LibX 2.0

Brian Robert Nicholson

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Science and Applications

Godmar Back, Chair
Eli Tilevich
Stephen Edwards

November 27, 2011
Blacksburg, Virginia

Keywords: LibX, library, OPACs, browser extensions
Copyright 2011, Brian Nicholson
LibX 2.0

Brian Robert Nicholson

(ABSTRACT)

As Internet applications continue to gain popularity, users are becoming increasingly comfortable with using the Web as part of their daily lives. Content is becoming digitized on a massive scale, and web browsers are emerging as the platform of choice.

Library catalogs, or OPACs, have become widely digitized as part of this trend. Unlike modern search engines, however, many OPACs require antiquated, boolean-based search queries. Consequently, OPAC usage has declined. Libraries have recently begun to introduce modernized services that enable Google-like queries with convenient syntaxes; however, these services are not widely adopted since Google remains more accessible and familiar.

LibX 2.0 is a browser extension for Mozilla Firefox and Google Chrome that provides an interface for locating library resources. LibX 2.0 gives users instant access to library searches, links, and proxies. It provides support for the modernized search services that libraries are beginning to offer. Additionally, as a browser extension, LibX 2.0 is more accessible than the OPACs themselves.

LibX 2.0 is the next iteration of the popular LibX extension. LibX 2.0 borrows several software engineering concepts for its design, including code reuse and modularity. As a result, we have created and updated many components to be compatible with these software engineering goals. We have designed a new user interface, inspired by Google Chrome, whose design we share between browsers. We have developed a framework for library applications, or LibApps, which enable user-created, extensible code. We have also developed custom caching, internationalization, and user preferences libraries to support our new design.
Acknowledgments

Firstly, I would like to thank my advisor, Dr. Godmar Back. Under his guidance, I have acquired knowledge across a broad spectrum of computer science topics. Dr. Back’s Operating Systems and Cloud Computing courses have given me an appreciation for systems programming and an intricate understanding of the lower-level parts of my research. I greatly appreciate the time he has dedicated toward our weekly meetings; his availability and willingness to help made him easy to approach whenever I encountered problems.

I would like to thank Dr. Eli Tilevich for his role as a committee member and his Software Engineering and Internet Software Design courses, which helped me learn the fundamental engineering and Internet concepts used in LibX. He has convinced me to never pursue a Ph.D. unless, of course, it could be achieved with a coursework-only option.

I would like to thank Dr. Stephen Edwards for also accepting to be a committee member and for his Programming Languages course. In particular, his teachings on the Scheme programming language prepared me for the functional aspects of JavaScript.

I would like to thank Sony Vijay, Annette Bailey, and Kiri Goldbeck who, as LibX team members, have provided invaluable contributions toward LibX.

Finally, I would like to thank my parents for their support and encouragement.
## Contents

1 Introduction 1

1.1 Motivation ............................................. 1

1.2 Our Solution: LibX ................................. 2

1.3 Core Contributions .................................... 3

1.3.1 Browser Portability .................................. 3

1.3.2 Updatability ......................................... 8

1.3.3 Extensibility ........................................ 9

2 Background ............................................ 10

2.1 Overview of Technologies .............................. 10

2.1.1 Web Languages ..................................... 10

2.1.2 Browser Extensions ................................. 11

2.1.3 The Document Object Model ......................... 12

2.1.4 jQuery ............................................... 14

2.2 In-depth JavaScript ..................................... 16

2.2.1 Closures ........................................... 16
2.2.2 The “this” keyword .............................................. 17
2.2.3 Classes ............................................................. 18
2.2.4 Prototypal Inheritance .......................................... 19
2.2.5 Asynchronous model ........................................... 24

2.3 Extension Models ................................................... 26
2.3.1 Google Chrome .................................................. 27
2.3.2 Mozilla Firefox .................................................. 29

3 User Interface ......................................................... 32
3.1 Search Interface .................................................... 33
3.2 Context Menu ....................................................... 36
3.3 User Preferences Interface ....................................... 38
3.3.1 Layouts ........................................................... 39
3.3.2 Browser Preferences .......................................... 42
3.3.3 Context Menu Preferences .................................. 43
3.3.4 LibApp Preferences ........................................... 43
3.4 Loading Editions .................................................... 44

4 LibApps ................................................................. 47
4.1 Motivation ........................................................... 47
4.2 Design Model ....................................................... 48
4.3 Implementation ..................................................... 50
4.3.1 Atom Feeds ................................................. 50
4.3.2 Using Feeds in the Client ................................. 53
4.3.3 Tuple Spaces ............................................... 56
4.4 Security Concerns ........................................... 58
4.5 The LibApp Builder ........................................... 59

5 Core API ......................................................... 61

5.1 User Preferences API ........................................ 63
5.1.1 XML Preference Files ..................................... 63
5.1.2 Saving Preferences ......................................... 64
5.1.3 Component Preferences .................................... 65

5.2 Internationalization API ...................................... 67
5.2.1 Firefox’s Approach ......................................... 68
5.2.2 Chrome’s Approach ......................................... 69
5.2.3 LibX’s Approach ............................................ 69

5.3 Storage .......................................................... 72
5.3.1 HTML5 Local Storage ...................................... 73
5.3.2 SQLite and Web SQL ....................................... 73
5.3.3 File I/O ...................................................... 74

5.4 Memory Cache .................................................. 74
5.5 Object Cache .................................................... 75
5.5.1 Validation ................................................... 77
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.5.2 Cache Updates</td>
<td>79</td>
</tr>
<tr>
<td>6 Browser-specific Implementation</td>
<td>83</td>
</tr>
<tr>
<td>6.1 Google Chrome</td>
<td>84</td>
</tr>
<tr>
<td>6.1.1 Execution Environment</td>
<td>84</td>
</tr>
<tr>
<td>6.2 Mozilla Firefox</td>
<td>88</td>
</tr>
<tr>
<td>6.2.1 Execution Environment</td>
<td>88</td>
</tr>
<tr>
<td>6.2.2 LibApps</td>
<td>91</td>
</tr>
<tr>
<td>6.2.3 Search Interface Popup</td>
<td>92</td>
</tr>
<tr>
<td>7 Related Work</td>
<td>94</td>
</tr>
<tr>
<td>7.1 Zotero</td>
<td>94</td>
</tr>
<tr>
<td>7.2 Conduit Toolbar</td>
<td>95</td>
</tr>
<tr>
<td>7.3 Greasemonkey</td>
<td>95</td>
</tr>
<tr>
<td>8 Conclusion</td>
<td>96</td>
</tr>
<tr>
<td>8.1 Future Work</td>
<td>96</td>
</tr>
<tr>
<td>8.2 Summary</td>
<td>98</td>
</tr>
</tbody>
</table>
List of Figures

1.1 The LibX 1.5 toolbar. .................................................. 4
1.2 The toolbar expanded with multiple search fields. .................. 4
1.3 The search interface redesigned for LibX 2.0. ....................... 5
1.4 The LibX 1.5 preferences interface. ................................. 6
1.5 The LibX 2.0 user preferences page. ............................... 7

2.1 The DOM tree for the given HTML document. ....................... 13
2.2 Original File menu. .................................................... 31
2.3 File menu with XUL overlay applied. ............................... 31

3.1 The LibX button. ....................................................... 33
3.2 The full LibX search interface. ...................................... 34
3.3 Summon, with preview support. ...................................... 35
3.4 The mini LibX search interface. .................................... 36
3.5 Firefox’s dynamic context menu. .................................... 39
3.6 The context menu in Chrome. ....................................... 39
3.7 The browser preferences tab. ....................................... 43
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.8</td>
<td>The context menu preferences tab.</td>
</tr>
<tr>
<td>3.9</td>
<td>A LibApp preference.</td>
</tr>
<tr>
<td>3.10</td>
<td>Searching for an edition.</td>
</tr>
<tr>
<td>3.11</td>
<td>Edition auto-detection on the page.</td>
</tr>
<tr>
<td>4.1</td>
<td>Alert users to ACM digital library subscription LibApp.</td>
</tr>
<tr>
<td>4.4</td>
<td>Two separate feeds containing modules, LibApps, and packages.</td>
</tr>
<tr>
<td>4.5</td>
<td>The tree representation of Feeds A and B.</td>
</tr>
<tr>
<td>4.6</td>
<td>Modules communicate through a tuple space.</td>
</tr>
<tr>
<td>4.7</td>
<td>The LibApp Builder interface.</td>
</tr>
<tr>
<td>5.1</td>
<td>The interaction of LibX components.</td>
</tr>
<tr>
<td>5.2</td>
<td>Cache layers for a request, traversed in order until a hit is found.</td>
</tr>
<tr>
<td>5.3</td>
<td>Logic flow for an object cache get() request.</td>
</tr>
<tr>
<td>6.1</td>
<td>Chrome execution environment.</td>
</tr>
<tr>
<td>6.2</td>
<td>A basic, non-RPC call to someFunction.</td>
</tr>
<tr>
<td>6.3</td>
<td>An RPC call to someFunction through magicImport.</td>
</tr>
<tr>
<td>6.4</td>
<td>Firefox execution environment.</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Motivation

Since its inception, the Web has become increasingly user-friendly. Users are now reliant on search engines that utilize link popularity ranking algorithms and can fetch relevant results with little user effort. This is in contrast to library catalogs, or OPACs, that use outdated boolean search algorithms and require more precise queries to obtain relevant matches. While they may take more effort to use, OPACs are linked to libraries, so their results will usually be of higher academic value than those returned by standard search engines.

However, rather than learning how to use these OPACs for scholarly work, students are turning to general search engines because of their simplicity [8]. As a result, library catalog use is diminishing, despite the high-quality content libraries provide that is not as readily available elsewhere. As this disconnect between users and their libraries continues to increase, users are becoming less knowledgeable about what their libraries can offer. To remedy this problem, libraries need to be accessible in a way that is compatible with their users’ normal workflow.

Libraries are beginning to offer services, such as Summon [27], that unify resources and
provide a broad range of search results from simple queries. Although powerful, these services are not yet being fully utilized since they are hindered by the interfaces that query them.

1.2 Our Solution: LibX

LibX [2] is a browser extension that makes library resources more accessible than standard OPAC interfaces do. LibX integrates with the browser to connect users to their libraries with a seamless interface, allowing many search services, including Summon, to be readily available. LibX includes the following features:

Library editions. A LibX edition specifies all of the features a library supports, where its catalogs are located, and the name and images to be used in LibX. Editions enable libraries to be compatible with LibX at a fine-grained level.

Search interface. Whenever a user needs to search their library, they can instantly open the LibX interface to perform a search. Because LibX provides a portal to the user’s library, the user does not need to visit any webpage to perform a search. Instead, the search can be executed directly through the browser.

Proxy support. Libraries often restrict access to their electronic resources to members of their institutions; generally, only on-campus machines are granted access. Proxies allow off-campus users to log in and make requests through the proxy server, which acts as a middleman for authentication. However, proxy logins can be inconvenient; if a user encounters a resource they would like to use before logging in, they need to go to their library’s page, log in to the proxy service, then navigate to the page through the library. LibX removes this inconvenience by supplying proxy redirection links to give instant, per-resource access.

Context menu integration. LibX uses the browser context menu to integrate into the browser interface. If text is selected, the context menu can show options to search a resource using the selected text as search terms, providing an alternative search method to the LibX
search interface. If the page is proxyable, the context menu gives the user an option to reload
the page using the library’s proxy.

**LibApps.** LibX can integrate cues into any page the user visits. For instance, if the user
searches a book on Amazon, a small image will appear that links directly to the corresponding
resource in the user’s library. These integrations are accomplished using *LibApps*, bundles
of code that describe page transformations on specific sites. LibApps can be assembled by
the edition maintainer, so each library has its own set of LibApps that it chooses to give to
users.

**User preferences interface.** Many aspects of LibX can be customized, including the
displayed context menu options, LibApp-related options, and browser-related options. Ad-
ditionally, this interface is extensible, so maintainers can add their own custom preferences
for LibApps and other components.

### 1.3 Core Contributions

Prior to LibX 2.0, LibX 1.5 was a fully functional and popular browser extension used by
hundreds of universities. LibX 1.5 was released by the LibX team after several years of
development starting in 2005. LibX 2.0 is the next release of LibX that expands upon the
existing LibX 1.5 implementation. My research was the design and implementation of LibX
2.0; thus, the focus of this thesis is the set of contributions that differentiate LibX 2.0 from
its predecessor. These contributions, which I was responsible for, are outlined in this section.

#### 1.3.1 Browser Portability

LibX is designed to work across multiple browsers. Where possible, we want to make LibX
look and function the same, regardless of which browser it is installed in. If we maintained a
separate codebase for Firefox and Chrome, it would be necessary to implement every func-
tional addition separately for each browser; furthermore, if we added support for additional browsers in the future, each would require its own codebase to maintain. Therefore, we wanted as much of the LibX code as possible to be browser-independent.

Search Interface

In LibX 1.5, LibX was displayed to the user as a toolbar (Figure 1.1). While functionally sufficient, the toolbar design suffered from multiple usability issues.

Firstly, toolbars consume fixed screen space. Many extensions opt to use toolbars, resulting in a cluttered browser interface for users who like to have many extensions. This problem is further exaggerated in LibX if the user decides to search for an item using multiple fields since each additional field consumes the height equivalent of another toolbar (Figure 1.2).

![Figure 1.1: The LibX 1.5 toolbar.](image)

![Figure 1.2: The toolbar expanded with multiple search fields.](image)

Secondly, the LibX toolbar was written in XUL (for Mozilla Firefox) and Windows Forms (for Internet Explorer). These are browser-specific languages, so each change would have to be made per individual browser. If expanded to additional browsers, this browser-specific toolbar implementation would only increase the maintenance.

Finally, Google Chrome, one of the browsers we decided to support for LibX 2.0, does not support toolbars in its API. Instead, Chrome extensions use popup pages written in HTML, CSS, and JavaScript. Whereas Firefox and Internet Explorer require specialized languages for implementing toolbars, Chrome’s approach enables standard HTML-based
pages to be used for extension interfaces. Interaction with the browser is provided with Chrome’s extension API; this API is available to the popup page via JavaScript.

