order to be fully specified–for example, a rule that only allows two users to read a file must provide the URIs of the two users to be granted access. An example of how such a policy can be used is shown in Figure 4-5 Providing a simple method for defining access avoids

```
$.rdfwidgets.load("http://www.example.com/commentlist.rdf");
$("#comments").annotator({
        acl:{ type:"public-readonly",
              user:"http://web.mit.edu/jambo/www/foaf.rdf#jambo" }
        });
```

Figure 4-5: An annotator widget instantiated with ACL. Here, the `public-readonly` access mode is used, which requires the definition of a `user` who owns the new data. If a default user has been defined, as in Section 4.2.3, then this parameter is not necessary.

the need for developers to muck around with actual access files–instead, they simply select a policy and provide the arguments to fill in the blanks.

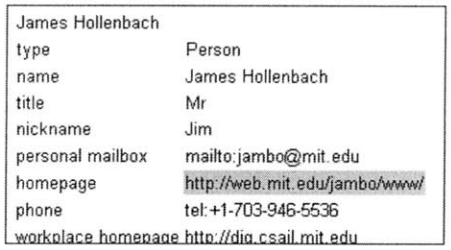
While the widget libary supports the submission of data with ACL metadata for use with FOAF+SSL, the enforcement of access control rules is left entirely to the server receiving the request. There is no need for special configuration in order to provide users with the data they are allowed to see–the browser will simply request that the user provide a certificate for each FOAF+SSL-secured data source requested by a given application. In practice, this operation is only supported in Firefox and Safari for cross-origin requests. A full description of the tradeoffs of this system and its requirements for future cross-browser compatibility are provided in Section 6.2.

Current implementations of the RDF Access Control system essentially control access on the document level. While it is theoretically possible to provide an RDF Access Control system that allows for different access rules to be defined for every resource or every triple in an RDF store, such levels of granularity tend to be too specific in most practical use cases. A full discussion of the tradeoffs in the RDF Access Control system's granularity can be found in [11].

4.2.5 Events and Callbacks

A simple set of events and callbacks are defined for common events that occur within the library. Application developers can bind listeners for these events to widgets in order to perform different tasks according to user input.

The `rdfselect` event is fired whenever an element is selected by a mouse click.



James Hollenbach	
type	Person
name	James Hollenbach
title	Mr
nickname	Jim
personal mailbox	mailto:jambo@mit.edu
homepage	http://web.mit.edu/jambo/www/
phone	tel:+1-703-946-5536
workplace homepage	http://dig.csail.mit.edu

Figure 4-6: Node selection in the widget library. Here, the node `http://web.mit.edu/jambo/www/` has been selected.

The library keeps track of a unique selected element for the entire window. This element carries the CSS class `rdfselected`, and is therefore styled in a way that indicates its status; the default style provides a light blue background for the selected element, as shown in Figure 4-6. When the `rdfselect` event is fired, in addition to carrying a reference to the target element, the event contains the RDF term currently displayed by the newly selected node. This allows for event-driven behaviors such as displaying a set of comments regarding whichever node the user selects.

On a per-widget level, the `beforesubmit` and `aftersubmit` callbacks provide a method for modifying a SPARQL update before it is sent to a server and a method for checking the response sent back by the server, respectively. Using the `beforesubmit` callback, auxiliary data can be added to an INSERT or DELETE request, or an extra parameter, such as a session key, can be appended to the outgoing XMLHttpRequest. The `aftersubmit` callback can be used to provide more advanced visual feedback, using coloring or jQuery animations, to indicate the success or failure of a request. The two callbacks can even be used together to provide custom feedback while a submission is pending in addition to the default behaviors, which disable input elements until the the request is completed.

More globally, when new data is loaded into or removed from the local data store, an `rdfchanged` event is triggered on the `document` element. This allows for widgets and application code to implement features such as logging of updates, providing a display of the history of a user's actions, and even potentially the ability to undo or redo operations such as deleting a triple from a given SPARQL endpoint.

4.2.6 Appearance

Each widget has a comprehensive set of CSS classes attached to its nodes that allow for widgets to be styled to match a Web site's layout. A widget's CSS class name consists of the the string `rdf` concatenated with the widget's name–for example, the outermost node of the `edit` widget consists of a `span` element with the style class `rdfedit`. Additionally, the style class `rdfselected` defines the style for the currently selected term in the window. Finally, while no style class is provided, callbacks and custom events allow developers to listen globally to determine whether or not all of their requested data sources have loaded.

4.3 Built-In Widgets

The widget library comes packaged with 17 predefined widgets, over half of which provide editing capabilities for the data they display. A full list of these widgets along with brief descriptions of their capabilities is provided in Table 4.2. Rather than describing all of these widgets in detail, this section provides a summary of a few widgets that embody much of the functionality provided by the library.

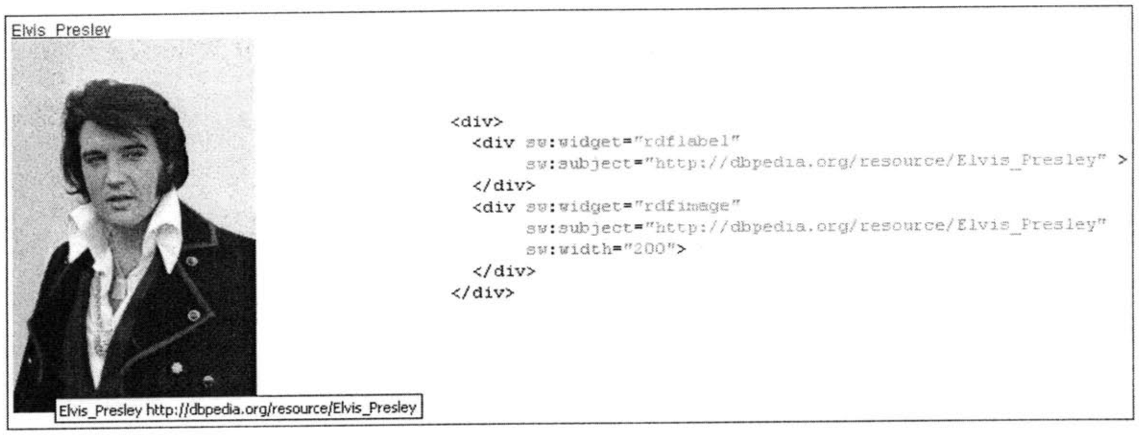
4.3.1 Labels and Simple Editing

The most basic functionality the library provides is viewing and editing a single term (subject, predicate, or object) in a triple. To view a term's value, the library provides the `label` class. The `label` widget can function in two different ways: it can either be provided a single term value, such as a person's WebID, and try to find a human-readable label for that resource in the local data store (as shown for Elvis in Figure 4-7), or it can take two of the subject, predicate, and object of a triple and display the value of the missing term. This second construct can be used, for example, to display a person's homepage given their WebID. It is also possible to optionally override the human-readable label and instead display the actual URI associated with a given resource.

Widget Name	*Description*
annotator	Display a popup for writing and viewing annotations about resources described on the page in RDFa.
checkbox	A set of checkboxes, each corresponding to a possible value for a triple.
combobox	A combo box, where the currently selected value corresponds to the current value of a term in a triple.
createinstance	Given a set of properties, display a form for creating a named graph about a new object.
edit	A single editable input box for modifying the value of a term in a triple.
image	Display any image property found to depict the provided subject.
instancedropdown	An autocomplete menu for finding any instances of a given class in the local store.
instancelist	A simple list of all instances of a given class.
label	Display a text label for a given subject.
radio	A set of radio buttons, each corresponding to a possible value for a triple.
resource	Display all of the properties of a given subject in an editable property table.
selecttable	Display all of the resources in the store that have a set of properties.
toolbar	Show a set of image representations of a given class in a toolbar, where each item is selectable.
tripleedit	A simple widget for editing each of the subject, predicate, and object of a single triple. Largely used as the basis for more complex widgets.
triplelist	Display all of the triples that match the provided subject, predicate, and/or object.
Diagnostic Widgets	
sourcelist	Show all of the currently loaded data sources.
sourceview	Dump the entire dataset in the local store in a given format.

Table 4.2: Descriptions of the widgets provided with the widget library.

Some elements are more logically displayed as images, or are put into better context when accompanied by images. The `image` widget uses RDFS relationships to find a valid visual representation of a given subject. This means that a developer does not need to track down a specific property that holds an image of a given resource,



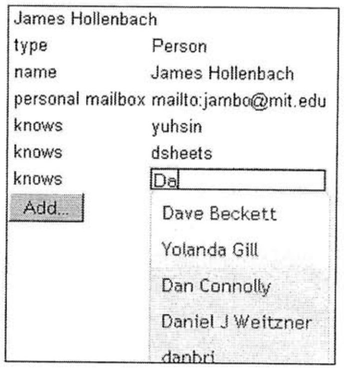
```
<div>
  <div sw:widget="rdflabel"
       sw:subject="http://dbpedia.org/resource/Elvis_Presley" >
  </div>
  <div sw:widget="rdfimage"
       sw:subject="http://dbpedia.org/resource/Elvis_Presley"
       sw:width="200">
  </div>
</div>
```

Figure 4-7: A `label` element and a `image` element for Elvis Presley, and the source code to produce it. The data for this display came from Elvis' data on DBPedia.

or fumble around with deciding which properties are valid for display–instead, such facts can be found in the RDF that has been loaded.

