
Whitman Books Online Documentation

Whitman CS-300 SP2018

May 14, 2018

Table of Contents:

1	Introduction	1
2	Login Page	3
3	Navigating the Market	5
4	Navigating your Profile	7
5	Buying a Book	9
6	Selling a Book	11
7	Front End Description	15
8	API Guide	17
9	API Autodoc	23
10	Analysis Document	35
11	Design Document	43
12	Ethical Concerns	59
13	Helpful Links	61
14	Meeting Notes Workflow	63
15	Requirements Document	65
16	Meeting Notes	67
17	Privacy Policy	79
	Python Module Index	81

CHAPTER 1

Introduction

Whitman Books Online is a platform for buying and selling used textbooks. At every transition between semesters, there's a rush to offload textbooks from the past semester and buy new ones for the upcoming semester. Currently, this is done through either word of mouth or, more commonly, email listservs within the whitman network. Unfortunately, these are quite inefficient and unreliable systems. Additionally, because buying and selling textbooks happens pretty much only during semester *transitions*, the event crowds out other items being sold more evenly throughout the school year. Whitman Books Online is a solution to this problem. This unique platform *specifically* for textbooks aims to diminish the traffic on email listservs during the proverbial textbook rush and create an efficient and pleasant experience for both buyer and seller.

CHAPTER 2

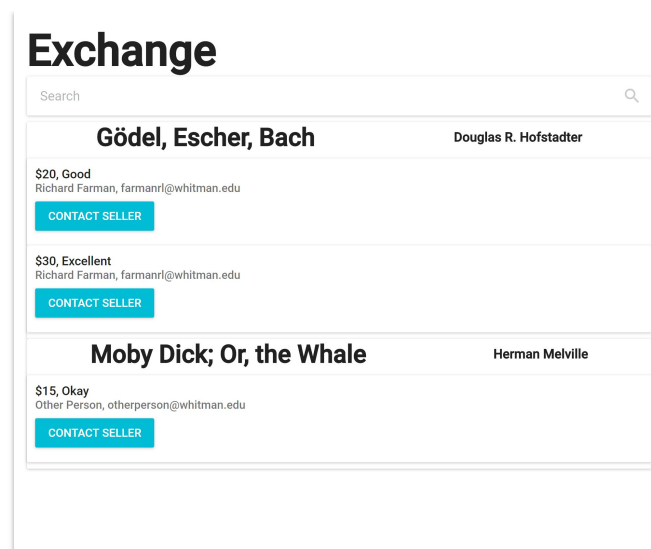
Login Page



- Clicking the login button opens a prompt
- This service is exclusively for Whitman students and requires you to use a Whitman email account
- Your user profile is created automatically from your Whitman account

CHAPTER 3

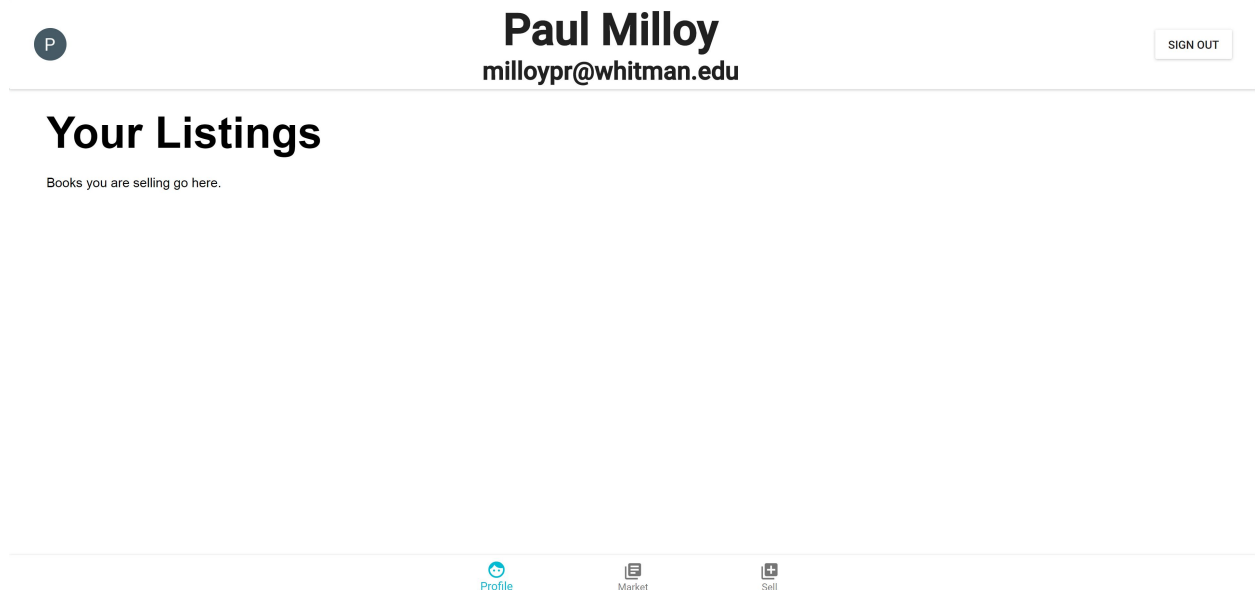
Navigating the Market



- The Market tab displays all listings posted of books for sale
- Each listing displays the title of the book, the author of the book, the price and condition of the book being sold, the name and email of the Whitman student who posted the listing and a Contact Seller button to contact that student and ask to purchase their book
- Listings can be navigated using the Search tab (*Currently not functioning, Advanced Search to be added)

CHAPTER 4

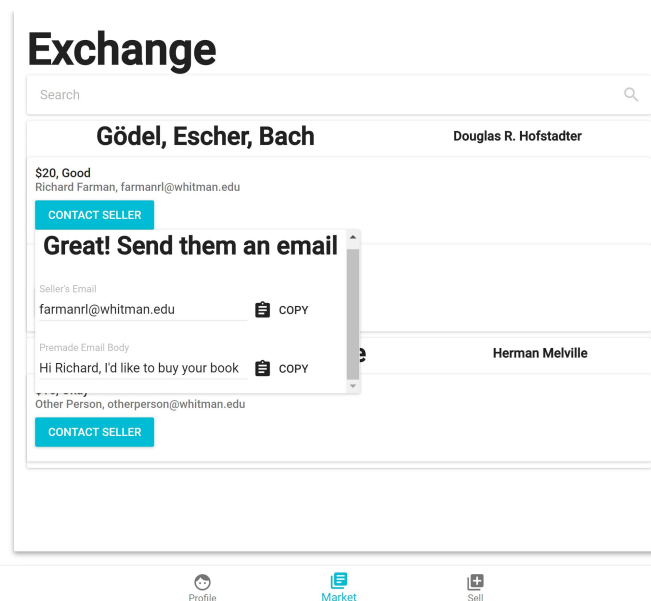
Navigating your Profile



- Click on the Profile tab at the bottom of the screen to navigate to your profile
- Your Profile page will display all of the listings you have posted to the market (*Currently books you fill out to sell will not be added under your listings)

CHAPTER 5

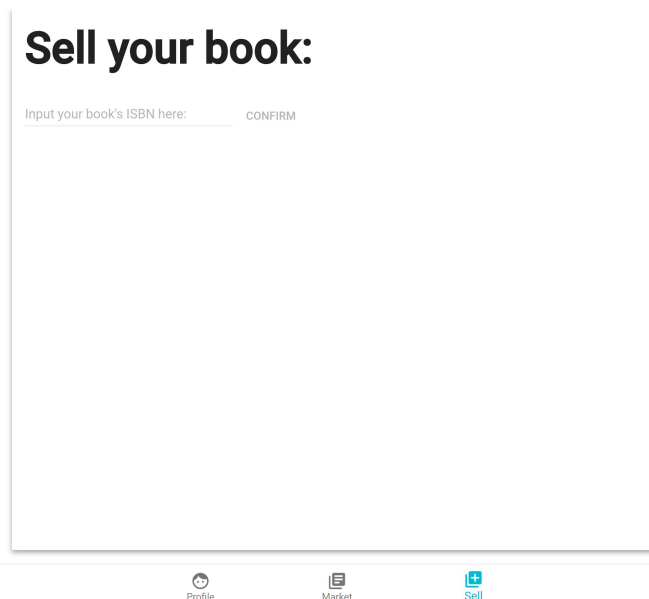
Buying a Book



- Click on Contact Seller
- A drop-down menu will appear that includes both the seller's email and a pre-made email message asking to purchase the book they've listed.
- Clicking the Copy button next to the email or pre-made message to copy either
- In your email, paste the message into the body of the email and send the email to the email address of the seller
- Wait for a response from the seller to coordinate buying their book!

CHAPTER 6

Selling a Book



Sell your book:

Input your book's ISBN here: CONFIRM

Profile Market Sell

- Click on the Sell tab at the bottom of the screen
- A prompt appears requesting an ISBN for the book you are trying to sell. This can typically be found on the back cover of the book

Sell your book:

Input your book's ISBN here:

1111111111

[CONFIRM](#)



BookData Dummy Title for Series Checkin
L. G. Minasean



Publisher: Psychology 2.0 Books

Published Date: 2001-10 This famous work is a textbook that emphasizes the conceptual and historical continuity of analytic function theory. The second volume broadens from a textbook to a textbook-treatise, covering the "canonical" topics (including elliptic functions, entire and meromorphic functions, as well as conformal mapping, etc.) and other topics nearer the expanding frontier of analytic function theory. In the latter category are the chapters on majorization and on functions holomorphic in a half-plane.

