**Dawa Sherpa, Jesse Weaver, Taylor Schmidt**
**ByWater Solutions**
**June 24, 2014**

# Table of Contents:

Introduction…… 4
- Product Description (Koha)
- Client Description
- Product Vision

Requirements……... 5
- Functional Requirements
- Non-Functional Requirements

System Architecture…… 6
- High Level Design
  - figure 1.1
- Detail Design

Technical Design…… 8
- AngularJS
  - figure 1.2
  - figure 1.3
  - figure 1.4
  - figure 1.5

Design & Implementation Decisions/Lessons Learned…….. 11

Results…… 12

Appendix A: Product Installation Instructions……..13

Appendix B: Coding Guidelines……14
- Complete coding guidelines:

Appendix C: Acceptance Tests…….. 16

References…… 18

# Introduction

### Product Description (Koha)

Koha is the world's first open source Integrated Library System (ILS), software used by libraries to manage their patrons and collections. It is in use worldwide, and its development is driven by the libraries themselves. Since its inception in 1999, Koha has been expanding to meet the needs of its ever growing client base. It includes many useful features central to library operation including circulation, cataloging, acquisitions, serials, reserves, patron management, branch relationships, and more.

### Client Description

ByWater Solutions was created in 2008 as a Koha support and implementation company. ByWater is actively involved in the Koha community and offer many services dealing with the ILS. These include support tasks dealing with technical consulting, bug fixes, and Koha customization to meet the needs of specific libraries. ByWater also deals with installation and data migration for libraries switching over to the Koha system, as well as training to help libraries learn to use the ILS. Lastly, ByWater Solutions is also heavily involved in the development of Koha's open source software, and are major contributors to the worldwide Koha community. ByWater Solutions currently employs a staff of 15 people, and has expanded to a partner base of over 800 libraries, making them one of the largest Koha service providers in the world.

### Product Vision

Develop a RESTful API along with front-end AJAX web application to expose this API for the open source ILS known as Koha. The web application should make use of the AngularJS framework to dynamically load information. The product created will specifically deal with Koha's circulation tasks associated with the checkin/checkout of items within the library. During this project, a nearly complete checkout interface and API will be implemented at a minimum, with as much of the rest being implemented as time permits. It is hoped that the beginnings of this project will eventually replace the current interfaces and back-end structure within Koha.

# Requirements

### Functional Requirements

- Web Service API:
    - Querying all relevant information about bibs (bibliographic records), their items, and borrowers, including holds, fines, checkouts, and personal information
    - Checking out, checking in, renewing and placing holds on items
    - Checking in items while forgiving fines
    - Paying fines and renewing expired borrowers
    - Correctly transfers items between libraries if needed
    - Does all of the above while respecting circulation rules and limits, lost statuses and existing holds
    - Not specific to the needs of the frontend, intended as a generic API
    - Complete unit-test suite
- Front end:
    - Exposes the above functionality through a web-based AJAX interface
    - Displays warnings and errors to users while supporting internationalization
    - Plays alert sounds
    - Can print receipts (common receipt printers function easily with browsers)

### Non-Functional Requirements

- RESTful HTTP API.
- Uses AngularJS.
- Can be integrated into existing Koha code base.
- Uses Git version control to eventually merge the project with the master branch of Koha

# System Architecture

### High Level Design

Our system involves the interaction of library staff and the Koha database for circulating items within the library. The library staff will interact directly with Koha through a web browser. Any actions taken by this person will be transferred through AJAX calls via the internet, sending requests to our API. Our API will then handle these requests, retrieving and/or updating any relevant information in the library's Koha database. After this interaction occurs, The API sends back the information to the browser, which updates the user interface accordingly.