In addition to simply providing a means for displaying the current value of a triple in the local data store, the `edit` widget allows for data to be modified and sent back to a specified SPARQL endpoint. On its face, the `edit` widget looks like a normal `label`, but when an `edit` widget is selected, the user may click again in order to transform the label into an input box, much as files can be renamed when browsing a file system. Where possible, the widget will use an autocomplete menu to suggest user input for editing triples–for example, if a user is editing a `foaf:knows` relationship, as shown in Figure 4-8, metadata from the FOAF ontology can be used to suggest only people as possible input options (A discussion about which techniques are used for performing these inferences is provided in Section 6.3).

The `label` and `edit` widgets provide the basis for a large number of functions that can be accomplished by the library. Since they provide the most basic way of viewing and editing RDF data, they are wrapped by many other widgets, such as the `resource` and `selecttable` widgets described in the next section, to provide editing features for more complex collections of data.



James Hollenbach

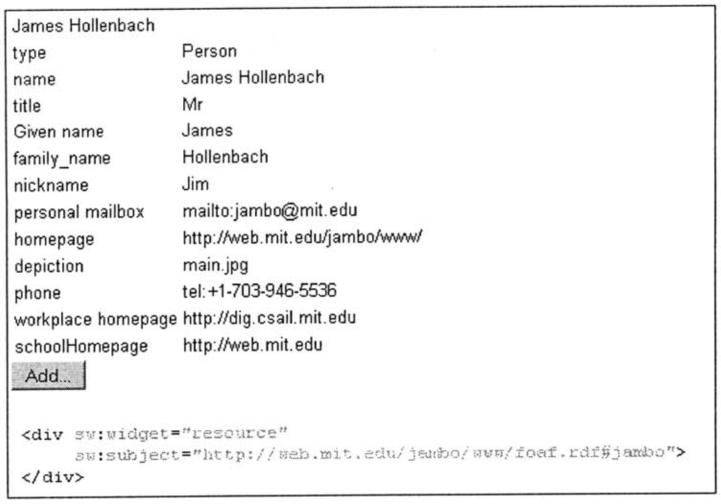
type	Person
name	James Hollenbach
personal mailbox	mailto:jambo@mit.edu
knows	yuhsin
knows	dsheets
knows	Da

Figure 4-8: Editing a property displayed in an `edit` widget. Note that since the `foaf:knows` property is being edited, only instances of `foaf:Person` are displayed.

4.3.2 Tables for Viewing and Editing Related Triples

From the basic `label` and `edit` widgets, it is possible to construct a wide variety of table-based widgets where the table cells are `label` and `edit` elements for specific triples. One such widget is the `resource` widget, shown in Figure 4-9, which very closely mimics the outline view from Tabulator. Given a resource's URI, the `resource` widget draws a table containing the predicate and object of each statement about that resource contained in the local store. Each predicate and object field can be edited, if a SPARQL endpoint is provided, and a button at the bottom of the widget allows for arbitrary new data to be inserted. For the amount of processing done, the code required to invoke a `resource` widget is very simple: the developer is only required to provide the subject URI that is to be described in the widget.

While exploring a single resource's properties is a convenient operation, there are a huge number of other potential methods for displaying RDF as tabular data. A separate widget, `selecttable`, demonstrates the flexibility provided by simply wrapping the `edit` class into a table. The `selecttable` widget performs what amounts to a SPARQL SELECT query over the local data store, and displays all of the matches discovered in a data table, where the rows represent the unique bindings found for the query. The example in Figure 4-10 shows how the `selecttable` widget supports fill-in-the-blank style updates. With the `requireall` option set to `false`, it is possible to find bindings where *at least one* of the variables selected by the query is bound.



James Hollenbach

type	Person
name	James Hollenbach
title	Mr
Given name	James
family_name	Hollenbach
nickname	Jim
personal mailbox	mailto:jambo@mit.edu
homepage	http://web.mit.edu/jambo/www/
depiction	main.jpg
phone	tel:+1-703-946-5536
workplace homepage	http://dig.csail.mit.edu
schoolHomepage	http://web.mit.edu

```
<div sw:widget="resource"
     sw:subject="http://web.mit.edu/jambo/www/foaf.rdf#jambo">
</div>
```

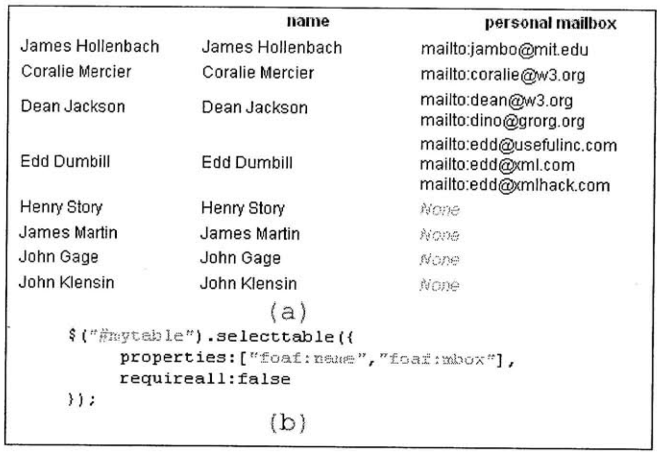
Figure 4-9: The `resource` widget. By default, the widget is editable. Clicking the "Add.." button produces a dialog for creating a new triple, and existing triples can be edited in place.

While the example in Figure 4-10 requests information about both names and e-mail addresses, it will display results where one of the two is unknown. If a SPARQL endpoint is provided, the user can then freely fill in missing data about e-mails or correct the spelling of a person's name on the fly.

Since tables are often an easy way of analyzing large amounts of data, the widget library offers several other widgets for different methods of displaying RDF in data tables, including the `triplelist` widget used in the introductory example for this chapter. A complete list of these widgets can be found in Table 4.2.

4.3.3 Widgets Based on Form Elements

While data tables and input boxes are useful for viewing and editing a wide variety of information, there are times when the set of inputs end-users may provide needs to be constrained to a small set of options. This includes situations where the set of acceptable options available to the user may not be immediately obvious, such as the question regarding a user's personal preferences in Figure 4-11. In these cases, special form elements such as the `radio` widget can be used to constrain the options presented to users as well as their options for how to modify that data (for example, a user cannot indicate that they have two different favorite bizarre FOAF properties).



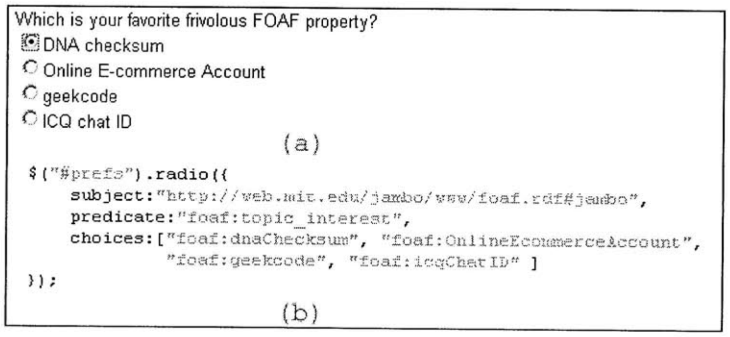
	name	personal mailbox
James Hollenbach	James Hollenbach	mailto:jambo@mit.edu
Coralie Mercier	Coralie Mercier	mailto:coralie@w3.org
Dean Jackson	Dean Jackson	mailto:dean@w3.org mailto:dino@grorg.org
Edd Dumbill	Edd Dumbill	mailto:edd@usefulinc.com mailto:edd@xml.com mailto:edd@xmlhack.com
Henry Story	Henry Story	*None*
James Martin	James Martin	*None*
John Gage	John Gage	*None*
John Klensin	John Klensin	*None*

(a)

```
$("#mytable").selecttable({
     properties:["foaf:name","foaf:mbox"],
     requireall:false
});
```

(b)

Figure 4-10: The `selecttable` widget. The output is shown in (a). Since the `requireall` option is set to `false`, incomplete matches are permitted. Each entry displaying *None* does not have a value, but if a SPARQL endpoint has been specified, then the missing values may be added in by the user. The source code for displaying this widget is shown in (b).



Which is your favorite frivolous FOAF property?