- Adding the ISBN will bring up a publisher description of the book you are trying to sell. Confirm that the description matches the book you are selling before continuing. If it does not, double-check that you have added the correct ISBN

Sell your book:

Input your book's ISBN here:

1111111111

[CONFIRM](#)



BookData Dummy Title for Series Checkin
L. G. Minasean



Publisher: Psychology 2.0 Books

Published Date: 2001-10 This famous work is a textbook that emphasizes the conceptual and historical continuity of analytic function theory. The second volume broadens from a textbook to a textbook-treatise, covering the "canonical" topics (including elliptic functions, entire and meromorphic functions, as well as conformal mapping, etc.) and other topics nearer the expanding frontier of analytic function theory. In the latter category are the chapters on majorization and on functions holomorphic in a half-plane.

Bad

Ehh

Good

New

[SUBMIT](#)



Profile



Exchange



Sell

- Choose from the Condition drop down menu the option that best describes the condition of your book. Be accurate and honest in choosing the most apt description.

Sell your book:

Input your book's ISBN here:

1111111111

CONFIRM



BookData Dummy Title for Series Checkin
L. G. Minasean



Publisher: Psychology 2.0 Books

Published Date: 2001-10 This famous work is a textbook that emphasizes the conceptual and historical continuity of analytic function theory. The second volume broadens from a textbook to a textbook-treatise, covering the "canonical" topics (including elliptic functions, entire and meromorphic functions, as well as conformal mapping, etc.) and other topics nearer the expanding frontier of analytic function theory. In the latter category are the chapters on majorization and on functions holomorphic in a half-plane.

Condition

Poor

Input your desired price:

\$20

SUBMIT



Profile



Market



Sell

- Finally, add the Price you are wishing to sell the book for. Be sure to only add numbers into this prompt in order to have a valid price.
- After clicking submit, a listing for your book will appear in the market.

CHAPTER 7

Front End Description

Written by Richard Farman

Edited and converted to reStructuredText by Jeremy Davis

Updated 2018-05-13

7.1 Overview

The front end of the app is built with a modern web development stack. We use the *Model/View/Controller* paradigm to separate responsibilities and allow our app to have a dynamic and interchangeable structure.

Our components are essentially JavaScript classes that render and return HTML/CSS. We connect these classes to various JavaScript functions that use Redux to dispatch actions, like getting a list of books from the database. These dispatched actions then modify our data model, which is then displayed to the user by React.

The front end allows us to make requests and display responses from a database. So using our web client, any Whitman College student can login to the app, submit a book with a listing attached to the server, and see that listing propagated to *all other clients* using the app. An exchange shows all the listings that have been added by other users and allows you to contact sellers with an offer for their book. Your profile shows all of your personal listings for easy management.

Our front end is designed with a simple, cohesive, and modern experience in mind for both clients and developers alike. With these technologies, we are best prepared to take on the challenges of project scope and scale as we continue building our project.

7.2 Technologies Used

HTML (HyperText Markup Language) is the standard markup language used to create websites. HTML is rendered into *Document Object Model* (DOM), which is read by the browser to construct a web page. CSS describes the presentation of HTML content. Putting these two together, we can start building for the web.

Javascript is a programming language that can add interaction and extend the functionality of websites. JavaScript runs on the client side of the web, which can be used to control how the web page reacts on the occurrence of an event. JavaScript is an easy to learn and powerful scripting language that is widely used for controlling web page behaviour.

We use the **React.js** library to create declarative and composable user interfaces that form the view for our clients. Using Redux.js on top of React allows us to add interactive actions to our app, serving to form the model and controller.

Redux is a *predictable state container* for JavaScript apps. It helps you write applications that behave consistently, run in different environments (client, server, and native), and are easy to test. On top of that, it provides a great developer experience, such as live code editing combined with a time traveling debugger.

The whole state of your app is stored in an object tree inside a single store. The only way to change the state tree is to emit an action, an object describing what happened. To specify how the actions transform the state tree, you write pure reducers.

Redux Thunk middleware allows you to write action creators that return a function instead of an action. The thunk can be used to delay the dispatch of an action, or to dispatch only if a certain condition is met. The inner function receives the store methods *dispatch* and *getState* as parameters.

Reselect is a simple library for creating memorized, composable selector functions. *Selectors* can compute derived data, allowing Redux to store the minimal possible state. Selectors are efficient. A selector is not recomputed unless one of its arguments change. Selectors are composable. They can be used as input to other selectors.

Written by Sean Miller.

Converted to reStructuredText by Tyler Phillips.

Updated 2018-04-11.

Run the API: `python3 app.py`

All files must be in the same directory.

Reset the database: `rm data.db`

The API will run on `http://127.0.0.1:5000/some_endpoint`. Each time the API runs, it will generate a new `data.db`.

8.1 1. Adding Books to the Database

To add a book to the database, send a POST request to:

`/book/isbn_of_book`

The body of the request should be the JSON body return from a Node.ISBN query. All JSON parsing of the body will happen on the back end.

8.2 2. Adding a User to the Database

To add a user to the database, send a POST request to:

`/user/google_token_of_user`

The body of the request should look like this:

```
{
  "imageUrl": "http://hello.com",
  "name": "Big Will",
  "email": "ashleygw@whitman.edu",
  "givenName": "Thick Ricky",
  "familyName": "Slick Nicky"
}
```

8.3 3. Adding a Listing to the Database

To add a listing to the database, send a POST request to:

```
/listing/isbn_of_book
```

The body of the request should look like this:

```
{
  "price": 27.99,
  "condition": "good",
  "google_tok": "1A",
  "status": "selling"
}
```

`google_tok` must be the Google token of the user making the listing. At the moment, the only `condition`s that I expect are "new", "good", "ehh", and "bad" so that Python string comparisons can uphold that order (In this case, we are using first-character comparison.). For the MVP, `status` should always be "selling". Price must be a float, *not* a string. If `google_tok` does not match a user in the database, or if the `isbn` in the endpoint does not match a book in the database, the request will fail.

8.4 4. User -> Listing -> Book

This series of steps will be executed when a user wants to see their own listings on their user profile.

First, a GET request is sent to:

```
/book/google_token_of_user
```

The body will look like this:

```
{
  "givenName": "Thick Ricky",
  "familyName": "Slick Nicky",
  "name": "Big Will",
  "listings": [
    1,
    2
  ],
  "google_tok": "1A",
  "email": "ashleygw@whitman.edu",
  "imageUrl": "http://hello.com"
}
```

`listings` is a list of all `listing_ids` that correspond to each of the user's listings. The other information can be used to load the user's profile, including the image tied to their Google account, name, email address, etc.

Second, use `listings` to make the following GET request:

```
/listing/listing_ids,seperated,by,commas
```

The response will look like this:

```
{
  "listings": [
    {
      "status": "selling",
      "listing_id": 1,
      "timestamp": "2018-04-16 19:57:26.674665",
      "condition": "ehh",
      "price": 24.99
    },
    {
      "status": "selling",
      "listing_id": 2,
      "timestamp": "2018-04-16 19:57:35.568820",
      "condition": "good",
      "price": 27.99
    }
  ],
  "isbn": [
    1
  ]
}
```

`listings` is all of the listings tied to the user. `isbn` is the list of ISBNs corresponding to the books paired with these listings. Notice that, in this case, both listings are for the same book, which has an `isbn` of 1. Use the `listings` data to edit the user's home page.

Third, use `isbn` for the following GET request:

```
/book/isbn,seperated,by,commas
```

The response will look like this:

```
{
  "books": [
    {
      "subtitle": "The Musical",
      "listing_ids": [
        1,
        2
      ],
      "canonicalVolumeLink": "TRIPLElol",
      "title": "Moby Dick",
      "isbn": 1,
      "thumbnail": "http://blahBLAHblah",
      "smallThumbnail": "http://blahblah",
      "authors": "Will Smith, Edgar Wright",
      "publishedDate": "1975",
      "categories": "Artificial Intelligence, Computer Science",
      "infoLink": "doublelol",
    }
  ]
}
```

(continues on next page)

(continued from previous page)

```
        "previewLink": "lolwhatisthis"
      }
    ]
  }
```

This information will be used to construct the book objects. Notice that users, listings, and books will need to be matched thorough `listing_id`. This will be cumbersome for the front end, but, hopefully, it will be more streamlined in the final product.

8.5 5. Book -> Listing -> User

This pipeline will be used when a user is looking to buy a used textbook from our site.

First, the user will search by author, title, subtitle, category, or date published. The response the user types into the search bar will be used in a GET request to the following endpoint.

Important: The user's search query must have spaces replaced by underscores ("_") and must be converted to all lower case before being sent to the back end through the endpoint:

```
/booklist/search_value
```

The response will look like this (for `/booklist/will_smith`):

```
{
  "books": [
    {
      "subtitle": "The Musical",
      "listing_ids": [
        1,
        2
      ],
      "canonicalVolumeLink": "TRIPLElol",
      "title": "Moby Dick",
      "isbn": 1,
      "thumbnail": "http://blahBLAHblah",
      "smallThumbnail": "http://blahblah",
      "authors": "Will Smith, Edgar Wright",
      "publishedDate": "1975",
      "categories": "Artificial Intelligence, Computer Science",
      "infoLink": "doublelol",
      "previewLink": "lolwhatisthis"
    }
  ]
}
```

This data can be used to construct the book objects.