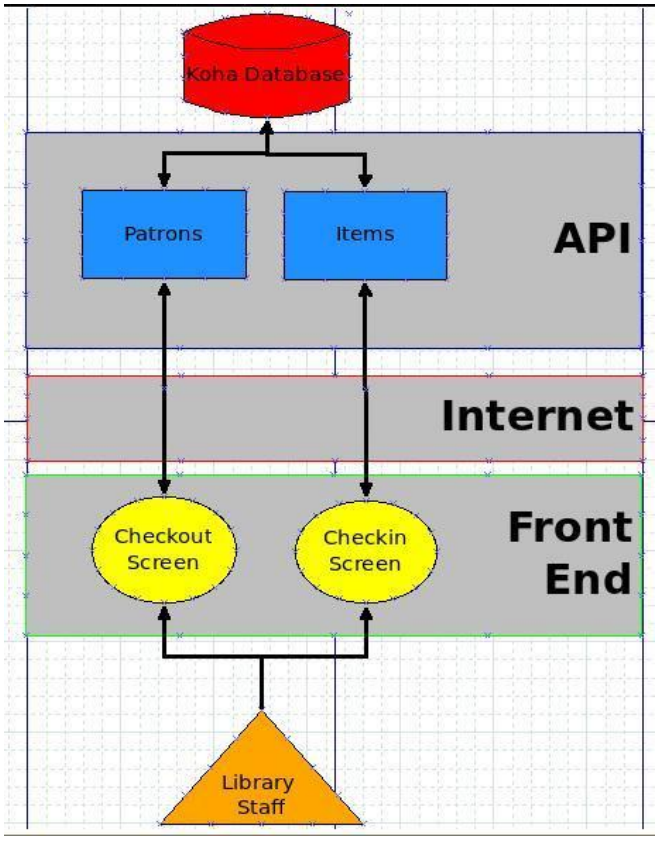



figure 1.1

### Detail Design

Database:
Our project interacts with the existing Koha SQL database, which holds all of the information needed by a library, including item information and patron information, as well as all of the circulation tasks that have taken place.

API:
Our circulation API is divided into two modules entitled Patrons and Items. The API is organized this way because circulation functions in a library either occur on a patron specific basis or on an item specific basis. This division ensures the API will be RESTful in interfacing with the front end, extremely modular, and easily expandable in the future if additions and/or modifications need to be made.

API calls for tasks such as checking out items, placing holds, and paying fines will always be accompanied by information for a specific library patron. These tasks are

intrinsically connected to a patron, so it made the most sense to place them within a patron module.

All other circulation tasks within a library are specific to individual items. These include checking an item in, adding new items, and making modifications to items' statuses. These tasks within the API will always need to be called with an item number.

Front-end:
The front-end we worked on is composed of two separate web applications. One deals with the checkout of items, and the other deals with checkins. Both of these pages are meant to replace their respective existing interfaces in Koha. These applications display their information using the AngularJS framework in order to update and display elements dynamically instead of reloading the entire page when information needs to be updated. This allows the page to work in a fast and efficient manner, especially when dealing with large amounts of information. Both of these applications use RESTful AJAX calls to communicate with our newly created API which handles the retrieval and updating of the database information.

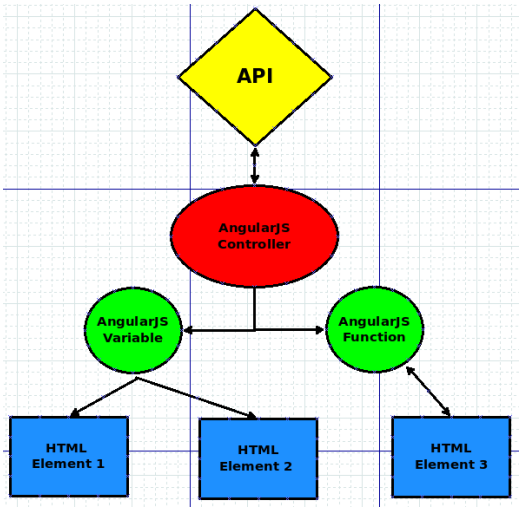
# Technical Design

### AngularJS

One of the suggestions made by our client for this project was the use of the AngularJS Javascript framework. This framework provides a very straightforward way of dynamically updating web pages one element at a time, without reloading the entirety of content. This is especially useful in the context of Koha, as library staff can experience somewhat large wait times when trying to update or view a large amount of data in the application. With AngularJS integrated into the User Interfaces (UI), large amounts of information will no longer need to be passed between the client and server, reducing client side wait time dramatically in some cases.