These web languages used in Chrome popups are supported by all browsers. Additionally, the popup design allows the interface to be shown only on demand, reducing screen consumption. Because of these benefits, we chose this design to replace the LibX toolbar. The redesigned search interface for LibX 2.0 is shown in Figure 1.3.

![Figure 1.3: The search interface redesigned for LibX 2.0.](image)

Since the Chrome extension API is not available to other browsers, the LibX popup page uses only the browser-independent LibX API. The LibX API provides a wrapper around the Chrome extension API while allowing us to reimplement required features in other browsers.

**User Preferences Interface**

Like the search interface, the user preferences were implemented in XUL and Windows Forms in LibX 1.5 (Figure 1.4). Although this design integrated seamlessly into the browser, it suffered from the same browser-dependent issues as the LibX toolbar. Additionally, Chrome again lacks support for this design in its API.
Similar to the search interface redesign, we redesigned the user preferences interface to use HTML, CSS, and JavaScript. Instead of being tightly integrated into the browser interface as it was in LibX 1.5, the user preferences interface in LibX 2.0 is now displayed in its own HTML page.

File and Code Structure

In LibX 1.5, much of the code was contained in large, monolithic files. The code was largely Firefox-dependent, making it unusable in Google Chrome. As a result, we decomposed the existing code to simpler, more modular files. Browser-dependent parts were extracted from the codebase, leaving generic, browser-independent code that is used by both browsers. We split the code into three directories: \texttt{ff}, \texttt{gc}, and \texttt{shared}, corresponding to Firefox code, Chrome code, and browser-independent code. Each browser imports code from the \texttt{shared} directory and its own browser-dependent directory.

Since JavaScript does not enforce any file or code structure, we followed a design similar to Java’s package mechanism. All LibX code is namespaced, and each module is placed in its own file. Each level of the namespace hierarchy is represented as a subdirectory in the file

Figure 1.4: The LibX 1.5 preferences interface.
structure, like Java packages. This structure is mirrored in each of the \texttt{ff}, \texttt{gc}, and \texttt{shared} directories.

In addition to browser-independent code, we have also made our API environment-independent. Even within a single browser, LibX code is run in multiple environments, and code that works in one environment may not run in another. To handle this, we follow the same pattern we use for browser portability: use independent code where possible and factor out environment-specific pieces.

All modern browsers support JavaScript 1.6, making this the single language used for the entire LibX API. Some browsers, particularly Firefox, may use JavaScript versions with bleeding-edge features that have not yet been widely adapted. We avoid using these features to simplify cross-browser compatibility.
1.3.2 Updatability

Editions

All edition maintainers use a tool called the *Edition Builder* to create and modify their editions. In LibX 1.5, each LibX edition was distributed as a separate Firefox extension generated by the Edition Builder. If edition maintainers needed to make a change to their editions, they had to rebuild the extension—that is, generate a browser-compatible package that could be loaded as an extension. These updated extension files were then detected by users’ browsers and triggered the browser extension update process.

There are a couple of drawbacks to this approach. In Firefox, extension updates are intrusive as they require user confirmation and a browser restart. Another drawback is that extension updates require downloading the entire updated extension, even if only a small portion has changed. This wastes network resources.

In LibX 2.0, editions are no longer distributed as individual extension files. Instead, there is only a single generic LibX extension that users download; editions are then dynamically fetched and loaded. As a result, edition maintainers are no longer required to undergo the edition building process used in LibX 1.5; changes to editions are automatically pulled and loaded in the extension without any user intervention.

Bootstrapped Scripts

Fixes and new features released by the LibX development team also required extension rebuilds, so the LibX code itself was subject to these same update limitations. To solve this, we split the LibX codebase into *core* and *bootstrapped* code. Core code is code bundled with the extension; that is, updates to the core require an extension rebuild. Bootstrapped code, on the other hand, is code that is dynamically retrieved and executed at runtime. This code can be updated during runtime since the files are not part of the static extension package.
Since the core code is not updatable, we try to make components bootstrapped whenever possible. Some components, particularly those involved with LibX initialization, must be part of the core. However, these components are relatively stable and require changes infrequently.

1.3.3 Extensibility

A popular feature of LibX is its ability to integrate with web sites to provide context-sensitive cues. In LibX 1.5, each cue was implemented as a separate JavaScript script, and all LibX editions used the same set of cues. Since the code was created and served by LibX developers, these scripts gave edition maintainers little control over the cues. In LibX 2.0, these scripts have been discarded in favor of extensible, user-created LibApps.

LibApps are library applications that execute context-sensitive, per-page transformations. They are composed of modules, which are small, modularized pieces of JavaScript code that perform a single function. There are many modules from which LibApps can be built, and a set of compatible modules can be combined to form a specific LibApp. Users are subscribed to a collection of LibApps, distributed in packages. Edition maintainers can fully customize the modules, LibApps, and packages distributed with their editions. LibApps, modules, and packages are user-created components constructed using the LibApp Builder (Section 4.5). They are integrated into the standard LibX installation, extending the features LibX provides.

A separate extensible component is the user preferences interface. Since LibApps can apply various page transformations such as cues, popup dialogs, and autolinking, end-users will likely want to configure LibApps according to their personal preferences. Because LibApps are extensible components, it follows that the preferences interface for customizing them must also be extensible. LibApps can specify layouts, which can be custom-created, to use in the preferences interface. When rendered, these layouts are combined to form a single, unified page.
Chapter 2

Background

This paper assumes the reader is familiar with common programming principles and vocabulary. Nevertheless, LibX development requires familiarity with certain web and browser technologies that programmers may not have experience with. This chapter explains those technologies essential to understanding LibX’s implementation.

2.1 Overview of Technologies

2.1.1 Web Languages

LibX is implemented with a collection of standard web technologies including HTML, CSS, XML, and JavaScript.

**HTML** is a markup language for displaying pages [23], and it is the most prevalent web language. HTML support is a necessity for all web browsers, so any LibX components written in HTML are guaranteed to work across browsers. This makes it the preferred language for the LibX interface, and HTML is used to display the LibX search interface and preferences page.
CSS is another widely-used language for webpages. Whereas HTML is generally used to show a page’s content, CSS is used to style the content and produce the layout [6]. CSS allows styles to be reused and specified separately from HTML code; therefore, we use CSS in LibX when possible for maintainability.

XML is a general markup language used frequently in web applications. XML documents are composed of nested elements that contain data generally intended for distribution [7]. XML documents follow a specific syntax that allows them to be parsed by a number of applications, including web browsers; this compatibility is the primary advantage of XML. In LibX alone, a wide variety of data can be encoded to XML, including LibX edition configurations, LibApp feeds, and LibX preference files.

XUL is an XML user interface language developed by Mozilla [18]. XUL is used to specify the layout of XUL-based applications, including Mozilla Firefox. Like HTML, XUL relies on CSS and the DOM, making XUL development similar to HTML development.

JavaScript, unlike the languages listed above, is an actual programming language. The LibX API is written entirely in JavaScript, making JavaScript the most important language to understand for LibX development. Because of this, Section 2.2 is devoted entirely to JavaScript programming.

2.1.2 Browser Extensions

As browsers evolve, users demand more features to enhance their browsing experience. For instance, a user may want to tweet the current page to Twitter, sync bookmarks across computers, or have a console for web development. It would be impractical to try to include all of these features directly in the browser; therefore, extensions are available as smaller modules that can be individually downloaded separately from the browser itself.

One popular extension is Adblock Plus [21], which blocks ads such as banners and Flash content. When a page loads, Adblock Plus scans the document for elements matching ad
patterns and removes them. These patterns are stored in filter lists, which are updated regularly.

Another popular extension is called Greasemonkey [5]. This extension enables support for user scripts on a per-site basis. Scripts are available at sites such as Userscripts.org and are downloaded separately from Greasemonkey. Thus, Greasemonkey itself is extensible, and provides an additional layer for customization without requiring users to bundle these scripts into more complex extensions.

These examples show the versatility of browser extensions. Many extensions, including Greasemonkey, Adblock Plus, and LibX, perform context-sensitive tasks based on pages the user is viewing, unifying the page and browser to create a seamless browsing experience.

2.1.3 The Document Object Model

The DOM Tree

The document object model, or DOM, is an in-memory representation of an HTML or XML page and its elements [15]. All valid HTML and XML pages can be represented as a tree where each node corresponds to an element in the document. For example, given the following HTML,

```html
<html>
<body>
<div id="fruits">A list of fruits:</div>
<ul>
<li class="fruit">apple</li>
<li class="fruit">banana</li>
<li class="fruit">orange</li>
</ul>
</body>
</html>
```

the following DOM tree can be formed:
Figure 2.1: The DOM tree for the given HTML document.

The above HTML includes `id` and `class` attributes. The `id` attribute must name a unique key, whereas the `class` is used to identify similar elements. These attributes are used for styling in CSS and in the DOM API for DOM manipulation.

**The DOM API**

All documents are rooted at a `document` element. From the root, a number of methods can be used to efficiently locate children in the tree. The following are some of the most commonly used DOM functions:

**getElementById:** searches for the element containing a unique ID. this method is available only on the root `document` element.

**getElementsByClassName:** searches all children of the current element for elements whose class attribute match the given class name.

**getElementsByTagName:** searches all children of the current element for elements of the specified type.
In web applications, the DOM is accessed and manipulated with JavaScript. The window object, available to all pages, contains a document property that acts as the entry point to the DOM. The following JavaScript snippet searches for a list with ID list, traverses its <li> children, and alerts the text of each:

```javascript
var list = document.getElementById('list');
var items = list.getElementsByTagName('li');
for (var i = 0; i < items.length; i++) {
    // textContent is not supported in all browsers
    alert(items[i].textContent);
}
```

Even in this simple example, the code is rather verbose. A number of libraries exist that can greatly simplify DOM access and manipulation.

### 2.1.4 jQuery

The most popular JavaScript library, jQuery [25], is used extensively throughout LibX. Created by John Resig, jQuery is a powerful JavaScript library used to make DOM interactions more concise. jQuery accepts CSS selectors to match elements on the page, enabling far more expressive syntax for element selection. Like in CSS, selectors in jQuery can match multiple elements, so operations can be performed on entire sets of elements; this avoids explicitly creating an array to operate on each element individually. The following code uses only the DOM API:

```javascript
var list = document.getElementById('list');
var items = list.getElementsByTagName('li');
var fruits = [];
for (var i = 0; i < items.length; i++) {
    if (items[i].className == 'fruit')
        fruits.push(items[i]);
}
alert(fruits.length + ' fruits found');
```

In jQuery, this can be reduced to:

```javascript
var fruits = $('list li.fruit');
```
alert(fruits.length + ' fruits found');

After filtering a set of elements, jQuery’s extensive API can be used to manipulate them. Some of jQuery’s many functions include:

**text:** returns the text inside of the matched elements.

**append:** appends the given content to each matched element.

**each:** executes the given function for each matched element.

**get:** fetches a remote resource, executing the given callback upon successful retrieval.

In addition to its filtering functions, jQuery accepts CSS selectors to access elements. The previous DOM example,

```javascript
var list = document.getElementById('list');
var items = list.getElementsByTagName('li');
for (var i = 0; i < items.length; i++) {
    // textContent is not supported in all browsers
    alert(items[i].textContent);
}
```

can be rewritten without a for loop or using the browser-dependent `textContent` property:

```javascript
// the # symbol indicates an ID in CSS
$('list li').each(function () {
    // each() sets JavaScript's this to refer to the matched element
    alert($(this).text());
});
```

Because jQuery is not compatible with XUL, jQuery is not used internally to implement the LibX API. However, most LibX UI components are created with simple HTML, so many of them utilize jQuery to simplify DOM manipulation. LibApps, in particular, are executed on standard webpages, and they frequently use jQuery selectors for scraping text and appending elements.
2.2 In-depth JavaScript

JavaScript is an expressive, functional language that is more closely tied to Lisp and Scheme than other C-based languages [9]. Closures and prototypal inheritance, two features that are prevalent in JavaScript code [10], do not exist in most C-based languages. These features are used frequently throughout LibX.

2.2.1 Closures

A closure is the ability of an inner function to use the referencing environment created by its outer containing function. Even if the inner function is called outside of its enclosing function, it can refer to variables inside the closure. For example:

```javascript
var closure = function () {
    var n = 42;
    return function () {
        return n;
    };
};
var c = closure();
var n = 7;
console.log(n);    // 7
console.log(c());  // 42
```

Here, the `closure` function creates a closure for `n`. It returns an anonymous function whose free variables are bound to `closure`, the containing function. When this function is called, the free variable `n` refers to the `n` defined by `closure` and the function returns 42, even though `n` is 7 where the function is actually called.

In JavaScript, the `var` keyword is used to declare–and optionally define–a variable in the scope of the current function. If `var` is omitted in a definition, the variable is placed in the global scope; typically, this means it will be attached to the `window` object. Therefore, accidentally omitting `var` when defining a variable will not produce any syntax errors,
potentially resulting in scope-related bugs as seen here:

```javascript
var closure = function () {
    n = 42; // not bound to this function
    return function () {
        return n;
    }
};

var c = closure();

var n = 7;
console.log(n); // 7
console.log(c()); // 7
```

Because `var` was not used to define `n`, `n` was inadvertently placed in the global scope. Many developers choose to define variables by attaching them as properties to the `window` object, making it explicit that the variables should be global.

### 2.2.2 The “this” keyword

The `this` keyword in JavaScript is different from most languages and is the cause of frequent errors. In a function, the value of `this` depends on how the function was called. If a function is called using the dot operator, `this` refers to the object to which the function is attached. If called as a standalone function, `this` refers to the global object—usually `window`. The difference is illustrated here:

```javascript
var obj = {
    val: 'hello',
    func: function () {
        return this.val;
    }
};

console.log(obj.func()); // 'hello'
var func = obj.func();
console.log(func()); // undefined
```
Even though the same function is being called in both cases, it is being called in different contexts. Since `func` is first invoked using the dot operator as part of the `obj` object, `this` refers to `obj`; thus, `this.val` is equivalent to `obj.val`, or “hello”. In the second invocation, the reference to `func` is stored in a variable, allowing it to be called independently. In this case, `this` refers to the global object, and `this.val` is equivalent to `window.val`, which is undefined.

