UMass BookShare

Chris Paika, Piers Calderwood, Walter Doan,
Amy Jiang, Bianca Tamaskar, Qiwen Wang

Software Design Specification

Draft 1
Feb 18, 2015

CSRocks Inc.

Revisions

Version	Primary Author(s)	Description of Version	Date Completed
1	UMass BookShare	First Draft	2/16/2015
2	Peter, Piers	Changes after Sunday Meeting	2/22/2015
3	UMass BookShare	Final Draft before submission	2/24/2015
4	UMass BookShare	Updates before Beta Release	3/31/2015
5	Peter	Updates before 1.0 Release	4/29/2015

System Architecture

1. Introduction

UMass BookShare is a web application that will serve as an online marketplace for UMass Amherst students to buy, sell, lend, and borrow textbooks. It aims to provide a more convenient and sustainable way for students to gain access to the books they need. Users can post books they have and are willing to share with others. They can also search for a book and view listings of which students have it and how they are willing to share it (sell, rent, lend). Additionally, the application has a wishlist feature that notifies a user when a book on his or her wishlist has been posted by a seller. The buyer and seller will negotiate a meeting place and payment details through a third-party communication channel.

Our system follows the Model-View-Controller style and Client-Server style. In our system, the "model" is the PostgreSQL database that will store information on users, listings, history, and more. The "controller" is the server which handles requests by the user and manipulates or queries the database accordingly. The server will be implemented utilizing Node.js and Express. The "view" is the User Interface that the user will interact with when using UMass BookShare. We selected to use the Model-View-Controller style for the high-level architecture because it decouples different aspects of the system - processing the data, manipulating the data, and viewing the data are all isolated (*see Figure 1*). This design, a standard practice in industry, provides greater flexibility because the different parts can be modified without greatly affecting the functionality of the others. After discussing the requirements of our system, we were able to design a class diagram to illustrate the various classes that will be necessary, as well as the interactions and associations between them (*see Figure 2*).

This System Design Specification and Planning Document describes the software architecture of UMass BookShare, as well as some key design decisions that were made. Additionally, this document details how the team will proceed in carrying out this project and what risks we may face. The Test Plan and Documentation Plan are also described here.

