The LibX LibApp Builder

Sony Vijay

Thesis 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 H. Edwards

September 20, 2013
Blacksburg, Virginia

Keywords: LibX, Browser Plugin, Digital Library, End-User Programming, LibApp Language Model, LibApp, LibX Feed, AJAX, XML Database

© Copyright 2013, Sony Vijay
ABSTRACT

LibX is a browser extension that provides direct access to library resources. LibX enables users to add additional features to a webpage, such as placing a tutorial video on a digital library homepage. LibX achieves this ability of enhancing web pages through library applications, called LibApps. A LibApp examines a webpage, extracts and processes information of the page, and modifies the web content. It is possible to build an unlimited number of LibApps and enhance web pages in numerous ways. The developers of LibX team cannot build all possible LibApps by themselves. Hence, we decided to create an environment that allows users to create and share LibApps, thereby creating an eco-system of library applications.

We developed the LibApp Builder, a cloud-based end-user programming tool that assists users in creating customized library applications with minimal effort. We designed an easy-to-understand meta-design language model with modularized, reusable components. The LibApp language model is designed to hide the complex programming details from the target audiences who are mostly non-technical users, primarily librarians.

The LibApp Builder is a web-based editor that allows users to build and test LibApps in an executable environment. A built-in publishing mechanism allows users to group LibApps into packages and publish them in AtomPub format. Any user can directly reuse or adapt published components as required. Two error checking mechanisms have been built into the LibApp Builder viz., type checking and semantic checking to enhance user experience and reduce debugging effort. Additionally, the web interface displays help tooltips to guide users through the process of building a LibApp.

We adhered to good software engineering practices such as the agile development model and the model-view-controller design paradigm. The LibApp Builder is built with the ZK AJAX framework and provides a rich interactive user interface. The LibApp Builder is integrated with an optimized full-text, fuzzy search engine and facilitates faceted search by exploiting the BaseX XML database system and XPath/XQuery processor. Users can locate and reuse existing language components through the search interface. To summarize, the LibApp Builder is a community platform for librarians to create, adapt and share LibApps.
Acknowledgements

“When you set sail for Ithaca, wish for the road to be long, full of adventures, full of knowledge.”*

I set sail to Virginia Tech with bright hope and aspiration, the journey has been rich and enlightening. Firstly, I express my deep gratitude to Dr. Godmar Back for providing me the opportunity to work on the LibApp Builder research project. Dr. Back is a wealth of knowledge and working with him has been very enriching and insightful. Within a month of meeting him, I was amazed at his zeal for perfection and attention to detail, which inspired me to write perfectly structured code and stretch my limits. The best part about working with Dr. Back is that he promptly answers any number of questions in detail with utmost patience. We have had many lengthy discussions about the research project, which helped me immensely. I thank Dr. Back for his constant support and academic guidance.

I thank Dr. Stephen Edwards and Dr. Eli Tilevich for being on my thesis committee. Academic system in the United States is in many ways different compared to that of India and I found my way through it beginning with the Programming Languages course by Dr. Edwards in my first semester. I thank him for paving my path into the graduate life. I thank Dr. Tilevich for his constructive academic and professional advice. He also created a lively atmosphere at Corporate Research Center, CRC with his witty remarks and humorous one liners.

I am grateful to Tilottama Gaat for helping me overcome my initial hiccups when I started my thesis research. She has been a helpful mentor, counsellor and a friend.

I thank Balaji, Vyas, Hari and Vishwas for making Corporate Research Center a friendly environment to work. I totally miss those refreshing tea breaks and delightful discussions, which ranged from philosophy to religion and from science to graduate life.

I found a home away from home at Blacksburg and it wouldn’t have been possible without my roommates Manisha Rai and Harshda Vabale. We bonded like sisters and shared many beautiful moments together. Harshda often pampered us with great food and her cheerful disposition. Her biryani is so delicious that she could give some of the best Indian restaurants a run for their money. My other roommate, Manisha was as nocturnal as I was. It was quite comforting to know that she would be awake when I returned home from CRC way past
mid-night and she mostly welcomed me with a cup of tea. I looked forward to and will always cherish those refreshing late night conversations.

I thank two of my dearest friends; Anusha Parthasarathy and Siddhartha Gutha for being my pillars of strength and helping me maintain my emotional balance. I thank Venkat Medhaj who helped me in getting acclimatized to Blacksburg. I thank Mayank Daga for giving me company to work whenever possible.

I am thankful to Starbucks and Math Emporium for providing a perfect environment to work. I thank the Safe Ride service for escorting me from CRC to Math Emporium countless times in the wee hours.

I thank my husband, Kartheek Pathuri, for showering me with love and being my companion and strength in the last mile of my research work.

“The Lestrygonians and the Cyclopes, an angry Poseidon - do not fear. You will never find such on your path, if your thoughts remain lofty, and your spirit and body are touched by a fine emotion.”

I cannot express my gratitude to my parents in words for encouraging me to pursue my masters’ degree. My father: an Ashtavadhani, a multi-linguist, and a genius. I thank him for standing by me fiercely in this choice despite many odds. My mother: an attorney, a dreamer, and a winner. I thank her for building my emotional strength both directly and indirectly from a very young age. I also thank my brother for his extensive support and love.

“Always keep Ithaca in your mind. To arrive there is your final destination.”

*Lines from one my favorite poems, Ithaca by Constantine P. Cavafy.*
Contents

1 Introduction 1
  1.1 Motivation ......................................................... 1
  1.2 LibX LibApp Builder ............................................. 4
    1.2.1 Live LibApps ............................................... 7
  1.3 Core Contributions ............................................. 8
    1.3.1 Web Editor .................................................. 8
    1.3.2 Storage Mechanism for Distributed Language Components . 9
    1.3.3 Representing Components in a Hierarchical Structure .... 11
    1.3.4 Assistive User Interface ................................... 12
    1.3.5 Building a Community Platform ............................ 13
    1.3.6 Search Interface ............................................ 14
    1.3.7 Error Checking .............................................. 15
    1.3.8 Meta-Design Approach ..................................... 16
    1.3.9 Summary of Contributions .................................. 17

2 Background 20
  2.1 LibX Browser Plugin ............................................ 20
  2.2 The Edition Builder ............................................. 21
  2.3 LibApp Language Model ......................................... 22
    2.3.1 Syntax Model ............................................... 22
    2.3.2 Semantic Model ............................................. 28
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.4.2</td>
<td>Edit XML Attribute</td>
<td>89</td>
</tr>
<tr>
<td>4.5</td>
<td>ZK-Server Centric Framework</td>
<td>90</td>
</tr>
<tr>
<td>4.5.1</td>
<td>ZK Include Component</td>
<td>90</td>
</tr>
<tr>
<td>4.5.2</td>
<td>ZK Generic Forward Composer</td>
<td>91</td>
</tr>
<tr>
<td>4.5.3</td>
<td>ZK Annotations</td>
<td>93</td>
</tr>
<tr>
<td>5</td>
<td>Related Work</td>
<td>96</td>
</tr>
<tr>
<td>6</td>
<td>Conclusion</td>
<td>100</td>
</tr>
<tr>
<td>6.1</td>
<td>Summary</td>
<td>100</td>
</tr>
<tr>
<td>6.2</td>
<td>Future Work</td>
<td>102</td>
</tr>
<tr>
<td></td>
<td>Bibliography</td>
<td>104</td>
</tr>
</tbody>
</table>
List of Figures

1.1 Show item availability on amazon.com. This figure includes the amazon.com book search result page for the book Linear Algebra (Dover Books on Mathematics) by Georgi E. Shilov. Used under fair use, 2013.

1.2 Grouping modules to create a LibApp

1.3 Distribution of LibApps to end users

1.4 Alert users to ACM digital library subscription

1.5 A directed acyclic graph of entries

1.6 Package Tree Component

1.7 A Help tooltip: Describes the functionality of an include clause

1.8 An Error Message: A feed must be published at a valid absolute URI

2.1 Communication of modules via tuple space. This figure includes the amazon.com book search result page for the book Linear Algebra (Dover Books on Mathematics) by Georgi E. Shilov. Used under fair use, 2013.

3.1 User Registration

3.2 User Login

3.3 User Account Settings

3.4 Feed Menu

3.5 Creating a Feed

3.6 Opening a feed

3.7 Editing feed properties

3.8 Publishing a feed

3.9 Opening feed revisions
Chapter 1

Introduction

1.1 Motivation

Traditionally, academics heavily relied on library catalogs or Online Public Access Catalogs (OPACs) for conducting scholarly research. OPACs are tied to libraries and therefore provide content of high academic value. Libraries are beginning to offer powerful services, such as Summon [5] and Primo [38], but their interfaces remain complex, rendering them difficult to use. However, rather than understanding how to use OPACs, students are relying on general search engines because of their simple, usable interfaces. Consequently, the use of library catalogs is diminishing among students. To address this issue, Back et al. developed LibX [16], a browser plugin that facilitates instant access to library services.

LibX enhances the functionality of web pages by integrating useful library services into
specific sites. For example, if a user searches for a book on amazon.com, a notification message is displayed in the search result which indicates the availability of the book in the user’s library collection, as shown in Figure 1.1. Initially, LibX 1.5 performed such transformations in web pages by executing user scripts. Each user script was comprised of a single large piece of non-modular JavaScript code snippet. In this earlier model, librarians developed their own customizable scripts for specific web pages and distributed their LibX scripts to users.

Users created monolithic scripts for LibX 1.5 but this design was not very efficient because other librarians found it difficult to customize or share these non-modular user scripts. One had to copy and understand the entire JavaScript code to make minor modifications to a LibX script. The user script model was also inadequate because librarians often have a non-technical background. Hence, Nicholson and Back built a new version of the browser...
plugin i.e., LibX 2.0, that performs web page transformations by executing library applications, known as LibApps. A LibApp executes context-sensitive, per-page transformations to integrate library services on specific web-sites. An end-user programming language \[34\], called LibApp language model was designed through which librarians can create LibApps, without using complex JavaScript programming.

Instead of exposing the complete JavaScript code of a library application directly, a lower-level abstraction was created by decomposing the script into smaller units called modules. Typical tasks involved in building a LibApp include examining a web page, extracting and processing information, and modifying the web page. Each of these tasks is separated and executed through modules. A module contains a small piece of JavaScript code along with a metadata description of its input and/or output and performs a specific task. A group of compatible modules are pieced together to create a LibApp. For example, the LibApp which displays the availability of a book on amazon.com search result page is comprised of four sub-tasks: 1) scraping the ISBN of the book from the search result page 2) finding the position at which the title of the book is displayed in the web page 3) determining availability of the book in user’s primary catalog using the ISBN and 4) displaying the availability of the book next to the title in the web page. Each of these tasks is performed by a distinct module that contains a piece of generic code snippet and these modules are grouped together to form the Show Item Availability on Amazon LibApp as shown in Figure 1.2.

Analogous to storing files and folders on a computer, a hierarchical structure was designed to store the components of the LibApp language model. A group of related LibApps are
further grouped together into a component called package which in turn may be grouped into higher-level packages. In short, a **package** is defined as a tree of packages, LibApps and modules. These language components are published and distributed to end users via the AtomPub format [25], similar to RSS. Each published package, LibApp, or a module is represented inside an **Atom feed** as an `<atom:entry>`. Throughout this document, an entry refers to an atom entry which could be a package, LibApp, or a module.

LibApps deliver the same tasks that were earlier achieved through user scripts but are much easier to understand. Potentially, countless number of LibApps can be developed but it not practical for a small team of LibX developers to build all possible library applications. A suitable environment is required to facilitate users to utilize the LibApp language model for building and sharing LibApps. Therefore, we designed an end-user development framework called the LibX LibApp Builder to assist users in building LibApps with minimum programming effort.

### 1.2 LibX LibApp Builder

The LibApp Builder creates a cohesive platform for librarians to create, adapt, and share LibApps. Our aim is to garner maximal user contribution through this platform, thereby creating an eco-system of library applications. LibX 2.0 enhances the functionality of web pages by executing LibApps, each containing an unordered collection of modules.