2.2.3 Classes

Though JavaScript does not support classes as first-class constructs, its objects can be created from functions. The `new` keyword, which is used with a function name, uses the function as a constructor for instantiation. When executed in this context, the function’s `this` keyword refers to the newly instantiated object, so `this.foo = 'bar'` can be used to attach the `foo` property to the object. On the other hand, the `var` keyword can be used in constructors so that variables can be referenced only in the closure of functions defined in the constructor. Together, `var` and `this` can be used to emulate private and public fields that are common in typical object-oriented languages. The following example shows how a class might be defined in a classical OO language:

```javascript
class A {

    // private field
    private str;

    // constructor
    function A() {
        str = 'a';
    }

    // public field
    public toString() {
        return str;
    }
}
```
In JavaScript, this same class can be created using functions, `var`, and `this`:

```javascript
// constructor function
function A() {
    // private field
    var str;

    // constructor logic
    str = 'a';

    // public field
    this.toString = function () {
        return str;
    }
}

var a = new A();
console.log(a.str);     // undefined
console.log(a.toString()); // 'a'
```

### 2.2.4 Prototypal Inheritance

JavaScript is a prototype-based language, meaning it does not use classes to achieve object-oriented functionality. Instead, existing objects can be used as prototypes to form inheritance relationships. This technique was first introduced in the Self programming language [30].

All functions in JavaScript have a **prototype** property. This property can refer to any object. When the function is used as a constructor to create an object, the **prototype** property specifies the object’s prototype. When an object has a prototype, the prototype is used as a fallback to look up any properties not present in the object itself.

Objects used as prototypes may themselves use other objects as prototypes. When a property of an object is accessed, the object itself is searched for the property. If the object does not have the property, its prototype is searched. If the prototype does not have the property,
the prototype’s prototype is searched. This recursive set of prototypes is called a *prototype chain*. The following example shows a multi-level prototype chain:

```javascript
var o1 = {  
a: 'a',  
b: 'b'
};

function O2() {}
O2.prototype = o1;
var o2 = new O2();
o2.c = 'c';

function O3() {}
O3.prototype = o2;
var o3 = new O3();
o3.d = 'd';
o3.b = 'new b';

// level 0 (object has property)
console.log(o3.d); // 'd'

// level 1 of prototype chain
console.log(o3.c); // 'c'

// level 2 of prototype chain
console.log(o3.a); // 'a'

// o3 now has property b
console.log(o3.b); // 'new b'

// o1.b remains unchanged
console.log(o1.b); // 'b'
```

In this example, `o3` has the properties `a`, `b`, `c`, and `d`, accessible through its prototype chain. Writing `o3.b` adds the property `b` to `o3` and does not affect the `b` in `o3`'s prototype chain.

The behavior exhibited here is similar to classical inheritance where an object’s prototype is analogous to a super class. Using classical inheritance, a subclass would be specified in a manner similar to the following:

```javascript
class Animal {
```
JavaScript has no `inherits` keyword, so inheritance is achieved by manually setting the prototype for `Dog`:

```javascript
// constructor
function Animal() {
    this.toString = function() {
        return 'animal';
    };
}
// constructor
function Dog() {
    this.toString = function() {
        return 'dog';
    };
}
Dog.prototype = new Animal();
```

Since the prototype must be set to an object, `new` is used with the `Animal` constructor to create an object for the `Dog` prototype. Though syntactically different from classical inheritance, the result is similar:
var animal = new Animal();
var dog = new Dog();

// subclass properties hide parent properties
console.log(animal.toString());  // 'animal'
console.log(dog.toString());     // 'dog'

// 'is-a' relationships
console.log(animal instanceof Animal);  // true
console.log(animal instanceof Dog);    // false
console.log(dog instanceof Animal);    // true
console.log(dog instanceof Dog);       // true

Note that instanceof determines whether the given constructor is in the object’s prototype chain.

Inheritance.js

Inheritance.js is a library that emulates classes in JavaScript. It is used as a core part of LibX, and all LibX classes are created using Inheritance.js. Inheritance.js supplies two functions used commonly throughout LibX: create() and mixin().

create() creates a class using the given object argument. If create() is given two arguments, an inheritance relationship is implied, and the new class inherits from the super class, specified by the first argument. If an initialize property is included in the class definition, it is immediately executed when the class is instantiated—in other words, the initialize function is the constructor. The above Animal and Dog classes can be rewritten to use Inheritance.js:

var Animal = Class.create({

    // constructor
    initialize: function () {},

    toString: function () {
        return 'animal';
    }
})
var Dog = Class.create(Animal, {

    // constructor
    initialize: function () {}

    toString: function () {
        return 'dog';
    }
});

With Inheritance.js, JavaScript’s prototype inheritance model is effectively hidden. There is one disadvantage to using Inheritance.js: it does not support private instance variables. This is a consequence of the object literal notation being used to form the class definition rather than a constructor function, preventing closures from being created for the methods.

Code reuse is often achieved using inheritance; however, subclasses may want to use the functionality of multiple superclasses. Mixins solve this problem by avoiding the complexities of multiple inheritance [11]. Rather than require entire classes be used for inheritance, mixins are generally fragments of classes that are not independently useful. Given JavaScript’s ad-hoc object composition and lack of rigorous class structuring, mixins are well-suited for the language.

The mixin() function provided by Inheritance.js can be used to merge properties of different objects. It accepts three arguments: a destination object, a source object, and a clobber flag. The properties of the source object are merged into the destination object, and the clobber flag specifies whether existing properties in the destination will be replaced by those in the source. An example of its usage is shown here:

```javascript
var obj1 = {
    a: 'a
    b: 'b'
};
var obj2 = { b: 'B' };
```
```javascript
var obj3 = {
  c: function () {
    return this.a + this.b;
  }
};
Class.mixin(obj1, obj2, true);
Class.mixin(obj1, obj3, false);

console.log(obj1.c()) // 'aB'
```

### 2.2.5 Asynchronous model

JavaScript is single-threaded. It is intended to be used in event-based systems. When an event occurs, a handler executes until completion and is never interrupted. As a result, any code that takes a long time to execute will block the execution of other code until it completes. For this reason, synchronous I/O should be avoided. JavaScript’s is prevalent on the Web, where frequent I/O–network requests and responses–is the norm.

In synchronous code, function calls wait for a function to execute and return, then use the result immediately in the following code. This means that synchronous functions should execute relatively quickly to prevent the execution of following code from being delayed. Quick execution, however, cannot always be guaranteed. I/O transactions, especially over a network, require an indefinite amount of time before responses are received.

For these cases, a different code structure can be used to wait for the responses asynchronously. Rather than immediately expecting a return value, client code passes one or more callback functions to an asynchronous function. A callback function accepts the results as parameters which are used in the callback body. Asynchronous functions call callbacks only once a result is available; consequently, code after an asynchronous function call can execute without waiting for the function.

A synchronous call to `xyz()` would look like the following:

```javascript
var result = xyz();
```
```javascript
alert('got result ' + result);
// more code here
```

Since `xyz()` is synchronous, the code following the execution is blocked until `xyz()` returns—even if the code is independent of `xyz()`’s result.

If `xyz()` is asynchronous, a callback function must be passed to handle the return value:

```javascript
xyz(function(result) {
    alert('got result ' + result);
});
// more code here
```

The program can continue execution while the result-dependent code waits to be called by `xyz()`. The result is not immediately available after calling the function; therefore, JavaScript code must be written with callbacks for all asynchronous functions.

A frequently-used I/O API is `XMLHttpRequest`. This is available to all pages as part of the global `window` object. The following are the most basic properties of `XMLHttpRequest`:

- **open()**: function that specifies the request method, URL, and whether the request should be asynchronous.
- **readyState**: an integer, 0-4, that represents the current state of the request. 0 indicates an unsent request; 4 indicates that the request is done.
- **readystatechange**: event fired whenever the readyState property changes.
- **status**: the HTTP status of the request.
- **responseText**: the text response body.
- **responseXML**: the document response body.
- **send()**: function that sends the request.

With this API, a simple request can be formulated:
Calling `addEventListener()` registers a callback that will be fired on the `readystatechange` event. In this example, the asynchronous flag is set to true. Following JavaScript’s single-threaded execution rules, the current event handler—the one issuing the request—executes to completion before other handlers are fired. This means the “executing XHR” message is guaranteed to be shown before the response message.

If the asynchronous flag were set to false, the request would block until the response is received, and the order of the log messages would be switched. If the request takes a long time, this means that any code after the request—even if not dependent on the request—will not execute until the response is received. As a result, asynchronous I/O should be preferred over synchronous I/O whenever possible.

### 2.3 Extension Models

LibX 2.0 currently works in two browsers: Google Chrome and Mozilla Firefox. Extensions are widely popular, in part because they are designed using existing web technologies. Extension developers for both browsers must be familiar only with languages such as HTML/XML and CSS, and extension logic for both browsers is written entirely in JavaScript. Despite these similarities, however, each browser uses a completely different approach for extension
implementation.

The extensible parts of browsers are called extension points. In Firefox, every part of the browser is exposed and extensible. Chrome, on the other hand, requires extensions to use its API to interact with the browser. These two approaches follow the push/pull model [16].

### 2.3.1 Google Chrome

Google Chrome has a specific API that extensions must use to interact with the browser. This fits the pull model: the browser allows only specific extension points to be modified. This has the advantage that the UI remains consistent since all extensions are required to conform to a specific layout. However, this has the disadvantage that it limits the freedom of extension developers; extensions’ layouts and code are constrained to the Chrome extension API.

Chrome uses a multi-process architecture, where each tab, extension, and the browser runs in its own process. This separation offers better responsiveness, stability, and security than single process browsers [24]. On the other hand, this separation also complicates the extension model since extensions frequently need to interact with these separate processes.

Chrome uses an extension manifest as an entry point to the extension. This file, named manifest.json, lists the extension name, version, update URL, and permissions. manifest.json also references other files in the extension, specifying how they will be used.

Each Chrome extension can have an HTML background page, specified in the manifest.json file. The background page, which is invisible, resides in the extension process, and it can communicate with each page process. Typically, the background page includes and executes JavaScript code with minimal HTML.

The extension manifest can specify a browser action, which is a button that appears to the right of the URL bar. This button performs a custom action when clicked, and it can be bound to either a JavaScript handler or HTML popup page. The Chrome API does not
support toolbars, so browser actions with popups are commonly used as an alternative.

To execute code on pages, extensions can also specify content scripts in manifest.json. Content scripts, composed of JavaScript, are designed to run on a per-page basis inside of the page process. To prevent pages from exploiting extension code, content scripts are run in isolated worlds. This means that content scripts cannot directly access functions or variables defined on a page, and vice-versa. Content scripts can, however, manipulate the page's DOM, so page-extension communication, if necessary, is still possible through modifications to the DOM tree.

Since content scripts and the background page run in different processes, communication between them is accomplished using asynchronous message passing. Either can listen for messages by registering listeners. Messages consist of JavaScript objects in JSON format. Object attributes can be used to differentiate requests, allowing listeners to handle and respond appropriately.

The following is a simple example of Chrome's message-based communication ¹:

Content script:

```javascript
chrome.extension.onRequest.addListener(function(request, sender, sendResponse) {
  if (request.greeting === "hello")
    sendResponse({farewell: "goodbye"});
  else
    sendResponse({}); // snub them.
});
```

Background page:

```javascript
chrome.tabs.getSelected(null, function(tab) {
  chrome.tabs.sendRequest(tab.id, {greeting: "hello"}, function(response) {
    console.log(response.farewell);
  });
});
```

This message passing model invokes all callbacks asynchronously. This can complicate exten-

---

¹http://code.google.com/chrome/extensions/messaging.html
sion code, especially the code of extensions that, like LibX, require extensive communication between the content scripts and background page.

2.3.2 Mozilla Firefox

Extensions in Firefox are created using a *push* model: the internals of the browser are exposed, and all modifications are made to the browser itself rather than via a strict API. This gives extension developers complete control over the browser, but also complicates extension development since all aspects of the browser serve as extension points.

A simple Firefox extension contains the following file structure:

install.rdf
chrome.manifest
chrome/
content/
    overlay.xul
skin/
locale/

Like Chrome, Firefox extensions have manifest files that describe the extension components. **chrome.manifest** is Firefox’s extension manifest that lists the *chrome providers* for an extension. There are three types of chrome providers: content providers, skin providers, and locale providers. A content provider, typically a XUL file, describes the extension’s layout and content. A skin provider, usually CSS files and images, defines the style for an extension. The locale provider supplies localization strings, allowing the extension to be internationalized.

Whereas Chrome extensions also include their versions and update URLs in the extension manifest, Firefox extensions store this information in a separate *installation manifest* called
**install.rdf.** This file also specifies the minimum and maximum Firefox versions the extension is compatible with.

The Firefox window exposed to the user is a single composite document formed by the browser XUL and its extensions, or collectively, the *chrome*. The Firefox application window is specified entirely with XUL. Elements in the Firefox UI are associated with a unique ID. For instance, the global Firefox window’s ID is *main-window*, and the navigation bar’s ID is *nav-bar*. The XUL composing the browser windows forms the foundation for extensions to build upon, and these IDs are used as points for extensions to hook into. This is done using *overlays*. The `<overlay>` element in a XUL file contains UI elements whose IDs match those of existing elements, resulting in the elements being merged. For example, given the following browser and extension XUL, the composite XUL is formed.