Second, use `listing_ids` to perform the following GET request.

Important: This is where condition and price ordering comes into play. To retrieve listings without any ordering:

```
/listings/listing,ids,separated,by,commas+
```

To retrieve listings ordering by lowest -> highest price:

```
/listings/listing,ids,separated,by,commas+price
```

To retrieve listings ordering by best -> worst condition:

```
/listings/listing,ids,separated,by,commas+condition
```

The response will look like this:

```
{
  "listings": [
    {
      "condition": "ehh",
      "status": "selling",
      "listing_id": 1,
      "google_tok": "1A",
      "price": 24.99,
      "timestamp": "2018-04-16 19:57:26.674665"
    },
    {
      "condition": "good",
      "status": "selling",
      "listing_id": 2,
      "google_tok": "1A",
      "price": 27.99,
      "timestamp": "2018-04-16 19:57:35.568820"
    }
  ],
  "google_tokens": [
    "1A"
  ]
}
```

Use this data to construct the listing objects.

Third, use google_tokens for a GET request to the following endpoint:

```
/userlist/google,tokens,separated,by,commas
```

The response will look like this:

```
{
  "users": [
    {
      "name": "Big Will",
      "listing_ids": [
        1,
        2
      ],
      "givenName": "Thick Ricky",
      "google_tok": "1A",
      "imageUrl": "http://hello.com",
      "familyName": "Slick Nicky",
      "email": "ashleygw@whitman.edu"
    }
  ]
}
```

Again, the listing_id will be used to match books to listings to users.

8.6 6. Loading the Home Page

When a user first enters Whitman Books Online, they enter a home page with listings ordered from most to least recent. To get most-recent listings:

```
/listings/home
```

This will return a JSON object similar to this:

```
{
  "listings": [
    {
      "condition": "good",
      "price": 27.99,
      "listing_id": 2,
      "status": "selling",
      "timestamp": "2018-04-16 19:57:35.568820"
    },
    {
      "condition": "ehh",
      "price": 24.99,
      "listing_id": 1,
      "status": "selling",
      "timestamp": "2018-04-16 19:57:26.674665"
    }
  ],
  "google_tokens": [
    "1A"
  ],
  "isbns": [
    1
  ]
}
```

Notice that the listings are in order from most to least recent. From here, the user and book objects will be loaded *separately*. Use the data from those queries to construct the full home page.

8.7 7. Deleting Objects

To delete a user, send a DELETE request to:

```
/user/google_token_of_user
```

To delete a listing, send a DELETE request to:

```
/listing/listing_id
```

I don't foresee us wanting to remove book objects from the database (at least for the MVP), but it works as you'd expect: Send a DELETE request to:

```
/book/isbn
```

9.1 book

class `api.book.Book`

Bases: `flask_restful.Resource`

The Book object handles API requests such as Get/Post/Delete.

none.

classmethod `as_view(name, *class_args, **class_kwargs)`

Converts the class into an actual view function that can be used with the routing system. Internally this generates a function on the fly which will instantiate the `View` on each request and call the `dispatch_request()` method on it.

The arguments passed to `as_view()` are forwarded to the constructor of the class.

decorators = `()`

delete (`isbn`)

Deletes a book from the database.

Parameters `isbn` (`str`) – The isbn of the book being deleted.

Returns What happened with the delete call.

Return type message

dispatch_request (`*args, **kwargs`)

Subclasses have to override this method to implement the actual view function code. This method is called with all the arguments from the URL rule.

get (`isbn`)

Get request, looking for all books based on isbn.

Parameters `isbn` (`str[]`) – A list of isbn to query with.

Returns A list of jsonified books.

Return type json[]

method_decorators = []

methods = set(['POST', 'DELETE', 'GET'])

parser = <flask_restful.reqparse.RequestParser object>

post (*isbn*)

Posts a book to the database.

Parameters **isbn** (*str*) – The isbn of the book being posted.

Returns What happened with the post call.

Return type message

provide_automatic_options = None

representations = None

class api.book.BookList

Bases: flask_restful.Resource

The BookList object handles the entire list of books in the database.

none.

classmethod **as_view** (*name*, **class_args*, ***class_kwargs*)

Converts the class into an actual view function that can be used with the routing system. Internally this generates a function on the fly which will instantiate the View on each request and call the `dispatch_request()` method on it.

The arguments passed to `as_view()` are forwarded to the constructor of the class.

decorators = ()

dispatch_request (**args*, ***kwargs*)

Subclasses have to override this method to implement the actual view function code. This method is called with all the arguments from the URL rule.

get (*search*)

Gets a list of all books in database that match a search.

Parameters **search** (*str*) – The string to search with.

Returns A list of jsonified books that match the search result.

Return type json[]

method_decorators = []

methods = set(['GET'])

provide_automatic_options = None

representations = None

class api.book.BookModel (*title*, *subtitle*, *authors*, *isbn*, *categories*, *publishedDate*, *smallThumbnail*, *thumbnail*, *previewLink*, *infoLink*, *canonicalVolumeLink*)

Bases: sqlalchemy.ext.declarative.api.Model

The BookModel object stores information about the book, as well as the listing objects that are associated with it.

title

string – The title of the book.

subtitle*string* – The subtitle of the book.**authors***string* – The author/authors of the book.**isbn***int* – The isbn number for the book.**categories***string* – The categorise of the book.**publishedDate***string* – The published date of the book.**smallThumbnail***string* – A string referencing the small thumbnail of the book.**thumbnail***string* – A string referencing the thumbnail of the book.**previewLink***string* – A link to preview the book.**infoLink***string* – An info link for the book.**canonicalVolumeLink***string* – A canonical volume link for the book. listings (Listing): The current listings of the book.**authors**

authors = relationship(“AuthorModel”, secondary=association_table1, back_populates=’works’)

bare_json()

Returns a jsonified book item, including a list of listing ids.

Parameters none. –**Returns** A json item representing a book.**Return type** json**book_json_w_listings()****Returns** a jsonified book item, including a list of jsonified listings.**Parameters** none. –**Returns** A json item representing a book.**Return type** json**book_json_wo_listings()**

Returns a jsonified book item, not including listings.

Parameters none. –**Returns** A json item representing a book.**Return type** json**canonicalVolumeLink****categories**

delete_from_db()

Deletes the book from the database.

Parameters *none.* –

Returns *none.*

classmethod find_by_isbn(isbn)

Finds a book by isbn number.

Parameters **isbn** (*str*) – The isbn number we are searching for.

Returns The book which matches the isbn.

Return type *Book*

get_listings()

Get a list of book listing jsons.

Parameters *none.* –

Returns A list of jsonified listings.

Return type *json[]*

infoLink

isbn

listings

metadata = **MetaData(bind=None)**

previewLink

publishedDate

query_class

alias of `flask_sqlalchemy.BaseQuery`

save_to_db()

Saves the book to the database.

Parameters *none.* –

Returns A json item representing the book.

Return type *json*

smallThumbnail

subtitle

thumbnail

title

9.2 listing

class `api.listing.Listing`

Bases: `flask_restful.Resource`

The Listing object handles API requests such as Get/Post/Delete/Put.

none.

classmethod as_view (*name*, **class_args*, ***class_kwargs*)

Converts the class into an actual view function that can be used with the routing system. Internally this generates a function on the fly which will instantiate the `View` on each request and call the `dispatch_request()` method on it.

The arguments passed to `as_view()` are forwarded to the constructor of the class.

decorators = ()

delete (*ids*)

Deletes a listing from the database.

Parameters *ids* (*str*) – The id of the listing being deleted.

Returns What happened with the delete call.

Return type message

dispatch_request (**args*, ***kwargs*)

Subclasses have to override this method to implement the actual view function code. This method is called with all the arguments from the URL rule.

get (*ids*)

Get request, looking for all listings matching an id in ids.

Parameters *ids* (*str*[]) – A list of ids to query with.

Returns A list of jsonified listings.

Return type json[]

method_decorators = []

methods = set(['PUT', 'POST', 'DELETE', 'GET'])

parser = <flask_restful.reqparse.RequestParser object>

post (*ids*)

Posts a listing to the database.

Parameters *ids* (*str*) – The listing id of the listing being posted.

Returns What happened with the post call.

Return type message

provide_automatic_options = None

put (*listing_id*, *price*, *condition*, *isbn*, *google_tok*, *status*)

Either posts listing to database, or updates it.

Parameters

- **listing_id** (*int*) – An id to represent the listing, generated by the table.
- **price** (*float*) – The price of the listing.
- **condition** (*str*) – The condition of the listing.
- **isbn** (*int*) – The isbn of the listing.
- **google_tok** (*str*) – The google token of the user who made the posting.
- **status** (*str*) – The status of the listing.

Returns A jsonified listing object representing what was put.

Return type json

representations = None

class api.listing.**ListingModel** (*price, condition, isbn, google_tok, status*)

Bases: sqlalchemy.ext.declarative.api.Model

The ListingModel object stores information about the listing, as well as the book and user objects associated with it.

listing_id

int – An id to represent the listing, generated by the table.

price

float – The price of the listing.

condition

string – The condition of the listing.

isbn

int – The isbn of the listing.

book

BookModel – The book being represented by the listing.

google_tok

string – The google token of the user who made the posting.

user

UserModel – The user who made the posting.

status

string – The status of the listing.

timestamp

int – The time the listing was posted.

bare_json ()

Returns a json object representing the listing.