Librarians of different universities adapt LibX as per the requirements of their institution by
creating custom versions of LibX, known as LibX Editions, through a tool called the Edition Builder \cite{23}. A LibX edition describes all the features supported by a library. Edition maintainers create customized editions of LibX and distribute them to end-users of LibX. Edition maintainers add a set of LibX packages to their editions and the users of these editions are automatically subscribed to the underlying LibApps of these packages. LibX users download and install the LibX browser extension and then load a specific LibX edition into their extension. All features of the loaded edition are then applied to the installed LibX extension. LibX users are subscribed to the LibX Core Package which contains a default
set of LibApps. A few LibApps of this default core package are used as examples in this document.

Figure 1.3 shows how the LibApp Builder and the Edition Builder relate to the LibX 2.0 plugin. A LibApp Builder user creates LibApps, groups them into packages, and distributes these packages via the LibApp Builder. An edition maintainer creates a custom edition of LibX 2.0 plugin for a specific library and adds chosen packages of LibApps to the edition. End-users download the LibX plugin, subscribe to a chosen LibX edition and use the services offered by the LibApps within the subscribed edition.
1.2.1 Live LibApps

A set of LibApps were created using the LibApp Builder and were grouped into a package called the LibX Core Package. Some examples of LibApps from the Core Package are shown below.

**Alert users to ACM digital library subscription**: Inserts a static cue in the ACM portal page that plays a tutorial video when clicked, as shown in Figure 1.4.

**Display Google Book results in user’s library page**: When a user searches for a book in their primary library, results from Google Books are fetched and displayed in the library.
1.3 Core Contributions

The LibApp language model was designed to simplify the process of creating library applications by hiding complex programming details from the higher-level components of the language model, known as LibApps. Our objective was to build an interface that enables non-programmers to build LibApps using this language model. We implemented the LibApp Builder interface by incorporating the syntactic and semantic rules of this language model to minimize the development and debugging effort of users. Our contribution was to ensure that the interface meets all the requirements of the language model.

1.3.1 Web Editor

Requirement: Each XML field of an atom entry either describes the metadata of the entry, performs a small task, or specifies a requirement. For e.g., the include clause in the Amazon ISBN Scraper module as shown in code snippet 1.1 specifies that the module should be executed on amazon.com web page. Similarly, the content field of the module briefly describes the functionality of the module. Users need an environment to create, view,
and modify each field of an atom entry separately.

**Contribution:** The LibApp Builder is designed as a web application wherein different fields of an atom entry are associated with suitable user interface components such as text fields, check boxes, drop-down lists etc. A user may create, edit, or delete atom fields of an entry by adding, modifying, or removing corresponding UI components. The interface provides an executable environment through which users can build and test library applications. Web applications are easy to access and save users from the burden of downloading and installing client-side applications. We have designed the LibApp Builder as a web editor to allow librarians across the world to use the application.

### 1.3.2 Storage Mechanism for Distributed Language Components

**Requirement:** The language components of a LibX package i.e., modules, LibApps, and packages are connected together in a directed acyclic graph, allowing users to share atom entries across different feeds. A partition of a graph which can be linked to other partitions is known as a LibX feed. A feed is a collection of entries which is updated to a new revision every time it is published, similar to that of a software version. Entries of a feed can be linked to entries of other feed partitions through cross-feed references. We required a suitable storage mechanism that stores the LibX feeds and connects entries between different feeds.

**Contribution:** We used an XML database storage scheme to store LibX feeds on a LibX server. When a user creates a feed through the LibApp Builder interface, the newly created
feed is stored on the LibX server as a linear XML document. All feeds stored on the LibX server are known as internal feeds. All internal feeds are local to each other and are termed as peer feeds. However, feeds may also be available on external servers which are known as external feeds. Entries can be shared between both internal and external feeds i.e., a parent entry may contain a reference to a child entry that points to an entry of the same feed, a peer feed, or an external feed. Figure 1.5 indicates that the module 105 of Feed A is shared between LibApps 103 and 104 of the same feed whereas LibApp 103 is shared between packages 101 and 201 of peer feeds A and B respectively. Additionally, module 305 is shared between LibApp 203 of an internal feed B and LibApp 302 of an external feed X. A `<libx:entry>` element was introduced in the LibApp language model to indicate the
parent-child relationship between entries. When a child entry is added to a parent entry, a `<libx:entry>` is added to the parent entry. A `<libx:entry>` element includes a `src` attribute which refers to the unique identifier of the child entry. As part of our contribution, we set up different addressing notations to distinguish references to entries of the same feed, a peer feed, or an external feed which are illustrated in code snippet 3.4. An entry of the same feed is addressed by its id, e.g., `src="106"`, an entry of a peer feed is addressed by its relative id in the format `author-emailid/feedname/entryid`, e.g., `src="vsony7@vt.edu/sonytest/108"`, and an entry of an external feed is addressed by the absolute HTTP URLs at which the entry is located, e.g., `src="http://top.cs.vt.edu/libx2/brn@vt.edu/brntest/126"`.

1.3.3 Representing Components in a Hierarchical Structure

**Requirement:** The components of the LibApp language model are grouped into a hierarchical structure i.e., a package may contain only LibApps or other packages and a LibApp may contain only modules. A LibX feed containing packages, LibApps, and modules is stored in the database as a linear XML document which by itself does not reflect the hierarchy of its underlying components. Our requirement was to create a view in the LibApp Builder interface that represents the hierarchical structure of the entries within a feed to help users manage the feed easily.

**Contribution:** The LibApp Builder interface enables users to fetch LibX feeds from the
internal database and modify the underlying entries. We created a tree component in the LibApp Builder interface that represents each LibX feed as a hierarchical structure of entries as shown in Figure 1.6. Each row in the tree component corresponds to a single atom entry and displays the entry title, entry type i.e., a package, a LibApp, or a module, and the reference type. The type of reference indicates whether the entry belongs to the same feed, a peer feed, or an external feed through the notations (S), (O), (E) respectively. Users may create, edit, or delete entries within the tree component. Any change made to the metadata or content of a feed is immediately updated in the database and vice-versa. The interface allows users to operate two feeds at a given time which makes it easy for users to share entries between feeds through simple user actions such as drag and drop, cut and paste, copy and paste etc.

1.3.4 Assistive User Interface

Requirement: New users of the interface may not be familiar with the LibApp language model and may need assistance to understand the LibApp Builder interface. Our goal was to reduce the learning curve of users by guiding them through the process of creating LibApps.

Contribution: We incorporated sufficient guidelines in the LibApp Builder interface, through numerous information and help tooltips to simplify the process of creating, executing, testing and sharing LibApps. Figure 1.7 illustrates an example of a help tooltip from the interface which explains the functionality of an include clause.
1.3.5 Building a Community Platform

**Requirement:** The amount of user contribution directly reflects the success of an end-user programming tool. Our requirement was to create an environment wherein contributing and sharing programs is made easy so that users can quickly adapt to the role of co-programmers.

**Contribution:** The LibApp Builder interface allows users to publish LibX feeds and the underlying published entries are made available to all users. Hence, users can adapt or reuse a published entry by copying or importing the entry. Modules are generic pieces of code snippets and in most cases can be reused without having to change the underlying code. A user may create a copy of a LibApp that was published by another user and customize it by simply reusing the underlying modules and modifying the higher-level LibApp.
Sony Vijay

Chapter 1. Introduction

Figure 1.7: A Help tooltip: Describes the functionality of an include clause

fields. Therefore, users may create unlimited number of LibApps and contribute support for emerging library services in a way that can be shared.

1.3.6 Search Interface

**Requirement:** Users can share and reuse existing components easily only if they can quickly locate LibApps and modules of their interest that were published by others. Users need a suitable medium to fetch the published entries.

**Contribution:** We implemented a *search interface* within the LibApp Builder through which users can instantly locate and reuse published LibApps and modules. We used a faceted UI to allow users to filter out LibApps and modules as required from the search results. A user may reuse a search result in two ways, either by creating a copy of the corresponding entry or by directly including a reference to the corresponding entry under a valid parent entry.
1.3.7 Error Checking

**Requirement:** A user can create and test a program in minimal time if the number of errors is small. The LibApp language model is comprised of various syntactic and semantic rules. For instance, the language model allows *argument passing*, a standard programming practice. A parent entry may call a child entry by passing arguments and it requires that the argument value matches with the parameter type. Our requirement to ensure that the code created by users is devoid of errors and inconsistencies and help them build applications quickly.

**Contribution:** In standard development environments, a user would write a program and then build it after which the compiler would raise errors and exceptions. In contrast to the standard time-consuming and tedious debugging process, we implemented an error-checking algorithm that validates the user inputs on-the-fly while a user is writing a program and displays descriptive in-line error messages as soon as an input condition fails. The error messages display sufficient information to help users fix the errors immediately. For e.g., the user must specify a valid absolute URI at which a feed must be published while creating a new feed. If the user inputs an invalid URI, a corresponding error message is displayed in the interface as shown in Figure 1.8. Similarly, when passing arguments from a LibApp to a module, an error is raised if the argument value does not match the parameter type of the module.
1.3.8 Meta-Design Approach

**Requirement:** Meta-design characterizes objectives, techniques, and processes for creating new media and environments allowing “owners of problems” (that is, end users) to act as designers [22]. The LibApp language model adopts a meta-design paradigm to assist users in effortlessly adopting to the role of meta-designers. Our goal was to build an interface that adheres to the principles of the meta-design approach.

**Contribution:** The LibApp Builder allows users to perform function calls by passing arguments from a parent entry to the parameters of a child entry. The argument-passing allows users to create new LibApps from existing LibApps in few simple steps. For instance, a user would create a copy of an existing LibApp and modify the title and description as needed. Then, the user would alter the include field to indicate on which web page the new LibApp should be executed. Lastly, the user would reuse the underlying modules of the LibApp by passing a new set of arguments without modifying any JavaScript code. Hence, users can act as co-programmers and build LibApps by simply modifying high-level fields. However, this approach is under the assumption that the modules are generic enough to be shared.
between multiple LibApps. We assume two types of users, basic and advanced. An advanced user, typically a developer from the LibX team or a librarian with programming background would create and publish generic modules, making them available to basic users who would then copy and reuse the published modules. Another concern is that some users may find it difficult to understand the functionality of various entry fields despite the help tooltips. A module contains JavaScript code and metadata fields, which must be combined to create programs. Therefore, the language model can be considered as a hybrid language model because users need to be familiar with some minimum programming basics, for e.g., regular expression patterns.

1.3.9 Summary of Contributions

To summarize, the LibApp language model was designed as a meta-design model to enable users with non-programming background to build library applications, LibApps. A LibApp transforms a web page by integrating library services into it. The complex programming details, JavaScript code snippets, are hidden in lower-level components called modules. A user may adapt an existing LibApp by simply reusing the underlying modules and modifying the higher-level LibApp fields. We designed a web editor called the LibApp Builder interface that abides by the rules of the LibApp language model. The interface allows users to act as co-programmers by providing an executable environment for users to build and share LibApps. A LibX feed contains a group of atom entries i.e., packages, LibApps, and modules which form the basic building blocks of the LibApp language model. We designed a storage
mechanism to store the LibX feeds as XML files and designed a suitable user interface to tie various XML fields to different view components. The hierarchical structure of entries within a LibX feed is depicted through a tree component in the interface to help users manage a feed easily. The interface assists users in understanding the LibApp language model and the steps involved in building a LibApp by displaying several help tooltips. We included an error-checking algorithm within the interface to reduce the debugging effort of users. An error message is displayed whenever an input condition fails to help users fix their errors immediately. The LibApp Builder is incorporated with a search interface which allows users to locate and reuse or adapt published entries. Therefore, the interface acts as a community platform for users to create and share LibApps.
Listing 1.1: Amazon ISBN Scraper

```xml
<entry xmlns:libx="http://libx.org/xml/libx2" xmlns="http://www.w3.org/2005/Atom">
  <id>http://libx.org/libx2/libapps/171</id>
  <title>Amazon ISBN Scraper</title>
  <updated>2012-06-14T11:32:56-04:00</updated>
  <author>
    <name>LibX Team</name>
    <uri>http://libx.org/</uri>
    <email>libx.editions@gmail.com</email>
  </author>
  <content type="html">Writes all valid ISBNs on Amazon page to the tuple space.</content>
  <libx:module>
    <libx:include>/amazon\./</libx:include>
    <libx:require>jquery</libx:require>
    <libx:body>/*
    * Find the ISBN to which an item refers on Amazon
    */
    var isbnNodeArray = $="b:contains('ISBN')"
    var isbnVal = isbnNodeArray[0].nextSibling.nodeValue;
    if (isbnVal == null)
      return;
    var isbn = libx.utils.stdnumsupport.isISBN(isbnVal, false);
    if (isbn == null)
      return;
    libx.space.write({
      isbn : isbn
    });</libx:body>
  </libx:module>
</entry>
```

Listing 1.2: Addressing Child Entries

```xml
<libx:package>
  <libx:entry src="106" />
  <libx:entry src="vsny7@vt.edu/sonytest/108" />
  <libx:entry src="http://top.cs.vt.edu/libx2/brn@vt.edu/brntest/126" />
</libx:package>
```
Chapter 2

Background

2.1 LibX Browser Plugin

LibX is a client-side extension that enhances the functionality of web pages by integrating library services and resources into specific sites. The first version of the browser extension, LibX 1.5 \cite{17}, was released in 2005 to provide instant access to library services. LibX 1.5 relied on monolithic user scripts to embed context-sensitive cues into web pages. However, modifying a user script required altering a large piece of JavaScript code which made it difficult for librarians to customize, extend, and share these code snippets. A newer version, LibX 2.0 \cite{35} was thus developed and has been released in 2012 to increase the code modularity and re-usability. A language model has been developed to facilitate librarians to create extensible code snippets called library applications or LibApps. LibX 2.0 executes
the subscribed LibApps and integrates cues into web pages.