2. System Architecture

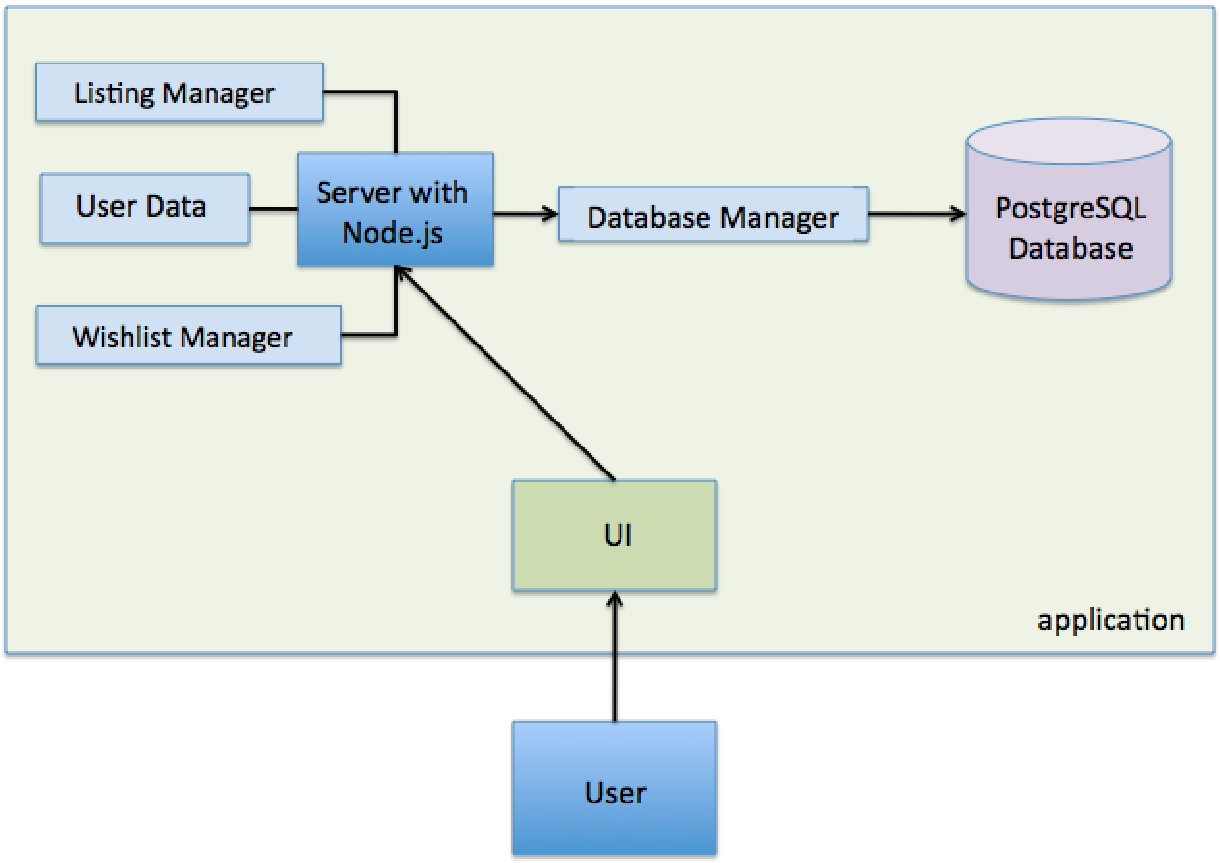

Figure 1: This diagram is the high-level software architecture view. The architecture style is Model-View-Controller. The user "views" using the UI and "controls" by making requests of the server. The "model" is the PostgreSQL database that contains all listing data, user data, etc.