Parameters *none.* –

Returns A jsonified listing.

Return type json

book

bu_bare_json ()

Returns a json object representing the listing. Used when going from books to users.

Parameters *none.* –

Returns A jsonified listing.

Return type json

condition

delete_from_db ()

deletes the listing to the database.

Parameters *none.* –

Returns none.

classmethod find_by_isbn (isbn)

Finds all listings matching an isbn.

Parameters `isbn` (*int*) – The isbn to search with.

Returns A list of listings.

Return type *ListingModel*[]

classmethod `find_by_listing_id` (*listing_id*)

Finds all listings matching a listing id.

Parameters `listing_id` (*int*) – The listing id to search for.

Returns A list of listings.

Return type *ListingModel*[]

`google_tok`

`isbn`

`listing_id`

`listing_json_w_book` ()

Returns the listing jsonified, with a reference to the book being represented.

Parameters `none`. –

Returns A jsonified listing.

Return type json

`listing_json_w_book_and_user` ()

Returns the listing jsonified, with a reference to the book being represented and the user who posted it.

Parameters `none`. –

Returns A jsonified listing.

Return type json

`listing_json_w_user` ()

Returns the listing jsonified, with a reference to the user who posted.

Parameters `none`. –

Returns A jsonified listing.

Return type json

`metadata` = `MetaData` (`bind=None`)

`price`

`query_class`

alias of flask_sqlalchemy.BaseQuery

`save_to_db` ()

Saves the listing to the database.

Parameters `none`. –

Returns none.

`status`

`timestamp`

`user`

```
class api.listing.allListings
```

```
    Bases: flask_restful.Resource
```

The allListings object handles the entire list of listings in the database.

none.

```
classmethod as_view(name, *class_args, **class_kwargs)
```

Converts the class into an actual view function that can be used with the routing system. Internally this generates a function on the fly which will instantiate the `View` on each request and call the `dispatch_request()` method on it.

The arguments passed to `as_view()` are forwarded to the constructor of the class.

```
decorators = ()
```

```
dispatch_request(*args, **kwargs)
```

Subclasses have to override this method to implement the actual view function code. This method is called with all the arguments from the URL rule.

```
get(search)
```

Gets a list of all listings in database that match a search.

Parameters `search` (`str[]`) – A list of search terms defining what to search with.

Returns A list of jsonified listings that match the search result.

Return type `json[]`

```
method_decorators = []
```

```
methods = set(['GET'])
```

```
provide_automatic_options = None
```

```
representations = None
```

9.3 user

```
class api.user.User
```

```
    Bases: flask_restful.Resource
```

```
classmethod as_view(name, *class_args, **class_kwargs)
```

Converts the class into an actual view function that can be used with the routing system. Internally this generates a function on the fly which will instantiate the `View` on each request and call the `dispatch_request()` method on it.

The arguments passed to `as_view()` are forwarded to the constructor of the class.

```
decorators = ()
```

```
delete(google_tok)
```

Deletes a user from the database.

Parameters `google_tok` (`str`) – The google token of the book being deleted.

Returns What happened with the delete call.

Return type `message`

```
dispatch_request(*args, **kwargs)
```

Subclasses have to override this method to implement the actual view function code. This method is called with all the arguments from the URL rule.

```

get (google_tok)
    Get request, looking for all users with google token.

    Parameters google_tok (str[]) – A list of google tokens to query with.

    Returns A list of jsonified users.

    Return type json[]

method_decorators = []

methods = set(['POST', 'DELETE', 'GET'])

parser = <flask_restful.reqparse.RequestParser object>

post (google_tok)
    Posts a user to the database.

    Parameters google_tok (str) – The google token of the user being posted.

    Returns What happened with the post call.

    Return type message

provide_automatic_options = None

representations = None

class api.user.UserList
    Bases: flask_restful.Resource

    The UserList object handles the entire list of users in the database.

    none.

    classmethod as_view (name, *class_args, **class_kwargs)
        Converts the class into an actual view function that can be used with the routing system. Internally
        this generates a function on the fly which will instantiate the View on each request and call the
        dispatch_request() method on it.

        The arguments passed to as_view() are forwarded to the constructor of the class.

    decorators = ()

    dispatch_request (*args, **kwargs)
        Subclasses have to override this method to implement the actual view function code. This method is called
        with all the arguments from the URL rule.

    get (tokens)
        Gets a list of all users in database that match any token from a list of tokens..

        Parameters tokens (str[]) – A list of tokens to query with.

        Returns A list of jsonified users that match the tokens.

        Return type json[]

    method_decorators = []

    methods = set(['GET'])

    provide_automatic_options = None

    representations = None

class api.user.UserModel (google_tok, imageURL, email, name, givenName, familyName)
    Bases: sqlalchemy.ext.declarative.api.Model

```

The `UserModel` object stores information about the user, as well as the listing objects that are associated with it.

google_tok

string – The google token for the user.

imageURL

string – The URL referencing the user's image.

email

string – The user's email.

name

string – The user's first name.

givenName

string – The user's given name.

familyName

string – The user's last name.

listings

ListingModel[] – All listings posted by the user,

bare_json()

Returns a jsonified user item, with a list of listing ids.

Parameters `none`. –

Returns A json item representing the user.

Return type `json`

delete_from_db()

Deletes the user from the database.

Parameters `none`. –

Returns `none`.

email**familyName****classmethod find_by_email(email)**

Finds a user by email.

Parameters **email** (*str*) – The email of the user we're searching for.

Returns The user who matches the email.

Return type *UserModel*

classmethod find_by_google_tok(google_tok)

Finds a user by google token.

Parameters **google_tok** (*str*) – The google token of the user we're looking for.

Returns The user who matches the google token..

Return type *UserModel*

get_listings()

Get a list of book listing jsons posted by the user.

Parameters `none`. –

Returns A list of jsonified listings.

Return type json[]

givenName

google_tok

imageURL

listings

metadata = **MetaData**(bind=None)

name

query_class

alias of flask_sqlalchemy.BaseQuery

save_to_db()

Saves the user to the database.

Parameters none. –

Returns A json item representing the user.

Return type json

user_json_w_listings()

Returns a jsonified user item, with a list of jsonified listings.

Parameters none. –

Returns A json item representing the user.

Return type json

user_json_wo_listings()

Returns a jsonified user item.

Parameters none. –

Returns A json item representing the user.

Return type json

Due Date: Wednesday, 2/21/2018

10.1 Risk Factors

10.1.1 Time

Because this is an academic course, we are constrained to one semester to develop this application. However, by taking the following into account, we believe we can complete the project within that time frame:

1. **Scope of the project:** We chose this project based on a preliminary estimate that the scope of this project matches the time frame we have available.
2. **Time management:** We have developed a timetable with the steps we need to complete in order to finish this project by the end of the semester. We have also devised a system for managing smaller tasks using the tickets function of GitLab.
3. **Ample work time:** Our weekly lab provides us with a large block of time to work on the project as a group, and each team will also meet outside of class twice a week.

10.1.2 Technical Feasibility

One technical factor we are concerned about is the system uptime and performance (including network performance) of our application. Since this is meant to be a public-facing application, we need to achieve a certain level of performance for the application to be usable. We are currently researching and testing hosted and self-hosted options for mitigating this concern. Another potential concern is the interface between the front and back ends. We plan to mitigate this by using a Flask HTTP REST API and by thoroughly documenting this API so that both front- and back-end teams are clear on how the interface is supposed to work.

10.1.3 Personnel Expertise

We are all students, not expert programmers. Our team includes a range of experience levels with the various technologies and techniques we will be using, and no one team member is experienced with all of them. However, those with experience with a particular technology will teach others how to use it, and everyone will learn things as they go. Collectively, we believe our team has the expertise to complete this project.

10.1.4 College Administration

There has been some concern that the Whitman Bookstore would object to our project because it could take business away from them. Our “Ethical Concerns” document analyzes the ethical concerns of that. From a risk-management perspective, we have concluded that the College would have no authority to shut down our project. Therefore, we are not concerned about this risk factor.

10.2 Resources

10.2.1 Front-End

Languages, Libraries, Frameworks, Tools

- JavaScript
- HTML/CSS
- React
- Bootstrap
- Google’s Material-UI
- GitLab

Reasoning

The front-end software will be written in JavaScript, CSS, and HTML. Our UI will be based on React, which is a JavaScript library for web UIs. We may use some pre-made React components from Bootstrap and/or Material UI.

All of these resources were chosen because they are industry standards and because many modules have been developed for them which we can draw from. The JavaScript/CSS/HTML combination is the clear industry standard for coding webapps. React is a widely-used library for developing front-ends, as evidenced by the fact that Facebook, Instagram, Snapchat, Discord, and many other prominent apps use React for their front-end. . We determined that React would be relatively easy to learn and would enable us to do everything we want to do. Finally, Bootstrap and Material UI are both UI libraries that utilize React. They are both widely used, and the decision between the two would be a purely stylistic decision. We may use one or the other, or we may use some components from each.