2.2 The Edition Builder

A custom version of LibX that meets the requirements of a specific library is known as a LibX edition. All features supported by a particular library are described by the respective edition.

A tool called the Edition Builder was designed to allow librarians or edition maintainers to create custom LibX editions. After creating an edition, a maintainer can configure its resources in multiple ways, for e.g., the Shortcuts Management feature is used to configure a set of shortcut links, and the Catalogs and Databases Management feature is used to add library catalogs and other library resources such as database lists, journal lists etc. Auto Discovery is an important feature of the tool which is used to automatically discover library resources such as OPACs and OpenUrlResolvers, allowing maintainers to select the auto-discovered settings. Additionally, the tool incorporates a revision management feature to allow maintainers to maintain and manage multiple versions or revisions of an edition, thereby providing a mechanism to provide new updates to the end-users in an organized manner.

While working on the LibApp Builder project, we have also contributed our time towards the maintenance and support of the Edition Builder. We built a searching and browsing mechanism into the Edition Builder through which edition maintainers can locate chosen LibX packages and them to LibX editions. When an edition maintainer adds a package
i.e., a collection of LibApps to an edition, the end users of the edition are subscribed to
the underlying LibApps. The end users can thereafter use the extended features offered by
the LibApps that belong to a LibX edition by loading the edition into their LibX browser
extension.

2.3 LibApp Language Model

LibX 2.0 plugin executes LibApps and enhances the functionality of web pages by perform-
ing context-sensitive, per-page transformations. As discussed in section 1.1 LibApps are
built using the modular LibApp language model. An unordered group of modules are pieced
together to build a meaningful LibApp and collections of LibApps are grouped into packages
and distributed to users through an Atom feed. A LibX feed is a collection of packages,
LibApps, and modules available at a given URL in AtomPub format. The client-side imple-
mentation of LibX 2.0 and some features of the LibApp language model have been described
in Brian Nicholson’s Master’s thesis [35]. The two main components of the language model
i.e., the syntax model and the semantic model are discussed in this section.

2.3.1 Syntax Model

The syntax model defines the structure and format of various properties and input fields of
a LibX feed and its atom entries. For e.g., an include clause directs a LibApp or a module
to run on a specific website, the value of which must be of a valid regular expression format.
The different rules of the syntax model are listed below.

**Entry Metadata**

Each package, LibApp, or a module is represented as an `<atom:entry>` within a LibX feed. Each entry contains a set of standard atom elements that describe the metadata of an entry. The fields used to describe the metadata are: id, title, updated, author, content. The author field contains three sub-elements namely, name, uri, and email, to describe the author information. The atom fields used to store the metadata of an entry are shown below:

1. **id** - uniquely identifies the entry.
2. **title** - contains a short title of the entry.
3. **updated** - indicates the most recent updated time of the entry in ISO format
4. **author** - wraps author details of the entry within three child elements: name, uri, and email.
5. **content** - briefly describes the task performed by the entry.

**LibX 2.0 Foreign Elements**

To separate the metadata from the inner code details, foreign mark up elements that use the LibX 2.0 namespace ([http://libx.org/xml/libx2](http://libx.org/xml/libx2)) are incorporated in an en-
try. Each <atom:entry> contains either a <libx:package>, <libx:libapp>, or <libx:module> foreign element, indicating whether the entry is a package, LibApp, or a module. Each package, LibApp or module element contains additional LibX 2.0 fields that specify a certain action. These fields are checked before executing a LibApp or a module. The different uses of LibX 2.0 elements are discussed below.

Addressing Child Entries

Packages can contain LibApps or other packages whereas LibApps are composed of modules. The children of packages and LibApps are listed using <libx:entry> elements. Each <libx:entry> element contains a src attribute that refers to a URL, which references to the <atom:id> of a child entry. A parent entry of a feed, i.e., a package or a LibApp may contain references to entries of the same feed, a peer feed, or an external feed, as explained in Section 1.3.2.

Specifying Websites

A LibApp or a module can be directed to run on specific websites through two clauses namely, include and exclude clauses, which contain regular expressions that specify URL patterns. When a user visits a website, the current URL is matched against includes and excludes patterns. If the URL matches an include pattern then the entry is executed and conversely if the URL matches an excludes expression then the entry is not executed on the web page. However, excludes takes precedence over includes which means that
an entry is not executed if a URL matches both includes and excludes fields. Examples of includes and excludes fields are shown in code snippet 2.1.

Overriding Entries

Librarians can add multiple packages to their customized LibX editions. By default, users are subscribed to the LibX Core Package consisting of a default set of LibApps. Librarians can add additional packages that may customize LibApps from the Core Package functionality to suit their specific libraries. In such situations, the original LibApps that have been altered must be suppressed for the customized versions to execute. An overrides URL is added in the altered LibApp or a module which specifies the URI of the entry to be replaced.

Internationalization

The LibApp language model allows librarians across the world to translate LibApps to their native languages. LibApps are designed to support Internationalization, a common design practice used for providing international language control. The language of a LibApp or a module can be defined through a `<libx:locale>` element. LibApps or modules can contain one or more locale elements.

Each locale element contains a language attribute, an optional default attribute, and a
Listing 2.2: English Locale

```xml
<libx:locale default="true" language="en_US">
  
  "shownotify" : { "message" : "Display popup notification" },
  "dontshowagain" : { "message" : "Don’t show this to me again" }

</libx:locale>
```

JSON object. The language attribute specifies the language to be applied for the localization. Before executing a LibApp or a module, the locale fields are parsed to determine the language. If an entry contains multiple locale elements, the default locale is used as a fall-back language. The JSON object describes the localization code in i18n format of Google Chrome [2]. The default language is specified as English through the locale field in Display HTML notifications module, as shown in code snippet 2.2.

**Module Body**

A module is the smallest unit of the language model that embeds a piece of JavaScript code snippet and metadata to perform a small task. Each module contains a body element to accommodate a JavaScript code snippet that describes the functionality of the module. The body of the Display Availability module, a child entry of the Show Item Availability on Amazon LibApp from the LibX Core package is shown in code snippet 2.3.
Module Requirements

The requires field of a module specifies the list of resources required for the module to execute. These resources may include JavaScript libraries, CSS stylesheets, or jQuery files. A resource can be described using a complete URI or a short-hand notation, for e.g., jquery.

The Display Availability module requires a jQuery resource as shown in code snippet 2.3.

Listing 2.3: Module: Display Availability

```xml
<?xml version = "1.0"?>
<entry xmlns="http://www.w3.org/2005/Atom" xmlns:libx="http://libx.org/xml/libx2">
  <id>http://libx.org:8080/editionbuilder/feeds/libx.editions@gmail.com/core/143</id>
  <title>Display Availability</title>
  <updated>2012-05-10T11:31:04-04:00</updated>
  <author>
    <name>LibX Team</name>
    <uri>http://libx.org/</uri>
    <email>libx.editions@gmail.com</email>
  </author>
  <content type="html">Displays an item’s availability.</content>
  <libx:module xmlns="http://libx.org/xml/libx2">
    <libx:include>/.*/</libx:include>
    <libx:guardedby>{ availability: libx.space.WILDCARD }</libx:guardedby>
    <libx:guardedby>{ position: libx.space.WILDCARD }</libx:guardedby>
    <libx:require>jquery</libx:require>
    <libx:body>for( var avail in tuple.availability){
      var msg = tuple.availability[avail] +
        (tuple.availability[avail] == 1 ? "copy" : "copies");
      msg += " : " + avail.trim().replace("\s", "");
      $(tuple.position).append(function(){
        return "&lt;/br&gt;&lt;span style='" +
            " font-size: 14px; font-weight: bold;" +
            "white-space: nowrap; border: 1px solid;" +
            "color: green; padding: 0 5px;'&gt;" +
            msg + ";
        });
    }
  </libx:body>
</libx:module>
</entry>
```
All required resources must be loaded for a module to execute successfully.

### 2.3.2 Semantic Model

The semantic model constitutes the semantic rules of the language model. The semantic rules define the hierarchical structure of the language components and the mechanism implemented to facilitate communication between modules.

#### Package Hierarchy

The language components of the LibX language model are designed to be modular and reusable. The complex programming details of a library application are abstracted into smaller units called modules. We defined a hierarchical tree structure to store the atom entries of a feed in an organized manner. The parent-child relationship of atom entries must conform to the defined tree structure. A parent package contains a group of related LibApps and other child packages. As explained in section 1.1 a parent LibApp contains a set of compatible child modules.

#### Module Communication

A LibApp contains a group of compatible modules that communicate with each other to execute a meaningful task. The execution order of modules must be specified to form a functional LibApp. For example, consider an existing LibApp, Show Item Availability on
Amazon: When a user searches for a book on Amazon, a notification message is displayed in the search result which indicates the availability of the book in the user’s primary catalog. This task is achieved by segmenting the LibApp into four modules as described below.

- **Amazon ISBN Scraper**: writes all valid ISBNs found on a Amazon web page to the tuple space.

- **Determine Availability by ISBN**: determines whether the item corresponding to the ISBN is available in the user’s primary library.

- **Find Position of Title in Amazon**: writes a tuple containing the position of the Amazon book title, the location at which the notification message should be displayed.

- **Display Availability**: displays a notification message next to the book title, indicating the availability of the book in the user’s primary catalog.

The functionality of the LibApp, *Show Item Availability on Amazon* is shown in code snippet 2.4. This LibApp works correctly only if the modules communicate properly. The module which scrapes the ISBN of the book on a Amazon web page must provide the ISBN to the sibling module which determines the availability of the corresponding book in the user’s primary catalog. The availability of the book should be provided to the module which displays a notification message in the web page. The module which displays the notification also needs to know the location at which the notification should be displayed. Hence, the module which finds a specific location of the web page, e.g., position of the book’s title,
must provide this information to the module which displays the notification message. After receiving the required information, the Display Availability module displays a notification message, indicating the availability of the book in the user’s primary catalog. A simple approach is to arrange the modules in a predefined order of execution and force each module to call its successor. However, such a model increases coupling among modules and hinders code re-usability. Alternatively, we designed the language model in such a way to facilitate communication between modules by placing data into a shared space, called tuple space.