3. Design View

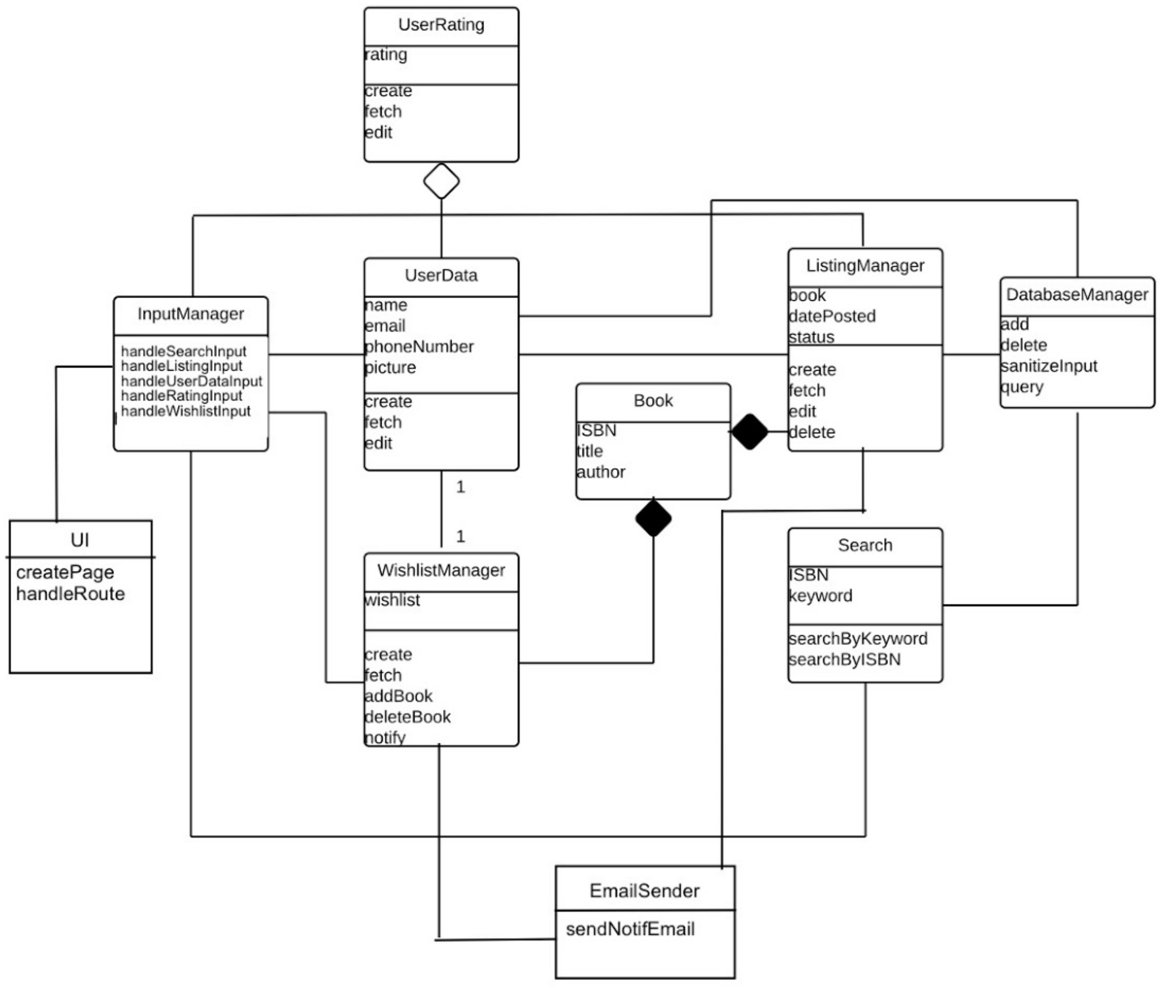

Figure 2: This *UML Class Diagram* illustrates the various classes we will use to implement the application.