**Browser XUL:**
```
<menu id="file_menu">
  <menuitem name="New"/>
  <menuitem name="Open"/>
  <menuitem name="Save"/>
  <menuitem name="Close"/>
</menu>
```

**Extension XUL:**
```
<?xml version="1.0"?>
<overlay id="singleItemEx"
  xmlns:xhtml="http://www.w3.org/1999/xhtml"
  xmlns="http://www.mozilla.org/keymaster/gatekeeper/there.is.only.xul">
<menu id="file_menu">
  <menuitem name="Super Stream Player"/>
</menu>
</overlay>
```

**Composite XUL:**
```
<menu id="file_menu">
  <menuitem name="New"/>
  <menuitem name="Open"/>
```

Example from https://developer.mozilla.org/en/XUL_Overlays
<menuitem name="Save"/>
<menuitem name="Close"/>
<menuitem name="Super Stream Player"/>
</menu>

Figure 2.2: Original File menu.

Figure 2.3: File menu with XUL overlay applied.
Chapter 3

User Interface

This chapter discusses the various user interface components within LibX:

**Search interface.** The search interface is the primary interface with which users interact. This interface allows a user to select a catalog in their edition and search the catalog with whatever fields it supports, such as by author, title, or keyword. The search interface also provides links relevant to the edition and a proxy button to reload the page with the library’s proxy.

**Context menu.** The context menu can be used as an alternative to the search interface for catalog searches. When a user selects text on the page, the context menu offers selections to search the text in the user’s catalogs. The context menu also offers the same proxy options shown in the search interface.

**User preferences interface.** The user preferences interface used to set LibX-related preferences. The user preferences interface is extensible: client code, such as LibApps, can specify layout files associated with any preferences it sets. These layouts are then loaded together to form a single composite interface.
3.1 Search Interface

LibX uses a popup search interface to provide an interface for its most commonly used features. It appears when clicking the LibX button, which is a small button to the right of the URL bar that shows the current edition icon (Figure 3.1).

![Image of LibX button](image)

Figure 3.1: The LibX button.

The search interface has the following sections:

**Search:** Searches the selected catalog for the user-given terms

**Links:** A collection of edition-specific links

**Preferences:** Shows the LibX preferences page

**About:** Displays information about the current LibX build and edition

**Maintainer:** Provides information useful to edition maintainers

**Proxies:** Allows the user to reload the page using the libraries proxies, if any

**Developer:** Provides information useful to LibX developers (enabled in About tab)

**About:** Shows information regarding the LibX edition and build; also provides a link to change the edition

The Maintainer, Proxies, and Developer tabs are shown only conditionally.

The full search interface is shown in Figure 3.2. The top link allows the user to select a catalog, whereas the links next to each search field allow the user to select a different search
option. Clicking the arrow to the right of the search field creates another input box, and clicking the red X removes the field.

![Figure 3.2: The full LibX search interface.](image)

Some libraries offer services that report resource listings using XML. The Virginia Tech library, for example, supports Summon, which searches across a broad range of resources and generates an XML file containing the results. For these libraries, LibX takes advantage of this service by fetching and displaying the resources directly in the search interface popup. These services can be queried using the “Preview” button (Figure 3.3). Summon XML results contain a wealth of information, including abstracts, resource ISBNs, and article types. We embed this information using tooltips.

In addition to this full-featured search interface, the user can also choose to switch the popup to a smaller, minimal view (Figure 3.4). This view supports only a single catalog and search field, and it may be preferred in cases where the user wants to see more of the page without it being covered by the larger search interface popup.

The search interface is created entirely in standard web languages: HTML, CSS, and JavaScript. This is in accordance with our goal of creating reusable components since all
browsers support these languages. jQuery is used extensively in the page: the dynamic search field add and remove buttons, the links to change catalogs and search options, and the navigation links all rely on jQuery.

The localization API is used for all strings displayed in the interface. To integrate localization with the HTML-defined layout, the user interface’s JavaScript initialization localizes all elements with the set-locale class. Elements with this class must contain an i18n property that will be replaced with a language-specific string. For example, this element refers to the tab_search i18n property:
The search interface’s JavaScript calls the LibX i18n API, looks up the `tab_search` property, and replaces the text with the translated message:

The decision to use a popup for the LibX was due in part to Google Chrome’s extension constraints; the Chrome API provides few options for extension interfaces. Toolbars, used frequently in Firefox and Internet Explorer, are not supported in Chrome. Firefox gives extensions complete freedom over browser integration, so it is possible to recreate Chrome’s popup UI in Firefox. Thus, we elected to use a popup to display the search interface. The Firefox popup implementation is discussed further in Section 6.2.3.

### 3.2 Context Menu

LibX supports a context menu that provides context-sensitive search options. For instance, if the user selects text, the context menu displays options to search the selected text; if the user right-clicks a link, the context menu gives an option to follow the link using the edition’s proxy.

Selection-based searches provide options to search the user’s catalog. The list of catalogs
displayed in the context menu is determined by the edition maintainer, but also configurable by the user.

The context menu also displays proxy entries. If the user right-clicks a link, the context menu offers an option to follow the link using the edition’s proxy. If the user clicks outside of a link, the context menu will instead offer an option to reload the current page using the proxy.

In Firefox, context menu entries are dynamically populated (Figure 3.5). If the selection matches the regular expression for an ISBN, PubMed ID, or DOI, these are the only search options presented in the context menu. Additionally, some libraries support proxy querying that indicate whether a given page can be reloaded by the proxy. For these libraries, Firefox dynamically updates the context menu to report whether the page is proxyable. Unfortunately, Chrome’s context menu API does not yet support the ability to dynamically display entries based on context, so these features are not available in Chrome (Figure 3.6).

The context menu was implemented using the object-oriented factory pattern [12]. Each option for each catalog is class instance. Catalog, OpenURL, proxy, and Scholar entries are instances of factory classes that subclass the generic ContextMenuItem class. The ContextMenuItem class, inspired by the Google Chrome context menu API ¹, contains the following properties:

visible: boolean flag that indicates whether this entry should be shown

label: the text that appears when this entry is displayed

contexts: an array of strings that indicate when the entry should be displayed

onclick: handler executed when the entry is clicked

onshowing: handler executed when the entry is displayed (supported in Firefox only)

¹http://code.google.com/chrome/extensions/contextMenus.html
For onclick, the supported strings are "all", "page", "selection", and "link". "all" always shows the entry, "selection" shows the entry for text selections, "link" shows the entry when the user right-clicked a link, and "page" shows the entry when the user right-clicks on the bare page (and not on a link or with a text selection).

The onshowing handler, available only in Firefox, enables the dynamic context menu functionality. This handler uses context-sensitive information from the page, such as the page's URL or the selected text, and updates the context menu while it is being displayed.

The context menu is itself implemented as a class, available as libx.ui.ContextMenu. This class contains four methods:

- **registerItem()**: maps the given ContextMenuItem to a numeric ID
- **addItem()**: adds the given ContextMenuItem entry to the actual browser context menu
- **removeItem()**: removes the context menu entry corresponding to the given ID
- **update()**: updates the context menu entry corresponding to the given ID

Because the addItem(), removeItem(), and update() are browser-dependent methods that interact directly with the browser UI, these methods are represented as abstract functions in the browser-independent context menu implementation. Thus, each browser must include browser-dependent context menu code that implements these methods.

### 3.3 User Preferences Interface

The user preferences interface is a composition of extensible layouts. Whereas the LibX preferences API is used by most LibX components to store various user-related options, the user preferences page serves as a front-end to these preferences, allowing users to manipulate the preferences for all components in a single, unified interface.
Figure 3.5: Firefox’s dynamic context menu. In accordance with the selection, only ISBN entries are displayed. Additionally, the current page is dynamically reported as non-proxyable.

3.3.1 Layouts

The LibX preferences API provides an in-memory representation of the user preferences, which is accessible using \texttt{libx.prefs}. Each property of \texttt{libx.prefs}, such as \texttt{libx.prefs.browser} and \texttt{libx.prefs.contextmenu}, is displayed using the preferences interface. Displaying each of these preferences is accomplished using \textit{layouts}.

The user preferences page is a dynamically generated page built from a collection of layouts.
The layout code is based on JsPlate, part of the JsDoc toolkit. Layout files used a special language that extends standard HTML. These extensions are implemented using regular expressions that replace the layout code and generate pure HTML. These constructs, specified in the JsPlate layout engine, consist of the following:

**JavaScript code.** JavaScript code can be incorporated directly into the layout template using the `{! [code] !}` syntax, like so:

```javascript
{! alert(foo); !}
```

**JavaScript output.** In addition to evaluation, JavaScript values can be included in the layout using the `{+ [value] +}` syntax:

```html
<div class="{+ item.clazz +}">
{+ item.value +}
</div>
```

In this example, `item` is some JavaScript object with `clazz` and `value` properties. Assuming these are respectively set to "foo" and "bar", the resulting HTML would be:

```html
<div class="foo">bar</div>
```

**Conditional statements.** Layouts support if/else statements, which take the following form:

```html
<if test="num==42">
    <div>foo</div>
</if>
```

```html
<else/>
    <div>bar</div>
</else>
```

In this example, the foo `<div>` is only displayed if `num` equals 42; otherwise, the bar `<div>` is shown.

**For loops.** The template engine supports for...in loops, as shown here:

```html
<for each="prop" in "thing">
    <div>{+ prop +}</div>
</for>
```

2http://code.google.com/p/jsdoc-toolkit/wiki/JsPlate
In this example, each property prop in the thing object is outputted inside a <div>.

Comments. Finally, the JsPlate engine supports comments in {# [comment] #} blocks.

Additionally, we have modified JsPlate to support additional constructs in LibX:

Localization definitions. To provide compatibility with the LibX internationalization API, we have created localization constructs that allow messages to be directly included in the layout files:

```template
{DefaultLocale=en_US}
...
{BeginLocale=en_US}
...
{EndLocale}
{BeginLocale=fr}
...
{EndLocale}
{BeginLocale=jp}
...
{EndLocale}
```

The first line specifies the fallback language to be used if the user’s current language is not supported. BeginLocale and EndLocale contain the actual localization JSON. There can be multiple such locale blocks, each which defines a localization for a different language. The actual message definitions in each block must follow the i18n JSON format described in Section 5.2.

Localization messages. Localized messages can then be placed in the layout HTML using {L [property] L}. These blocks are replaced with the localized message corresponding to the given properties. The template engine uses the i18n API to find the messages, which are included in the template using the locale syntax described above.

Processing directives. Finally, we have implemented a custom processing directive, which uses the {P [object], [layout] P} syntax. This directive enables a template to specify children sub-templates to be processed; in other words, this construct enables nested template processing. The first argument is a preference object, which is a child of libxprefs.
The second optional argument specifies a layout to use for processing.

To determine which layout file to load for a given preference object, the engine searches for layouts in the following order:

1. if the optional `layout` argument is supplied, this layout is used.
2. the given preference object may have a `_layout` property that indicates which layout to use. if so, this layout is used if the `layout` argument is not defined.
3. otherwise, a layout matching the `_nodeType` property of the given preference object is used as a last resort. all preference objects have this property.

These rules allow clients to specify a custom layout for their preferences, using a fallback if they do not provide one. Thus, the combination of the processing directive and these rules enables the interface extensibility.

The LibX preferences page shows different tabs to divide the preferences. The tabs include a Browser tab, a Context Menu tab, and a LibApps tab. Each tab is processed using a separate layout. Inside each tab is a collection of preferences relative to that section, and each of these is also defined by a layout. Thus, the layouts form a rooted tree, where each node is processed before its children. `libx.prefs.tmpl` acts as the root layout; this is the first layout processed when the LibX preferences page is opened. This layout contains child layouts corresponding to each tab, and each of these child layouts contains children corresponding to actual segments within the tab. Any layout can have any number of child layouts, and layouts are recursively processed—starting at the root—and completed when the entire tree has been traversed.

### 3.3.2 Browser Preferences

Browser preferences are general, global preferences that are not edition-specific. One example of a browser preference is the new page preference, which specifies whether a new page should
be opened in a new tab, in the current tab, or in a new browser window.

![Browser Preferences](image)

Figure 3.7: The browser preferences tab.

### 3.3.3 Context Menu Preferences

Context menu preferences allow users to select which catalogs and corresponding search options (such as keyword, title, or author) will appear in the right-click context menu. Each search option for a catalog can either be enabled or disabled; thus, the entire context menu preference tree consists of boolean values.

### 3.3.4 LibApp Preferences

The LibApps tab allows users to enable or disable LibApps. The collection of LibApps is presented in a checkbox tree. Additionally, LibApps can specify their own custom preferences as shown in Figure 3.9.
3.4 Loading Editions

To use LibX, users must first load a LibX edition. Editions can be loaded manually in the edition interface, by clicking a specially-crafted link on a web page, or by auto-detection.
Via search interface. Public editions can be loaded from the search interface popup. Upon clicking “Change Edition” in the “About” tab, users will see a search input box. Here, they can type the name of their library. LibX uses the search input to query a JSON service that reports public editions. The service is queried as the user types, providing an experience similar to Google’s autocomplete.

![Figure 3.10: Searching for an edition.](image)

Via links. Editions can also be loaded via links on pages. This is most useful when libraries are linking to their LibX edition; after clicking the link to install LibX, the edition is automatically loaded. This is accomplished using cookies.

We host a simple PHP file, called latest.php, that always links to the latest LibX build. If the edition GET parameter is given, this script also sets a libxedition cookie that saves the given edition. When LibX is loaded, libxedition cookie is detected and the edition is loaded.

Via auto-detection. Finally, editions can be automatically detected using metadata in pages. Whenever a user visits a page, LibX checks for a <meta> tag that specifies an
edition and version. For auto-detection to work, the name attribute of the metadata must be "libxeditioninfo". The content attribute contains the edition ID and edition version, separated by a semicolon. The following is an example of a valid <meta> tag:

<meta name="libxeditioninfo" content="vt;1.5.19" />

If a tag following this format exists in the page, the user will see the option in the search interface as shown in Figure 3.11.