Listing 2.4: LibApp: Show Item Availability on Amazon

```xml
<?xml version = "1.0"?>
<entry xmlns:libx="http://libx.org/xml/libx2" xmlns="http://www.w3.org/2005/Atom">
  <id>http://libx.org:8080/editionbuilder/feeds/libx.editions@gmail.com/core/141</id>
  <title>Show Item Availability on Amazon</title>
  <updated>2012-05-10T09:24:27-04:00</updated>
  <author>
    <name>LibX Team</name>
    <uri>http://libx.org/</uri>
    <email>libx.editions@gmail.com</email>
  </author>
  <content type="html">Shows the availability of an item (such as books) on Amazon website</content>
  <libx:libapp>
    <libx:include>/amazon\.com//</libx:include>
    <libx:entry src="142">
      <args xmlns="http://libx.org/xml/libx2"/>
    </libx:entry>
    <libx:entry src="143">
      <args xmlns="http://libx.org/xml/libx2"/>
    </libx:entry>
    <libx:entry src="171">
      <args xmlns="http://libx.org/xml/libx2"/>
    </libx:entry>
    <libx:entry src="172">
      <args xmlns="http://libx.org/xml/libx2"/>
    </libx:entry>
  </libx:libapp>
</entry>
```
Tuple Spaces

Tuple spaces are used for storing and sharing data that flows between modules. The use of tuple spaces was first seen in the Linda coordination language developed by David Gelernter and Nicholas Carriero [22]. In this work, a tuple space acts as a shared, virtual, distributed, memory which is used to store and retrieve tuples. Clients communicate with each other via tuples placed in a tuple space. In general, a tuple consists one or more keys and zero or more argument values. Clients using the tuple space specify tuple patterns consisting of a key and zero or more parameters. Clients can act as both producers and consumers. A producer client produces one or more tuples and places them into the tuple space. A consumer client specifies tuple patterns and takes in a tuple only if the specified pattern matches. Then, the consumer executes a piece of code that requires the taken tuple. After consuming the tuple, the consumer can act as a producer and place one or more tuples into the tuple space which can then be consumed and processed by other clients. This creates an input/output relationship in the tuple space, where the tuple taken by a client is the input and the tuple written by the client is the output.

A LibApp, when executed, creates a separate tuple space for all its modules. Each module may contain one or more produces and guardedby clauses. A guardedby clause specifies a tuple pattern, indicating a tuple required by the module. A produces clause places a tuple in the tuple space. If a module contains produces clauses then each produces clause places a tuple in the tuple space when the module is executed. If a module contains guardedby clauses then its code can be executed only if the module receives tuples that
match the patterns specified by these clauses. When a module that contains `guardedby` clauses is executed, its code is wrapped in tuple space take listeners and these listeners wait for tuples that match the specified patterns. The inner code is executed after each wrapping take listener receives a required tuple placed by other modules. After executing the code, the module may place one or more tuples in the tuple space via `produces` fields. Other modules that are guarded by these newly placed tuples are then executed. Therefore, a LibApp breaks if one if its modules remains waiting for a tuple that is never produced, e.g., if a module needs an ISBN to check the availability of a book then another module of the LibApp must produce the ISBN.

The execution order of modules depends on the available tuples in the tuple space rather than clients calling each other directly, resulting in low coupling among modules. This makes it easy to write modular code because each module is unaware of other modules interacting with it. Moreover, modules are designed as generic code snippets facilitating code re-usability. Figure 2.1 illustrates the module interaction in the LibApp, *Show Item Availability on Amazon.*

`guardedby` and `produces` clauses are the fundamental fields used during module communication. The two `guardedby` fields of the *Display Availability* module are shown in code snippet 2.3. This module is guarded by two clauses, availability of the book and position of title in the book result page. A `guardedby` element is a JavaScript Object Literal that specifies a tuple pattern. The parent LibApp, *Show Item Availability on Amazon,* will be executed only if other sibling modules of the module produce these fields. The *Determine*
Figure 2.1: Communication of modules via tuple space. This figure includes the amazon.com book search result page for the book Linear Algebra (Dover Books on Mathematics) by Georgi E. Shilov. Used under fair use, 2013.

**Availability by ISBN** module produces an availability tuple and the **Find Position of Title in Amazon** module produces a position tuple. The produced tuples are then consumed by the modules which are guarded by the corresponding tuple patterns. A `produces` field contains a comma separated list of strings that correspond to the keys in `guardedby` clauses. A module writes a tuple into the tuple space when its body is executed.
2.4 Overview of Technologies

We have used a number of technologies to implement the LibApp Builder. This section provides an overview of the core technologies used during the implementation.

2.4.1 ZK AJAX Framework

ZK is an AJAX framework that enables users to build rich user interfaces for web and mobile applications [10]. ZK is a server-centric, event-driven, and component-based framework that eliminates the need to learn JavaScript or AJAX for building Rich Internet Applications. The three main components of the framework include an Ajax-based event-driven engine, a rich set of XML User Interface Language (XUL) and XHTML components, and a markup language called ZK User Interface Markup Language (ZUML) [21]. An event-driven engine provides a programming model similar to that of desktop programming model. Readily available XUL and XHTML components make it easy to build the user interface of a ZK application, these components are manipulated based on events triggered by user interaction. The style, behavior, or function of the components can be configured to suit the needs of a user. The ZUML mark-up language simplifies the task of designing user interfaces and provides a similar experience as that of designing HTML pages.
ZK Architecture

The architecture of ZK is comprised of three important elements namely, the ZK Loader, ZK AU (asynchronous update) Engine, and ZK Client Engine. The ZK Loader and ZK AU Engine run at the server side and are composed of Java servlets. The ZK Client Engine runs at the client side and is composed of JavaScript code snippets. The architecture of ZK is illustrated in this section. Firstly, the ZK Loader receives and interprets an incoming HTTP URL request and generates an HTML page in response and creates ZK components at the server side. The ZK Loader then sends the generated HTML page containing standard HTML, CSS, and JavaScript code to the client and the ZK Client Engine. The Client Engine monitors queued JavaScript events on the client side that are triggered by user activity such as a button click or a mouse movement. When such JavaScript events are triggered, the ZK Client Engine sends the triggered events or Ajax requests to ZK AU Engine. After receiving the Ajax requests, the ZK AU engine updates the ZK components on the server side if necessary and sends Ajax responses to the Client Engine. On receiving a ZK AJAX response, the ZK Client Engine updates the corresponding content in the Document Object Model (DOM) tree in the browser.

2.4.2 BaseX XML Database

BaseX provides a scalable, high-performance, light-weight XML Database engine and offers an efficient XPath/XQuery 3.0 Processor, including full support for W3C Update and
Listing 2.5: A FLWOR Expression

```xml
declare namespace atom='http://www.w3.org/2005/Atom';
declare variable $feed_document as xs:string external;
declare variable $entry as xs:string external;

for $entry in doc($feed_document)/atom:feed/atom:entry
  let $author := $entry/atom:author/atom:email
  where $author := $userid
  return $entry/atom:title
```

Full Text Recommendations [7]. The BaseX engine applies full-text indexing and provides an optimized full-text retrieval feature. BaseX facilitates approximate searches in full texts through its fuzzy querying feature. Additionally, BaseX offers an interactive Graphical User Interface that allows users to explore and analyze large XML and JSON documents and collections through various hierarchical visualizations. Users can execute and evaluate queries in real-time using the visual front-end.

BaseX adopts a Client/Server architecture model that features ACID safe transactions, user management, and logging by handling concurrent read and write operations of multiple users. XML data is stored on a BaseX server and a client communicates with the server to execute database commands, perform queries, or listen to events. BaseX is platform-independent and a BaseX client can be written in multiple programming languages using the several lightweight bindings provided by BaseX. The BaseX query processor and some of its extensions are described below.
XQuery

XQuery [8] is a widely implemented query and functional language used to combine documents, Web pages, and databases. A set of XQuery specifications were developed by W3C XML Query Working Group in collaboration with W3C XML Schema Working Group and the W3C XSL Working Group. XQuery is used to extract and manipulate data from not only XML documents but also from any document that can be represented as an XML document such as relational databases or office documents [9]. A specific part of an XML document is addressed using an XPath expression syntax in an XQuery. Additionally, **FOR**, **LET**, **WHERE**, **ORDER BY**, and **RETURN** expressions, FLWOR expressions, are used for performing joins which are similar to SQL expressions. The code snippet 2.5 is an example of a FLWOR expression which returns a list of titles of those entries created by a given user.

BaseX provides an implementation of the W3C XPath and XQuery languages, which includes a complete implementation of the XQuery Update Facility (XQUF). The XQuery implementation of BaseX includes a simple map operator and expressions such as **group by**, **try/catch**, and **switch**. One of the unique features offered by BaseX XQuery are function items, also called lambda functions. Lambda functions enable abstracting over functions and thus provide means for additional modularization of code. Additional features offered by BaseX XQuery include expanded QNames, namespace constructors, string concatenations, external variables, serialization parameters, context items, and annotations.

One common API standard used by Java developers for processing and parsing XML docu-
ments is XQJ, the XQuery API for Java. XQJ, also known as JSR 225 (Java Specification Requirement 225) [45], was designed through the Java Community Process (JCP) [3]. XQJ can be used to invoke XQuery expressions against any XML or relational database and process the query results.

Full-Text Processor

BaseX query processor supports the W3C XQuery Full Text 1.0 Recommendation. The query compiler applies a full text index structure appropriately when possible in an attempt to optimize and speed up queries. BaseX offers three evaluation strategies for XQFT queries namely, the standard sequential database scan, a full-text index based evaluation, and a hybrid strategy that combines both strategies. The BaseX query processor optimizes queries by automatically selecting the suitable evaluation strategy depending on the input data and the existence of a full text index.

The full-text index provides the means to support various match options specified in the XQuery Full Text Recommendation. Indexing options include language, stemming, case sensitive, diacritics, and stop word. BaseX provides multilingual support. Depending on the language chosen by the user, the tokenization and stemming procedures of an input text are determined. The default languages supported by BaseX are English and German, and support for other languages can be obtained by adding additional libraries to the classpath. BaseX implemented an internal scoring model wherein the score of a full-text result is com-
Computed using the count of the terms found and the frequency of these terms in a single text node. A higher rank is attributed to those terms that are found in short texts. Another feature implemented in BaseX full-text is the use of thesauri. BaseX defined an XSD Schema based XML format for thesauri so that users can provide suitable thesauri.

The Fuzzy Querying feature is also supported by BaseX in addition to the official recommendation that allows approximate searches in full-texts. Fuzzy search is implemented based on the Levenshtein distance [4] and is applied by default to the standard full-text index. For example, the query shown in code snippet 2.6 is used to retrieve a list of titles of those entries which approximately match a given keyword. Two other important features of BaseX are mixed content search and functions. BaseX allows searching across mixed content, which are elements that contain a combination of text and markup. Moreover, BaseX extended the official language recommendation by adding a set of full-text functions. BaseX functions provide additional benefits, for e.g., a user can explicitly request the score value of an item or mark the matching text of a full-text request.

Listing 2.6: A Fuzzy Query

```
declare namespace atom='http://www.w3.org/2005/Atom';
declare variable $feed_document as xs:string external;
declare variable $entry as xs:string external;

for $entry in doc($feed_document)/atom:feed/atom:entry
  let $author := $entry/atom:author/atom:email
  where $author := $userid
  return $entry/atom:title
```
Java Bindings

The Java Bindings feature of BaseX enables developers to directly access and execute Java code from XQuery. Java code can be invoked from XQuery in two ways. One way to identify Java classes is to use namespaces, i.e., by specifying the fully qualified class name in the namespace URI. All static functions of an identified class can then be called using the namespace. The second way to integrate Java code is to import classes as modules. A module import creates a new instance of the addressed class which can then be used to access the Java class in the query body. Alternatively, a context-awareness mechanism is used to notify a Java class of the current context variable. An instantiated class can obtain access to the context variable by inheriting the abstract QueryModule class of BaseX. All static and dynamic properties of the current query can be accessed via the context variable, i.e., an instance of the QueryContext class. Annotations can be used in addition to modify the default properties of functions.
Chapter 3

User Interface Design

The LibApp language model, described in section 2.3 was designed to hide the complex programming details, JavaScript code snippets, from LibApps, through a lower-level abstraction called modules. However, users need a suitable environment to build, share, and customize LibApps. Our goal was to provide a usable interface that enables users with minimal technical background to build LibApps with reduced effort. We designed the LibApp Builder interface to address the different user requirements involved in building a LibApp. The user interface allows users to create LibX feeds, modify and publish entries of a feed. The revision management feature allows users to retrieve, modify, and re-publish a previous version of a published feed. Sufficient help tooltips are included in the interface to guide users through the process of building LibApps. An error-checking algorithm is integrated into the interface, which displays error messages whenever an input condition fails to help users fix their errors immediately. We designed a search mechanism which allows users to
locate and reuse published entries, thereby providing a community platform for users to share LibApps. The various design concepts used to implement the LibApp Builder are described in this chapter.