4. Process View

<u>Use Case: Create Listing</u>

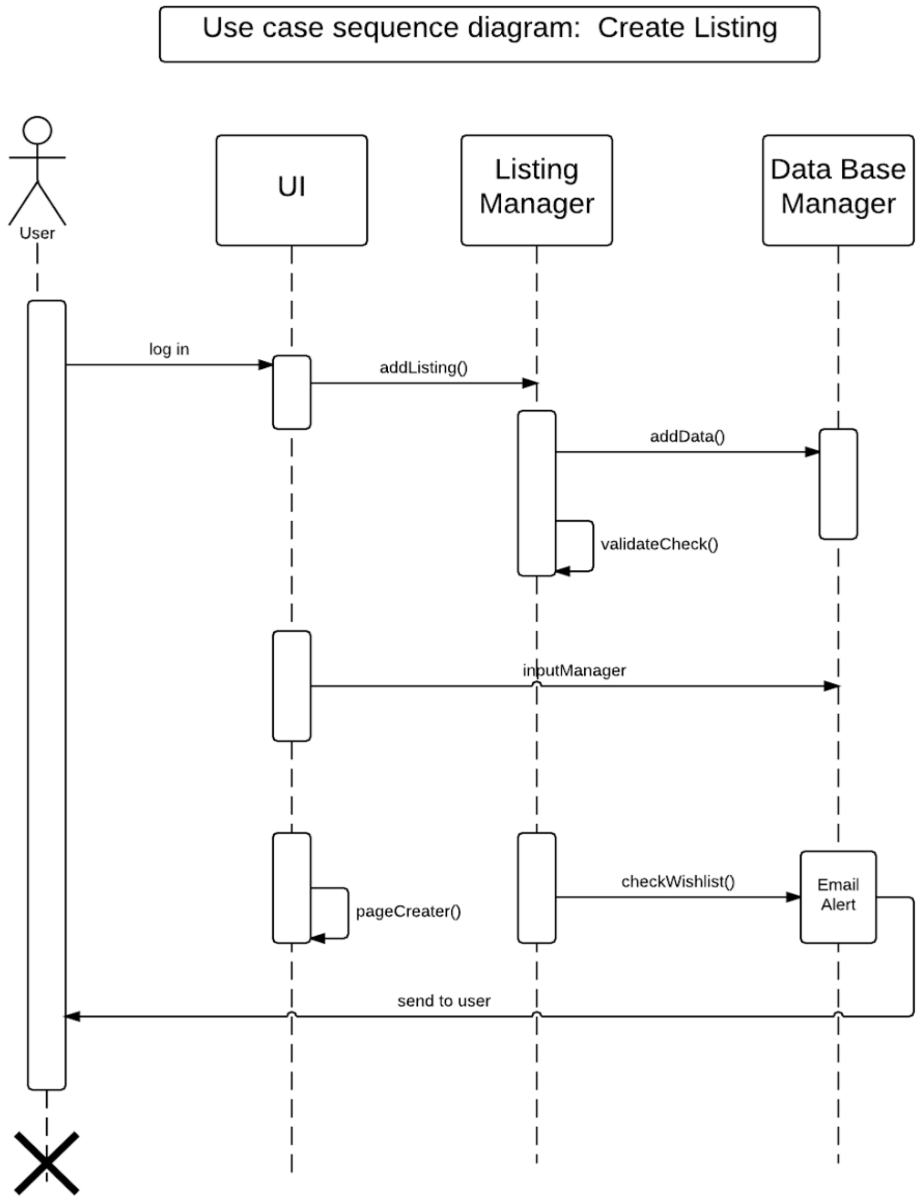

Figure 3: *UML Use Case Sequence Diagram* for creating a listing

The user logs in from the front page, and adds listings to the Listing manager through the UI. The listing manager checks validation first then adds data to data manager, input manager send message directly to database. Email center is under database, and send message to user once it checked form wishlist.