[✓ DNA checksum]
Online E-commerce Account
geekcode
ICQ chat ID

(a)

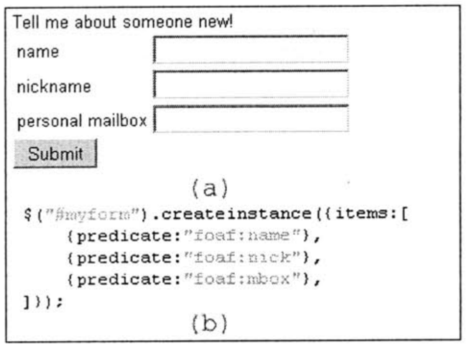
```
$("#prefs").radio({
    subject:"http://web.mit.edu/jambo/www/foaf.rdf#jambo",
    predicate:"foaf:topic_interest",
    choices:["foaf:dnaChecksum", "foaf:OnlineEcommerceAccount",
             "foaf:geekcode", "foaf:icqChatID" ]
});
```

(b)

Figure 4-11: The `radio` widget. In (a), The selected element represents the single value from the list that is currently present in the local data store. The source for this display is shown in (b).

Similar widgets provide this type of functionality with check boxes and combo boxes.

Sometimes, rather than simply editing data about an extant resource, the editing process involves the creation of data about a previously undocumented resource. In this case, it makes more sense to provide a form displaying all of the required fields for creating a new instance. In the widget library, this functionality is provided out of the box with the `createinstance` widget. The example shown in Figure 4-12 demonstrates how, given a list of input properties such as name, nickname, and e-mail, the `createinstance` widget automatically generates a form for creating a new description of a person. When the user submits the form, a SPARQL query is


Tell me about someone new!

name

nickname

personal mailbox

```
$("#myform").createinstance({items:[
    {predicate:"foaf:name"},
    {predicate:"foaf:nick"},
    {predicate:"foaf:mbox"},
]});
```

Figure 4-12: The `createinstance` widget. In (a), the form generated for collecting user input is shown. In (b), the source for the displayed form, each of the anonymous objects passed in the `items` array can contain additional information about how to process the user's input, in addition to the standard `predicate` field.

automatically generated to POST the form's contents to a SPARQL endpoint. The `createinstance` widget provides relatively limited options in terms of form layout by design, as it is intended only as a way to create extremely simple forms; for instance, the widget only allows for statements to be created regarding a single subject. The form generator described later provides a huge amount of freedom if a developer wishes to create a more involved form, at the cost of slightly more involved code. A discussion of the tradeoffs involved in providing more complex form descriptions is provided in Section 6.4.

4.4 Extending the Widget Library

While individual widgets are often helpful in creating a new Web site, there are many instances where the widgets provided by a library are not an exact match for the desired behaviors, layouts, and data filters demanded by an application. For this reason, the widget library is written to be easily extended with new widgets. Wherever possible, the opportunity is provided to access internal convenience functions for processing user input and displaying useful, human-readable output.

4.4.1 The `rdfwidget` Base Class

The jQuery UI interface widget library provides a base class for creating jQuery UI widgets. This class handles set-up and cleanup for widgets, as well as providing basic features such as namespaced events and CSS class names for the DOM nodes that are displaying a certain widget. This base class was wrapped and extended with the function `$.rdfwidget`, which provides additional convenience functions that are specific to operations involving RDF.

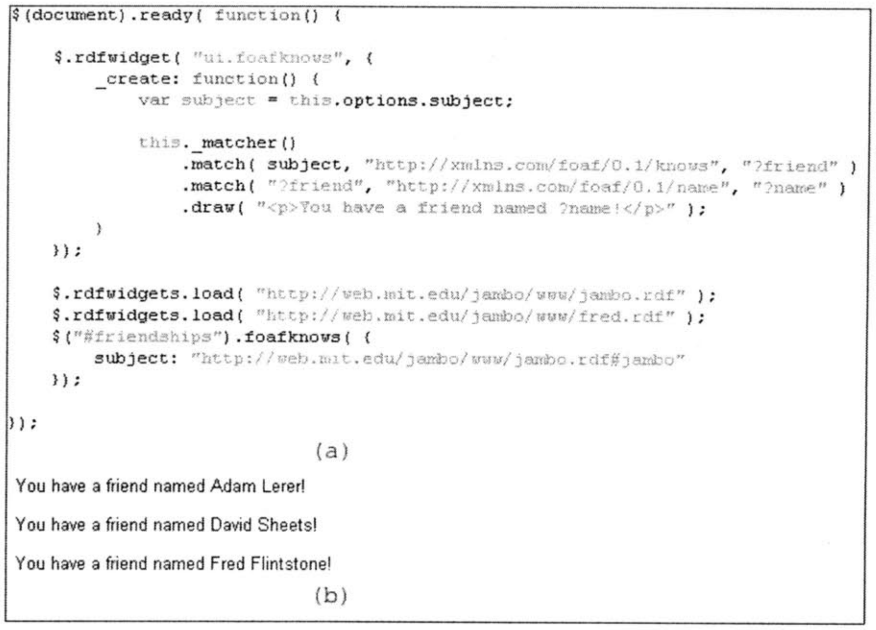
By creating a widget using the `rdfwidget` base class, a widget developer is guaranteed that all instances of the new widget will be notified when new data has been loaded into the local store[2]. The widget author simply needs to filter this data to decide whether or not it is relevant. Additionally, support is automatically provided for invoking the widget through HTML using the `sw:widget` attribute, exactly as other widgets in the library can be invoked. The next few sections briefly describe some of the core functions provided for handling data within widget code.

4.4.2 Matching, Templating, and Query-generation

The most powerful programming mechanism for widget authors is the matching mechanism provided with every widget instance, which closely mimics the matching system provided in rdfQuery. But whereas the rdfQuery matching system is geared primarily towards processing data in the local store, the same system has been extended in the widget library to collect user input and generate SPARQL queries to remote servers, in addition to providing the same data-filtering mechanisms on the local store.

Figure 4-13 shows the source code for a simple example widget, called `foafknows`. This widget, given a person in the `subject` field, will find all of the names of that person's friends and display them in a pretty-printed list. The output shown in Figure 4-13 demonstrates the output generated by the `foafknows` widget using the same data set as the introductory example in Section 4.1. Using nothing but the

[2]Discovering which data is relevant to a given widget can sometimes be a costly operation; the current implementation of the widget library does not attempt to distinguish relevant data automatically. See Section 6.5.3



```
$(document).ready( function() {

    $.rdfwidget( "ui.foafknows", {
        _create: function() {
            var subject = this.options.subject;

            this._matcher()
                .match( subject, "http://xmlns.com/foaf/0.1/knows", "?friend" )
                .match( "?friend", "http://xmlns.com/foaf/0.1/name", "?name" )
                .draw( "<p>You have a friend named ?name!</p>" );
        }
    });

    $.rdfwidgets.load( "http://web.mit.edu/jambo/www/jambo.rdf" );
    $.rdfwidgets.load( "http://web.mit.edu/jambo/www/fred.rdf" );
    $("#friendships").foafknows( {
        subject: "http://web.mit.edu/jambo/www/jambo.rdf#jambo"
    });

});
```

(a)

```
You have a friend named Adam Lerer!

You have a friend named David Sheets!

You have a friend named Fred Flintstone!
```

(b)

Figure 4-13: A custom widget created using the matcher object. The source code in (a) both creates the new widget `foafknows` and immediately loads data and creates an instance of the widget. Since the matcher instance will automatically refresh its output when new data is loaded, the output in (b) is produced when the two data sources have finished loading.

matching mechanism, this widget is able to provide clean, human-readable output based on the RDF data in the backend using about 4 lines of meaningful code.

The widget-internal `_matcher` function used in the example greatly facilitates the creation of new widgets while simultaneously ensuring the consistent behavior of widgets written by different authors. For example, by using a matcher generated with the `_matcher` function, the source filters provided when the widget was instantiated will be honored automatically. Additionally, this function will ensure that the widget automatically refreshes itself whenever new data is loaded into the internal store. These features allow widget authors working with the library to assume that their code will behave as other widgets in the library do.

The `match` function is used to identify triple patterns in the store similar to a SPARQL WHERE clause. Variables are defined as strings beginning with a question mark, such as the `?friend` and `?name` variables in the example. Sequences of calls to `match` are chained together, so the second match call in the example will build on top of the previous `match` to find the names of all of the subject's friends. The two `match` calls in the example are equivalent to the SPARQL query "`SELECT * WHERE { <subject> foaf:knows ?friend. ?friend foaf:name ?name.}`".

After a set of variables has been bound, the matcher's result set can be drawn using the `draw` function. This function takes in an HTML template string, and uses that string to draw individual representations of each set of bindings into the widget's DOM node. This HTML string can even contain widget definitions written in the widget library's HTML syntax. Bindings are inserted into the template with a regular expression; anywhere that a `?varname` string is found in the template, the binding for the variable `varname` is plugged in. This allows for the creation of a list of all bindings, as is shown for the `?friend` binding in the output of Figure 4-13. A similar function to `draw`, called `paginate`, allows for extremely long result sets to be automatically be divided into navigatable pages. This allows developers to cope with the potentially large number of bindings that can be generated by the matching process.