3.1 User Interface Design

We adopted a minimalistic design in the LibApp Builder interface, allowing users to accomplish their tasks in a simple and efficient manner. This section describes the different features of the interface design.

3.1.1 User Authentication

The interface validates user information to authenticate access to the LibApp Builder. A user with a valid email address must register with the LibApp Builder to create LibApps, as shown in Figure 3.1. The users must be registered so that we can store the author information of each LibX feed. A LibApp is capable of modifying a web page and is prone to security threats. We assign a permission called trusted to those users whom we trust to handle security attacks. Only these feeds published by trusted users are made available to other users. A registered user can sign in to the LibApp Builder using the registered email address and password, as shown in Figure 3.2. The user password is encrypted using a hash function and stored in the database in a hash format to ensure security. Since we do not store user passwords in the original text form, we implemented a password reset feature instead of a
password recovery feature. If a user chooses to reset their password, a new random password is generated and sent to their registered email address. Additionally, the account settings feature allows a user to modify their personal information such as user name, organization, password etc., after signing in to their account as shown in Figure 3.3.

3.1.2 Feed Manipulation

The primary goal of a user is to create and publish LibX feeds through the LibApp Builder interface. A LibX feed may contain references to entries of the same feed, a peer feed, or an external feed, as described in section 2.3.1. Hence, users need a facility to work across feeds.
and the ability to copy or reference entries from one feed to another easily. We equipped the user interface with two identical feed panels, enabling users to manipulate two different feeds simultaneously. Each feed panel is comprised of a feed menu and a feed tree. A feed menu is used to modify the properties of a feed itself and a feed tree component is used to represent the entries of a feed in a hierarchical structure.
Feed Menu

We designed the feed menu to enable users to manage their LibX feeds. The different options of the feed menu shown in Figure 3.4 are discussed below.

1. **Creating a feed:** The basic properties of a feed must be set during the creation of a feed which are: name, publishat, and description. Let us consider that a user creates a feed called ‘main’ to create LibApps. The creation of the ‘main’ feed is shown in Figure 3.5. When a user creates a new feed, the feed is activated in the corresponding feed area immediately.

2. **Opening a feed:** A user may open one of the feeds they own in the feed area as shown in Figure 3.6. After opening a feed, the user can modify the properties of the feed or the underlying entries. However, an administrator has the privilege to open and manage feeds created by other users.

3. **Editing feed properties:** A user can modify the publish URL and description of the currently open feed at any point. Figure 3.7 shows the modified description of the ‘main feed’. However, the feed name cannot be altered once a feed has been created because it is used for creating a unique file name in the database.

4. **Publishing a feed:** After creating a feed, the user can activate and test the functionality of its LibApps using the LibApp Builder interface. After successful execution of the LibApps, the user may use the publish option to publish the feed as shown in
5. Managing Revisions of a feed: A user may publish a feed multiple times to make the latest updates available to the end-users of the feed. Each time the current editable version of a feed is re-published, the previously published version is saved as an archive in the database. A user may use the revision option in the feed menu to retrieve all archives of the active feed in the feed area as shown in Figure 3.9. The user may then open a chosen archive in the active feed area but only in a read-only mode. However, the user may revert an archive into the current editable version as shown in Figure 3.10.

6. Deleting a feed: The delete operation of a feed as shown in Figure 3.11 deletes the current editable version of the feed and all its existing revisions i.e., the published and archived versions if any.
Feed Tree

LibX packages are connected to each other in a directed acyclic graph as described in section 1.3.2. To allow users to manipulate each LibX feed separately, we expand this graph into a hierarchical structure and represent each entry of a feed as a tree item in a ZK tree component. The visual tree representation of a feed illustrates the parent-child relationship of the underlying entries. The feed tree facilitates users to arrange packages, LibApps, and modules, similar to arranging files and folders on a computer. At any given instance, users can manage two feed trees in two feed panels of the interface. Users can exchange entries across different feeds by dragging and dropping entries from one feed tree to another. When a user drops an entry into another feed, a copy of the dragged entry is created and added
as a child to the target entry. Additionally, users may drag and drop entries into the same feed which is analogous to a cut and paste action i.e., the reference of the dragged entry is removed from the original parent and included under the new parent.

Apart from exchanging entries between feeds, users need a facility to perform entry-level tasks such as adding a child entry, deleting an entry etc. We added an entry-level context menu for each item in the feed tree through which users can manipulate each entry individually. Figure 3.12 shows a snapshot of different entry-level menu options.

1. **Creating a child entry:** Adhering to the hierarchy structure of our language components, we allow users to add only valid child entries under a parent entry. LibApps and packages can be created under a parent package whereas only modules can be added under a parent LibApp. For example, if a user creates a new package called LibX Main Package in the ‘main’ feed, then the user is allowed to add only a package or a LibApp
2. Including an external entry: A user may include a reference of an entry that belongs to an external feed under a suitable parent entry of an active feed using the “Insert External” option by providing the fully published URL of the external entry.

3. Deleting an entry: A user may delete an entry in two ways - (i) Removing from parent: When a child entry is removed from a parent, only its reference is deleted from the parent but the entry itself is not deleted permanently. If the entry is referred as a child in other parent entries, those references are retained. However, if the entry is not referred in any other parent, it becomes a root entry. A root entry is a top level entry in a feed hierarchy that has no parent entry. (ii) Deleting an entry: An entry that belongs to the same feed can be deleted using the ‘delete’ option which will delete the entry permanently from the feed. All references to the deleted entry are removed from the feed. References to entries that belong to other LibX feeds or external feeds
can only be removed from their parent entries but cannot be deleted permanently.

4. **Copying an entry:** A clipboard is maintained in the LibApp Builder internally, allowing users to store a copied entry. The clipboard is common to both the feed panels and this allows users to paste or include the copied entry under a suitable parent within the same feed or a different feed.

5. **Cloning and Pasting an entry:** After copying an entry to the clipboard, users can clone and paste it under a suitable parent. A new duplicate copy of the clipboard entry is created under the selected parent.

6. **Including an entry:** The reference of the clipboard entry can be included under a suitable parent entry of the active feed using the include option. A duplicate copy of the clipboard entry is not created when an entry is included.
3.1.3 Entry manipulation

A LibApp is created by grouping a set of compatible modules together and passing suitable arguments to the parameters of the child modules. A collection of LibApps are grouped into packages and distributed to other users via LibX feeds. Each entry of a LibX feed contains a list of XML elements such as include, exclude, locale which are described in section 2.3. Users need the ability to manage these entry fields to create meaningful LibApps. However, displaying the complete set of entry fields in a single place defeats the purpose of providing a usable interface. Hence, we used a tabbox UI component i.e., a container for displaying a set of tabbed pages of elements. The area assigned for the tabbox component in the interface is termed as entry edit area wherein different types of entry fields are distinguished by grouping each field type into a separate tab of the tabbox. The entry edit area is kept common to both feed panels in the interface to provide a simple and clean user experience. Consequently, only one feed panel between the available two feed panels can be active at any given time. However, users can deactivate the currently active feed panel and activate the other panel by simply performing any user action such as selecting a feed menu option or selecting a feed tree item in the other feed panel.

When a user selects an item in the feed tree, the fields of the associated entry are partitioned and displayed in different tabs of a tabbox. Each tab allows users to control a specific functionality of an entry such as specifying websites for an entry, passing arguments to a child entry etc. Each tab is integrated with sufficient help information, guiding users to manage the corresponding entry fields. In this chapter, we describe the process of adding
Figure 3.13: Modifying Entry Description

LibApps to the newly created LibX Main Package of the ‘main’ feed. The user may select the new package in the feed tree and edit its metadata in the Description tab, as shown in Figure 3.13. In each tab, the user may add, delete, or modify an entry field and each such action is validated by an error-checking algorithm.

3.1.4 Error Checking

We designed an error-checking algorithm to reduce the debugging effort required by users and enrich the LibApp development process. The algorithm is incorporated with several type-checking and semantic-checking rules to validate user actions. If the user performs an invalid action, such as adding an incorrect regular expression to an include field then a useful error message is displayed in the interface immediately. Each error message contains adequate information to help users fix their errors quickly.
Type Checking

We implemented several type checking rules in our language model to help users build meaningful LibApps. The different rules of the algorithm are described in this section.

1. Regular Expressions: The include and exclude clauses are used to specify the websites on which an entry should run. These fields accept a regular expression that match a URL pattern for e.g., the expression used to specify the URL of the ACM portal page is shown in code snippet 3.1. When a user creates or edits these fields, the algorithm matches the input against a regular expression validator and accepts the input only if it matches the format of a regular expression.

2. Tuple Patterns: A guardedby clause is used to specify a tuple pattern required by a module through a valid JavaScript Object Literal as shown in code snippet 3.2. The validity of each guardedby clause is checked by the algorithm.

3. Tuple Keys: A produces clause of a module is used to produce a tuple required by a sibling module and accepts a comma separated list of Strings. For e.g., code snippet 3.3 shows the produces clause used for producing the above guardedby clause. The
algorithm checks whether each `produces` clause conforms to the valid String format.

4. Argument Type: The algorithm ensures that each argument value matches the corresponding parameter type when a parent entry passes an argument to a child. Figure 3.14 shows a snapshot of an error message that occurs when a user incorrectly passes a String value from a LibApp to an integer parameter of its module.

5. Feed Names: A feed name must contain more than 2 characters and can include only lower case characters and digits. Each feed is stored in the database with a unique file name that is created using the feed name. Thus, spaces and special characters are not accepted in the feed name to avoid invalid file names. Figure 3.15 indicates an invalid feed name.
Semantic Checking

In addition to type checking, we implemented semantic checking rules to help users group entries consistently. The first rule is to adhere to the hierarchical structure of entries. A package may contain other packages or LibApps whereas a LibApp must contain only modules. This rule restricts users from grouping entries in any other way i.e., a user cannot add a module in a package, a package in a LibApp, or any entry in a module. When a user creates or adds a child entry either through the entry-level menu or a search result, the validity of the parent-child relationship is checked. Additionally, we incorporated a rule to check whether the modules of a LibApp are grouped together meaningfully. Modules of a LibApp communicate through a tuple space and when a module specifies a tuple pattern through a guardedby clause then its sibling module must produce a matching tuple for the LibApp to be executed correctly. For e.g., if a module needs an ISBN to check the availability of a book then another module of the parent LibApp must produce the ISBN. Figure 3.16 shows an error message that prompts the user to produce an ISBN tuple for the Link Amazon by
Figure 3.16: An Error Message: missing isbn tuple


3.1.5 Search Interface

The LibApp Builder is integrated with a Quick Search interface through which users can locate published LibApps and modules instantly. We implemented fuzzy full-text search logic using BaseX XQuery API to enhance the search experience of our users. Users can locate matching entries by providing approximate or incomplete search keywords. Moreover, we implemented Faceted Search UI to simplify the search process. When a user searches for a set of keywords, all matching LibApps and modules are displayed in the search results by default and thereafter, the user can filter the search results as needed. Figure 3.17 displays all search results matching the keyword video, and Figure 3.18 displays the filtered out LibApps. Each search result includes adequate information to help users choose an entry such as entry metadata, a hyperlink to the raw XML code, and a list of matching extracts. A user can adapt a search result by creating a copy of the entry or reuse a result by including it directly in a suitable parent entry.
Copying a search result

Copying a search result allows users to adapt the entries in a chosen way. When a user copies a search result into a parent entry, a duplicate copy of the search result entry is created and

Figure 3.17: Default Search Results
added under the parent, allowing the user to adapt the duplicated entry in a chosen way.

**Adapting an ACM-specific LibApp into a JSTOR-specific LibApp**

The various steps involved in adapting the *Alert users to ACM digital library subscription* LibApp, referred to as ACM LibApp are discussed in this section. From the description of the ACM LibApp, the user understands that the entry is used to embed a tutorial video in the ACM portal page. Now, the user may wish to create a similar application except the user may want to include a JSTOR tutorial in the JSTOR homepage instead of the ACM tutorial. This can be accomplished by simply adapting the high-level LibApp fields without modifying the inner JavaScript code of the modules. The original ACM LibApp is decomposed into four generic modules as shown below:

- **Display a help icon that plays a video when clicked**: Given a YouTube ID and jQuery selector, writes a tuple specifying an icon to display the notification.
• **Place a clickable image into a page:** Creates an icon on the page, performing the given action when clicked.