Use Case: Search

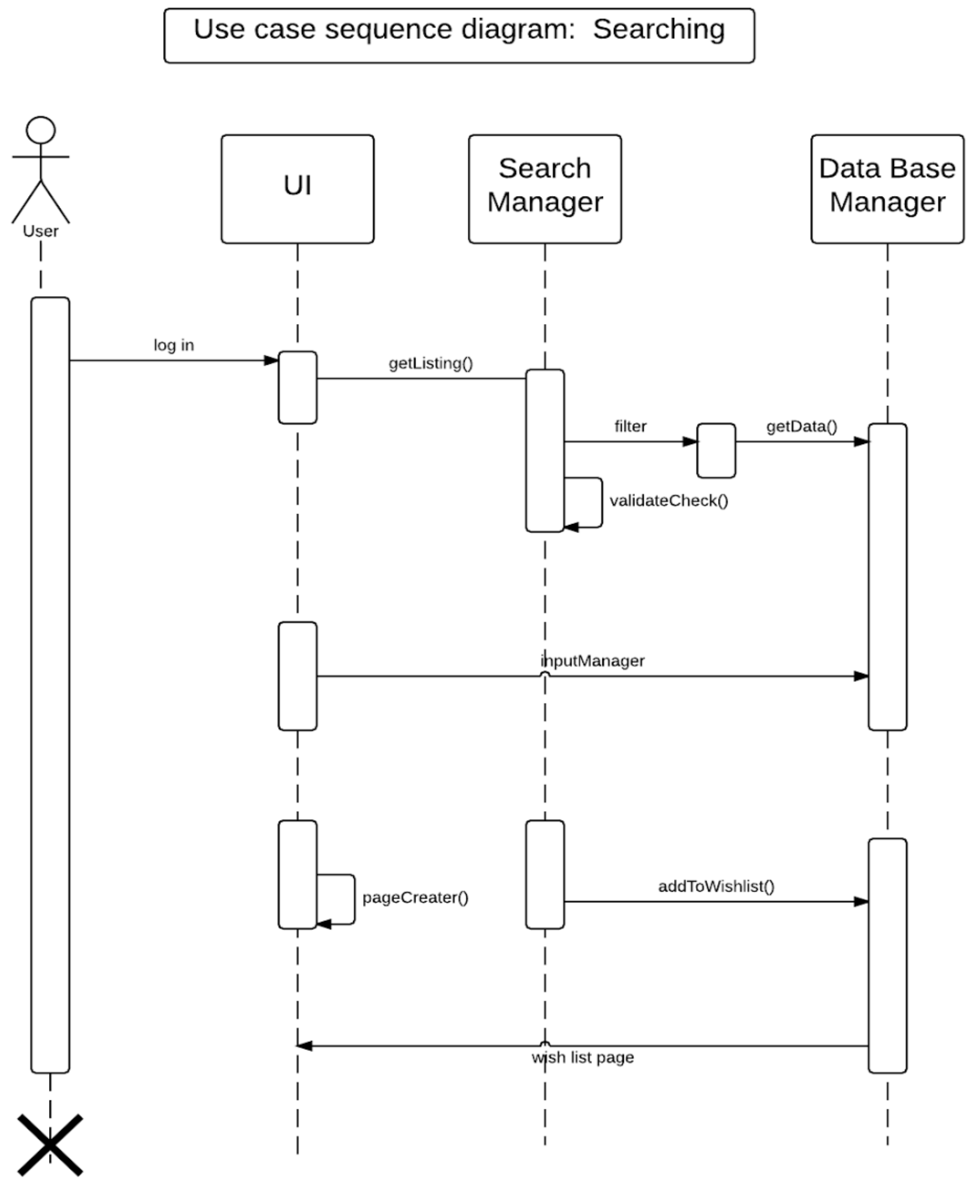

Figure 4: *UML Use Case Sequence Diagram* for searching

The user login from the front page, and gets listings from the search manager through the UI, the search manager checks validation first then get data from data manager through filter first, input manager send message directly to database. Search manager also creates the wishlist, and sends it to the database.

5. Database Schema

We chose to use a relational database since we can easily index large portions of the system and the entire group has previous experience with similar systems.

Our primary entities are Users, Book, and Listing, which store information about the system's users and books that users wish to either sell, rent out, or lend. Each User has a unique username and each Book has a unique ISBN-10 and ISBN-13 although we only require the ISBN-13. All interactions in the database are dependent on these two sets. Other entities are dependant on User or Book.

Some specific fields to make note of are sex, books, and cover photos. We store sex as a CHAR since it is equivalent in size to a BOOLEAN and it is more obvious what "m" and "f" mean. Books are stored locally after being accessed once from the OpenLib DB because it is very likely the books will be accessed again and this allows for faster processing. Cover photos are also stored in the table if they are available.

```
create table Users(
	username CHAR(20),
	password CHAR(128),
	firstName CHAR(30),
	lastName CHAR(30),
	email CHAR(50),
	phone CHAR(12),
	institution CHAR(60),
	age INTEGER,
	sex CHAR(1),
	profilepic VARCHAR(255),
	PRIMARY KEY(username)
);

create table Book(
	title VARCHAR(255),
	author CHAR(60),
	isbn10 CHAR(10),
	isbn13 CHAR(13),
	publicationDate DATE,
	version SMALLINT,
	cover VARCHAR(255),
	PRIMARY KEY(isbn13)
);

create table Wishlist(
	username CHAR(20),
	isbn13 CHAR(13),
	wishDate DATE,
	PRIMARY KEY(username, isbn13),
	FOREIGN KEY(username) REFERENCES Users,
	FOREIGN KEY(isbn13) REFERENCES Book
);

create table Listing(
	listID CHAR(32),
	username CHAR(20),
	isbn13 CHAR(13),
	forRent BOOLEAN,
        rentPrice CHAR(10),
        forSale BOOLEAN,
        sellPrice CHAR(10),
        forBorrow BOOLEAN,
        available BOOLEAN,
        listDate TIMESTAMP,
        description CHAR(500),
        PRIMARY KEY(listID),
        FOREIGN KEY(username) REFERENCES Users,
        FOREIGN KEY(isbn13) REFERENCES Book
);

create table Rating(
        ratedUser CHAR(20),
        ratingUser CHAR(20),
        rating INTEGER,
        PRIMARY KEY(ratedUser, ratingUser),
        FOREIGN KEY(ratedUser) REFERENCES Users(username),
        FOREIGN KEY(ratingUser) REFERENCES Users(username)
);
```

6. Design Alternatives and Assumptions

Chat:

We decided to not include chat or Private Messaging between users. Firstly, we did not wish for communication to be handled by the site as this forces us to spend time securing and possibly moderating chat. It also allows the person with the books to decide how they are contacted and for people looking for a book to mistakenly PM someone who only checks their email. Unfortunately this means users will spend less time using the service since they do not need to regularly login to check messages.

Apache:

One potential alternative designs was to use Apache instead of Node.js. The reason why we discussed potentially using Apache is because the learning curve of PHP could have been easier than learning Javascript. However, because the team collectively wanted to learn Javascript and Node.js we ultimately decided on choosing Node.js based on personal interests.

There were many other options we could have gone with instead of just deciding between Node.js and Apache but we just felt that those two would provide the most documentation and online resources for us to be successful.