While matching and displaying matches alone is a powerful function, the fill-in-the-blank style of requesting data can be extended to provide other highly useful functions for widget authors. For example, as was noted in the previous section, the style used by the `match` function essentially amounts to creating a SPARQL WHERE clause. By turning the input generated by a developer into a WHERE clause, a CONSTRUCT query can be generated and sent to a remote SPARQL server. Figure 4-14 shows a SPARQL query generated by the library using the set of matches from the `foafknows` widget. By using this SPARQL query functionality, a widget author can generate a SPARQL query using the matching mechanism and the widget library will handle the request, parsing, and processing of the response graph sent by the server automatically.

4.4.3 Library Functions Based on Options

As was briefly alluded to in the previous section, one of the most helpful features of the widget library base class is that it provides functions that automatically follow the norms established by the pre-supplied widgets listed in Table 4.2. Without diving

```
$.rdfwidgets.matcher()
            .match( "<http://web.mit.edu/jambo/www/foaf.rdf#jambo",
                    "foaf:knows",
                    "?friend" )
            .match( "?friend",
                    "?pred",
                    "?obj" )
            .query( "http://www.example.com/sparql" );
```

(a)

```
CONSTRUCT {
   <http://web.mit.edu/jambo/www/foaf.rdf#jambo> foaf:knows ?friend.
   ?friend ?pred ?obj.
} WHERE {
   <http://web.mit.edu/jambo/www/foaf.rdf#jambo> foaf:knows ?friend.
   ?friend ?pred ?obj.
}
```

(b)

Figure 4-14: An example SPARQL query generated by the matcher. The call to query in (a) causes the response graph for the corresponding CONSTRUCT query in (b) to be loaded asynchronously.

into a list of all of the API functions available (such a list can be found in Appendix A, these functions enable widget authors to process user input with namespaces, create helpful autocomplete boxes, perform data filtering as the rest of the library does, and automatically generate SPARUL queries. A link to the full API documentation for the library, which lists and briefly describes all of these functions, is provided in Appendix A.

4.5 RDF Access Control for Named Graphs

As described earlier, the widget library provides mechanisms for interacting with a SPARQL endpoint that supports RDF Access Control on named graphs. As the concept of RDF Access Control based on FOAF+SSL is relatively new, no stable implementation of such a service for named graphs has come into existence yet–current versions only grant access to flat files. However, for this thesis, a proof-of-concept server implementing the RDF Access Control ontology for named graphs was built on top of ARC2 [25] to demonstrate a working instance of a SPARQL server with distributed access control.

4.5.1 Access Control Ontology

The access control library defines rules using the W3C ACL ontology [26]. A pair of example policies are provided in Figure 4-4. The three types of access that can be granted are `Read`, `Write`, and `Control`. The `Read` and `Write` modes control access to the named graphs stored on the server. `Control` is essentially a special form of `Write` access: having `Control` access to a graph allows a user to read and edit the ACL metadata for that graph. In addition to granting accesss to users, the access control library supports granting access more broadly to groups and classes of users. A special-case user class, `foaf:Agent`, is used to indicate that any authenticated agent may access a graph.

4.5.2 Flow of Authorization

When a user attempts to read or write a data on the SPARQL server, a series of SPARQL queries are executed in order to determine whether or not the currently defined set of access rules grants that user permission to perform the requested action. First, a master control graph is used to identify where access control data for the graph to be read or modified is located. The current implementation stores the access control for each graph in a separate meta-graph. If an access control graph is found, then the authentication process proceeds as expected: the server checks if the user is granted access by the user's identity or membership in a group or class.

If no access control graph is found, then there are two possible courses of action. First, the server checks to see if the user is granted `Control` access under the default set of rules, which are defined in a separate `default` access control graph–`Control` access must be provided by the defaults for a user to be able to create a new set of access rules for a new named graph. If the user is found to have `Control` access, then the server performs one of two operations: if the user has provided an ACL URI using the `acl` query parameter, then the rules found at that URI are loaded into the store as the new access control graph. Otherwise, the server simply copies the `default` graph into the new access control graph. Regardless of the action taken, if the user is granted permission to modify a graph based on the `default` rules, then a new access control graph will be created and the `default` graph will not be accessed for future operations on the target graph.

Chapter 5

Examples

This section provides two example applications that can edit and view Semantic Web data in the wild. In addition to live examples, the widget library's documentation includes a growing set of specific examples involving individual widgets and library functions. Further information about these examples, including links to the live versions, is provided in Appendix A.

5.1 RDF Annotator

5.1.1 Concept

Annotations are a commonly cited use case for the structure provided by the Semantic Web. Since the Semantic Web allows anything to have its own URI, an annotation program can provide a method for sharing information about any concept.

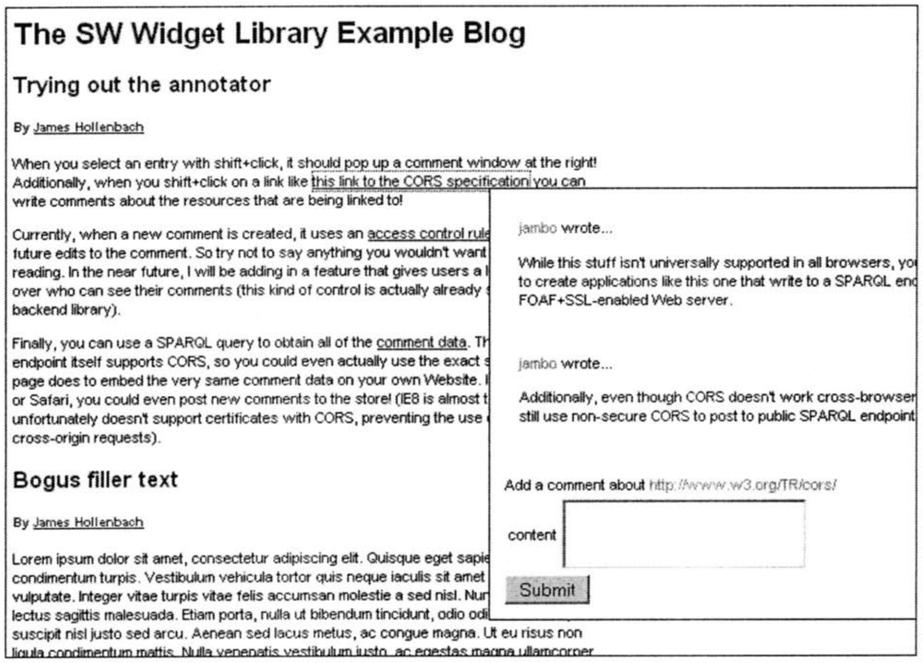
Of course, not all annotation information is necessarily meant to be public. Some annotations might be meant only for members of a certain group, such as comments about a closed meeting. Other annotations may even be strictly private, such as an annotation on a product noting that it would make a good Mother's Day present. A good annotation tool should allow for different levels of privacy to be set when annotations are written.

One of the greatest strengths of the Semantic Web is that it enables the description of arbitrary concepts in the body of any document on the Web. Particularly with RDFa, it is possible to embed RDF in an HTML document and create correspondences between certain DOM nodes and resources on the Semantic Web. Two different HTML documents that use the same data sources for annotations can display the same set of annotations describing any resource on the Semantic Web, and features like smushing can allow for an application to realize that two sets of annotations are actually about the same thing.

There have been several attempts at creating universal annotation services similar to the one described here. Annotea [3], an INRIA and W3C project, uses RDF to accomplish this task by interacting with distributed annotation servers. By design, Annotea is intended to operate on top of a user's browser, rather than operating within any single Website. This creates a high activation energy for using Annotea, since it specifically requires that the user install software in order to use it. At the same time, Annotea's privileged status as a client-side application makes it easy to ensure consistency as to which annotation servers are used across different Web sites. Social bookmarking sites like `del.icio.us` provide less structured ways of sharing and annotating links, but at the same time are limited in that they are centralized services that do not immediately link into the Semantic Web. The application described here tries to land somewhere in between these two extremes by providing a distributed service like Annotea that only allows for very simple textual annotations.

5.1.2 The Annotator Application

The widget library's `annotator` widget provides a compact, simplified method for putting Semantic Web annotations on any Web site. As is shown in Figure 5-1, the `annotator` widget in its simplest form simply pops up over any valid resource when the user clicks on the page while holding the shift key. This causes the annotator to search for a valid URI to create an annotation about–either the `href` element of an anchor tag, or a subject defined somewhere in the page's RDFa. Once this URI is found, the annotator looks in the set of annotations that have been pulled down for any available comments about the selected resource and displays those comments to



The SW Widget Library Example Blog

Trying out the annotator

By James Hollenbach

When you select an entry with shift+click, it should pop up a comment window at the right!
Additionally, when you shift+click on a link like this link to the CORS specification you can
write comments about the resources that are being linked to!