![Figure 3.11: Edition auto-detection on the page.](image-url)
Chapter 4

LibApps

4.1 Motivation

In addition to the basic search interface and context menu, LibX can integrate with pages users visit to provide context-sensitive page transformations. For instance, ISBNs can be automatically linked to locate a resource in a user’s library. This functionality was implemented in LibX 1.5 using JavaScript scripts maintained by LibX developers. Given the power and flexibility of these scripts, edition maintainers wanted to customize them for fine-grained interactivity with their libraries. As a result, we implemented these context-sensitive features as extensible components.

Our initial extensibility model, used in LibX 1.5, allowed edition maintainers to create their own customizable scripts. The model was very similar to Greasemonkey; edition maintainers developed user scripts that would run on the specified pages. However, whereas Greasemonkey scripts are distributed to users by programmers, these LibX scripts were distributed by the edition maintainers. Edition maintainers are generally unfamiliar with JavaScript, so the Greasemonkey model was not effective since maintainers could not implement their customizations.
To give maintainers the same power without requiring JavaScript knowledge, we implemented a user programming language in LibX 2.0. Rather than directly exposing the JavaScript code to the edition maintainers, we developed an abstraction layer that hides the JavaScript code and allows maintainers to work with just the higher level components. We refer to these components as *LibApps*.

## 4.2 Design Model

LibApps, short for library applications, provide the same context-sensitive capabilities as user scripts. These applications offer a wide range of features that tie pages to the user’s library at a fine-grained level. Examples of LibApps include:

**Alert users to ACM digital library subscription:** places static cues on the ACM portal page. When clicked, the cues display a popup tutorial video (Figure 4.1)

**Autolink ISBNs:** ISBNs are automatically linked to the user’s library to determine resource availability (Figure 4.2)

**Link Amazon by ISBN:** uses the ISBN displayed with books to place cues by the book’s title. These cues search the user’s library for the displayed book (Figure 4.3)

LibApps are implemented using JavaScript. However, this JavaScript is hidden to provide a more user-friendly programming model. Rather than directly expose the underlying LibApp code, we have designed a lower-level abstraction called *modules*. Modules are basic building block elements used to form LibApps. A module is a relatively simple piece of JavaScript code that has a single, well-defined function. For instance, a LibApp that displays resource availability could be broken down to the following modules:

1. scrape the page for text strings matching the ISBN pattern
2. determine the availability of the resource corresponding to the ISBN

3. display a notification on whether the item is in stock

Whereas a collection of modules form a LibApp, a collection of LibApps form a package. Packages are used as a grouping mechanism analogous to directories in a file system. Directories can contain subdirectories and files; likewise, packages can contain subpackages and LibApps.

The collection of modules, LibApps, and packages creates a hierarchy. Packages, at the highest level of this hierarchy, contain LibApps and packages; LibApps contain modules;
and modules, at the lowest level of this hierarchy, are composed entirely of JavaScript code.

4.3 Implementation

4.3.1 Atom Feeds

LibApps, packages, and modules are published using the Atom publishing protocol \(^1\). Each LibApp, package, or module is represented in an Atom feed as an `<atom:entry>` element. All `<atom:entry>` elements contain standard Atom fields, such as `<atom:id>`, `<atom:title>`, and `<atom:author>`. In addition to these fields, the `<atom:entry>` element contains children that use the foreign LibX 2.0 namespace.

Packages are specified with the `<libx:package>` element. Its children—which can be either LibApps or other packages—are listed as `<libx:entry>` elements, children of the the `<libx:package>` element. Each `<libx:entry>` element contains a `src` attribute that contain the ID of the entry. The following example shows the LibX Core Autolinking package entry:

```
<entry xmlns="http://www.w3.org/2005/Atom">
  <id>http://libx.org/libx2/libapps/12</id>
  <title>LibX Core Autolinking</title>
  <updated>2009-06-10T14:58:49Z</updated>
  <author>
```

\(^1\)http://bitworking.org/projects/atom/rfc5023.html
This package contains three `<libx:entry>` elements, each of which in this case are LibApps. The `src` attribute specifies either absolute or relative HTTP URLs. If relative, the entry must be in the same feed as the parent package. The package is agnostic to whether its children are LibApps or other packages; this can be determined only by looking at the entries themselves.

LibApps are specified as `<libx:libapp>` elements. Like `<libx:package>` elements, `<libx:libapp>` elements contain `<libx:entry>` children. However, these children must refer to module entries since LibApps are composed entirely of modules. The following example shows the Autolink ISBN LibApp included in the LibX Core Autolinking package:
Modules are specified as `<libx:module>` elements. Modules do not contain child entries; they contain only code. Module code is enclosed by a `<libx:body>` element, as shown here:

```xml
<entry xmlns="http://www.w3.org/2005/Atom">
  <id>http://libx.org/libx2/libapps/17</id>
  <title>Add Tooltips to DOM elements</title>
  <updated>2009-06-10T14:58:49Z</updated>
  <author>
    <name>LibX Team</name>
    <uri>http://libx.org</uri>
    <email>libx.org@gmail.com</email>
  </author>
  <content type="html">Add a tooltip of the given type given node.</content>
  <libx:module xmlns:libx="http://libx.org/xml/libx2">
    <libx:include>./</libx:include>
    <libx:require>legacy-cues</libx:require>
    <libx:guardedby>
      { 
        alnode: libx.space.WILDCARD, 
        type: libx.space.WILDCARD
      }
    </libx:guardedby>
    <libx:body>
      <![CDATA[
        var node = tuple.alnode;
        var type = tuple.type;
        var ondemand = tuple.tooltip['ondemand'];
        var tooltipValue = tuple.tooltip['value'];
        var query = tuple.query;
        var catalog = tuple.catalog;

        if ((tooltipValue === null) || ondemand && (type !== null)) {
          var addTooltipFunc = eval("libx.cues.add"+ type +"MetadataTooltip");
          addTooltipFunc(node, catalog, query);
        } else if (tooltipValue !== null) {
          node.title = tooltipValue;
        }
      ]]>}
    </libx:body>
  </libx:module>
</entry>
```
4.3.2 Using Feeds in the Client

The LibX Core Package contains the standard set of packages, LibApps, and modules that are most frequently used. Edition maintainers can release their own feeds that run either with or instead of the LibX Core Package. End-users are subscribed to the set of feeds associated with an edition, which are run with the LibX extension.

For the client to convert the subscribed feeds to usable code, it must first parse the feed. The LibX API provides the PackageWalker class to handle the retrieval and walking of entries in a feed. Given a package, LibApp, or module URI, a PackageWalker downloads the containing feed as an XML document, locates the entry using XPath expressions, and uses the supplied callback to handle the entry. PackageWalkers follow the visitor pattern [22], so clients can supply a visitor function as the PackageWalker recursively walks the LibApp hierarchy.

Two simple feeds are shown in Figure 4.4. The tree corresponding to these feeds, which represents the in-memory tree traversed by the PackageWalker, is shown in Figure 4.5. For brevity, several atom fields have been omitted.

Entries can contain a number of fields used for LibApp execution. These fields are includes, excludes, requires, overrides, and guardedby.

includes and excludes clauses are among the first fields checked when running a LibApp or module. These fields are regular expressions that test the current page’s URL. If the page’s URL matches an includes expression, the entry is executed; if the page’s URL matches an excludes expression, the entry is not run. If a URL matches both includes and excludes clauses, the excludes takes precedence, and the entry is not executed.

Modules may specify requires fields that list prerequisite resources. These resources are necessary for the module’s execution, so the module does not execute until all required resources have been loaded. Required resources are either JavaScript scripts or CSS stylesheets. One frequently required resource is jQuery.
Users may subscribe to multiple feeds. Most commonly, users will subscribe to the generic LibX Core Package, containing the default set of LibApps, and some additional feed that is more directly tied to their libraries. These additional feeds might alter LibApps from the LibX Core Package to implement some additional enhancements. Consequently, the LibApps that have been changed must be suppressed. This is accomplished with the overrides field.

Figure 4.4: Two separate feeds containing modules, LibApps, and packages.
The overrides field, used in both LibApps and modules, specifies an entry URI that the current entry replaces.

One example of override use is modifying a LibApp that displays cues next to COinS, which are metadata embedded in the page used to identify a resource [14]. Several libraries do not want COinS cues shown on their catalogs, so they can customize the LibApp to contain an excludes clause to omit certain pages. Other than the additional excludes clause, this “new” LibApp is functionally equivalent to the original. Thus, the original LibApp should be suppressed by the modified one, which specifies the original LibApp’s URI in its overrides field.

As shown previously, an ISBN lookup LibApp may be decomposed into the following modules:

1. scrape the page for text strings matching the ISBN pattern
2. determine the availability of the resource corresponding to the ISBN

3. display a notification on whether the item is in stock

The module execution order is essential. That is, the ISBN must be scraped from the page before checking the resource’s availability, and the resource’s availability must be known before the notification can be shown. One obvious approach would be for each module to call its successor; however, this introduces coupling and decreases module reusability. Instead, we use a guardedby clause to enforce an execution order. These guardedby clauses are the backbone of the LibX tuple space implementation, discussed below.

4.3.3 Tuple Spaces

Tuple spaces were first introduced by David Gelernter in the Linda coordination language [13]. In general, tuple spaces are shared object spaces where objects, or tuples, can be stored and retrieved. Clients that use this tuple space specify a pattern that is used to check properties of tuples in the tuple space, and tuples are taken only if the pattern is matched. Once a client has taken a tuple, it executes code that uses the tuple’s properties. When finished, the client can write one or more tuples back to the tuple space for other clients to take and process. This forms an input/output relationship, where the taken tuple is the input, and the written tuple is the output.

Tuple spaces result in low coupling among modules: execution order is based entirely on the tuples available in the tuple space rather than clients directly calling each other. This enables modular code that is agnostic to other clients interacting with it.

In LibX, each executing LibApp creates a separate tuple space for all of its modules to use. When each module is executed, its code is wrapped inside of tuple space take listeners; this way, the inner module code is executed only after the wrapping take listeners have each received a tuple. These listeners wait for tuples matching patterns in the module’s guardedby fields. During the module’s execution, it can place tuples in the tuple space.
Other modules that are guarded by these tuples can then execute. Thus, `guardedby` fields enforce module ordering: module code can be run only once the necessary tuples exist. A simple interaction of modules using the tuple space is illustrated in Figure 4.6.

![Figure 4.6: Modules communicate through a tuple space.](image)
4.4 Security Concerns

LibApps allow the execution of code from unknown sources. This introduces a number of security concerns, for which we have made the following assumptions:

Other extensions are not malicious. Other extensions in the browser are capable of interfering with LibX. This is especially true in Firefox, where extensions have complete, unrestricted control over the browser [4]. Although this is a concern, it is a flaw in Firefox’s extension model, so guarding LibX from other extensions is beyond our control. Therefore, we can only assume that other extensions are benign.

No man-in-the-middle attacks. We must also assume that users are not being subjected to man-in-the-middle attacks. If a malicious party controlled a proxy between the user and the LibApp server, the party could intercept requests and return malicious code instead of valid LibApps.

These are relatively safe assumptions, so ignoring these concerns should cause little harm. However, there are several concerns we must proactively guard against:

Malicious pages. We cannot assume that all webpages are not malicious. In both Firefox and Chrome, LibX has elevated permissions. Calls through the LibX API could run JavaScript code in any domain, and in Firefox, calls through the LibX API could be used to attack the browser itself. If the libx object were exposed to the page, malicious websites could execute attacks simply by exploiting the API. Chrome provides a security model using content scripts, which are isolated from background pages. Thus, unless the API is carelessly exposed via the DOM, code run in content scripts is inherently safe. Firefox, on the other hand, requires sandboxes and object wrappers to emulate this safe environment. LibApp security in Firefox is discussed in further detail in Section 6.2.2.

Malicious LibApp developers. Another concern is malicious LibApps. Because LibApps allow executing arbitrary pieces of code on any page a user visits, they could be abused to violate the user’s privacy, steal sensitive data, or aggressively advertise with popups or page
redirections. Currently there is no safeguard against these kinds of attacks. In the future, however, we plan to implement whitelists in the client. These lists will contain trusted publishers, and publishers not on the whitelist will require explicitly allowing them to run.

4.5 The LibApp Builder

To facilitate users creating LibApps, we have developed the LibApp Builder. The LibApp Builder, currently being developed by Sony Vijay, provides an interface for building package trees. Packages, LibApps, and modules can be imported from other feeds, and they can then be manipulated using the drag-and-drop interface. The corresponding XML is automatically generated, and maintainers can associate their custom feeds with their editions. The LibApp Builder also allows the edition maintainer to modify includes, excludes, requires, and overrides fields for each LibApp. The interface tree presents only the names of modules; the underlying JavaScript code can be separately accessed by users familiar with JavaScript. As a result, this interface allows maintainers with no JavaScript experience to build complete LibApps.
Figure 4.7: The LibApp Builder interface.
Chapter 5

Core API

This chapter discusses the underlying components available through the LibX API. The following is an overview of these components and how they interact with each other and the user interface components.

User preferences API. The user preferences interface is the front-end to the underlying preferences API. This API is also used by all other interface components as described above. The search interface, context menu, and LibApps interact with preferences API directly; the preferences interface, on the other hand, is simply the UI layer that presents these options to the user.

Internationalization API. LibX supports a variety of different languages by grouping strings in language-dependent files. All components that present an interface to the user—the search interface, context menu, LibApps, and preferences interface—make calls to the internationalization API to retrieve these language-specific strings.

Object cache. The object cache, as the name implies, is used to cache different objects. Objects are resources that are fetched externally, such as JSON, XML documents, scripts, and images. Objects in the object cache are saved to disk and persist across browser sessions. The object cache is the core of the LibX API and used by all of the aforementioned interfaces.
and APIs.

**Memory cache.** The memory cache, like the object cache, is used to cache responses. Whereas the object cache caches to disk, the memory cache only caches to memory, and the cache does not persist between browser sessions. The memory cache wraps the `XMLHttpRequest` object. The object cache is built on top of the memory cache, so all external requests from the object cache are forwarded to the memory cache. Although LibX components use the object cache for the majority of requests, the memory cache is used for dynamic queries such as search requests.