Development Plan

1. Team Structure

Structure and Responsibilities

- **Project manager: Chris**
  Organizes meetings, reviews drafts, works with customer.
- **Testing - Qiwen**
  Qiwen responsible for developing our test strategy and both our automated and manual tests. Peter will help as he is needed.
- **UI - Amy**
  Amy is responsible for mock-ups, and other User Interface design decisions. She will be involved in Test and System architecture as needed.
- **Database - Piers, Peter**
  Piers is responsible for developing the database schema, as well as ensuring every group member can configure their development environment to work with the database correctly. He is also responsible for handling database interaction in the system architecture.
- **System architecture - Bianca, Walter, Peter, Amy**
  Bianca, Walter, Peter, and Amy are responsible for the planning and design of system architecture. This will transition to server-side coding as we transition to alpha release.

Communication

The team communicates mainly through Google Hangout, group emails, and Trello. We post documentation and other group resources on our Slack page where they are accessible at any time, and communicate for planning purposes via email. We have weekly meetings in our discussion section on Wednesday and also meet on Thursday or Friday in the USpace (depending on team members' changing schedules). We utilize a shared Google Docs folder to collaborate on documents and presentations. As we transition to alpha, beta and final releases we will utilize github.com for source control, and maintain communication on the Slack platform.

2. Project Schedule

Task/Milestone	Estimated time	Date due	Resource(s)
Set up initial db	1 day	Mar 2	Piers

Set up server	3 days	Mar 6	Qiwen
Define site structure w/ navigation flow	2 day	Mar 6	Amy
Alpha Release		***Mar 9***	
Design and implement initial GUI for web app	4 days	Mar 14	Amy
Implement Create Account	1 day	Mar 17	Peter
Implement Create Listing	1 day	Mar 17	Walter
Implement Search Function	2 days	Mar 20	Bianca
Implement Display Listing	1 day	Mar 20	Walter, Amy, Peter, Bianca
GUI talks to db	2 days	Mar 25	Piers, Amy, Walter, Peter, Bianca
Customer Testing	1 hour	Mar 31	all
Beta Release		***Apr 1***	
Continued GUI development	4 days	Apr 13	Amy
Wish-list	1 day	Apr 16	Peter, Amy, Piers, Bianca, Walter
View/Edit Profile	1 day	Apr 16	Peter, Amy, Bianca, Walter
Edit/Delete Post	1 day	Apr 16	Peter, Amy, Bianca, Walter
Display Recent Listing	1 day	Apr 21	Peter, Amy, Bianca, Walter
Notifications	1 day	Apr 21	Peter, Amy, Bianca, Walter, Qiwen

Testing/Debugging GUI and feature functions	4 days	Apr 25	Piers, Qiwen,
Customer Testing	1 hour	Apr 27	all
1.0 Release		***Apr 29***	

3. Risk Assessment

Risk	**Chance of occurring (H, M, L)**	**Impact if it occurs (H, M, L)**	**Steps taken to increase chance it won't occur**	**Mitigation plan should it occur**
Project schedule inaccuracy	High	Med	- Divide work into manageable pieces - Overestimate time required	Divide work into discrete tasks. Prioritize which tasks are essential and then work to complete these first.
Learning curve for skills	High	Med	- Ask for help - Study/tutorials - Start tasks early	We will study tutorials in order to understand the basics. We will also pair up so that inexperienced team members can code with more experienced team members.
Communication with team members	Med	Low-Med	- More meetings - Use different communication methods (email, chat, discussion)	Discuss other ways to communicate effectively based on team members' specific needs
Permissions from institution or other applicable sources	Med	Med	- Request permission if necessary	Change project where applicable
Customer	Med	Med	- Reach customer requirements	Seek customer's

satisfaction			- Meet with customer frequently to hear their opinions.	opinion frequently and make changes to enhance customer satisfaction

Test and Documentation Plan

1. Test Plan

Unit Test Strategy

- Connection Test:
    - server connection
        - get status code from http.get function
        - if status code equal to 200
        - return done
    - home page connected
        - get status code from http.get function
        - if status code equal to 302
        - return done
    - sign up page connected
        - get status code from http.get function
        - if status code equal to 200
        - return done
    - profile page connected
        - get status code from http.get function
        - if status code equal to 302
        - return done
    - wishlist page connected
        - get status code from http.get function
        - if status code equal to 302
        - return done
    - trying connected a not exist link
        - get status code from http.get function
        - if status code equal to 404
        - and will be direct to a 404 error page
        - return done
- Form entries:
    - Username
        - *Username is available* - Prevents users from having identical names