Currently, when a new comment is created, it uses an access control rul
future edits to the comment. So try not to say anything you wouldn't want
reading. In the near future, I will be adding in a feature that gives users a
over who can see their comments (this kind of control is actually already
backend library).

Finally, you can use a SPARQL query to obtain all of the comment data. Th
endpoint itself supports CORS, so you could even actually use the exact
page does to embed the very same comment data on your own Website.
or Safari, you could even post new comments to the store! (IE8 is almost t
unfortunately doesn't support certificates with CORS, preventing the use
cross-origin requests).

Bogus filler text

By James Hollenbach

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Quisque eget sapi
condimentum turpis. Vestibulum vehicula tortor quis neque iaculis sit amet
vulputate. Integer vitae turpis vitae felis accumsan molestie a sed nisl. Nu
lectus sagittis malesuada. Etiam porta, nulla ut bibendum tincidunt, odio od
suscipit nisl justo sed arcu. Aenean sed lacus metus, ac congue magna. Ut eu risus non
ligula condimentum mattis. Nulla venenatis vestibulum justo, ac egestas magna ullamcorper

jambo wrote...

While this stuff isn't universally supported in all browsers, yo
to create applications like this one that write to a SPARQL en
FOAF+SSL-enabled Web server.

jambo wrote...

Additionally, even though CORS doesn't work cross-browse
still use non-secure CORS to post to public SPARQL endpoint

Add a comment about http://www.w3.org/TR/cors/

content

Submit

```
$(document.body).annotator({endpoint:"http://www.example.com/sparql"});
```

Figure 5-1: A running instance of the annotator application, and the Javascript code used to instantiate it. Here, the user has selected a link to the CORS specification, triggering the known annotations regarding the specification to be displayed.

the user, as well as providing an input box for submitting a new annotation about the resource.

Since the widget library is capable of relying on external data sources via CORS, the annotator can be pointed at an arbitrary number of annotation sources[1]. All of these sources will be aggregated automatically in the library's unified store and displayed in a single, coherent interface. Barring certain restrictions due to the continued development of the CORS specification[2], it is also possible to request resources that are potentially secured by the RDF Access Control system. Since the widget library relies on the browser's internal systems for performing FOAF+SSL authentication, the comments displayed to the user will be automatically filtered depending on their identity.

To create a new annotation, a user simply types their annotation into the input box and presses submit. If the submission requires the use of FOAF+SSL and the user has not yet authenticated with the server, their certificate will be requested. Finally, at the server's discretion, the user may or may not be allowed to submit the new comment; accordingly, the widget library provides the appropriate feedback to the user about the result of their submission. The `annotator` widget also allows developers to specify the access control policy to be used with new comments.

The number of annotations to be displayed can potentially be huge. To circumvent issues with never-ending lists of comments, the `annotator` widget makes use of the built-in `paginate` function described earlier to paginate the comments it finds when the library's internal store is queried for annotations.

5.1.3 Benefits of the Annotator

The widget library's `annotator` widget provides all of the benefits one would expect from an annotator on the Semantic Web. Perhaps most importantly, it allows developers on different Web sites to use exactly the same data set when displaying

[1]One open challenge not solved by the annotator application is helping developers choose which annotation sources would be appropriate for inclusion on their pages.

[2]A full description of the current state of CORS in major browsers is provided in Section 6.2

annotations. If a user creates an annotation about a resource on one page, that same annotation will appear when the user attempts to view annotations about that resource on a completely different page. This ease of data-sharing between sites is the hallmark of the Semantic Web.

The code for creating an `annotator` is simple–the developer simply has to point the annotator at an annotation source and a SPARQL endpoint. The entire process involves a few script tags and the single line of Javascript code shown in Figure 5-1. The annotator can layer on top of virtually any existing Web site to enrich it with data written all over the Web.

Finally, the RDF Access Control library, if used, provides the privacy and security that one would expect from any Web application, and therefore the privacy and security that the Semantic Web must also provide–for example, allowing for a blog post or comment to be shared only between friends. It may be the case that many users never write a single private annotation. Nevertheless, having the option to do so gives the user confidence that the system they are using has been created with privacy in mind–it is simply necessary to make sure that users are aware that they have such an option. Section 6.5.5 discusses methods for integrating this ACL data into widget displays.

The Annotator application shows how a simple application can take advantage of Semantic Web technologies to enrich the content describing any resource on the Web. A universal annotator that can be simply tacked onto an already-existing Website provides an easy and useful way to bring the Semantic Web to Websites that do not have any Semantic Web infrastructure of their own.

5.2 Peel Sessions Browser

5.2.1 Concept

In addition to simple tools like the annotator, it is equally important to support the creation of large-scale applications that take advantage of data from multiple sources on the Semantic Web to create a single, unified interface for exploring information. This example demonstrates the widget library's capability of supporting such applications with an example based in music.

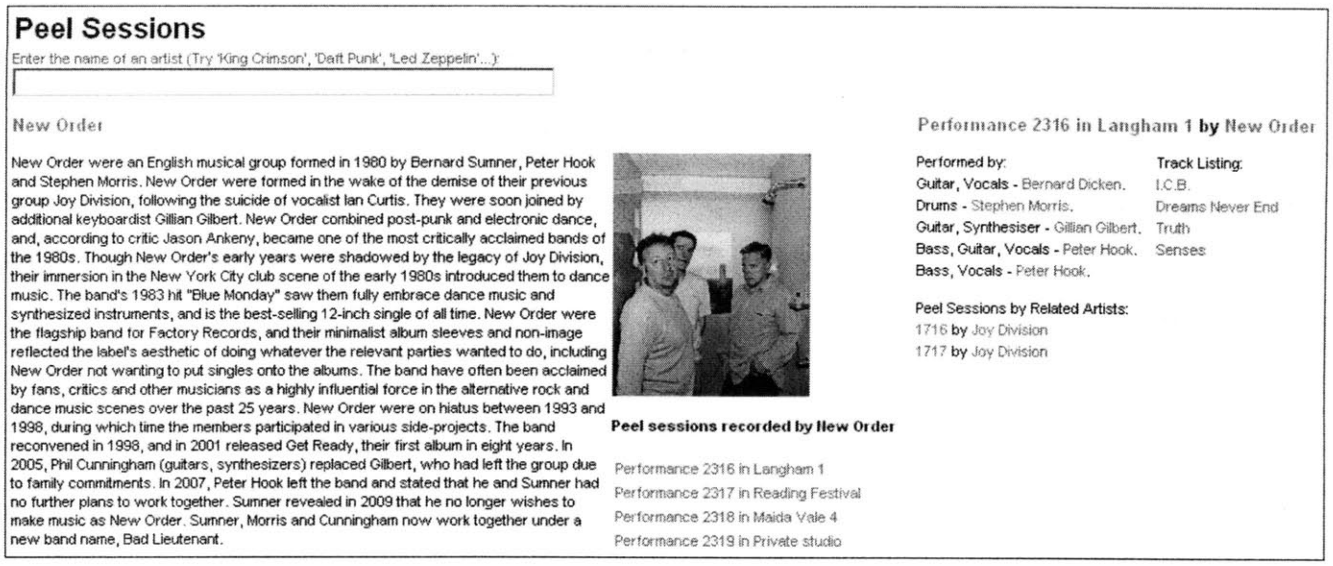
An application was created for exploring the links between musicians who have recorded sessions with the late BBC DJ John Peel. Peel recorded sessions in the BBC's studios with thousands of musicians, each featuring a small number of tracks–usually 3 to 5. In 2007, the BBC released all of its data about these sessions as RDF [19], which was eventually augmented with `owl:sameAs` links to artists, songs, and musicians on DBPedia.

The Peel data demonstrates a fine opportunity to link between multiple data sets in order to discover more about musicians. While the Peel data is decidedly sparse–it rarely features more than the track listings, artist names, and musician names–the information made available by following the `sameAs` links into DBPedia and outward from there is enormous. This application provides a way to link in the Peel data with the wealth of data freely available on the Semantic Web in an intuitive user interface. Furthermore, it promises the opportunity to develop an interesting Web application that could not easily exist without the Semantic Web, but nevertheless provides no indication from its interface that it relies on URIs, OWL, or SPARQL. Finally, much like the annotation example, it provides a way to write interesting applications without owning any Semantic Web infrastructure.

5.2.2 The Peel Application

The main focus of the Peel application is a description of the artist, shown at the left of Figure 5-2. The artist view takes full advantage of the widget library's features for creating custom layouts with the `match` function, as well as making use of the `label` widget for displaying the artist's name, the `image` widget for displaying the artist's picture, and the `triplelist` widget for displaying the list of Peel sessions recorded by the artist.