AngularJS works by binding HTML elements on a web page with variables and functions defined in Angular “controllers” within Javascript elements. Our controller communicates with our back-end API to retrieve all the necessary information regarding things such as the checkins and checkouts needing to be displayed by the application. Our controller will then update data in  HTML elements using directives defined by Angular, communicating with the HTML, telling it what to display and taking in information it needs. This is all done in real time, with no pause or reloading of the URL, making it very streamlined and efficient. Below is an example of a generic Angular setup where variables and functions are linked to specific HTML elements by the controller.

figure 1.2




To control the HTML, AngularJS comes with a large set of very useful features which can be easily integrated into any webpage. This was especially useful in our case because we needed to modify an existing interface to look and act the same as the previous version, but operate far more efficiently behind the scenes. This was done by only modifying HTML tags with AngularJS directives to either hide or show the entire element based on a variable or the result of a function. AngularJS also allowed us to capture form data and store it directly into variables within the controller. Using this approach, we were able to keep most of the existing code for each page, adding in our own modifications as needed. Below are a few direct

code examples from our project to demonstrate Angular's integration into our web application.

This code snippet from our project is a good example of one of Angular's directives known as "ng-show". This is a commonly used directive to dynamically display or hide an HTML element based on a boolean variable set within the controller, dictated by information received from the API.

figure 1.3

```
<li ng-show="questions.NOT_FOR_LOAN_FORCING">
    <span ng-show="questions.itemtype_notforloan">Item type is normally not for loan.</span>

    <span ng-show="questions.item_notforloan">
        Item is normally not for loan
```

This next code snippet shows how controller variables can be directly inserted into HTML using syntax unique to AngularJS. When a variable is encapsulated in curly braces, like `{{variable}}`, AngularJS knows to display that variables value dynamically in the web application, allowing it to be easily changed, without any HTML modification.

figure 1.4

```
<li ng-show="questions.AGE_RESTRICTION">
    Age restriction {{ questions.AGE_RESTRICTION }}.
    [% IF CAN_user_circulate_force_checkout %]
        Check out anyway?
    [% END %]
</li>
```

The final code example from our project shows Angular's ability to easily bind HTML forms to functions within the controller. This is done using the "ng-model" directive to bind an input field to a variable, and then the "ng-submit" directive is used to call a function within the controller when the form is submitted. In our case, the function called will send an AJAX "POST" command to our API.

figure 1.5

```
<form id="checkin-form" autocomplete="off" ng-submit="checkin()">
<div class="yui-u first">
        <fieldset>
<legend>Check in</legend>
        <label for="barcode">Enter item barcode: </label>
        <input name="barcode" id="barcode" size="14" class="focus" ng-model="barcode"/>
        <input type="submit" class="submit" value="Submit" />
```

For this project, AngularJS proved to be a very powerful tool. In most cases, a web application would need to experience a complete overhaul in order to incorporate a new framework or to see any extreme increases in efficiency. With AngularJS, most of the original web code was kept, we only needed to add in the appropriate directives and bind the necessary variables in order to interface with our newly created API. In many cases, we were able to actually simplify the existing Koha code. AngularJS was the technical design decision which made the most impact on our project, allowing us to overhaul Koha's circulation client in a relatively short period of time.

# Design & Implementation Decisions/Lessons Learned

- The use of the AngularJS framework for this project was one of the better design decisions we could have made. Angular's built in directives and functionality allowed us to easily convert Koha's old circulation UI without a total overhaul of the code. This allowed us to easily create a front-end which would dynamically load content, while maintaining the same look and functionality as the previous pages. AngularJS also allowed us to easily connect the front-end with our newly

created API by using functions within Angular to send AJAX calls using data lifted directly from the web application.