10.2.2 Back-End

Languages, Libraries, Frameworks, Tools

- Python
- Flask
- MySQL
- SQLAlchemy

- Gitlab

Reasoning

The backend software will be a daemon written in Python, exposing an API to the frontend using a Flask HTTP REST API. This daemon will communicate with a MySQL database through SQLAlchemy, a Python module that will allow us to represent relational structures as objects to better fit Python's programming paradigm. As for operations of the server, we will likely be leveraging AWS for hosting, although we have not finalized that decision.

The benefits of Python are its simplicity and our team's pre-existing experience with it. While it is a less-performant language than some others, its use in backend webapps (Flask, Django) indicates that this is not too significant a cost. Flask was also chosen because of its simplicity and because it has excellent documentation.

We chose MySQL as our database over SQLite for performance, at the cost of more complex deployment (which is something we determined our team will be able to handle). To ease the process of learning how to work with relational databases, we decided to leverage the SQLAlchemy mapper to make our code more or less RDBMS-agnostic and represent our back-end constructs as traditional objects and let SQLAlchemy deal with the lower-level work of how to represent them relationally.

10.2.3 Documentation

Languages, Libraries, Frameworks, Tools

- Markdown
- Gitlab Wiki
- Google Software Suite
- Draw.io
- Sphinx
- GitLab

Reasoning

Meeting Notes: All full-team and sub-team meetings are recorded and hosted through the GitLab project's wiki. The text itself is written in standard Markdown and is divided up between different folders for each team. On the homepage in the root directory, there's a table of contents with links to each day's meeting notes for easy navigation. To make this wiki printable so that we can include it in our documentation binder, a Python script is being written that will automatically concatenate all the pages of the wiki into a single Markdown file that is then converted to a PDF using Pandoc.

Code: We will be using Sphinx to help document our code. Sphinx is a tool that formats documentation and automates some of the documentation process by reading comments and docstrings in the source code. We chose Sphinx because we like the professional format of the documentation it produces and because it works with all of the languages we will be using on this project.

10.3 Project logistics

10.3.1 Projections

Cost

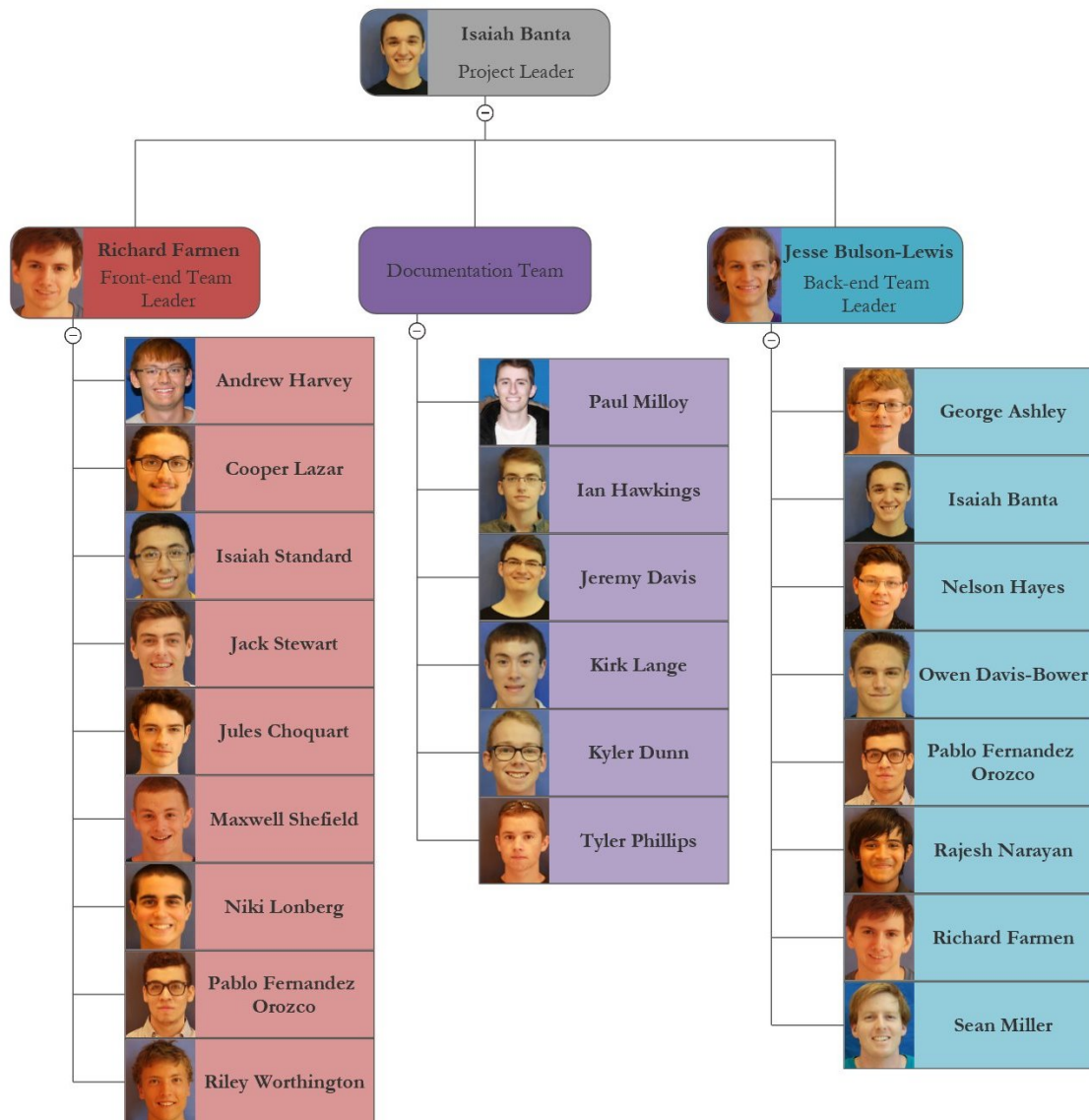
Almost all of the resources we are using for this project are either available for free online or provided by the College. Because this is an academic exercise, there will be no labor costs. The only thing we anticipate paying for is a domain name for our website.

Timeline

Currently, we estimate roughly six weeks (not including spring break) to push our MVP. We plan to delegate smaller, easier tasks over the two weeks we lose to spring break, such as fleshing out comments. We are currently aiming for our completed project to be done by May 1st. Giving us ample amount of time to add any additional features, debug, and do some post-delivery maintenance.

10.3.2 Team Organization

Organization Chart



Note: Since this Org Chart's Creation *Jeremy Davis* has been appointed Documentation Leader

Communication

Our team will communicate with each other using Slack, which is an online messaging service. We chose Slack because it allows each team member a large amount of choice in what things they want to be notified about and

because it allows us to easily create separate conversations for each sub-team and for the team as a whole.

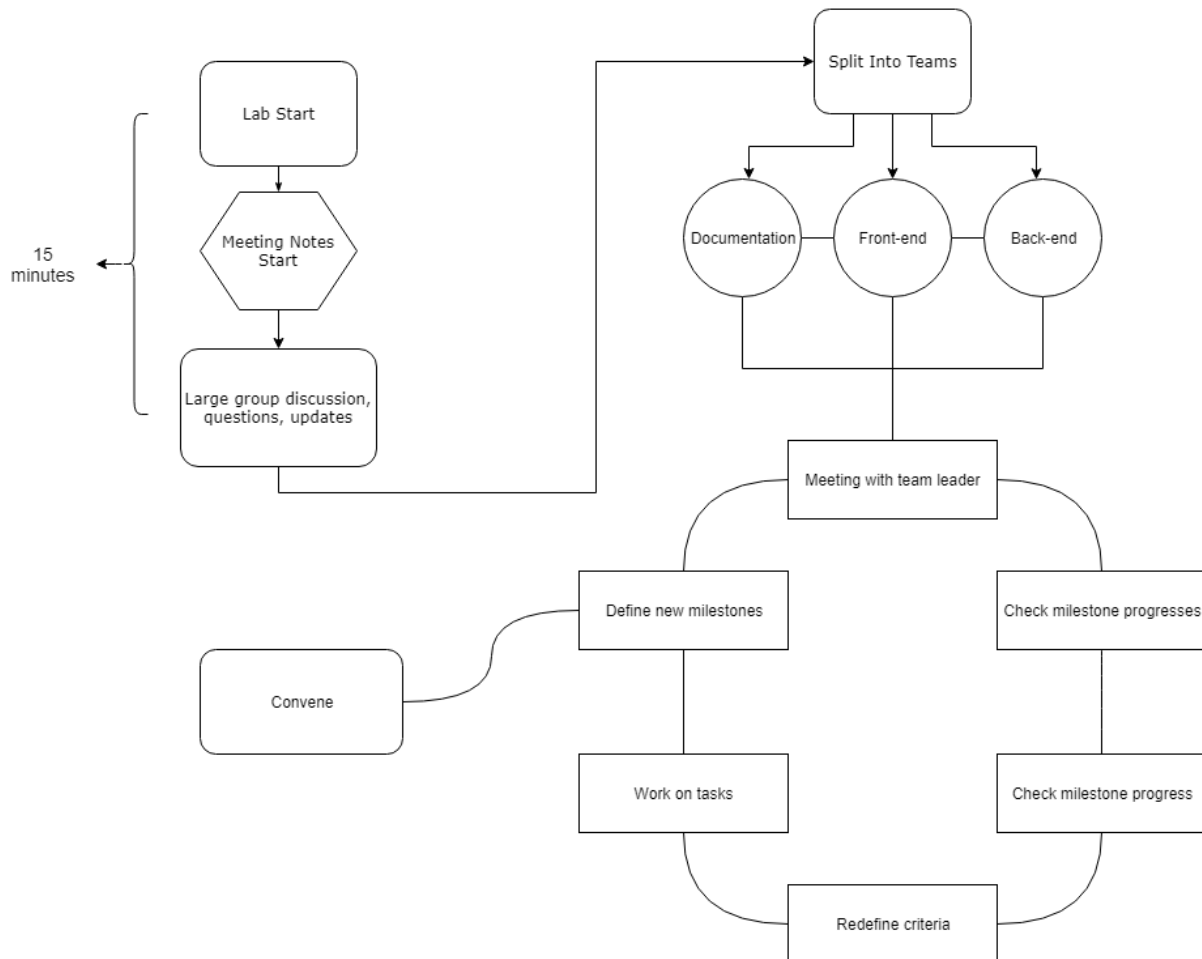
Meeting Space

Our team will meet in Olin 228 and Olin 124. Both of these rooms provide computers preloaded with a wide range of programming resources, printers, whiteboards, and enough space to easily accommodate our entire team. Meeting Times