        - *Username is not blank* - Make sure user actually has an identifiable name
        - Testing:
            - All elements in username will be tested manually
            - Tests will check if the proper system response to the user is given
            - This will be tested early during the beta release developmental stages
            - Does not need to be tested often

    - Password
        - *Minimum character length* - Strengthens security
        - Testing:
            - The elements in password will be tested manually
            - Tests will check if the proper system response to the user is given
            - Will be tested during the early stages of the beta release development
            - Does not need to be tested often
    - Email
        - *Correct format* - Prevents some typos and makes it easier for database input
        - *Not blank* - Must be valid email for account creation and verification
        - *Email not taken* - An email cannot be tied to more than one account
        - Testing:
            - Elements in email will be tested manually
            - Tests will check to see if the user is given useful feedback for their errors
            - Will be tested during the early stages of beta release development
            - Does not need to be tested often
- Post Listing
    - *Correct format* - Keeps the site clean, makes it easier for users to view a post
    - *Not blank* - No empty listings allowed to prevent flooding of posts
    - *Provides sufficient information* - Allows users to find a way to communicate with each other and to see what the post is actually for
    - Testing:
        - All elements in post listing will be tested manually
        - Tests will check to see if user is given useful feedback for their errors
        - Will be tested during all stages of development
        - Tested often because it is the main focus of the web app
- Wishlist
    - *User added items will appear on their wish-list* - Make sure the wish-list is actually working
        - Will be tested manually by checking to see if the 'wished' item actually shows up on a user's wish-list
        - Testing will be done lightly during the 1.0 developmental stages
    - *If a matching item is posted, send notification* **-** Allows users to be notified that an item they want is available and saves the user time

        - Will be tested manually by adding pseudo listings matching items on a wish-list. Will mainly be checking for an email to be sent to the user who is searching for the item.
        - Testing done during the 1.0 developmental stages
        - Will be tested often because of the connections it has with other elements of the web app.

- addUser
    - User registration web page will store in database
    - adding call addUserbasic function add new user (‘testuser’,0000,null,’qiwen’’wang’,555-5555-5555) should correctly store user: testuser in database, and can be found by getUser function.
    - if found return done, if not return error

- getUser
    - getUser will get user from database by user name
    - after addUser, user testuser, can be found by db.getUser(‘testuser’)
    - if use(testuser) can be found, return done, if not return error

- verifyUser
    - verifyUser is used to verify if the username match the user password
    - call verify(‘testuser’,0000) should return true if the username and password match up
    - After verified user call checkUser function from DB_interface, check if user information can be found.

- addBookbyISBN
    - addBook will add book with book from isbn13 only
    - add a book to testuser by calling function db.addBookbyISBN(‘978478914401’,’testuser’)
    - it should return done if book added and return error if not

- searchBook
    - searchBook can search books in database by term phrases
    - add a test bok first by db.addBookbyISBN(‘978478914401’,’testuser’)
    - create two variables searchBytermA and serchBytermB,
    - call db.search search two different terms for one book
    - assert if both of them getting same book
    - return done if its same, return error if not
    - this test have a render error to be fixed, i haven’t figure out what caused this error,but test passed properly. due to time issue cannot be fixed.

- addBookWish

  - addBookWish can add a book to user’s wish list by ISBN number
  - call function testbook.addBookWish(‘testuser’,978478914401)
  - should add the book in database
  - return done if added, return error if not

- addBookCover
  - add book cover use Node.js module node-isbn
  - covers added by input isbn number, and server will search book cover from google.com and return the link for image thubnail.
  - call function addBookCover(9784789014401,imgpath)
  - imgpath is a variable assigned the to path ../public/images
  - return done if image found, return error if not.

- viewListing
  - return books user have listed, and post on profile page
  - call function db.postBookListing(9784789014401,res) to call database
  - return done if get booklisting, return error if not

- Listing
  - listing can find a user listing
  - call function getListing(id,res)
  - and it will return the Listing for user

- deleteListing
  - delete listing can delete a book you added, and remove it from the view of your profile page.
  - call function db.deleteListing(listid, res,req)
  - and it should return done, if nothing goes wrong.