- Creating a RESTful API proved to be a very useful way to structure our code, as our newly created services are structured like actual resources. This ended up as a very modular and expandable API, greatly improving on the previous codebase.
- When interfacing directly with the Koha database, we found that using Perl's DBIx::Class object-relational mapper was very helpful. This allowed us to retrieve and/or update information in the Koha database without actually writing raw SQL commands. This sped up development time tremendously.
- In trying to overhaul the front-end web application of Koha's circulation UI, we attempted to integrate AngularJS with the DataTables JS library currently used in Koha. After wasting an entire day trying to mix the two together, we determined that the task was an impossible one, and had to scrap a large portion of the code we had written. Researching this problem after the fact led us to find out that many had attempted to integrate DataTables with other frameworks and libraries to no avail, concluding it to be an insurmountable task due to compatibility issues. In the future, we have now learned to thoroughly research a problem in its early stages before trying to push through it, only to realize success is not possible.
- Working with a group of three for the entirety of our project led to a fair amount of improvisation for some of the Agile and Scrum mentalities. Instead of pair programming with an odd man out, we experimented with Triplet programming throughout the project. This turned out to be very successful, leading to some interesting group dynamics when coding. Two members would take on the classic roles of a pair programming duo who are focused on the task at hand, while the third member would almost act as an overseer, catching syntax and coding errors.

# Results

The initial goal for this project was to create a Restful API for the Koha circulation client, to be exposed using an AngularJs web application with AJAX calls. This included both the checkout and checkin interfaces for items in the library. Our main objective was to get the checkout functionality in as complete of state as possible, and then move on to the checkin interface as time allowed.

So far, we have completed as much functionality as possible for the checkout API and front end using AngularJs. Our Interface implements around 95% of the features of the old checkout interface while making it more streamlined and dynamic so that pieces can be updated individually without reloading the entire page.

We have also made a good dent in the checkin interface and subsequent API. Koha is used by thousands of libraries worldwide. If our code is implemented into the system, and an update is released, librarians will have the option to use the original checkout/checkin clients developed for the integrated library system, or our AngularJS version.

Our API has been subjected to unit tests throughout every step of development to ensure that every component is working correctly. Our frontend has also been extensively tested using both Firefox and Chrome to ensure that it is functional in all browsers specified as compatible with the Koha system. We have had a first look into the future of Koha, Angular

Within the next few years, we hope to see Koha grow more dynamic, in order to improve processing times for library circulation. Hopefully, the introduction of AngularJS will allow for the addition of more single-page applications, which will make loading pages faster and simpler. It would also be nice to see our checkin/checkout screens fully developed and being used within the Koha community.

# Appendix A: Product Installation Instructions

**Koha Requirements**

To install Koha for immediate use we recommend

- A Linux server – Debian is what most people use
- Apache
- MySQL
- Perl
- Root access to the server
- A better than average level of skill with the command line, Apache, and MySQL tools

Koha is free software and is licensed under the GNU General Public License, either version 3 of the License, or (at your option) any later version

**Get Koha**

There are often two versions of Koha being released here during the same period of time. See the release schedule (http://koha-community.org/about/release-schedule/) for more information.

**Current Release**

Debian packages are available – Instructions here (http://wiki.koha-community.org/wiki/Debian)

**Ubuntu users** can use the packages as well by following these instructions: (http://wiki.koha-community.org/wiki/Koha_on_ubuntu_-_packages)

From download.koha-community.org **Not recommended, use Debian/Ubuntu packages** – Current Stable Release (.tar.gz) (http://download.koha-community.org/koha-latest.tar.gz)

**Install Koha**

Once you have downloaded Koha, please unpack it and find the installation and upgrade instructions in the INSTALL file for your system, or the general INSTALL file.

# Appendix B: Coding Guidelines

General rule: if you submit code that fixes existing code that violates those guidelines, QA can still be passed. You don't have to fix everything. If you want to fix more than just your fix, to respect current guidelines, you're welcomed of course. But in this case, please do it in a 2nd patch, to have reviewers being able to distinguish easily what's related to your bugfix and what is related to your guidelines-violation fixing.
Example: you submit a patch that does not respect perlcritic after applying the patch. The QA team will check for perlcritic after your patch, and if the patch does not add a new perlcritic violation, it's OK.

### Licence

Each file (scripts & modules) must include the GPL licence statement, typically at the top.

```
# This file is part of Koha.
#
# Copyright (C) YEAR  YOURNAME-OR-YOUREMPLOYER
#
# Koha is free software; you can redistribute it and/or modify it
# under the terms of the GNU General Public License as published by
# the Free Software Foundation; either version 3 of the License, or
# (at your option) any later version.
#
# Koha is distributed in the hope that it will be useful, but
# WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with Koha; if not, see <http://www.gnu.org/licenses>.
```

### Commit messages

When you submit code to the project (using Git) please write useful commit messages. Detailed guidelines on writing commit messages can be found at the wiki page on Commit messages (http://wiki.koha-community.org/wiki/Commit_messages).