• **Create a notification to play a YouTube video, based on Video ID:** Given a YouTube ID, write a notification that shows the corresponding video.

• **Display HTML notifications via an embedded panel using jGrowl:** Given a notification tuple, displays the jGrowl notification on the page.

A user can reuse and share a set of modules in multiple LibApps because they are generic and reusable blocks of code. When the user creates a copy of the ACM LibApp, the underlying modules are included in the copied LibApp directly. The user can adapt the duplicate LibApp by simply modifying the ACM-specific fields in the LibApp. This process can be summarized in four simple steps:

1. Copying the LibApp: Firstly, the user copies the ACM LibApp into the LibX Main Package of the ‘main’ feed. Typically, end-users of LibX editions may be subscribed to multiple packages. For example, the user may add both the LibX Main Package and the LibX Core Package to their customized LibX edition. The user may want to suppress the original ACM LibApp with the newly copied LibApp that the user is about to customize. The original LibApp can be overridden by checking the overrides component while copying it from the search results. The resulting visual tree representation of the feed is shown in Figure [3.19](#)
2. Modifying entry description: The user then changes the metadata of the entry to describe the functionality of the adapted LibApp. Figure 3.20 shows the copied title and description fields of the original LibApp and the modified fields are shown in Figure 3.21.

3. Specifying the website: The original LibApp runs on the ACM portal page but the user wants to run the new LibApp on the JSTOR homepage. The user accomplishes this by changing the include clause of the entry, as shown in Figures 3.22 and 3.23.

4. Modifying Arguments: The first module described above contains two String parameters namely, youtubeid and selector. The youtubeid accepts a String value corresponding to the Id of the YouTube video to be played and the selector parameter accepts a jQuery selector describing the element next to which the cue must be placed. The user visits YouTube and copies the Id of the chosen video and passes this
value to the `youtubeid` parameter. Then, the user visits the JSTOR web page and determines the id of the search box component to place a cue next to it and passes this value to the `selector` parameter. Figures 3.24 and 3.25 show the original and modified arguments of the LibApp. The user can now test the customized JSTOR LibApp and distribute it to the end users.

Figure 3.26 summarizes the above process of adapting the ACM-specific LibApp into a JSTOR specific LibApp.

**Including a search result**

A user can directly reuse a search result entry by including it under a suitable parent entry. An included entry points to the original search result entry i.e., a duplicate copy of the entry is not created. A user may include entries that belong to the currently active feed, a peer feed, or an external feed. A user can reuse the included entry without having to make any
A user may create a LibApp by grouping a set of compatible modules together. The functionality of certain modules may be specific to a particular web page or a particular feature of a LibApp. The JavaScript code of such a module needs to be modified to be able to execute it on a different web page. However, most modules are generic snippets of code that perform the same task on any web page and can be reused without any code changes. The user may reuse a search result entry by choosing the include option in the search result.
significance of including entries is described in the following use-case scenario.

**Adapting an Amazon-specific LibApp into a Barnes and Noble-specific LibApp**

Let us now consider adapting the *Show Item Availability on Amazon* LibApp discussed in Section 2.3.2 When a user searches for a book in Amazon, the LibApp searches for the book in the user’s primary catalog and displays a notification in the Amazon web page which indicates the availability of the book in the user’s library. The user may want to create a similar LibApp that runs on Barnes & Noble page. The process of creating the Barnes & Noble LibApp is summarized below.

1. Creating the LibApp: Firstly, the user creates a LibApp called *Show Book Availability on Barnes & Noble* under the ‘main’ package and adds suitable description to describe the functionality of the LibApp.

2. Copying and adapting modules: The user then copies the modules that are specific to Amazon web page and adapts them to suit the BN.com page as shown below:

writes all valid ISBNs on Amazon page to the tuple space. The user copies this module into the newly created Barnes & Noble LibApp and performs the following changes to it: (i) modifies the description of the module suitably. (ii) modifies the JavaScript code in the module body such that all valid ISBN patterns of Barnes & Noble page are scraped and written into the tuple space. (iii) modifies the include clause of the module such that the module runs on the Barnes & Noble page instead of the Amazon page.

- Creating Find Position of Title in Barnes & Noble Module: The user copies the module that finds where the book title is displayed in Amazon page and modifies it as required. The description and include field of the module are modified as explained above. The user modifies the JavaScript in the module body such that the tuple containing the position of the book title in BN.com is written into the
3. Including modules: The Amazon-specific LibApp contains two other modules which are generic in nature. The *Determine Availability by ISBN module* determines an item’s availability by ISBN in the user’s library and the *Display Availability* module indicates the availability of the book in the user’s library by displaying a notification message next to the book title. These two modules can be run on any web page since they are not specific to a single web page. Thus, the user directly includes these two modules in the new LibApp from the search results which completes the last step of creating the BN.com specific LibApp. Reusing existing entries will significantly reduce the time and effort required to build a LibApp. Figure 3.27 summarizes the process of adapting the Amazon-specific LibApp into a BN.com-specific LibApp.
3.1.6 Addressing entries

As described in Section 2.3.1, a `<libx:entry>` containing a `src` attribute is created when a child entry is added to a parent. Each `src` attribute refers to the atom id of the child entry and we designed a suitable technique to differentiate the different types of child entries, i.e., references to entries of the same feed, a peer feed, or an external feed which are illustrated in code snippet 3.4.

1. Same Feed Reference: The parent entry and child entry belong to the same feed. In this
Figure 3.27: Adapting Amazon-Specific LibApp into a BN.com-specific LibApp

case, the value of src attribute contains the id of the child entry, e.g., src="106".

2. Peer Feed Reference: The parent entry and child entry belong to different feeds. However, both of these feeds are maintained on the same server. The value of src attribute contains reference to the child entry in the format author-emailid/feedname/entryid, e.g., src="vsony7@vt.edu/sonytest/108".

3. External Feed Reference: The child entry belongs to an external feed maintained elsewhere. The value of src attribute contains an absolute HTTP URL at which
Listing 3.4: Addressing Child Entries

```xml
<libx:package>
    <libx:entry src="106" />
    <libx:entry src="vsony7@vt.edu/sonytest/108" />
    <libx:entry src="http://top.cs.vt.edu/libx2/brn@vt.edu/brntest/126" />
</libx:package>
```

the child entry is located, e.g., src="http://top.cs.vt.edu/libx2/brn@vt.edu/brntest/126".

A package does not indicate whether its children are packages or LibApps because the src attribute does not specify the entry type. One must examine the child entry of a package to determine its type. However, the child entry of a LibApp is always a module and modules themselves do not contain any children.

### 3.2 Life cycle of a feed

After creating a LibX feed comprising of meaningful LibApps, users need the ability to activate and test these LibApps. The LibApp Builder interface allows users to test the entries of a feed and upon successful execution of the entries, the user can distribute them to other users by publishing the feed. The complete cycle starting from creating a feed to publishing a feed is termed as the Life cycle of a feed. The different stages of the life cycle are described below and are illustrated in Figure 3.28.

1. Create: The user creates a new feed called ‘main’ by setting its properties as described in section 3.1.2. The newly created feed is saved in the database and referred to as the
development version. The user can fetch the development version of a feed from the
database through the LibApp Builder interface and modify the feed as required.

2. Edit: Once the feed is created, the user may add new entries, delete or modify exist-
ing entries to create useful LibApps. Section 3.1.5 describes the creation of two new
LibApps under the LibX Main Package.

3. Test and Publish: After creating the desired LibApps in a feed, the user activates and
tests the functionality of the LibApps. The user activates the ‘main’ feed and checks
the execution of the JSTOR LibApp by visiting the JSTOR home page. Activating
and testing the JSTOR LibApp is shown in Figures 3.29 and 3.30. Upon successful
execution of the LibApps of a feed, the user publishes the feed at a specific URL and
distributes the entries to other users. Figure 3.8 shows the URL at which the ‘main’
feed has been published. When a feed is published, an archived version i.e., a duplicate
copy of the current development version is created and saved with a time stamp in the
database. Also, a published version i.e., a transformed copy of the current development
version is saved in the database. A duplicate copy of the published version is also saved
in the file system which allows the LibX 2.0 client to fetch the published feeds directly
from the file system. In the published copy, all the entry ids of the feed are transformed
from entryid to puburl/entryid format where puburl is the Publish URL of
the feed. The feed id is transformed to the puburl itself and all the references to cross-
feed child entries are converted to the absolute Publish URLs of the corresponding
entries.
4. Re-edit: The user may wish to push more updates to a published feed and hence the development version is always maintained in the database. The user can edit the properties of a feed, and add, delete, or adapt the entries of a feed by modifying the development version. The archived and published versions of a feed are read-only. However, a user may convert an archived version into the development version and modify the same.

5. Re-publish: After re-editing a feed the user may test the newly incorporated changes and re-publish the latest development version. Every time a feed is re-published a new archived version of the feed is saved in the database as explained in step 3. Similarly, a new published version is created which replaces the previously published copy in the database and the file system. Hence, a single copy of the published version exists in the database and the file system at any point if the feed has been published at least once.

### 3.2.1 Revision Management

As explained above, the user is allowed to edit the development version of a feed whereas all other versions of a feed are read-only. However, the user may wish to retrieve a previously archived version, make changes to it, and publish it. Thus, we implemented a revision management feature in the interface that allows users to retrieve all the archived versions of a feed as described briefly in section 3.1.2. The user may select a chosen archived version
of the active feed and open it in the active feed area to inspect the functionality of its entries. The feed area allows users to revert an archived version into a development version, as shown in Figure 3.10a. Before reverting the archive into the development version, the current development version is archived in the database with a time stamp. Thus, if the user wants to undo the revert action, then they can fetch the previous development version that has been recently archived i.e., the most recent archived version of the feed and revert it back to the development version.

Figure 3.28: Lifecycle of a feed
3.3 Integration with the Edition Builder

As explained in section 1.2, edition maintainers create custom editions of LibX using the Edition Builder interface. However, they need a facility to locate and add published packages to their editions to subscribe their end users to the underlying LibApps. Thus, we enhanced the functionality of the Edition Builder by incorporating an interface which allows users to search for or browse across packages, and add chosen packages to their editions. Any registered user may create and publish LibApps in the LibApp Builder but not all applications created by users may be safe. We introduced a trust factor to users under the assumption that the LibApps created by trusted users are safe. An administrator assigns trust to users and only packages published by trusted users are displayed in the Edition Builder.

3.3.1 Package Browsing Interface

Users can browse packages using the package browsing interface as shown in Figure 3.29. The packages are sorted by author email address first and then by feed name. Sufficient information is displayed about each package to help users choose specific packages for their editions. This information contains the metadata, a hyperlink to the raw XML code, and
another hyperlink to the tree display of the underlying entries of a package.

3.3.2 Package Search Interface

Using the package search interface, edition maintainers can locate published packages using specific search keywords as shown in Figure 3.32. We used BaseX XQuery API to implement fuzzy search logic to make the search process easier for edition maintainers. As explained above, sufficient package information is displayed for each search result. Additionally, a list of matching extracts containing the search keyword are also displayed.
Thus, an edition maintainer can locate packages either by browsing or searching in the Edition Builder and add the chosen packages to their editions. Adding a package to an edition is shown in Figure 3.33.

![Figure 3.31: Package Browsing Interface](image-url)
Figure 3.32: Package Search Interface

Figure 3.33: Adding a LibX package to a LibX edition
Chapter 4

Implementation

4.1 BaseX XML Database

LibX feeds are published using the AtomPub protocol. Similar to RSS, an Atom works within an XML environment to offer feed information to subscribers. We required a suitable storage scheme to store the feeds and deliver feed information to users. As explained in section 2.4.2, BaseX provides a high-performance XML Database engine, an efficient XPath/XQuery Processor, and also features a full-text processor. Thus, we chose the BaseX XML database to store, process, and manage LibX feeds. Moreover, we used the BaseX full-text retrieval feature to implement a search interface that facilitates users to locate and reuse published entries.