Even though the interface uses a custom layout, it still benefits from the automatic update mechanisms provided by the widget library through the widgets it uses


Peel Sessions

Enter the name of an artist (Try 'King Crimson', 'Daft Punk', 'Led Zeppelin'...):

New Order

New Order were an English musical group formed in 1980 by Bernard Sumner, Peter Hook and Stephen Morris. New Order were formed in the wake of the demise of their previous group Joy Division, following the suicide of vocalist Ian Curtis. They were soon joined by additional keyboardist Gillian Gilbert. New Order combined post-punk and electronic dance, and, according to critic Jason Ankeny, became one of the most critically acclaimed bands of the 1980s. Though New Order's early years were shadowed by the legacy of Joy Division, their immersion in the New York City club scene of the early 1980s introduced them to dance music. The band's 1983 hit "Blue Monday" saw them fully embrace dance music and synthesized instruments, and is the best-selling 12-inch single of all time. New Order were the flagship band for Factory Records, and their minimalist album sleeves and non-image reflected the label's aesthetic of doing whatever the relevant parties wanted to do, including New Order not wanting to put singles onto the albums. The band have often been acclaimed by fans, critics and other musicians as a highly influential force in the alternative rock and dance music scenes over the past 25 years. New Order were on hiatus between 1993 and 1998, during which time the members participated in various side-projects. The band reconvened in 1998, and in 2001 released Get Ready, their first album in eight years. In 2005, Phil Cunningham (guitars, synthesizers) replaced Gilbert, who had left the group due to family commitments. In 2007, Peter Hook left the band and stated that he and Sumner had no further plans to work together. Sumner revealed in 2009 that he no longer wishes to make music as New Order. Sumner, Morris and Cunningham now work together under a new band name, Bad Lieutenant.

Peel sessions recorded by New Order

Performance 2316 in Langham 1
Performance 2317 in Reading Festival
Performance 2318 in Maida Vale 4
Performance 2319 in Private studio

Performance 2316 in Langham 1 **by** New Order

Performed by:	Track Listing:
Guitar, Vocals - Bernard Dicken.	I.C.B.
Drums - Stephen Morris.	Dreams Never End
Guitar, Synthesiser - Gillian Gilbert.	Truth
Bass, Guitar, Vocals - Peter Hook.	Senses
Bass, Vocals - Peter Hook.	

Peel Sessions by Related Artists:
1716 by Joy Division
1717 by Joy Division

Figure 5-2: The John Peel application. Left, a summary of the currently chosen artist is displayed. Right, a description of the currently chosen Peel Session for that artist is displayed. Related artists are determined by combining Peel session data with information about related artists on DBPedia. These connections are readily attained by smushing.

and through the `match` function. When the user selects an artist from the autocomplete menu at the top of the page–which is itself a `instancedropdown` widget–the application requests data from the Peel database about the artist. If this data contains any `owl:sameAs` relationships, those resources are pulled down as well. As each data source loads, the artist display is updated with new information. This causes a trickling effect where the artist's biography appears, followed by its Peel sessions, and finally a photograph pulled down from DBPedia.

When the user selects a Peel session from a given artist's list of sessions, the `rdfselect` event fires. This is caught by the application, which then requests the data for that session from the Peel database and displays it in another custom widget, shown at the right of Figure 5-2.

The data from DBPedia provides even more information about artists that share some relationship with the selected artist. This allows for the application to display a list of related artists, shown in the lower right corner of Figure 5-2. In this way, links are connected both from the Peel data to an external source, and then form a loop where even more information is learned about the resources within the Peel data as a result of the external source. These artists can then be selected in order to view information about their respective Peel sessions, allowing for a browsing experience that flows back and forth between artists and sessions.

5.2.3 Benefits of the Peel Application

First and foremost, the Peel example demonstrates the ease with which data can be aggregated and equated using the widget library. While the data from DBPedia does not provide any explicit links back to the Peel data, it can nevertheless be used to discover information about relationships between artists described in the Peel database. Applications like the Peel application allow for information to be discovered that is not immediately attained from any single data source.

Additionally, the Peel example shows that it is possible to create a larger-scale application that makes use of many of the functions the widget library provides. The Peel example uses five different core widgets, defines a custom layout with the `draw` function, and makes use of the `rdfselect` event to drive interaction between the artist and session views. Finally, it shows an instance where the trickle-in loading effect gives the user instant feedback that data is being loaded, despite the fact that some larger, slower data sources (namely, DBPedia), take a while to respond.

For the user's sake, the Peel application provides no indication that the application being used is driven by the Semantic Web. There are no outward-facing URIs and no clunky labels like "family_name". The layout is not a simple table layout–instead, the data is arranged in a way that makes sense for this specific dataset. The amount of effort required for the developer to achieve this consistently is relatively small–in fact, the entire Peel Session application can even be wrapped into its own widget and redistributed for use on other sites. As the number of applications created using the same library grows, the wealth of custom widgets geared toward specific applications–such as business cards, schedules, photojournals, and album summaries–will continue to make this process even easier.

Chapter 6

Discussion

Throughout the initial authoring of the widget library, there were many tradeoffs made and limitations reached. This chapter provides a discussion of the major tradeoffs made in the creation of the widget library, the difficulties met in dealing with the limitations of developing a Javascript-only library, and several potential paths for future development related to the widget library.

6.1 Complexity and Expressivity

One of the biggest tradeoffs in the creation of an interface library involves its complexity. The Semantic Web is inherently complex, and developing with it can be confusing for developers who are not used to everything having a URI or unfamiliar with common ontologies. Many developers are impatient to learn an entirely new system to develop an application that may not provide much improvement over a mashup developed with XML data. It is therefore at times prudent to develop a widget library that obscures as much of the Semantic Web as possible–even from developers.

Exhibit is a prime example of a tool that does this. While Exhibit is fully capable of handling RDF data, it is absolutely not a requirement; Exhibit users can import data from CSV files and quickly build a rich application with maps, timelines, and faceted browsing in literally minutes. However, applications like Exhibit trade away the ability to write truly varied applications in exchange for a simple programming model. While RDF can be imported into Exhibit, the semantics associated with that data will generally not be used to group together relevant data. Additionally, the Exhibit framework tends to force the faceted browsing paradigm on developers who use it. This can often be a blessing–Exhibit makes it possible for otherwise novice developers (or even those who have no programming skill) to develop slick, fast-operating Web sites based around structured data. However, there are often cases where more skilled developers want many of the great features provided by Exhibit, but nevertheless wish to break free of some of the design paradigms it imposes.

At the same time, the ability to lay out documents for development with RDF can be taken to the opposite extreme, where entirely too much expressivity is provided. This seemingly is the issue that has caused Fresnel [18] to never take off. Fresnel was originally intended to be a complete layout engine for describing how to display pieces of RDF in Fresnel "lenses". While the level of expressivity granted by Fresnel is unparalleled–it allows all kinds of filtering based on query paths, the definition of unique shapes for enclosing data, and much more–seemingly nobody took interest in learning how to describe the display of RDF data *in RDF*, as Fresnel requires. The learning curve for developing with Fresnel was simply too steep.

The widget library situates itself somewhere in between these two paths. Programming with the widget library is certainly nowhere near as simplified as Exhibit, but this trade in simplicity allows for relatively complete freedom in terms of layout and information processing. The widget library hides many aspects of live Semantic Web browsing, such as smushing, in places that novice developers need not look, but still leaves the option open for those who are willing to get their hands dirty. This decision allows for very simple applications, such as the annotator, which essentially requires a single line of Javascript, to come out of the same library as the Peel example, a full-fledged application that takes advantage of OWL, SPARQL, and Javascript event interactions. A good library should provide easy ways for new developers to get started, but simultaneously provide methods for heavy customization.

6.2 Cross-Browser Security and CORS

The CORS specification promises to greatly enhance the sharing of data between Web applications directly within the browser, in addition to bringing an end to the use of formats like JSONP to circumvent browser security. However, the CORS specification has yet to be finalized, and even when it is finalized, it will presumably face a long road to cross-browser reliability. This will most likely continue to pose a challenge to Semantic Web applications that aim to work securely in the browser for some time.

Current implementations of the CORS specification, while all functional for non-secured connections, are relatively inconsistent when dealing with certificates–particularly with the self-signed certificates that are the norm with FOAF+SSL. Ideally, all browsers will eventually support the use of certificates with CORS and allow users to see which remote resources are being accessed using the user's identity to retrieve data.

Currently, the latest version of Firefox and Safari support FOAF+SSL under CORS, but both require the user to wrangle with ominous warning screens about the security of certificates. In Safari in particular, the user must repeat the process of permitting secure communications with a particular server every time the browser is restarted. Internet Explorer simply does not support secured connections with CORS, due to security concerns. Future versions of these browsers should accept the use of self-signed certificates as identification, and provide users at least with the option to eliminate cumbersome warning screens.

If Javascript-based applications built around the Semantic Web become more popular, and FOAF+SSL or other single-identity systems gain a bigger foothold on the Web, then there may yet be a stronger push for better identity management in cross-domain interactions in all of the major browsers, enabling FOAF+SSL in this environment.