**Storage.** The storage component is a wrapper for the browser storage facilities. LibX components that need a persistent storage database, notably the object cache, all use the storage component.

![Diagram of LibX components interaction](image-url)

**Figure 5.1:** The interaction of LibX components.

The interaction of these components is illustrated in Figure 5.1. The components have been grouped into layers to represent their similar dependencies. The top layer consists of the interface components, the middle layer consists of the utility components, and the bottom layer consists of the data components.
5.1 User Preferences API

LibX has a number of features that can be customized on the LibX Preferences page. These include preferences for the browser, the context menu, and LibApps.

5.1.1 XML Preference Files

Using the LibX preferences API, preferences can either be dynamically created or imported from XML files. Elements specified in preference XML files can be either `<category>`, `<preference>`, or `<item>` nodes.

`<category>` nodes are simply used as a grouping mechanism for preferences, and can contain other `<category>` nodes or `<preference>` nodes as children.

`<preference>` nodes, as the name implies, represent actual preferences. `<preference>` nodes have a `type` attribute that must be either `boolean`, `string`, `choice`, or `multichoice`. Boolean and string `<preference>` nodes have no children; their values are specified in the `<value>` attribute. Choice and multichoice `<preference>` nodes, on the other hand, have `<item>` child nodes.

`<item>` nodes are analogous to radio and checkbox groups in HTML. If the `type` attribute of a `<preference>` node is `choice`, it can have only one selected item, like an HTML radio. If a `<preference>` node is of type `multichoice`, any of its items may be selected, like HTML checkboxes. Selected items are designed by setting the `<selected>` attribute to `true`. All `<item>` nodes are string types.

The following is a sample XML file:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE category SYSTEM "http://libx.org/xml/libxprefs.dtd">
<category name="browser" layout="group">
  <preference name="displaypref" type="choice">
    <item type="string" value="newtabsswitch" selected="true"/>
    <item type="string" value="newtab"/>
  </preference>
</category>
```
5.1.2 Saving Preferences

LibX preferences are accessible via the \texttt{libx.prefs} object. When preferences are loaded, either dynamically or as XML, they are attached to this object, and each node is available in its equivalent hierarchical JavaScript representation. For example, in the XML shown above, the displaypref preference could be accessed using \texttt{libx.prefs.browser.displaypref}.

To keep the \texttt{libx.prefs} object persistent, it is saved to LibX storage as a composite XML file. This requires that \texttt{libx.prefs} first be serialized, which is done using the internal serialization visitor. Once \texttt{libx.prefs} has been converted to an XML string and saved, it can be fetched and loaded as an XML document. This means that there is no special code required for loading user's previously saved preferences—they are loaded the same way as any preference XML file. These saved preferences are stored in a single composite XML document named \texttt{userprefs.xml}.

When preferences are loaded from multiple XML files, they may conflict with each other. To resolve this, preferences are merged into the \texttt{libx.prefs} object, and whether they will replace existing values is determined by an overwrite flag. We set this flag to true only when loading the \texttt{userprefs.xml} document (which are the preferences the user has previously saved). This has the desired consequence that if the user is loading a preference XML for the first time, the default values will be used; if the user has previously loaded these preferences, the previously saved values will be used instead.
5.1.3 Component Preferences

Browser Preferences

Since the core browser preferences, or built-in preferences, are identical across all LibX installations, they can be served from a single location. Thus, we have made the built-in preference XML file a bootstrapped resource. Serving the built-in preferences as a bootstrapped file, rather than including them in the LibX build, allows us to incrementally update LibX preferences without requiring a reinstallation.

Context Menu Preferences

Catalogs are specific to an edition, so they are specified in the edition configuration file. In the configuration XML, each catalog node can also have a contextmenuoptions attribute. If given, this attribute provides a semicolon-separated list of search options that will be enabled by default for this catalog. During the LibX initialization, these defaults are merged into the preferences using the dynamic preference API.

LibApp Preferences

While the built-in preferences are common to all LibX installations, each user can have a different set of subscribed LibApps, meaning a different set of LibApp preferences. LibApp preferences are meaningful only if used with their corresponding LibApps, so LibApp preferences are included in the LibApp feeds. Since LibApp feeds are XML files, preferences can be included directly in the feed in the same XML form described above. When feeds are walked using the PackageWalker, any discovered preferences are merged into the libx.prefs object. The following is an example of a LibApp preference as it would appear in a feed:

```xml
<libx:preferences>
  <libx:preference name="shownotify" type="boolean" value="true"/>
</libx:preferences>
```
In addition to preference XML being stored in the feed, LibApps can also create and use preferences dynamically using the LibX preference API. The `libx.prefs.getCategoryForUrl()` method takes an entry URL—either a LibApp or module—and creates a preference category node for that entry. It also takes a template argument, which provides a shorthand for constructing all child preferences for that category. If the preference already exists, the template preference will be merged with the existing preferences (with the overwrite flag set to false). This method also returns the category object so that the preferences can be manipulated programmatically. The following example illustrates the same preference being constructed and used:

```javascript
function showNotify(enabled) {
    if (enabled)
        alert('hello world');
}

// create this category and preference, being enabled by default
var cat = libx.prefs.getCategoryForUrl(libx.libapp.getCurrentLibapp(),
    [{
        name: "shownotify",
        type: "boolean",
        value: "true"
    }]
);

// may or may not show notification, depending on whether this preference has
// been previously loaded. if it was just created, a notification will be
// shown.
showNotify(cat.shownotify.value);

// now change the value of the preference
cat.shownotify.value = false;

// does nothing
showNotify(cat.shownotify.value);
```

In this example, we want our preference to be stored under the LibApp (as opposed to the module), so we provide `libx.libapp.getCurrentLibapp()` as an argument to `libx.prefs.getCategoryForUrl()`. If we want to construct preferences under a module,
we can instead use `libx.libapp.getCurrentModule()`. The decision to store a preference under a LibApp or module must be determined by the module developer. In general, if the preference should be customizable on a per-LibApp basis, the preference should be stored under the running LibApp; if all LibApps using the module will use the same preference, it should be stored under the running module.

Because preferences created in this manner are loaded only once the code is executed, these preferences will not initially appear on the LibX Preferences page.

### 5.2 Internationalization API

LibX is designed to be used with libraries and universities around the world. It is therefore essential that it supports multiple languages to cater to its diverse user base. This is accomplished by factoring out strings and grouping them into language-specific files. If the browser finds a file corresponding to the user’s current language preference, that file is used for strings; otherwise, the default language (usually English) is used as a fallback. This is a common practice in software design, and it is known as internationalization, or i18n for short.

In LibX, all of the components need to be localized. This includes the LibX search interface, context menu, preferences, preferences interface, and LibApps. Components that are built into the LibX extension, such as the search interface and context menu, are relatively straightforward to implement; their localization entries can simply be included in LibX’s primary localization file. On the other hand, remotely retrieved components, such as preference layouts and LibApps, require special consideration since they are included dynamically.
5.2.1 Firefox’s Approach

Firefox allows a locale directory to be specified in the extension manifest. In this directory, DTD files and properties files contain entries that map variables to language-specific strings. DTD files are XML files used by XUL, whereas properties files contain the localizations used by JavaScript.

To use a localization string in JavaScript, Firefox offers the string bundle XPCOM component\(^1\). This component reads properties files found in the extension’s locale directory. String bundles provide the `getString()` method, which gets the message for a variable in the bundle.

Properties files also support placeholders. Placeholders allow language-generic strings to be injected into language-specific messages using the `getFormattedString` method. For example, the following snippet uses placeholders to show a localized message for the context menu:

```plaintext
# an entry in a .properties file
catalogsearch=Search %1$S for ISBN %2$S
```

Here, the catalog name and ISBN would be passed as arguments to `getFormattedString()`.

The Firefox i18n framework works well for the built-in LibX components like the search interface and context menu; localization entries for these components can simply be included in the properties file bundled with the LibX XPI. Remote and bootstrapped items, on the other hand, are more complex. The string bundle component can accept `chrome://` only URLs\(^2\). This is a problem for remote resources in LibX since they can be referenced using only data URIs.

---

\(^1\)https://developer.mozilla.org/en/XUL/stringbundle

\(^2\)https://bugzilla.mozilla.org/show_bug.cgi?id=133698
5.2.2 Chrome’s Approach

In Chrome, all localization strings for each language are stored in a separate `messages.json` file. This is a JSON file that is structured according to the Chrome i18n specifications. The following shows the same catalog search message for Chrome:

```json
"catalogsearch": {
  "message": "Search $catalog$ for ISBN $searchstr$",
  "placeholders": {
    "catalog": { "content": "$1" },
    "searchstr": { "content": "$2" }
  }
},
```

As shown here, placeholders are also supported in Chrome. Though the syntax is different, the concept is the same. Also, localizations in Chrome can be used only in JavaScript; there is no equivalent to Firefox’s dereferencing scheme in XUL.

Like Firefox, Chrome’s i18n framework works well for components bundled with the LibX extension. Chrome requires each language has a single `messages.json` for all localization messages, so localizations for built-in components would go in this file. However, this single localization file requirement makes Chrome even more restrictive than Firefox: other localization files cannot be specified. This makes it impossible to use the i18n framework to load remote localizations for preference layout files or LibApps.

5.2.3 LibX’s Approach

Because each browser uses a different approach for internationalization, we would need to abstract away the differences to create a generic localization framework for LibX. This gives us two options: either change the localization files for each browser, or keep a single localization file and change the framework itself.

---

3http://code.google.com/chrome/extensions/i18n.html
The first approach would involve creating a single source localization file from which browser-specific localization files could be generated. This conversion would take place in the LibX build script. By using this method, we could use the internationalization API given by each browser since our script would generate localization files compatible with those APIs.

The second option would be to implement our own custom internationalization code. Rather than relying on either browser’s API and having to generate localization files for each browser, we could instead create the i18n logic ourselves, allowing all browsers to use the same i18n code (and thus the same localization files). This option has the added benefit of letting us tweak the framework to our needs since we would be in control of the underlying implementation. A custom i18n implementation allows us to process bootstrapped and remote localizations, a feature neither browser supports. Given the advantages, we decided to implement this option. We wanted the i18n framework to contain minimal parsing logic, support fallbacks for missing strings, and support importing messages from multiple sources.

By implementing the internationalization code, we can use any localization file format we choose. We decided to use the Chrome specification for our files, largely because they are in JSON—meaning we would not have to create a custom parser.

We also opted to implement the same fallback logic Chrome uses for missing strings; localization files will be maintained by a number of different users, and the fallback algorithm gracefully handles the inevitable inconsistencies that will arise.

The third requirement is that we should be able to use localization files for bootstrapped and remote components. Here, we borrow Firefox’s concept of string bundles, allowing client code to specify URLs of different localization files. Of course, to be useful, our implementation does not restrict URLs to chrome:// URLs. To use a string bundle object and read locale strings, `libx.locale.getBundle()` must be used.

For components bundled with the LibX extension, such as the LibX search interface or the context menu, we still use a single `messages.json` file containing the translations. These core components are not updated as frequently, so their localizations should not change
often. On the other hand, user preference layouts and LibApps each use different techniques for distributing their localization files.

**Built-in Preferences**

Localization files for preferences are served as separate JSON files. The URL for the localization is constructed relative to path of the preference XML. The directory holding the preference XML must contain a subdirectory named `locales`. This subdirectory must contain other subdirectories of the supported locales, such as `en_US` or `jp`. Each of these directories must contain a JSON file with the same name as the corresponding preference XML name.

Built-in preferences are processed and displayed using the `builtin.tmpl` layout. This layout is responsible for fetching the built-in localization files before processing and displaying the preferences.

**User Preference Layouts**

As described in Section 3.3.1, localization messages are integrated directly into the user preferences layout files. The following example shows the localization embedded into `about-libx.tmpl`, the template used to display information about the current LibX installation:

```json
{DefaultLocale=en_US}
{BeginLocale=en_US}
{
  "adaptedby" : { "message" : "Adapted By" },
  "libxversion" : { "message" : "LibX Version:" },
  "builddate" : { "message" : "Build Date:" },
  "authors" : { "message" : "Authors" },
  "homepage" : { "message" : "LibX Home Page" }
}
{EndLocale}
```

As shown here, the exact same i18n JSON format is used to define localizations in `BeginLocale` and `EndLocale` blocks.
LibApps

LibApps also use localizations for their messages. LibApp localizations are embedded into the feed, using the `<libx:locale>` element placed inside of the `<libx:module>` or `<libx:libapp>` element. The language attribute specifies which language the localization applies to. Exactly one of the `<libx:locale>` elements should also have the default attribute set to true to indicate the fallback language. The following example is the localization used for the jGrowl module in the LibX Core Package:

```xml
<libx:module xmlns:libx="http://libx.org/xml/libx2">
  <libx:locale language="en_US" default="true">
    { "shownotify" : { "message" : "Display popup notification" },
    "dontshowagain" : { "message" : "Don't show me this again" }
  }
</libx:locale>
</libx:module>
```

Each of these localization entries is parsed prior to running the LibApps, and each is available as a string bundle. To access them, the `libx.libapp.getStringBundle()` method can be used. This method takes a single argument, a URL, and returns the string bundle associated with that entry. The `libx.libapp.getCurrentLibapp()` and `libx.libapp.getCurrentModule()` methods return the current libapp and module; these can be used as the URLs to pass to `libx.libapp.getStringBundle()`.

5.3 Storage

Files cached in the object cache need to be stored to disk to make the cache persistent. Both browsers offer several different options for storage. Firefox provides HTML5 local storage, file I/O, and SQLite. Chrome provides HTML5 local storage and Web SQL.
5.3.1 HTML5 Local Storage

HTML5 provides per-origin web storage, or local storage. Local storage provides simple key/value-based storage that persists between browser sessions.

A page’s localStorage object is accessible in the window scope, and entries can be added to localStorage by attaching them as JavaScript expando properties. These properties will persist until they are removed. For instance,