As discussed in section 2.4.2, XQJ, the XQuery API for Java can be used to invoke XQuery
expressions against any XML or relational database and process the query results. However, the XQJ API does not utilize the client/server architecture of BaseX. Rather than using the XQJ API, we implemented our own BaseX client interface to retrieve and update information in a client/server fashion. We created an XQJ-like layer by creating several templated XQueries using the BaseX Java Binding feature. The templated queries are used for performing database operations and are similar to XQJ queries with variable substitution.

4.2 Model-View-Controller Architecture

We implemented a Model-View-Controller architecture for the LibApp Builder to separate user interface, application, and data storage logic. The different components of the LibApp Builder architecture are illustrated in Figure 4.1

4.2.1 Model

The LibApp Builder architecture is composed of two model components. A DOM tree model is used for interacting with the XML data storage and representing feeds in a tree format. Another model component is used for maintaining a temporary in-memory cache.
DOM Tree Model

A LibX feed containing packages, LibApps, and modules is stored in the database as a linear XML document which by itself does not reflect the hierarchy of its underlying components. We need to display each feed distinctly in the LibApp Builder interface but packages, LibApps, and modules of various LibX feeds are connected in a directed acyclic graph. We unfold this graph by creating an XML tree structure for each feed to represent the parent-child relationship of its entries. A top-level entry in the tree refers to a root entry of the
feed i.e., an entry with no parent. When a registered user logs in to the LibApp Builder and opens a LibX feed, the tree structure of the feed is retrieved from the database through an XQuery called `get_connected_tree.xq` and wrapped in a Document Object Model (org.w3c.dom). Each node in the DOM tree model is tied to an atom entry of the feed and can be used to fetch the full entry information from the database. The constructed DOM tree model is completely unaware of the way it is represented in the user interface. The View component is used to render the Model state in a visual interface.

A parent entry of a feed i.e., a package or a LibApp may contain references to entries of the same feed, a peer feed, or an external feed as described in section 1.3.2. However, multiple entries of a feed may share a common child entry. A separate child node is created in the tree model under each parent that refers to the same child entry. Code Snippet 4.1 shows the expanded DOM tree model of a feed that contains cross-feed references pointing to the modules of the LibX Core Package. For example, the tree model indicates that the module, *display HTML notifications via an embedded panel using jGrowl*, is shared by two LibApps.

**In-Memory Cache**

As explained in 4.1, we implemented templated XQueries to retrieve and update XML information stored in a BaseX database. We required a suitable Model component to manipulate the stored XML data. Thus, we introduced an in-memory cache model to perform database operations. When a user accesses a specific node of the DOM tree model 4.2.1 a unique copy of the corresponding XML entry element is restored from the database and kept in a
table in memory. All retrievals and updates to the XML element are forced to go through the in-memory cache. The in-memory cache acts an intermediate layer between the database and the interface. We use the cache layer to implement an observer-pattern that facilitates bi-directional updates between the View and the database. Each child element of an entry is associated with a DOM node which listens to changes made to the field and updates the View immediately. Conversely any change made to the View is updated in the database instantly.

A single entry is cached in memory at a given time and exactly one listener i.e.,
DOMSubtreeModified listener is attached to each cached entry to ensure automatic persistence of data. An XML entry is typically composed of numerous child elements. Each child element of a cached entry is tied to a DOM node which is in turn tied to a UI component in the View. For e.g., the text child of an XML element is tied to a org.w3c.dom.Text node and this DOM node is tied to a textbox component. We defined event listeners for each such DOM node that is tied to a specific child element. When the user modifies an entry field in the interface, firstly the DOM-tree node associated with the full entry triggers a DOM mutation event which in turn triggers the event listeners of the specific DOM node that is associated with the modified child element. The event listeners notify the corresponding Controllers which then update the database information and the view components simultaneously.

However, the drawback of our design is that the in-memory cache cannot be shared across users. If the same user logs in from two different IP addresses, the state of the cached entry is lost by one user. One way to overcome this is to implement a lock mechanism but we adopted a last-write wins approach for simplicity. We consider our solution to be acceptable under the assumption that a user is not likely to initiate simultaneous writes to an entry very often.
4.2.2 View

The LibApp Builder user interface is designed using ZUML, ZK User Interface Markup Language. The interface queries the data captured in the Model and displays the information consistently. When a user activity such as a mouse movement or a button click modifies the model state, the database and the corresponding UI components are updated immediately.

The DOM tree model of a feed is displayed in the interface via ZK tree component wherein each tree item refers to a specific entry. However, the full information of an entry is opened lazily i.e., when a user selects the corresponding tree item. Any change made to the tree model such as, adding, deleting, or modifying an entry is updated in the database and the interface simultaneously.

4.2.3 Controller

In general, a Controller is used to change the Model state. We implemented a tree controller to control the DOM tree model. Each user interaction with the tree component is translated into an instruction in the tree controller and delivered to the tree model.

When a user creates or opens a new feed, the tree controller retrieves the feed from the database, creates a DOM tree model, and defines event listeners for the tree model, namely `setNodeRemovedListener`, `addTreeDataListener`, etc. The tree model implements a tree suitable for ZK Tree Component by mapping methods such as `getChildCount()`, `getChild()`, `isLeaf()` etc., to the DOM equivalents. The
getRootNode() method returns the root node of the tree which refers to a root entry of the feed. The Tree Controller renders the feed tree model in the View through a method called initializeTreeItemRenderer(). Each DOM node of the tree model is tied to a Tree Row component and two event listeners are added to each row for handling the ON_CLICK and ON_DROP events.

1. Select and modify an entry: Each DOM node of the tree model refers to an entry ID of a specific feed document. When a user selects a tree row, the corresponding entry is retrieved and cached in a temporary memory. Each entry field such as title, includes, locale, etc., is tied to an in-memory DOM node and displayed in the interface through suitable UI components. When the user adds, modifies, or deletes an entry field, the database and the interface are updated as explained in the section, In-Memory DOM Model 4.2.1.

2. Drag and drop an entry: The ON_DROP event allows users to rearrange the entries of a feed by simply dragging and dropping the tree items.

Additionally, each tree row is associated with a context menu wherein each menu item is attached to an event listener, facilitating the user to create, delete, include, copy or paste entries in the feed tree. The functionality of copying an entry to the clipboard is shown in code snippet 4.2.
Listing 4.2: Copy Menu Item

```java
抄项 = addMenuItem("Copy", new EventListener() {
    public void onEvent (Event e) {
        Element entryToCopy = (Element) getRightClickedDOMNode();
        clipboard.set(new ClipboardEntry(docName, Entry.Type.lookup(
            entryToCopy.getAttribute("type")), entryToCopy));
    }
});
```

4.3 Storage Scheme

We store the user and feed information as XML files on a BaseX server and store the published feeds in a file system. The feed information is offered to end-users through a standard web server configuration of Apache.

4.3.1 BaseX XML Database

The design used to store LibX feeds and the associated metadata is described in this section.

Storing Feeds

We need a way to uniquely identify LibX feeds and group them by owner and/or state quickly. Hence, we used a simple technique to capture the state, userid, and name of a feed in the filename itself by using the following notation: prefix_feed_userid_feedname. The prefix indicates the state whereas the userid refers to the email address of the author. The special characters in an email address i.e., @ and . are replaced with _at_ and _dot_ to create valid document names for storing feeds. The three different file formats of the core
feed created by the LibX team are shown below:

1. \texttt{feed_libx\_dot\_editions\_at\_gmail\_dot\_com\_core}: development state.

2. \texttt{PUB\_feed_libx\_dot\_editions\_at\_gmail\_dot\_com\_core}: published state.

3. \texttt{TIMESTAMP\_feed_libx\_dot\_editions\_at\_gmail\_dot\_com\_core}: archived state.

Distinguishing feeds by email address and feed name allows multiple users to share common feed names, for e.g., another user whose email address is \texttt{vsony7@vt.edu} may also create a feed called core which will then be stored as \texttt{feed\_vsony7\_at\_vt\_dot\_edu\_core} in the database. However, the names of all feeds created by a specific user must be unique.

\textbf{Storing Metadata}

\textit{User Metadata}: A user needs a valid unique email address to create a LibApp Builder account. The information of all registered users is stored in the user metadata file, as shown in code snippet 4.3. A user element is created for each registered user with the following child elements: \texttt{userid}, \texttt{password}, \texttt{username}, \texttt{organization}, \texttt{uri}, \texttt{sessionfeeds}. The first five child elements are used to store the user details directly except the user password is encrypted using a hash function to ensure security. The sixth child element i.e., \texttt{sessionfeeds} contains information of the two feeds that were most recently accessed by the user. Additionally, each user element contains two attributes for
assigning specific permissions to the user. An `admin` attribute is used to indicate whether the user is an administrator or not. An administrator is allowed to modify any feed in the database whereas other users can modify only those feeds owned by them.

**Feed Metadata:** The metadata of each feed is stored in the feed metadata file, as shown in code snippet 4.4. A feed element is created for each feed with three child elements namely, `feedname`, `userid`, and `description` to store the feed details. Additionally, each feed element contains a `publishat` attribute for storing the publish URL i.e., the location at which the feed is published.

**Roots Metadata:** Roots metadata file is used to retrieve the top-level root entries of all LibX
feeds and assign ids for newly created entries. The document attribute of each roots element corresponds to a specific feed in the database and the child id elements refer to the root entries of the feed, as shown in code snippet 4.5. Additionally, we use the curid element, short for current id, to store the id of the most recently created entry in the database. A new positive integer id is generated for each newly created entry by incrementing the curid value.
4.3.2 File System

The file system path of a published feed is derived from the author’s email address and the feed name. A parent directory by the name of a user’s email address is created in the file system for each registered user under which the published feeds of respective users are stored. A published feed can be fetched using a relative path of format userid/feedname e.g., libx.editions@gmail.com/core, libx.editions@gmail.com/bookvendor, and vsony7@vt.edu/main. An entry of a published feed can be fetched using its entry id e.g., libx.editions@gmail.com/core/libxcore, vsony7@vt.edu/main/107.

Maintaining feeds on the file system allows edition maintainers to subscribe their end users to chosen packages of LibApps. An edition maintainer may add a package to their edition by simply adding the corresponding URL path of the package. The LibX plugin tracks the LibApps of an edition through a standard web server configuration of Apache and integrates library services into web pages.

4.4 Reusable Code

We implemented a reusable design for allowing users to modify and update XML elements through the LibApp Builder interface. The underlying code techniques used for controlling and editing the text nodes and attributes of an XML element are described in this section.
4.4.1 Edit XML Text

We implemented a class called EditableXMLTextField.java to control and update the text content of any XML element in the database. The text child of a given XML element is tied to a DOM Text node and is displayed through a textbox component in the interface. When a user edits the textbox content, the corresponding DOM node is updated via org.w3c.dom.CharacterData.setData() method in the ON_CHANGE event listener of the textbox. This triggers the DOMCharacterDataModified event on the Text node and the DOMSubtreeModified event on all its ancestors, thereby updating the database and all the corresponding views subsequently.

4.4.2 Edit XML Attribute

The attributes of an XML element are controlled and updated through a class called EditableXMLAttributeField.java which is very similar to EditableXMLTextField.java except that an attribute is updated via org.w3c.dom.Attr.setValue() method in the ON_CHANGE event listener of the corresponding textbox. A DOMAttrModified event is triggered on the DOM Attribute rather than the DOMCharacterDataModified event and the remaining logic remains same.

However, some attributes may contain boolean values instead of Strings and it is more appropriate to display these attributes through checkboxes in the interface. To con-
trol a boolean attribute, we implemented an EditableXMLCheckboxField.java class which is very similar to EditableXMLAttrField.java except that the org.w3c.dom.Attr.setValue() method is implemented in the ON_CHECK event listener of the corresponding checkbox.

Additionally, a set of possible values are defined for certain attributes e.g., a parameter type can either be a string, number, boolean, or enum, and such options are displayed for an attribute through a listbox component through EditableXMLChoiceField.java class. The functionality of this class is very similar to EditableXMLAttrField.java except that the org.w3c.dom.Attr.setValue() method is implemented in the ON_SELECT event listener of the corresponding listbox.