Integration Test Strategy

- use case simulation test
  - simulate a process of user use on website
  - use case- create listing: user login , list a book, check in profile and log out
    - user login
      - call function loginUser, input user name:cute, and password 000
    - user list book
      - call function addBookByISBN input isbn number 9784789014401
    - check added book

            - simulate user go back to profile page checkout if the book he just listed is there
        - user log out
            - call function req.session.reset()
            - simulate user log out

return done if every step passed, return error if not

System Test Strategy

- **Button functionality**
    - Submit Buttons
        - Required for completed of events
    - Search Button
        - Required to search for items
            - Elements in button functionality will be tested manually
            - Testing to see if the buttons actually do what they were originally required to.
            - Testing to see if the server actually receives the request of each button
            - Testing done during the beta development stages
                - Needs to be tested often to see if the buttons work well with other aspects of the web app.
- **Database**
    - Entry
        - Manual testing will be checking to see if an input is actually inserted into database
            - Will be tested often throughout the release as more features are added
- **Email notification**
    - Item added matches wish list
        - Manual tests will check to see if a user is notified via email that a posting matches an item on their wish-list
        - Will be tested often during the 1.0 developmental stages
    - Verification of account creation
        - Manual tests will check to see if a user is notified via email that their account has been created successfully and that they can verify their account
        - Will be tested lightly during the beta developmental stage

Usability Test Strategy

- **Browser compatibility / Organization**
    - CSS
        - Manually test if everything scales properly depending on window sizes of the accessing device.
        - Will be tested often throughout the developmental stages

        - Mobile
            - Manually test if everything scales properly on mobile devices
            - Will be tested during the 1.0 developmental stage

- **Navigation**
    - Test links
        - Manually test that all links will work by simply clicking them and seeing if the user will be redirected to the correct pages
        - Will be tested often from beta to 1.0 release
- **Error messages**
    - Page does not exist
        - Manually test to see if the correct page is shown if a user inputs or clicks a nonexistent link
        - Will be tested lightly during beta developmental stages
    - Form submit fail
        - Manually test if the correct error message shows up if a user fails to provide sufficient information
        - Will be tested lightly during beta developmental stages
    - Search found no matches
        - Manually test by inputting a search that will result in no results
            - Mainly checking for the correct feedback to be sent to users
        - Will be tested often from beta developmental stages and onward.
- **Easy to use**
    - User friendly
        - Manually test by bringing in different people to act as users to see how easily they navigate the app
        - Will be tested lightly during 1.0 developmental stage
- **Aesthetically pleasing**
    - Manually test by collecting feedback from group members and outside opinions on web app
    - Lightly test during 1.0 developmental stage

- **Feedback time**
    - Testing will be both manual and automated
        - Will navigate the site to see how quickly it loads
    - Test the run time of the database queries and will optimize if necessary
    - Will be tested often during 1.0 developmental stage

Adequacy of test strategy

- Github Issue Tracker
    - Has version control so we can check back at states of the project

        - Allows teammates to collaborate together easily
        - Can check to see changes other teammates made

Instruction for new test developers:

- platform structure
    - Using Mocha as main test framework
    - Due to a version issue, install mocha as global before running test
        - sudo npm install mocha -g
    - all test file stored at bookapp/test, adding new test can easily create .js file under /test

- modules used
    - supertest
    - chai
    - should
    - expect
- latest test result
    - latest test result can be find
- procedure of test
    - connection test will only pass when server is running on localhost:3000
    - run npm start and open another terminal run mocha
    - API for mocha, should expect can be find in
        - http://mochajs.org/
        - https://github.com/Automattic/expect.js
        - https://github.com/tj/should.js

2. Documentation Plan

User Guide in Web App

- How to Login
    - where to login with username and password
- New User Registration Steps
    - how to create account with required fields (name, email, institution)
- Update User Profile
    - make changes in profile form
- Create/Edit/Delete Posting
    - makes changes in a posting form
- Add items to WishList
    - make changes in the wishlist form

- Search & View Postings
    - how to search & view postings

Admin Guide

- Overview of System
    - Classes/Components
- Installation/Compile
    - Instructions to compile code in Github

Help Menu in Web App

- Contact Us
    - contains email to contact or a little message box that will send email
- FAQ
    - for the most common issues