Meeting Times

Full-Team Meeting Schedule: Tuesdays from 8:30 AM - 11:20 AM

Meeting Structure for Tuesday Labs



Sub-Team Meeting Schedule:

Team	Meeting Slot 1 (Day)	Meeting Slot 1 (Time)	Meeting Slot 2 (Day)	Meeting Slot 2 (Time)
Front-End	Wednesdays	7 PM	Sundays	2 PM
Back-End	Mondays	7 PM	Thursdays	4 PM
Documenta- tion	Thursdays	4 PM	Sundays	5 PM

Time Management

We are using a combination of the “issues” functionality of GitLab and a spreadsheet on Google Docs to keep track of tasks assigned to specific team members as well as to keep a long-term schedule.

Source-Code Management and Version Control

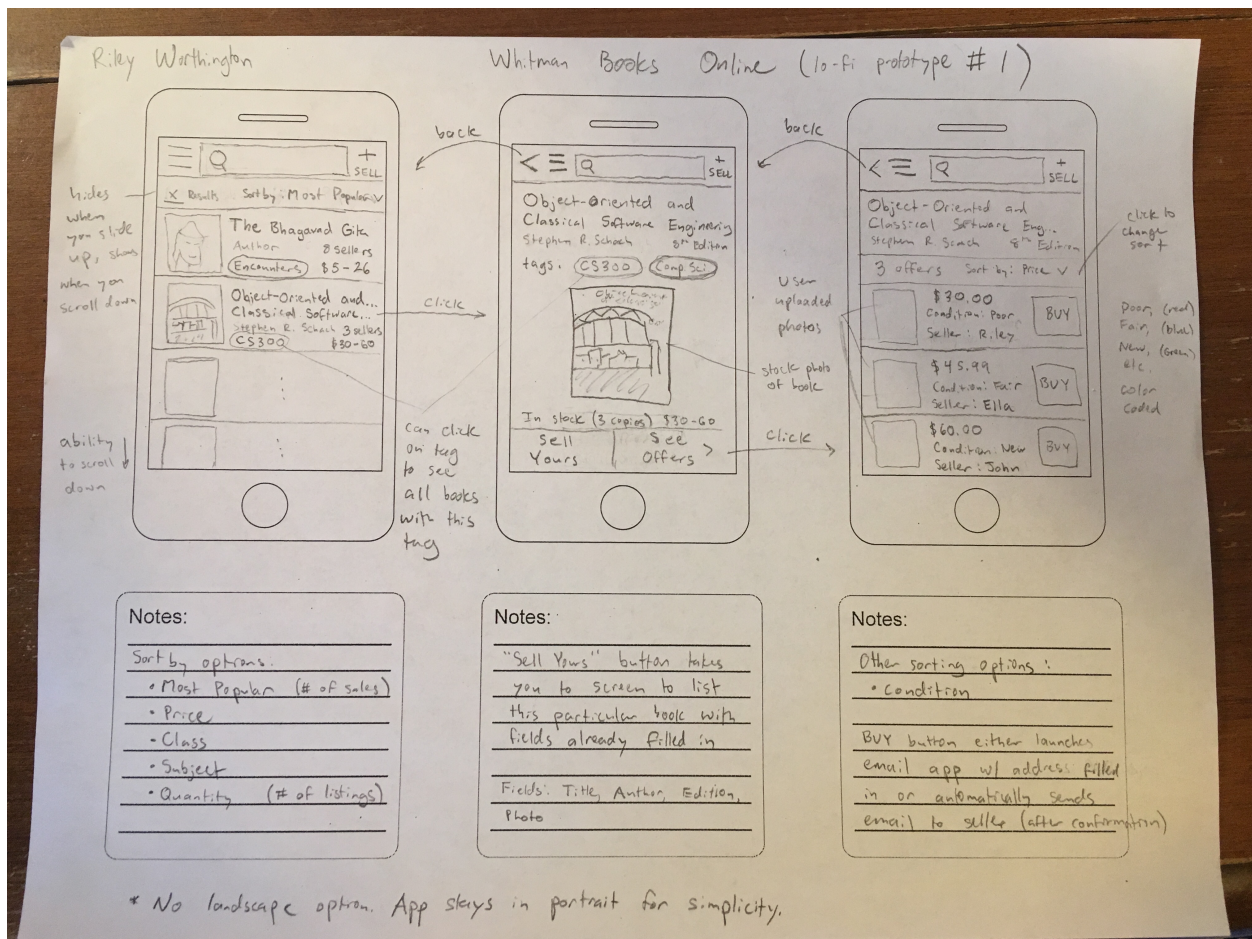
We are using Git for version control and storing our source code as well as our documentation on GitLab.

10.3.3 Schedule

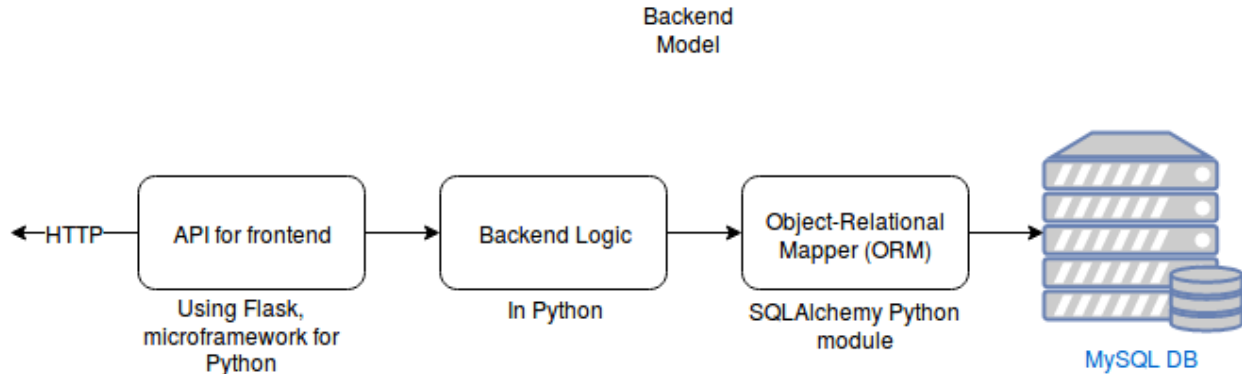
Minimum-Viable-Product

Preliminary Design Graphics

UI Drawings



Back-End Model



Post-Delivery Maintenance

We plan on allowing the final project to be used for as long as it is realistically usable and do minimal post-delivery maintenance on it. We as students will likely not have the time needed to work much on the project after this class is completed, so we will simply test our final product as much as possible before release.

10.4 Project value

We believe the benefits will largely be in terms of experience and benefit to the student body. We are gaining valuable experience in collaboration with each other and learning new software tools. Working in groups is today the only way to develop large software projects, and this class is allowing us to model that process. The community will benefit from having a webapp to use rather than the “forsale” email listserv. The webapp will be more organized and have more features such as searchable listings than a listserv, so the community as a whole will likely save some time when shopping for and selling books and have a more pleasant experience doing so.