6.3 Level of Inference

As was mentioned in previous sections, the widget library provides functionality for performing autocompletion on `edit` widgets and others when they are being edited. This can be used, for example, to provide a list of only people when a person is editing a `foaf:knows` attribute. While autocompletion is highly useful to users, and the structure of the Semantic Web should seemingly make it easy to decide which options to show, there are actually several different methods for performing inference and filtering user options that all have varying benefits depending on the amount of data available.

One of the fundamental issues underlying the selection of options to display is the "open world" principle underlying the Semantic Web. This principle implies that, unless explicitly specified with a property such as `owl:disjointWith`, it is impossible to know whether or not a given instance is an instance of a specific class. For example, the description of a class `ex:Man` may or may not contain information such as an `rdfs:subClassOf` property relating that class to `foaf:Person`. However, the mere absence of this information does not imply that an instance of `ex:Man` is not a `foaf:Person`. Therefore, failing to present a list of Men as options for autocompletion of the `foaf:knows` property, which has `rdfs:range foaf:Person`, might confuse developers and their users who know that all `ex:Man` instances are in fact people.

In an ideal world, even very simple `owl:disjointWith` and `rdfs:subClassOf` properties, and an `rdf:type` property for instances, would greatly facilitate the distinction between people, places, and documents. Thus, users who are asked to name an employee, rather than seeing a list of only `Employee` instances, could see a list of all known people, with places and documents removed since people, places, and documents are known to be disjoint. This is the only truly correct way to eliminate instance options from appearing in autocomplete menus.

In the absence of such globally-accepted and used basic properties, it is still important from a user interface standpoint to provide useful, uncluttered autocomplete options to the user. To this end, the widget library makes use of RDFS properties as *hints* as to which instances would make good menu choices. Note, however, that this behavior is not ideal, as it will explicitly miss instances of `ex:Man` from the earlier example if an `rdfs:subClassOf` property is not provided. Nevertheless, in the absence of widespread OWL ontology information, these RDFS hints can provide useful filtering for options to eliminate strange behaviors such as displaying locations when editing a `foaf:knows` property. If desired, this processing can always be switched off, causing all resources in the local store to be displayed during autocompletion.

However, the simple truth is that for many ontologies, little to no data useful for inference is provided; there is no definition of domain and range, and there may not even be an ontology defined at the specified URI at all. This presents a different problem for autocomplete functionality: how can useful options be provided to the user in the absence of any metadata? One potential answer, and the one used by the widget library, is to simply provide options that are used in existing data–for example, if there is a property called `author` from an unknown ontology that the user is trying to edit, the library displays all of the object matches for the pattern `?subject author ?object` in the local store. In this way, the user still gets a reasonable set of options despite the lack of an ontology. As already mentioned, another possible option always supported by the widget library is to display all known resources as options.

Within the widget library, facilitating user input is a primary goal; it is therefore important to always provide some sort of hint as to what input the user can provide for a given field. When available, the widget library takes advantage of RDFS metadata to pick out options that can obey range and domain restrictions. It also uses OWL for performing "smushing", using the Tabulator Library. However, in the absence of either of these, useful options can still be produced by inspecting the data available and reproducing options that are already present in the data.

6.4 Form Complexity

Providing a simple language for defining the input users can provide in forms can be a somewhat messy task; it is easy to get caught up in the seemingly endless number of possible data structures that a developer may want users to input. A large amount of work has been done both within and outside of the Semantic Web community to develop robust form languages for obtaining user input through forms. For the widget library, it was seen as advantageous to avoid complexity in favor of a simple method for creating new instance data.

It is possible to use OWL and other languages to constrain a user's options when creating new data about a resource–for example, allowing a person to only enter a single spouse, mother, or father. While such an implementation may be advantageous when OWL ontologies and the like are well-defined, these ontologies often do not exist or do not always provide adequate information required to create a specific form–for example, a developer may wish to combine data from two domains, but does not necessarily have the expertise to write their own OWL to reconcile the different required by the two ontologies. Such form models can often be too cumbersome for novice users.

Languages like XForms [8] go even further to promise effectively infinite expressivity in the creation of new form data; however, languages like XForms are very clearly too complex for practical use by unexperienced developers. For this reason, they were completely discarded as the basis for form creation in the widget library.

Instead of going with more complex systems, the widget library provides a simple method for defining forms: the `createinstance` widget. The main goal of the widget is to provide a very simple method for creating new linked data. The `createinstance` widget is a very simple interface for creating statements about a single resource–it allows the developer to provide mandatory and optional fields for the form, and all of the fields describe properties of a single object. By heavily limiting the options for what type of input can be provided, the `createinstance` interface stays simple. An extension to the matcher library, proposed in the next section, would allow for more complex forms to be created. By maintaining consistency across multiple aspects of the library with the matcher, teaching users to create complex forms in the widget library would not add too much of an additional learning curve.

6.5 Future Work

6.5.1 Linking in with Other Libraries

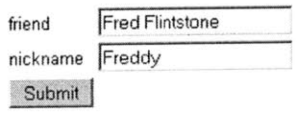
The widgets provided in the widget library are geared towards the viewing and editing of textual data. This decision was made largely on purpose; several libraries, including Exhibit and RPI's many GoogleViz visualizations [22] already provide highly sophisticated methods for browsing data in maps, charts, graphs, and timelines. While it would be a waste of time to try and reimplement these features, it would be extremely useful to link these visualization libraries up with the widget library by some sort of wrapper widget. This widget could, for example, funnel location data about a set of resources into Exhibit for display on a map, or funnel statistical data into the RPI visualization code to produce a pie chart. By using the widget library as the primary layer for processing the RDF coming from the Web, all of the benefits of live updating and editing provided by the widget library are maintained while also gaining the power of these excellent graphical libraries.

6.5.2 Custom Form Generation Using the Matcher

In Section 4.4.2, a mechanism was described for generating remote SPARQL queries using the widget library's matcher functionality. Similarly, the matcher functionality could be used to create a form requesting that the user fill in all of the variables defined the WHERE pattern of the query. Figure 6-1 depicts an example where a matching function for books is used to generate a form for entering the titles of new books into a system. This mechanism would allow for the construction of much more complex triple patterns than the single-subject forms generated by the `createinstance` widget. When the user submits the form, the widget library automatically would

```
$.rdfwidgets.matcher()
            .match( "<http://web.mit.edu/jambo/www/foaf.rdf#jambo",
                    "foaf:knows",
                    "?friend" )
            .match( "?friend",
                    "foaf:nick",
                    "?nickname" )
            .form( "http://www.example.com/sparql" );
```

(a)



friend	Fred Flintstone
nickname	Freddy

(b)

```
INSERT {
   <http://web.mit.edu/jambo/www/foaf.rdf#jambo> foaf:knows <http://web.mit.edu/jambo/www/fred.rdf#fred>.
   <http://web.mit.edu/jambo/www/fred.rdf#fred> foaf:nick "Freddy".
}
```

(c)

Figure 6-1: A form generated with the proposed matcher extension. Each variable used in the matcher pattern is requested in the user input. Depending on their context, certain variables, such as `friend`, will have autocomplete functionality based on RDFS information.

automatically generate the appropriate SPARUL INSERT query and send the entire pattern generated by the matcher, with the user-provided bindings, to a target SPARQL endpoint.

6.5.3 Optimization of Update Processing

Currently, individual widgets do little to no processing to determine whether or not an update to the local data store requires the widget to redraw. This means that whenever new data is loaded, the vast majority of widgets refresh on the spot, even if the change was the result of a single triple being updated by a user-driven edit.

There are a number of different ways to combat this problem, which vary inversely in their granularity and the amount of time spent determining if an update is relevant. Since DOM operations largely dominate the work involved in redrawing widgets, refreshing is most expensive for widgets like the `resource` widget, which can potentially display several hundred triples related to a single subject. Simply reading through each triple in the new data and picking out those relevant to the `resource` widget, in this case, will turn out to be much faster if the number of triples to be inserted is small. However, if the number of triples to be inserted is large, this operation will actually be slower than simply redrawing, since the `resource` widget code is highly optimized for the initial draw. Thus, there are non-linear tradeoffs in determining the redraw strategy of each widget that makes defining a single "best" redraw strategy difficult. While there may not be a single best redraw strategy, it is still fair to say that some widgets can be better optimized for high-frequency updates.

6.5.4 SPARQL Diffs

Currently, there is no standardized or even widely used method for sending out a Comet-like [23] SPARQL query that can receive updates to the remote triple store in the form of live diffs. This seems to be a major missing piece in the creation of Semantic Web applications that can fully compete with traditional Web applications–for example, extending the annotator example to stream live comments about a given resource on the Web would allow for the creation of a highly interactive service that operates on many different Websites, yet displays the same data set on all of them. This would allow for users to share information about how different Web sites are portraying the same resources, since annotations could contain links back to the location that a given resource was edited from.