```javascript
// add the foo property to localStorage
localStorage.foo = 'bar';

// foo can be accessed across browser sessions
alert(localStorage.foo); // alerts 'bar'

// remove foo from localStorage
localStorage.removeItem('foo');
```

While HTML5 local storage is easy to use, it has at least one disadvantage: most browsers impose a storage limit. This includes Firefox and Chrome, which each limit local storage to 5MB.

5.3.2 SQLite and Web SQL

Both browsers have support for SQL storage. Chrome supports Web SQL [1], whereas Firefox supports SQLite [19]. While each implementation uses a different API to query the SQL databases, the queries themselves are identical.

We opted to use SQLite and Web SQL for LibX storage. Unlike local storage, the SQL storage implementations do not impose storage size limits. Additionally, the Web SQL and SQLite APIs accept callback functions to asynchronously execute the SQL queries, improving efficiency.
5.3.3 File I/O

Firefox provides file I/O as a third storage option. Instead of being stored in a database, entries can be written directly to files on the disk. This can be useful for debugging purposes; that is, files can be directly added, modified, and deleted using standard tools. Additionally, files can be directly referenced using Firefox `resource:///` URLs, so their absolute paths can be specified in the `src` attribute for images and scripts. Entries in local storage and SQL databases cannot be referenced directly from HTML tags and instead must be incorporated using data URLs.

Despite its advantages, file I/O has several drawbacks. It is available only in Firefox, so supporting file I/O in the LibX API would result in fragmented, browser-specific code that we are aiming to avoid. Also, storing files directly to disk would require implementing file naming scheme. Whereas actual URLs can be used as the keys in local storage and SQL databases, file names cannot contain a number of characters found in URLs (such as the colon and forward slash). By using SQL databases, we avoid this issue.

5.4 Memory Cache

The memory cache is a general, in-memory caching class that wraps the standard JavaScript `XMLHttpRequest`. It exposes a convenient, jQuery-inspired API for all LibX XHRs. When a remote resource is fetched, the memory cache automatically caches the request and `XMLHttpRequest` object using an LRU cache. All future requests for this resource will fetch the cached version, and no XHR will be issued (unless the resource has since been removed from the cache, in which case it must be fetched again). Because the memory cache is maintained only in memory, the cache is discarded when the browser is closed.

One useful feature of the memory cache is its batching of requests for the same resource. If a resource is requested, it is asynchronously fetched while other code continues running,
and its callbacks will be executed once the response is received at some indefinite point in the future. During this waiting period, code may fire additional requests for the same resource. Rather than performing a separate request for each, the memory cache recognizes the requests are for the same URL as the initial request, and they are added to a temporary cache. When the response is finally received, callbacks for these entries are executed using the same response. This eliminates redundant requests from being sent, thus reducing the network overhead.

![Cache layers for a request](image)

Figure 5.2: Cache layers for a request, traversed in order until a hit is found.

### 5.5 Object Cache

The object cache is the storage-based caching implementation built on top of the memory cache. Like the memory cache, the object cache exposes an API similar to jQuery’s `ajax()` method.

Many remote resources used by LibX need to be fetched each time LibX is initialized; these resources include the edition configuration XML, the preference XML files, the bootstrapped code, and LibApp feeds. Thus, the majority of XHRs in LibX go through the object cache. If the resource is not found in the cache, the object cache forwards the request to the memory
cache, then saves the result to disk. As a result, the non-volatile object cache maintains its entries between browser sessions.

The object cache uses the `responseText` XHR property for all of its requests; this allows the responses to be stored directly to disk without having to explicitly serialize them. When requesting a resource from the object cache, the `dataType` property can be set to either `text`, `json`, or `xml`. If the `dataType` is `xml` or `json`, the object’s text is first retrieved, either via the cache or via and `XMLHttpRequest`. The text is then parsed as either XML or JSON and returned to the client.

In addition to the actual object, metadata for each entry is stored in the cache. Object cache metadata consists of the following entries:

- **lastModified**: the time and date the resource was last modified; given by the `Last-Modified` header of the XHR response
- **lastAccessed**: the time and date the resource was last retrieved
- **mimeType**: the MIME type for this resource; given by the `Content-Type` header of the XHR response
- **sha1**: the SHA1 hash of the response data, calculated using LibX’s hash utility
- **expired**: a flag indicating that the object cache should update the entry on the next request for this resource

To fetch a resource using the object cache, the client can call either `get()` or `update()`. If `get()` is called, the object cache first checks the metadata. If it does not exist, an `XMLHttpRequest` is issued. If the metadata does exist, the expired property is checked, and if it expired, `update()` is called. If the metadata is not expired, or if the update from the expired metadata fails, the item is retrieved from the cache.
When calling update(), the client can optionally pass the existing metadata for the resource. If the metadata is given, update() will issue a conditional GET request using the lastModified entry in the metadata. If the data has been modified since the lastModified date, the cache is updated and the new data is given to the client; otherwise, the server is expected to return a 304 response, and the object remains unchanged in the cache.

Figure 5.3: Logic flow for an object cache get() request.

5.5.1 Validation

When the object cache requests a resource, it is not guaranteed to succeed. The Internet is unreliable, so the object cache must be capable of handling failures. The most likely failure is that the resource cannot be reached due to being offline. Alternatively, HTTP errors, such as a 404 error, could occur even if the user is online. In these cases, the object cache checks
for an unsuccessful status code (anything other than 200 or 304) and fires the error callback. When an error is detected, the cache will not be modified.

A more complex scenario occurs when the resource is successfully fetched but the item is still considered invalid. This can occur during any man-in-the-middle attack, but it will most frequently occur when the user is behind a captive portal. A captive portal, or web authentication proxy, is a catch-all technique that requires a user to log in before visiting webpages. Captive portals are used commonly at universities, hotels, airports, and other public locations.

The captive portal intercepts the request, and instead of returning the requested resource, it returns the login page along with an HTTP 200 status code. To the browser—and the object cache—this response is indistinguishable from the actual resource. It would be bad enough to mistake the response as valid just once; it would be far worse to store the response in the cache and return the invalid object for future requests.

Since the XMLHttpRequest object yields no useful information about fake responses, we must examine the data itself. Different kinds of data exhibit defining characteristics, and we can use these characteristics to ensure that the data matches what we expect. For example, configuration XML files can be matched against XPath expressions, the updates.json file (described below) must be JSON-parsable and contain a files property, and layout files must begin with the \{libxtemplate\} header. These checks are performed by validator functions.

A validator function is passed to the object cache when a request is made. The object cache defines a number of public validator functions that can be used in most situations. Or, if necessary, client code can define and use its own validator for requests. If a request succeeds but fails validation, the response is considered an error, and it is not added to the cache. Validators are used only when a request is made since objects already in the cache do not need revalidation. In practice, therefore, validators will not be executed for most object cache requests. However, their use is essential for guaranteeing the cache’s integrity.
5.5.2 Cache Updates

When an entry in the object cache is stored on disk, all future requests for that item will use this cached entry, and no XHR will occur. This means that if a remote resource is updated, users would be stuck with the older, stale version since the updated version would never be fetched. Therefore, we need a way to periodically check for updates for entries in the object cache, and if updates are found, refresh these entries. We do this using update schedulers.

Update schedulers are separate from, but complementary to, the object cache. Given a certain interval, such as four hours, update schedulers will iterate through items in the object cache, then send an XHR with an If-Modified-Since header that is set to the date from that entry’s Last-Modified metadata. In other words, the update scheduler sends conditional GET requests over a specified interval. This minimizes the network resources when checking for updates, and the entire object is returned only if it is updated. If the object has not been updated, the server simply sends back a 304 Not Modified header with no body.

Even though conditional GET requests help reduce the server load, it can still be expensive for the thousands of LibX users to poll the servers for updates to all object cache entries over this interval. It is more efficient to instead check for a root entry for updates. If the root is updated, we then check its children for updates; if the root is not updated, we can assume that the children have not changed since the last update.

Configuration XML files, LibApp feeds, and bootstrapped scripts are largely independent from each other and have different update requirements; therefore, a custom update scheduler implementation is used for each.

Edition Configurations

The edition configuration XML file is the simplest case for updating. Each edition configuration file has an <additionalfiles> entry that lists dependent resources. Thus, the
edition configuration file can act as the root node. The update scheduler can simply check the edition configuration file’s last modified date, and if it is not modified, all dependent resources listed in its `<additionalfiles>` do not need to be checked. If a child entry does change, the modification date for the configuration file should be updated (by using the Linux touch command, for example).

When the update scheduler sees the root has been updated, it then iterates through and checks the modification date for each child. In this case, checking each child for updates is suboptimal compared to a solution that explicitly names children that have changed. However, edition configuration files generally list only a few dependent resources, so the penalty is minimal.

**Bootstrapped Scripts**

Unlike resources listed in the edition configuration file, bootstrapped files have no equivalent master file that can be used as the update scheduler root. Therefore, we must create the root that exists solely for the update scheduler, and update this root whenever a bootstrapped file has changed. We accomplish this using a Perl script that recursively iterates all files in the bootstrapped directory and builds a root file, `updates.json`, that lists as bootstrapped files as children. Whenever a bootstrapped file is modified, `updates.json` is updated so that the update scheduler can detect changes.

However, whereas the edition configuration root will list only a few children, the bootstrap root file has dozens more. If we were to update a single bootstrapped file and mark the root as updated using this method, a conditional GET would be executed for every child listed in the root, even though only one would be changed. Thus, instead of solely relying on last modified dates to perform these updates, we can also use SHA1 hashes. When the Perl script creates a directory listing of each file in the bootstrapped directory, it also includes a SHA1 hash of each file. Each entry in the object cache includes its SHA1 hash in its metadata. Consequently, when the update scheduler reads the root bootstrap file, it can compare each
SHA1 metadata entry from the object cache to the SHA1 listed in the root file. If the hashes do not match, the child needs to be updated. If the hashes do match, no conditional GET request is made, avoiding unnecessary update checks.

**LibApp Feeds**

LibApp feeds contain a collection of packages, LibApps, and modules, which can be recursively iterated using the `PackageWalker`. Each entry in a feed may or may not be local to that feed. When updating a feed, a full package walk is required in order to make sure that any externally referenced entries are also updated.

The collection of entries in feeds form a rooted, directed graph. Cycles, while uncommon, are still possible. We therefore use a marking algorithm when walking entries in feeds. Once an entry has been checked, it is marked as visited for this walk, and if it is a child of another entry, it will not be checked again.

Each subscribed feed acts as a root, so a different update scheduler is required for each feed the user is subscribed to. Unlike the edition configuration and bootstrap update schedulers, all children of the root must be checked for updates, regardless of whether the root itself has been updated. This is because feeds may refer to other external feeds which are maintained independently from their referrers.

**User Preference Layouts**

The last type of update scheduler is used for layouts. The layout update scheduler is required for third party layout files which are not in the LibX bootstrapped directory. Unlike the other update schedulers, the layout scheduler does not have a root file. Instead, it scans the object cache for files that do not begin with the LibX bootstrap URL prefix and do not end in `.tmpl`. It then pulls the metadata for each of these files and sets the `expired` property to `true`. 
The layout update scheduler is the most primitive of the schedulers. It does not initiate any update requests on its own; it only marks that entries need to be updated on their next use. The actual update occurs when the get() request is made to the object cache. The object cache sees that the expired flag has been set and updates the item accordingly.
Chapter 6

Browser-specific Implementation

Because Mozilla Firefox and Google Chrome use different extension models, parts of the API required browser-specific implementation. We designed LibX 2.0 with these two browsers as our target platforms; as a result, LibX 2.0 is fully implemented in each browser. This chapter discusses the browser-specific implementations for Firefox and Chrome.

Other browsers, such as Internet Explorer and Opera, are not yet supported for LibX 2.0. Since much of the codebase has been partitioned into separate browser-dependent and browser-independent directories, we anticipate that supporting these browsers will require mostly browser-specific implementation; the majority of the codebase should be reusable with little modification. However, we will not know if our design is successful until LibX 2.0 is ported to other browsers.
6.1 Google Chrome

6.1.1 Execution Environment

As discussed in Section 2.3.1, Chrome is divided into multiple processes. Each tab in the browser represents a separate process, and each extension uses a separate process for its background page. These processes communicate with each other using Chrome’s message passing API.

Like most Chrome extensions, LibX has its own background page, invisible to the user. This page is opened immediately when the browser is opened and remains open until the browser is closed. The background page includes all of the LibX global scripts. These are responsible for loading the core components, initializing LibX, and loading the user’s edition. Consequently, this background process contains the global libx object that provides access to the LibX API.

Content pages, which run the content scripts designated in LibX manifest.json file, are in processes separate from the LibX background page. However, they still need to interact with LibX and must do so using Chrome’s messaging API. Additionally, to fulfill our design goal of code reuse and promote code consistency, LibX content scripts should be able to access the LibX API. This poses an interesting problem: the content scripts need to access the libx object in a different process to use the API.

One potential solution is to wrap the entire LibX API around Chrome’s sendRequest and onRequest messaging functions, making these calls transparent. This technique is a JavaScript implementation of RPC [28]. The following shows an example of how it can be used. Assume we want to wrap the common case of jQuery’s get(), which accepts a URL and success callback:

```javascript
/* *******************************************************************************
* get() as it would be used on a standard webpage.
* for consistency, this call should look the same when using RPC.
*/
```
Figure 6.1: Chrome execution environment.