Converted to reStructuredText by Isaiah Banta

Completed: 04/03/2018

11.1 Objective

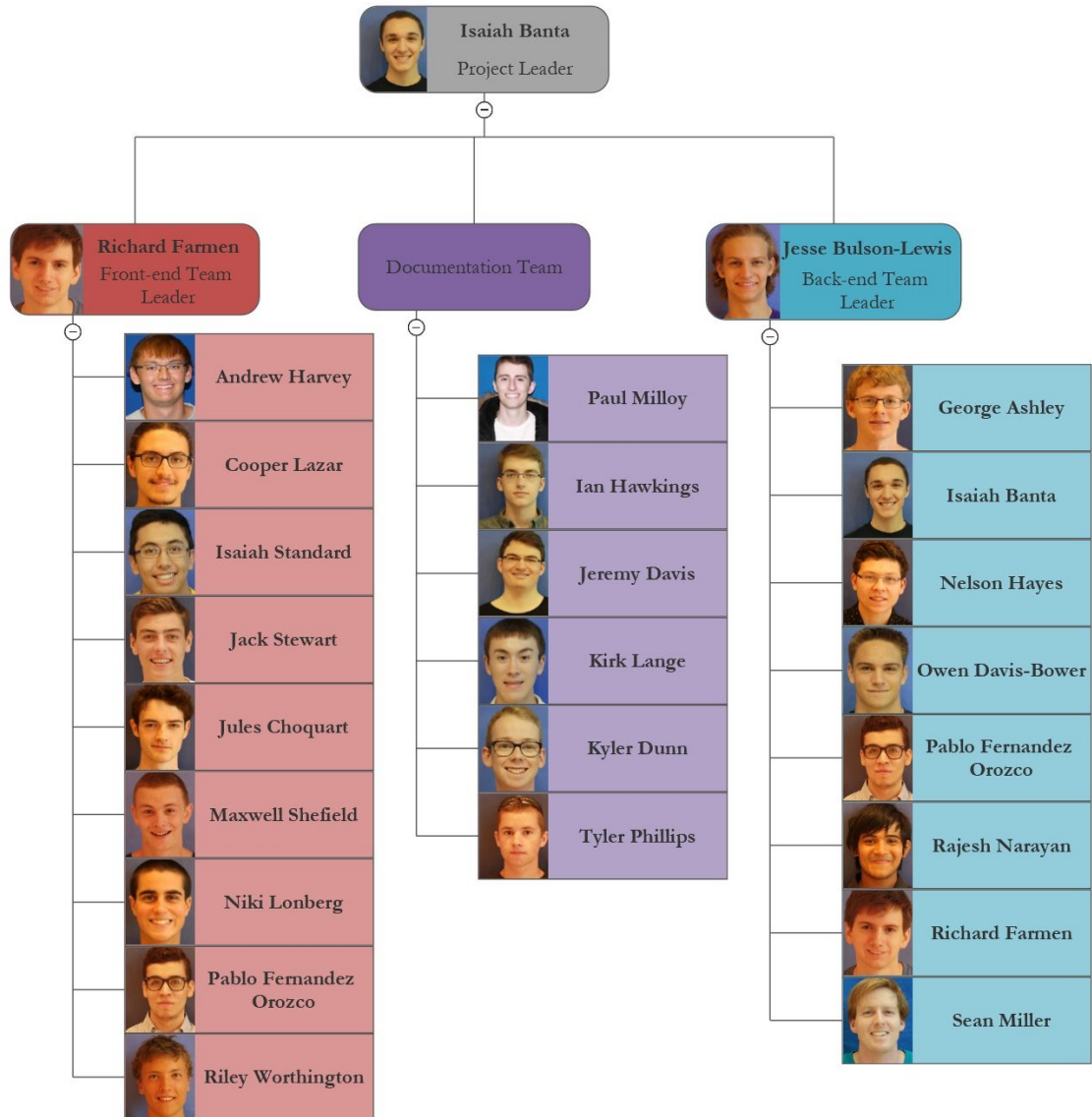
As stated in our requirements document, the objective of this project is to develop a web application to facilitate the sale and/or exchange of textbooks between students at Whitman College. The minimum viable product (MVP) will contain at least the following four functionalities:

- Create and log in to a user account.
- List a book for sale.
- Search for books.
- Reserve a book.

See our requirements document for more details on each of these items.

11.2 Team

Organizational Chart



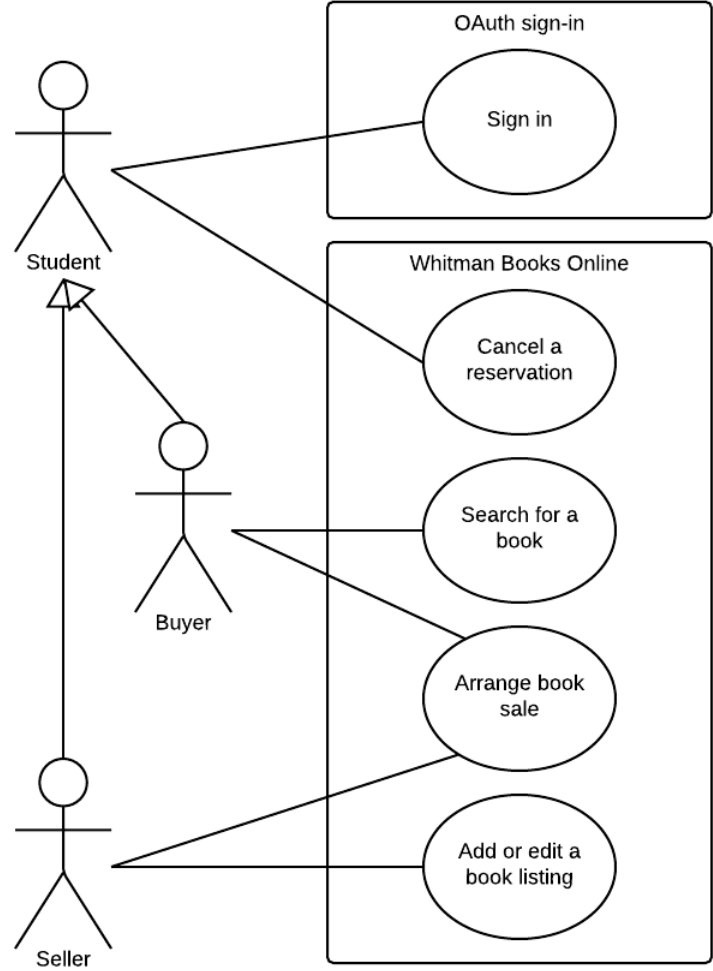
11.3 Front-End

11.3.1 User Stories

Seller As a **seller**, I want to **list my books** so that **I can sell them to people at Whitman with ease**.

Buyer As a **buyer**, I want to **find books I need** so that **I can buy them easily from people at Whitman**.

11.3.2 Use Case Diagram



11.3.3 User Interfaces

User Profile

**Riley Worthington**

SIGN OUT

Your Listings

To Kill a Mockingbird

Harper Lee

\$100, Good

\$120, Excellent

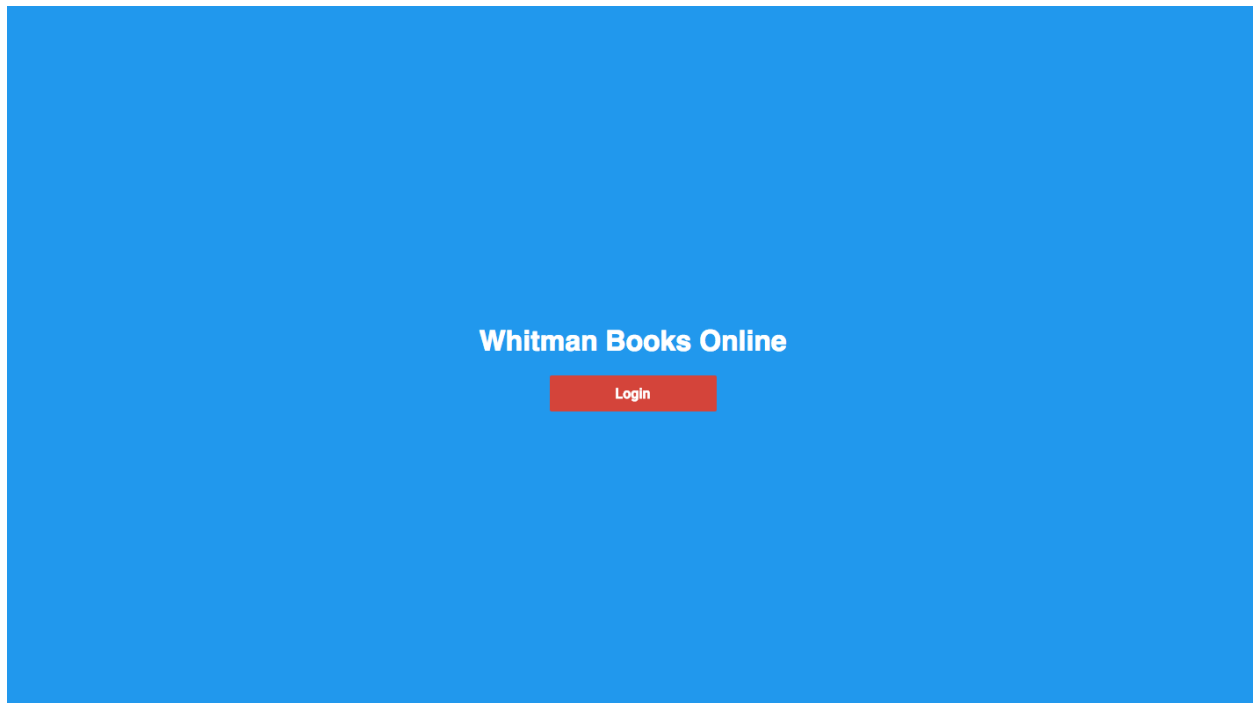
Gödel, Escher, Bach

Douglas Hofstadter

\$65, Okay

Contact:worthirj@whitman.edu

Login Page



11.3.4 Search

Initiating a Search To initiate a search, the user will simply have to type in the name of the book they are looking for in the provided search bar, towards the top of the application.