4.5 ZK-Server Centric Framework

We have implemented the LibApp Builder interface using ZK Server-Centric Ajax Framework. The important aspects of the interface implementation are discussed in this section.

4.5.1 ZK Include Component

We designed several ZK ZUML pages to facilitate various user actions and each ZUML page allows the user to modify a specific information in the database such as modifying an entry field, updating user account information, changing the properties of a feed etc. Rather than
displaying these ZUML pages in different windows or tabs, we display a single ZUML page at a time in the _Center_ region of our layout to simplify user experience based on the premise that a user performs only one action at a time.

A ZK include component can be used to include other ZUML pages dynamically by simply changing its `src` attribute. We added a ZK include component in the Center region of the layout to which a ZUML page is dynamically set depending on the active user action. The sample code snippet 4.6 shows how the `manageuser.zul` page is added as a child to the ZK include component when a user clicks the account settings button.

### 4.5.2 ZK Generic Forward Composer

In many cases, we create and add a set of child components to a parent component dynamically for e.g., when a user performs a search action, a row component is created dynamically for each matching search result and added to a parent grid. When a child row is created for a search result, several inner components are added to it for displaying entry details such as title, content, and matching extracts. One way of creating such dynamic components is by defining the UI in pure Java, i.e., each row and its child components can be defined in a Java class. However, this approach makes UI redesigning cumbersome and hinders code maintainability. Alternatively, we used ZK GenericForwardComposers [13] to dynamically control the UI components.

A ZK GenericForwardComposer is a skeletal composer that can be extended to write intuitive
event handler methods with auto event forwarding and auto-wired variable components in a ZUML Page. The sample code snippet 4.7 shows that each search result layout is controlled by a composer class called SearchResultComposer. The composer creates and defines the inner components of the layout such as entryTypeLabel and importButton.
4.5.3 ZK Annotations

ZK annotations [11] are used to add metadata to ZK components. ZK annotations can be retrieved at the run time. The value of a component can be loaded and stored by examining its annotation using ZK Bind [12], a data binding mechanism used to synchronize data between the View and the ViewModel. Any change made to the data in a UI component can be automatically carried to the target ViewModel by creating a predefined binding relationship.

As explained in section 4.2.1, the entries of a feed are tied to an in-memory DOM model and are displayed through a Tree Component in the interface. We added components in the user interface to display specific entry details such as title, publishat url of the currently selected tree item by setting up annotations and binding these components using

```xml
<zk xmlns:html="http://www.w3.org/1999/xhtml">
    <vlayout id="resultsBox" apply="org.libx.libappdatabase.
        SearchFeedsDisplay$SearchResultComposer">
        <vlayout id="searchresultheader">
            <hlayout hflex="1">
                <label id="entryTypeLabel" />
                <a id="entryTreeDisplayLink" target="_new" />
            </hlayout>
        </vlayout>
        <hlayout id="importBox" hflex="1">
            <button id="importReferenceButton" />
            <separator orient="vertical" spacing="30px" />
            <button id="importButton" />
        </hlayout>
    </vlayout>
</zk>
```
Listing 4.8: Adding annotations to UI Components

```xml
<vbox sclass="entryheaderidtitle">
    <a:bind content="entrydisplay.activeEntryCaption" />
    <html />
</vbox>
```

Listing 4.9: Setting up annotations

```java
// set up annotations
binder = new AnnotateDataBinder(container);
displayProperties = new DisplayPropertyBean();
binder.bindBean("entrydisplay", displayProperties); //NON-NLS-1$
```

a Java Bean class, as shown in code snippets 4.8, 4.9 and 4.10.
Listing 4.10: Java Bean class for binding annotations

```java
/* This bean is tied to various ZUL attributes of element in the GUI
* that reflect the current state of the selected entry.
* Must be public for bindings to work.
*/
public class DisplayPropertyBean {
    private Entry.Id selectedId;
    private Element selectedEntry;

    public String getActiveEntryCaption() throws Exception {
        String entryTitle = Entry.getTitle(selectedEntry);
        String entryUrl = getActiveEntryUrl();

        return String.format("%s: %s", getActiveEntryType(), entryTitle);
    }

    void setSelectedEntry(Entry.Id selectedId, Node selectedEntry) {
        this.selectedId = selectedId;
        this.selectedEntry = (Element) selectedEntry;
    }
}
```
Chapter 5

Related Work

Tools to assist with end-user programming have improved dramatically over the past decade, and with good reason. Interest in end-user programming has steadily increased, and end-users are increasingly adapting to the roles of co-programmers as Web 2.0 technologies promote frequent changes [36]. Programming attempts by untrained programmers are often subject to coding errors and inconsistencies [26]. Programming languages are simply too complex and it is difficult to fully understand a language without studying it in-depth.

Regardless of the methods used to simplify programming, it has been proven that certain barriers will always create difficulties for users. These barriers arise due to a combination of factors such as, a lack of complete understanding of what the user wants to accomplish, insufficient knowledge of which interfaces are available, what these interfaces actually do, how these interfaces interact, and insufficient familiarity with the debugging process [29, 31].
Such barriers are inevitable, so there is a learning curve to any given solution; however, if sufficiently motivated, users have demonstrated that they will commit considerable time to solving problems \[40, 43\]. This information is promising because many of our intended users, primarily librarians, may lack programming expertise. It is crucial that they make the effort to understand the LibApp language model and to help users learn the language model quickly, sufficient help tooltips and informative messages have been integrated into the interface.

Non-technical users can solve technical problems provided there is a simple way to translate their ideas into functional code. While experienced programmers perform such translations with little difficulty, studies have shown that the same process is typically unnatural and difficult for non-programmers \[28, 37\]. Thus, rather than have users embrace the typical text-only programming approach, recent projects have abstracted segments of code into reusable modules that encapsulate common requirements \[24\]. By minimizing user input, these approaches have greatly reduced syntactic errors. Thus, the components of the LibApp language model have been designed to be modular for e.g., a user can build a LibApp by piecing together a set of compatible modules.

Many projects have abstracted language constructs into graphical, puzzle-like representations \[27, 28, 32, 41\]. In addition to cleanly encapsulating code into modules, these jigsaw puzzle approaches intuitively display how modules can logically interact. The jigsaw puzzle approach has proven to be very effective, especially in the case of Scratch, which has nearly one million user-created projects. Another project, CAMP, uses a magnetic poetry
metaphor to build programs [44]. Other approaches use rule definitions to control the program logic [19 33]. Among these approaches, the jigsaw puzzle metaphor has seen the most success. This is likely because there is both a graphical representation of modules along with obvious cues of how modules fit together, an important consideration for users not familiar with common programming semantics. Similarly, the packages, LibApps, and modules of a feed are represented in the interface within a tree component and users can open and control two feed tree components at any given time. The hierarchical tree structure makes it easier for users to organize their feed components and moreover entries can be exchanged between the two open feeds by a simple drag and drop operation.

Users of the LibX LibApp Builder would still require some programming knowledge and the user inputs may contain errors and inconsistencies. A simple type checking mechanism can help prevent data incompatibilities. The LibApp Builder is thus integrated with a type checking algorithm that detects errors such as inconsistent argument values or invalid entry fields. Additionally, a semantic checking algorithm has been built into the interface that detects incompatible modules within a LibApp. One simple but effective method for indicating invalid field entries is to highlight such fields with some color, such as red, to point out an error [14] and to immediately display detailed error messages to describe the errors [15]. Descriptive in-line error messages are displayed in the LibApp Builder interface whenever a user input doesn’t match the syntax or semantic rules of the language model to assist users in fixing their code promptly.

User-created programs can fail due to reasons other than type checking and semantic errors.
Studies have shown that it is imperative that end-user programs update and reflect the changes made to a piece of code as soon as the changes are made so that users can test as they create; users are accustomed to software that provides immediate feedback when performing actions [18, 20, 29]. One debugging approach is to incorporate errors into the design model [30]. The LibApp Builder interface allows users to activate, test and execute their LibApps on web pages before having to publish the LibApps. Users can troubleshoot ambiguous problems through this activation and testing mechanism.

One concern in the LibApp Builder is that modules are developed to work for specific pages, but these pages are modified outside of our control. Semantic anomaly detection [39] is therefore an important consideration to ensure that modules are updated promptly. When a module is updated, any changes made to it must be reflected in every application that relies on it [42]. The LibApp Builder facilitates users to re-publish their feeds multiple times, thereby providing an efficient updating mechanism. However, the onus is on the feed maintainers to update their feeds when required. Each time a feed is published, the previously published version is saved as an archive to provide a fall-back option to the users. The feed publishing mechanism is intended to increase user contribution by creating a community platform for building and sharing LibApps.
Chapter 6

Conclusion

This chapter summarizes the design and implementation details of the LibApp Builder project and also proposes the future scope for improving the intuitiveness and visual appeal of the application.

6.1 Summary

The LibX LibApp Builder enables users to create library applications, known as LibApps with minimal programming effort. A LibApp enhances the functionality of web pages by integrating library services through context-sensitive per-page transformations. We used ZK AJAX framework to build a rich assistive user interface that is interactive and intuitive. The interface relies on a high-level language model that is designed to help non-technical users adapt as meta-designers. Numerous help and information tool tips have been incorporated
into the interface to familiarize users with the LibApp language model and minimize the learning curve of the LibApp development process. Additionally, we implemented syntax and semantic error-checking algorithms to reduce the debugging effort. In-line error messages are displayed in the interface to help users fix their code whenever an error occurs.

The design of LibApp language model includes standard programming practices such as argument passing and type-checking. The complex JavaScript code of a LibApp is hidden from users by decomposing the script into smaller units called modules. By passing arguments from a LibApp to a module, a user can customize the module without having to understand the underlying module code. Moreover, we implemented a tuple space mechanism that allows modules to interact by placing data in shared space. The tuple space model eliminates the need to arrange modules in a specific execution order which reduces coupling and increases re-usability of the modules. Hence, the LibApp language model provides a user-friendly programming framework which simplifies the process of building LibApps.

The modules and LibApps of a feed are grouped into packages and are published via Atom-Pub format. The standard format makes it easy to parse and serialize the packages of a LibX feed. We implemented a mechanism that allows users to activate and test the LibApps of a feed before publishing the feed. Upon successful execution of the LibApps, the users may distribute the entries of a feed to other users by publishing the feed. Keeping in mind that the users need a convenient way to locate and reuse such published entires, we built a faceted quick search feature in the interface. We used the BaseX XQuery API to build a fuzzy full-text search logic that returns matching entries for approximate or incomplete
search key words. Thus, the publishing and search features of the LibApp Builder create a community platform for users to build and share LibApps.

6.2 Future Work

The objective of the LibApp Builder project is to enable users with minimal programming knowledge to build complex library applications. In the future, we could perform a user study to test the success of the project. One way to perform the study is to log the activity of users given their consent. The logs can then be analyzed to measure the user effort involved in creating a LibApp. The user effort can be measured using various metrics such as, the time taken to create a LibApp, the number of errors occurred while creating a LibApp etc. Additionally, we could analyze the utility of the help tooltips by tracking how often users access these tooltips. Logging user activity during the LibApp development process will also help us provide better user support without having to acquire a detailed explanation of an issue. We could introduce a support button in the interface using which a user can send a brief note to the LibX team on encountering an issue. We could then analyze the log information of the corresponding user and offer quick assistance.

Additional ways to simplify user input can be built into the application. For example, in the current implementation, a user needs to directly input a JSON Object while creating a locale field which could cause syntactic errors and delay the development process. Alternatively, the interface can be improved to accept a number of key-value pairs which could then be
internally translated into a valid JSON object. Further abstracting out such programming details would make the programming experience more natural and reduce the number of errors.

The intuitiveness of user experience can further be increased by adopting a puzzle piece design in the interface, as seen in Scratch [27] [28] [32] [41]. For example, if a module of a LibApp contains a `guardedby` field that has not been produced by a sibling module then an incomplete puzzle piece could be displayed in a working area. The user could solve the puzzle by adding the required `produces` field in a sibling module. Such a design would improve the overall user experience of the interface.
Bibliography


[34] Brad A. Myers, Andrew J. Ko, and Margaret M. Burnett. Invited research overview: end-user programming. In CHI '06: CHI '06 extended abstracts on Human factors in computing systems, pages 75–80, New York, NY, USA, 2006. ACM.