### Refactoring code

Don't needlessly refactor code, **if it ain't broke, don't fix it**! Don't waste time on changing style from someone else's style to yours! If you must refactor, make sure that the commit for the refactoring is completely separate from a bugfix.

**Bug Numbers**

Contributors to Koha must reference a bug number with every commit. This helps us determine the purpose of the commit and establishes a more searchable and understandable history. If there is no bug open at bugs.koha-community.org that addresses your contribution, please open one and describe the bug or enhancement. Then, include Bug + the bug number at the beginning of your commit message and set the "Status" field to "Needs Signoff". This helps us keep track of patches that have been contributed but not yet applied.
It is also requested that you attach the patch file to the bug report. This allows others not on the patches list to pull in your code, test, and sign-off. You can use git bz for that.

**Complete coding guidelines:**
http://wiki.koha-community.org/wiki/Coding_Guidelines

# Appendix C: Acceptance Tests

**Koha Revision Control Overview for RMs**

Applying Patches

Ok, the release manager has received some patches and wants to apply them.
Git also provides a tool called git-am (am stands for "apply mailbox"), for importing such an emailed series of patches. Just save all of the patch-containing messages, in order, into a single mailbox file, say "patches.mbox", then run
$ git am -3 -i -s -u patches.mbox

Git will apply each patch in order; if any conflicts are found, it will stop, and you can fix the conflicts as described in "Resolving a merge". (The "-3" option tells git to perform a merge; if you would prefer it just to abort and leave your tree and index untouched, you may omit that option.)
Once the index is updated with the results of the conflict resolution, instead of creating a new commit, just run
$ git am --resolved

and git will create the commit for you and continue applying the remaining patches from the mailbox.
The final result will be a series of commits, one for each patch in the original mailbox, with authorship and commit log message each taken from the message containing each patch.

**Pushing changes to a public repository**

Now they want to update the public repository so the rest of the world can get the new code. The simplest way to do this is using git-push and ssh; to update the remote branch named "master" with the latest state of your branch named "master", run
$ git push ssh://yourserver.com/~you/proj.git master:master

or just
$ git push ssh://yourserver.com/~you/proj.git master

As with git-fetch, git-push will complain if this does not result in a fast forward. Normally this is a sign of something wrong. However, if you are sure you know what you're doing, you may force git-push to perform the update anyway by proceeding the branch name by a plus sign:
$ git push ssh://yourserver.com/~you/proj.git +master

Note that the target of a "push" is normally a bare repository. You can also push to a repository that has a checked-out working tree, but the working tree will not be updated by the push. This may lead to unexpected results if the branch you push to is the currently checked-out branch!
As with git-fetch, you may also set up configuration options to save typing; so, for example, after
$ cat >>.git/config <<EOF
[remote "public-repo"]
url = ssh://yourserver.com/~you/proj.git
EOF

you should be able to perform the above push with just
$ git push public-repo master

See the explanations of the remote.<name>.url, branch.<name>.remote, and remote.<name>.push options in git-config for details.

**Koha Revision Control Overview for QA Manager**

The QA manager will receive patches and will want to either apply them and then push them upstream to the RM's repo or escalate the patch for the RM to view.

Dealing with Patches

The QA manager needs to check if the patch is a bugfix or a new feature. If it is a bugfix they then need to check that it confirms to the coding guidelines. If it does they can then apply the patch (the same way as the RM does). Then they do a git-push to push this back to the RM's repository. If they are new features, the QA Manager will forward them to the Release Manager to look at.
For example

- Check mail
- Look at new patch email, its a bug fix, looks good, save to a file to_apply
- git-am to_apply
- git push
- Let the RM know.

OR
- Check mail
- Look at new patch email, ooh a neat new feature, forward to RM Manager

# References

appendix a: http://koha-community.org/download-koha/
appendix b: http://wiki.koha-community.org/wiki/Coding_Guidelines
appendix c: http://wiki.koha-community.org/wiki/Version_Control_Using_Git