Open Anzo [17] aims to provide a Javascript API for receiving diffs from a triple store, but presently it uses a custom library and does not seem to work with SPARQL queries. Even so, it may be possible to create a wrapper for OpenAnzo or other streaming servers that streams data into the widget library's internal store.

6.5.5 ACL Integration in Widget Interfaces

As was seen in the annotator example, it is possible for data to be created along with an associated access control policy for future use. However, when data is displayed back to the user, there is no information provided as to why they are allowed to see a given piece of data. This can cause user confusion when two different users do not see the same dataset on the same Website, and can even cause users to disclose information that they may not otherwise make public. For example, a user might write a public reply to a private annotation simply because they did not realize it was private, thereby disclosing information that they may otherwise not have mentioned. There are several approaches that can help mitigate this risk. One method makes use of the `link rel=meta` HTTP header to identify the location of the access control graph, pull it down, and display the access control data that grants a user access. This behavior would be similar to the policy processing and display used in the Tabulator Justification UI. Alternatively, the server could generate some sort of metadata as a part of the response to the SPARQL query that provides a plain-text string or proof tree describing why the user was granted access. This information could then be displayed along with the annotation to help the user understand why they are seeing a certain piece of data.

Chapter 7

Contributions

To ensure the sustained growth of the Semantic Web, powerful, exciting applications must be developed that could not exist without the Semantic Web. The creation and improvement of tools for working with Semantic Web data will greatly speed up the rate at which such applications are developed. To that end, this thesis has:

- **Defined a set of widgets for viewing, creating, and editing Semantic Web data.** The widgets defined allow for the creation of interfaces with common user interface paradigms that can automatically push user-generated content onto the Web. These widgets not only save developers time when creating applications that use Semantic Web data; they also ensure that end-users are not unnecessarily exposed to the inner workings of the Semantic Web, which fundamentally are not of interest to them.

- **Shown that the Widget Library can be extended to fit new needs.** Providing custom template, form, and query generation and a solid base of fundamental editing widgets ensures that developers will be able to create interfaces that are specifically tailored to their domain. Some services may require widgets that looks like business cards, while others need full descriptions of prescription drugs and their side effects. The huge variation in the data available on the Semantic Web precludes the possibility of a one-size-fits-all solution; instead, the Widget Library provides developers with the flexibility needed to build custom interfaces.

- **Created a server-side application for managing access to Semantic Web data stored in named graphs.** Providing meaningful services to users often implies the need for identification; certainly, at some point, many Semantic Web applications will generate user-specific data. A good widget library will therefore need to be able to understand the notion of access control policies used in conjunction with user-generated data. The access control policies currently provided are intentionally simple, but the system can be extended arbitrarily while still supporting the policies described here.

- **Demonstrated the functionality of the Widget Library with a pair of examples.** The annotation example shows how different developers can make use of the same publicly available information to create dynamic content on their own Web site. In a world where users, blog posts, and grocery store items all have URIs, the possibility for community review is endless. The Peel session example demonstrates how developers can pull together a large number of data sets simply by allowing the widget library to crawl between data sources, fulfilling the Semantic Web's promise of essentially free interoperability.

As the amount of data available on the Semantic Web continues to grow, the opportunity for gathering meaningful data simply by exploring links will continue to grow. The Semantic Web Widget Library is a powerful tool that allows developers to work together to create increasingly useful and unique distributed applications on the Semantic Web.

Appendix A

Links to Source Code, Documentation, and Examples

The main page for the project, which includes a brief introduction to coding with the widget library, download links to the project source, and links to all other resources, can be found at:

`http://dig.csail.mit.edu/2010/rdf-widgets/`

Code documentation, including examples for each widget, tutorials for writing new widgets and using the main library functions, and maintenance information for future project contributors can be found at:

`http://dig.csail.mit.edu/2010/rdf-widgets/docs/`

Finally, the examples from Chapter 5 are both available online:

Annotator:

`http://dig.csail.mit.edu/2010/rdf-widgets/examples/annotator.html`

John Peel :

`http://dig.csail.mit.edu/2010/rdf-widgets/examples/johnpeel.html`

Bibliography

[1] B. Adida, M. Birbeck, S. McCarron, and S. Pemberton. Rdfa in xhtml: Syntax and processing. `http://www.w3.org/TR/rdfa-syntax/`, 2008.

[2] K. Alexander. Morph: Talis semantic web formats converter. `http://convert.test.talis.com/`, 2009.

[3] Annotea project. `http://www.w3.org/2001/Annotea/`, 2001.

[4] Soren Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, and Zachary Ives. Dbpedia: A nucleus for a web of open data. In *In 6th International Semantic Web Conference, Busan, Korea*, pages 11–15. Springer, 2007.

[5] T. Berners-Lee, J. Hollenbach, Kanghao Lu, J. Presbrey, and mc schraefel. Tabulator redux: Browsing and writing linked data. `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.112.1782`, 2007.

[6] Tim Berners-lee, Yushin Chen, Lydia Chilton, Dan Connolly, Ruth Dhanaraj, James Hollenbach, Adam Lerer, and David Sheets. Tabulator: Exploring and analyzing linked data on the semantic web. In *In Proceedings of the 3rd International Semantic Web User Interaction Workshop*, 2006.

[7] C. Bizer. Disco hyperdata browser. `http://www4.wiwiss.fu-berlin.de/bizer/ng4j/disco/`, 2007.

[8] J. Boyer. Xforms 1.1. `http://www.w3.org/TR/xforms11/`, 2009.

[9] M. Hanson, D. Mills, A. Raskin, and A. Faaborg. Mozillawiki: Account manager. `https://wiki.mozilla.org/Labs/Weave/Identity/Account_Manager`, 2009.

[10] M. Hausenblas. pushback: Write data back from rdf to non-rdf sources. `http://code.google.com/p/pushback/`, 2009.

[11] J. Hollenbach, J. Presbrey, and T. Berners-Lee. Using rdf metadata to enable access control on the social semantic web. In *Proceedings of the Workshop on Collaborative Construction, Management and Linking of Structured Knowledge (CK2009)*, 2009.

[12] David F. Huynh, David R. Karger, and Robert C. Miller. Exhibit: lightweight structured data publishing. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 737–746, New York, NY, USA, 2007. ACM.

[13] B. Ippolito. Remote json: Jsonp. `http://bob.pythonmac.org/archives/2005/12/05/remote-json-jsonp/`, 2005.

[14] David R. Karger, Scott Ostler, and Ryan Lee. The web page as a wysiwyg end-user customizable database-backed information management application. In *UIST '09: Proceedings of the 22nd annual ACM symposium on User interface software and technology*, pages 257–260, New York, NY, USA, 2009. ACM.

[15] D. McGuinness and F. van Harmelen. Owl web ontology language. `http://www.w3.org/TR/owl-features/`, 2004.

[16] Musicbrainz d2r server. `http://dbtune.org/musicbrainz/`, 2009.

[17] Open anzo. `http://www.openanzo.org/`.

[18] Emmanuel Pietriga, Christian Bizer, David Karger, and Ryan Lee. Fresnel - a browser-independent presentation vocabulary for rdf. In *In: Proceedings of the Second International Workshop on Interaction Design and the Semantic Web*, pages 158–171. Springer, 2006.

[19] Y. Raimond. Dbtune: Bbc john peel sessions. `http://dbtune.org/bbc/peel/`, 2007.

[20] Rdfa 1.1 cors test results. `http://rdfa.digitalbazaar.com/tests/cors/results`, 2010.

[21] J. Resig. jquery: The write less, do more javascript library. `http://jquery.com/`, 2010.

[22] Demos: Data-gov wiki. `http://data-gov.tw.rpi.edu/wiki/Demos`, 2007.

[23] A. Russell. Comet: Low latency data for the browser. `http://alex.dojotoolkit.org/2006/03/comet-low-latency-data-for-the-browser/`, 2007.

[24] Andy Seaborne, Gheeta Manjunath, Chris Bizer, John Breslin, Soupriya Das, Ian Davis, Steve Harris, Kingsley Idehen, Olivier Corby, Kjetil Kjernsmo, and Benjamin Nowack. Sparql update. `http://www.w3.org/Submission/SPARQL-Update/`.

[25] Arc rdf classes for php. `http://arc.semsol.org/`, 2007.

[26] Basic access control ontology. `http://www.w3.org/ns/auth/acl`, 2009.

[27] Sig.ma. `http://sig.ma`, 2009.

[28] Henry Story, Bruno Harbulot, Ian Jacobi, and Mike Jones. FOAF+SSL: RESTful Authentication for the Social Web. In *Proceedings of the First Workshop on Trust and Privacy on the Social and Semantic Web (SPOT2009)*, 2009.

[29] J. Tennison. rdfquery: Rdf processing in your browser. `http://code.google.com/p/rdfquery/`, 2009.

[30] A. van Kestern. Cross-origin resource sharing. `http://www.w3.org/TR/cors/`, 2009.