```javascript
$.get('http://libx.org/', function (result) {
    console.log(result);
});

/******************************************************************************
* the wrapper created in the content script.
* this function transparently forwards the request to the background page.
*/

$.get = function (url, success) {
    chrome.extension.sendMessage({ get: url }, function (response) {
        success(response);
    });
};

/******************************************************************************
* the listener on the background page.
* this accepts the forwarded request, executes $.get(), then sends the response
* back to the caller.
*/

chrome.extension.onRequest.addListener(function (request, sender, sendResponse) {
    var url = request.get;
    if (url) {
```
With the wrapping function defined in the content script and the listener defined in the background page, calls to `get()` in the content script are transparently forwarded to the background page, which then forwards the response back to the content script where the callback is executed. `get()` is a wrapper for `XMLHttpRequest` and accepts a callback to be executed upon success. Because of this asynchronous callback design, this technique works well for `get()`; the success callback can wait until the background page has responded.

Unfortunately, much of the LibX API is synchronous. Synchronous wrappers cannot wrap asynchronous methods since a return value is required immediately. This leaves only two options: either rewrite the entire LibX API to support asynchronous callbacks, or make the `libx` object available locally so that it can be used without requiring messages to the background process. Other than rewriting the entire LibX API to be asynchronous, the only option is to recreate the entire LibX API in the content script’s process so it is available locally.

In order to make the `libx` object available for content scripts, the same initialization procedure used in the background page needs to take place for each page load. This includes running many of the scripts that define the LibX API, loading the preferences and localizations, and loading the edition configuration.

On the other hand, we cannot directly include several components that would not work in page processes, such as the memory cache. The memory cache is a wrapper for `XMLHttpRequests`, which can be cross-domain only if executed within an extension process. If the memory cache were included as a content script, the `XMLHttpRequest` object would be the one exposed by the page’s window, meaning it would not have cross-domain privileges.
Despite these limitations, these components must still be accessible since much of the API depends on them. Since they require access to the background page’s objects—such as its web database and privileged XHR—we must create custom implementations of these components, specifically designed to run in page processes. To do so, we can wrap Chrome’s messaging API as discussed above. The RPC implementation is the same for each method: create a wrapper that sends a request to the background page, call the method in the background page using the arguments given by the request, then forward the response back to the content script to execute the callbacks.

To simplify this process, `magicImport` was created as a convenience function. `magicImport` is a generic wrapper generator that can construct the RPC wrappers given a function name string. Assume the client page wants to call a function in the background page called `someFunction`. First, the wrapper scans the list of arguments, searching for function arguments. Each function is given an ID, then stored in an array that maps the ID to the corresponding function. The function argument is then replaced with a function stub holding the ID. The marshalled arguments are then sent to the background page. Here, each function stub is replaced with a wrapper function for `sendResponse`, the callback function provided by Chrome. The function reference is obtained using the function name string given by the client (in this case, ”someFunction”), and the function is called with the arguments given by the client. When the function invokes a callback, it fires the wrapper function generated by the stub. This, in turn, triggers the `sendResponse` call which sends the function ID and callback arguments back to the client. Finally, the client finds the ID saved in the function and executes actual callbacks given to the `someFunction` stub.
6.2 Mozilla Firefox

6.2.1 Execution Environment

In Firefox, extensions and pages all run in the same process. This means that content scripts can communicate directly with the global libx object without resorting to RPC. Regardless, LibX code still needs to be run in separate environments; that is, content scripts need to be run each page, and global scripts need to be run in a single global context.

Firefox does not support HTML background pages like Google Chrome. To make extension code behave like a background page, it must run only once when Firefox is loaded. This is accomplished by making a LibX XPCOM component. XPCOM, short for Cross Platform Component Object Model, allows for the creation of a module whose API can be accessed
anywhere in the browser code [29].

XPCOM components can be created using JavaScript. There is no available DOM, however,
and no `window` object containing common JavaScript utility functions. These functions are themselves implemented as XPCOM components, so they can still be used if imported. One such example is JavaScript’s `setTimeout` function. This function accepts a callback and a timeout value, and executes the callback after the timeout has occurred. To use `setTimeout` in the LibX component, the following code is used:

```javascript
function setTimer(callback, timeout, nsITimer.TYPE) {
  var timer = Components.classes['@mozilla.org/timer;1'].createInstance(Components.interfaces.nsITimer);
  timer.initWithCallback({
    notify: function (timer) {
      if (typeof callback === "string")
        eval(callback);
      else
        callback();
    },
  }, timeout, nsITimer.TYPE);
  return timer;
}

libx.utils.timer.setTimeout = function (callback, timeout) {
  setTimer(callback, timeout, Components.interfaces.nsITimer.TYPE.ONE_SHOT);
}
```

This code imports the `nsITimer` interface and initializes a timer using the interface’s API. The LibX API hides this component call and exposes a `setTimeout` function identical to what would be used on a webpage. This technique of wrapping components to expose their DOM `window` equivalents is used frequently in the LibX Firefox-dependent code. By factoring out XPCOM component calls, we can create a browser-independent API to help achieve our code reuse goal.

The LibX component merges the browser-independent and browser-dependent code to construct the `libx` object holding the LibX API. This object is then exposed when the component created, and the component can be imported to access the `libx` object. The XPCOM component is constructed only once per Firefox application instance and is used across Firefox windows. Thus, the LibX XPCOM component serves as the Firefox equivalent to Chrome’s background page.
Firefox executes content scripts by injecting them into each loaded page. Though pages are not isolated in separate processes as in Chrome, they still provide an environment different from the LibX component; namely, the DOM is available. Content scripts assume the `window` object and DOM are accessible.

In addition to global and content scripts, LibX uses a third set of scripts for Firefox. These scripts are called *window scripts*. Each browser window in Firefox is a different XUL window, meaning the LibX interface must be integrated into each. The window scripts are Firefox-specific since Chrome does not require per-window initialization. These scripts included in the `libx.xul` overlay which is automatically applied when a new Firefox window is opened.

![Firefox execution environment](image)

**Figure 6.4:** Firefox execution environment.

### 6.2.2 LibApps

In Firefox, extensions and pages all run in the same process. This means that we can directly access the LibX API from LibApps, making the inter-process messaging problem a non-issue. However, Firefox does not support isolated worlds and content scripts, so the LibApp code must be executed safely using other means.

By default, when the Firefox chrome accesses a window, this window is wrapped using an
XrayWrapper\textsuperscript{1}. When the Firefox chrome sets properties on wrapped windows, the page cannot access them, and vice-versa. This feature prohibits malicious websites from replacing the default JavaScript functions and exploiting code in the extension. With a few exceptions, a wrapped window can be treated as if it were the actual unsafe window.

In addition to the XrayWrapper, we also need to specify the environment to run the LibApps in; that is, LibApps need to run in the page context with access to the \texttt{window} object and DOM. This environment is created in Firefox using a \textit{sandbox}. In Firefox, the sandbox constructor accepts two primary arguments: the execution permissions and an object to use as the sandbox’s global scope. By using a sandbox, we can easily construct and tear down a separate environment to run LibApps without polluting the extension’s global scope.

### 6.2.3 Search Interface Popup

Implementing the search interface popup in Firefox required several iterations. The most obvious solution is to include an iframe inside of a XUL popup panel; however, this results in a typing bug under Microsoft Windows\textsuperscript{2}. As a workaround for this problem, a new Firefox window is created using \texttt{window.open()}. With the \texttt{chromehidden} flag, the window does not have the standard window decorations (such as the title bar). Also, with the \texttt{dependent} flag, the window does not appear as a separate entry in the Windows taskbar.

XUL popup panels have a \texttt{position} property that allows them to be positioned relative to the button, and this property is used on Linux and Mac. However, this property is not available using the workaround for Windows, so the position must be explicitly set. The position is calculated using the dimensions of the user’s screen, the LibX button, and the popup.

Finally, in order to correctly display the popup as it appears in Chrome, the popup window must dynamically resize based on its contents. Iframes have a static width and height, so we

\textsuperscript{1}https://developer.mozilla.org/en/XPCNativeWrapper

\textsuperscript{2}https://bugzilla.mozilla.org/show_bug.cgi?id=385609
must explicitly resize the iframe whenever the size of its contents change—for instance, when switching between the large and mini search interfaces. There is no DOM event fired specifically for changes to the document size, so we listen for the generic `DOMSubtreeModified` event, which is fired on every single change to the DOM tree.
Chapter 7

Related Work

7.1 Zotero

Zotero, like LibX, is an extension aimed toward simplifying research. Zotero is used to collect, sync, and share bibliographic resources. LibX, on the other hand, provides search capabilities and increased interactivity. The overlap between LibX and Zotero is minimal, and the two extensions complement each other in the research process [26]: LibX can be used to locate resources, and Zotero can then be used to compile a list for a bibliography.

Some features of Zotero may be worth integrating into LibX in the future, possibly even as LibApps. LibApps are extremely versatile, and the LibX API is rich enough that some of the core functionality of Zotero could be implemented. For instance, a LibApp could be written to handle ISBNs and their resources, LibX storage could be used to save them, and a layout could be designed to display them in the preferences. This functionality could all be implemented by a third party developer with no modifications to LibX itself.
7.2 Conduit Toolbar

Conduit is a tool used to create toolbars and web apps. Some libraries opt to use Conduit to power their own library toolbars. However, these toolbars generally contain just a subset of a LibX’s functionality.

The Conduit API is feature-rich, and the same Conduit web app can work across Internet Explorer, Firefox, Chrome, and Safari. Conduit is therefore an attractive framework for LibX as it could drastically reduce the codebase by eliminating all browser-dependent code.

7.3 Greasemonkey

Greasemonkey [5] is a Firefox extension used to load user scripts. When you install a user script to be used with Greasemonkey, it contains a set of filters that designate which pages the script should be run on. When visiting one of these pages, the script is injected into the page. Conceptually, this is identical to a LibApp, and the two are, in fact, quite similar. Most (if not all) scripts in Greasemonkey could be implemented as LibApps. The opposite may also be true; however, it would require much more work without the convenience of the LibX API.

Perhaps most importantly, LibApps have complete access to the LibX API. Parsing for COinS, ISBNs, DOIs, etc., along with support for OpenURL, edition catalogs, and proxies are features available to LibApps. Most of this functionality could be duplicated in Greasemonkey, but a great deal of configuration—from both libraries and end-users—would be required to implement the existing set of LibApps. The modular composition of LibApps makes it simple for developers and users alike to tweak, create, and share their applications.
Chapter 8

Conclusion

8.1 Future Work

LibX 2.0 was designed to work with Mozilla Firefox and Google Chrome. Other browsers, most notably Microsoft Internet Explorer, cannot currently use LibX. Internet Explorer’s extension model relies on the .NET framework, and it requires add-ons be written in C# or VB.NET. Consequently, an IE-specific framework for LibX should be implemented that is compatible with the JavaScript LibX API. If this backbone were developed, the rest of the browser-independent API should be usable with few changes—assuming our browser-portable design was effective.

Unfortunately, developing an IE framework that is fully compatible with the LibX API is a time-consuming task. LibX is actively developed, and there are still frequent changes to the API. These changes occasionally require modifications to the browser-dependent code, and in the case of IE, would likely require changes to the VB.NET or C# code. In fact, this problem has already occurred in the past. LibX 1.5 for Internet Explorer [3], developed by Nathan Baker, was a largely complete implementation. The advent of LibX 2.0, however, required too many changes to the codebase to keep the LibX for IE updated. Because of the
difficulty of maintaining an IE-compatible LibX, it may be most efficient to wait until the LibX API is relatively frozen before porting it to Internet Explorer.

Many users also prefer to use other browsers, including Opera and Safari. These browsers have considerably smaller user bases, so they will likely never be compatible with the LibX extension. However, the LibX API is written entirely in JavaScript, it is still possible to use the code in other, unsupported browsers—even if the LibX extension is not installed. This can be accomplished using bookmarklets.

Bookmarklets are pieces of JavaScript that, as the name implies, can be bookmarked. When clicked, a bookmarklet runs on the current page. Just like all scripts in a page, a bookmarklet is active only while that page is open. Navigating to another page requires the bookmarklet to be clicked to be loaded again. Since LibX is created entirely using standard web languages—HTML, CSS, and JavaScript—it is possible to include at least part of LibX in a bookmarklet. We’ve already created a simple bookmarklet \(^1\) that includes the basic search interface functionality in an iframe, like loading an edition, searching catalogs, and visiting the edition’s links. HTML5 local storage is used to save the user’s edition between page loads. This bookmarklet is hosted on libx.org, so most resources, including bootstrapped scripts, can be fetched without cross-domain concerns. Data from external sites can still be included using JSON-P [20]; alternatively, a proxy could be set up on libx.org that external resources and forwards them to the bookmarklet.

The existing bookmarklet was created with relatively little effort, so using the current LibX code in a bookmarklet should be similar. Some features would require more work to implement. LibApps, in particular, would require additional code since they access the page’s DOM outside of the bookmarklet iframe (and thus in a different domain). However, HTML5’s postMessage [17] enables cross-domain communication across windows and frames, and it would likely be sufficient to enable LibApps. In fact, using the postMessage message-based implementation to provide the LibX API would be similar to importing the API in content

\(^1\)http://libx.org/libx-bmtest/src/base/bookmarklet/
scripts in Chrome. If most of LibX’s features, including LibApps, were implemented using
bookmarklets, bookmarklets may even prove to be a suitable substitution for a LibX IE
extension.

8.2 Summary

LibX 2.0 shares the majority of its code between Mozilla Firefox and Google Chrome. We
accomplished this by separating the browser-specific code from the browser-independent
code; only the browser-dependent parts needed to be implemented in each. This foundation
should make future browsers and environments much easier to support. Our bookmarklet
implementation, designed from the same portable codebase, was implemented with relative
ease, evidencing our design’s effectiveness.

We have designed many LibX 2.0 components to be extensible. Existing components in
LibX, such as the user preferences interface and the internationalization implementation,
now support user-contributed content. LibApps, also designed for extensibility, allow users
to create their own library applications, including ones we did not anticipate when designing
LibX.

Finally, LibX 2.0 is designed to be updatable. Whereas the existing LibX implementation
required extension rebuilds for every change to the API or edition configuration, LibX 2.0
dynamically updates many of these components during runtime. Updatability is especially
useful in Firefox where extension updates are invasive.

As a whole, these contributions make LibX 2.0 more maintainable than its predecessor:
browser portability reduces duplicate implementation, extensibility removes burden from
the developers while gives power to the users, and updatability simplifies the release process.
Bibliography