Display of Search Results Once the user has entered in what they are searching for, they will see a general listing corresponding to the book they are looking for. These results do not correspond to individual listings from sellers. Similar to Amazon's marketplace, the user sees the general listing for the product they are looking for.

Listings Once the user has selected a general listing, different sellers for that product will be shown. These correspond to individual entries by sellers. The user can proceed from those individual listings to communicate with the seller to organize a purchase.

11.3.5 Components

Our application will be designed using declarative, composable pieces called Components. Putting these components together will form the view of our application. These components will interact with our controller, allowing the user to have meaningful interactions with our Back-End, which will form the model.

Component List `<App />`

`<Page />`

- A wrapper that displays content on the user's screen.

`<Login />`

- Handles user login information to determine valid login credentials and give access to profile if valid.

`<Navigation />`

- Accesses components selected by the user.

`<Profile />`

- Displays user's profile information.

<Market />

- Displays list of book listings currently for sale.

<SellBook />

- Processes user input of book being listed for sale, posting to Market's page

<Search />

- Searches through listed books.

<BookFeed />

- List of books displayed on the Market's page

<BookItem />

- Stores user input of books for sale contained in BookFeed's list of books.

<BookPage />

- Page used to display BookItem information.

<ListingFeed />

- List of book listings for a specific book.

<ListingItem />

- Individual listing of a book with price and selling information attached.

<BuyBook />

- Handles user's purchase of a book, processing payment and removing the book's listing if paid for.

<ListingPage />

- Page used to display ListingItem information.

<SellerPage />

- Page pulled from a user profile to be viewed by other users that displays past sales history.

11.4 Back-End

11.4.1 Infrastructure

Server We will be using two CentOS 7 servers with private networking. The app server will run the Node.js runtime, the application, and PM2, which is a process manager, and the web server will run Nginx which will work as a reverse proxy to the application. This server is also how users will access the application.

11.4.2 Databases

Frameworks - Flask - to connect our API - Flask SQLAlchemy - SQLAlchemy is an SQL toolkit for the Python language.

Information in the Database - Users - All of the users of Whitman Books Online. - Books - All of the current book listings on Whitman Books Online.

Method of Collection - Users - Ideally retrieve user information using their Whitman ID, each user should have a unique ID. - Books - User inputted information, collected while creating a listing.

11.4.3 Search Algorithms and Implementation

Fuzzy Search for Lookups & Extending in C Database lookups will likely constitute the majority of our processing requirements. Thus, it is important to have a efficient way of matching an item in our database to a textbook a user is trying to access. Due to long textbook titles and the high probability that the average user won't know the ISBN of the book they are looking for, we need a search method that doesn't require the user know the exact name their target. We implement numerous fuzzy matching techniques to guess the exact entry from possible misspellings. The first is Levenshtein distance, getting an exact number of character distances from a pre-hashed list of possible spellings of all of our entries. We also implement a trigram search, breaking up each term into randomly sized substrings, and searching those through our database. We will not implement Metaphone or its derivatives as it is unlikely that a user will mistake a "p" for a "b" or like errors in typing the textbook they are looking for. For all of these to work with reasonable speed they should not be implemented in Python. We will use the subprocess module to spawn pre-compiled C++ programs for the obvious speed advantages. As the speed requirements of our program become more apparent (server selection) other implementations may need consideration. It is possible that the native library search functions will be adequate given enough processing power.

11.4.4 Classes

- ListingModel - includes "helper" functions for the Listing class
- Listing - objects have a listing id, price, user id, book id, condition (good, poor, fair), and status (sold/not sold)
- Listing class also supports get, put, post, and delete http requests, which interacts with the listing table in the database
- The Book and User classes follow the same framework ...

11.4.5 API

List of API Functions The API is being designed to accommodate all frontend needs and will update and retrieve data from the database. Currently, the items stored in the database are simple: just an item with a price. This will be changed to accommodate book objects, with the following corresponding database columns: title, author name, price, Listing ID, ISBN, Whitman class. The book objects will also be related to users in our user table:

- GET - returns all items in the database
- GET(item) - returns all information stored in the database about the item.
- POST(item) - This function adds an item to the database, does nothing if the item already exists in the database.
- PUT(item, price) - this will be the most commonly used API function. If the item does not exist in the database, the item gets added to the database along with the price. If the item exists in the database, the PUT function will update the item in the database with the current price. If the item already exists and has the same price, nothing will change.
- DELETE(item) - If an item exists in the database, the item gets removed from the database.

Similar functions will have to be implemented for the users.

The API also incorporates two functions for security via the JWT library:

- REGISTER - takes a username and password, saves it to the user table, and prints an error message if the user already exists or if there is a problem with the username or password.

- **AUTHENTICATE** - takes a valid username and password, and generates a security token that is required in order to call GET, POST, DELETE, etc.

11.5 Documentation

11.5.1 Documentation Plan

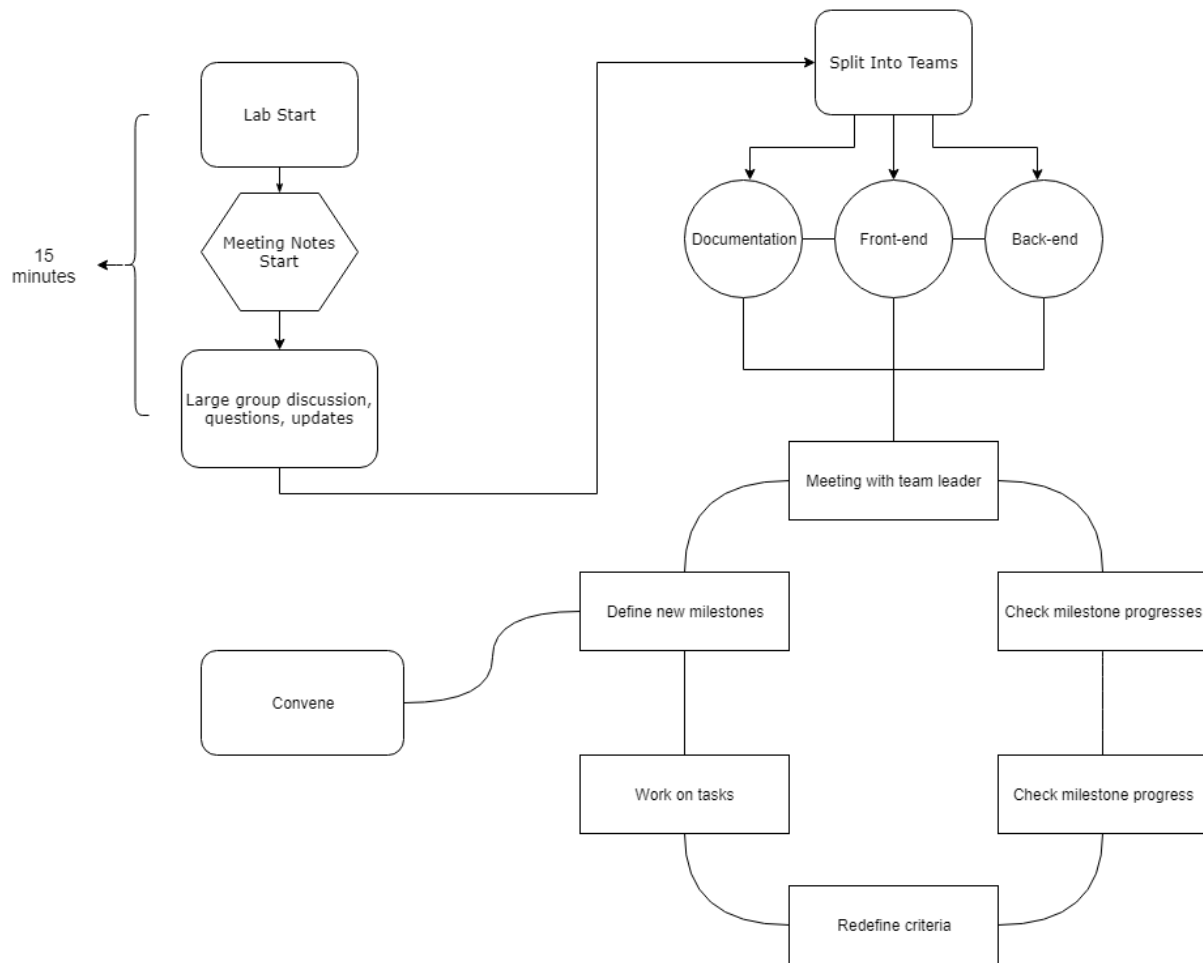
Sphinx We are using the Sphinx Documentation Tool and an associated Read The Docs page that will be fleshed out as we build the application itself.

Data Flow Following a team meeting, the respective team documenter will update the wiki with new tasks and a summary of the meeting itself. The following steps outline the process of adding information to the wiki This document is intended to suggest to documentation team members how to update the wiki with new tasks after a meeting.

- Take notes from the meeting a. Convert notes to reStructuredText
- Make a new wiki page or add to the one already created in /docs/source/meeting/ a. Meeting note file names should be named like so YYYY-MM-DD.rst, for example, 2018-01-30.rst
- For each task in the meeting notes, create a new GitHub issue a. Assign it to the person most responsible for it b. Add as much detail as possible from the meeting notes and specify any additional people working on that ticket c. Tag the issue with any relevant labels and add it to a milestone (if applicable)
- Notify Kyler about anything that would affect the big-picture schedule

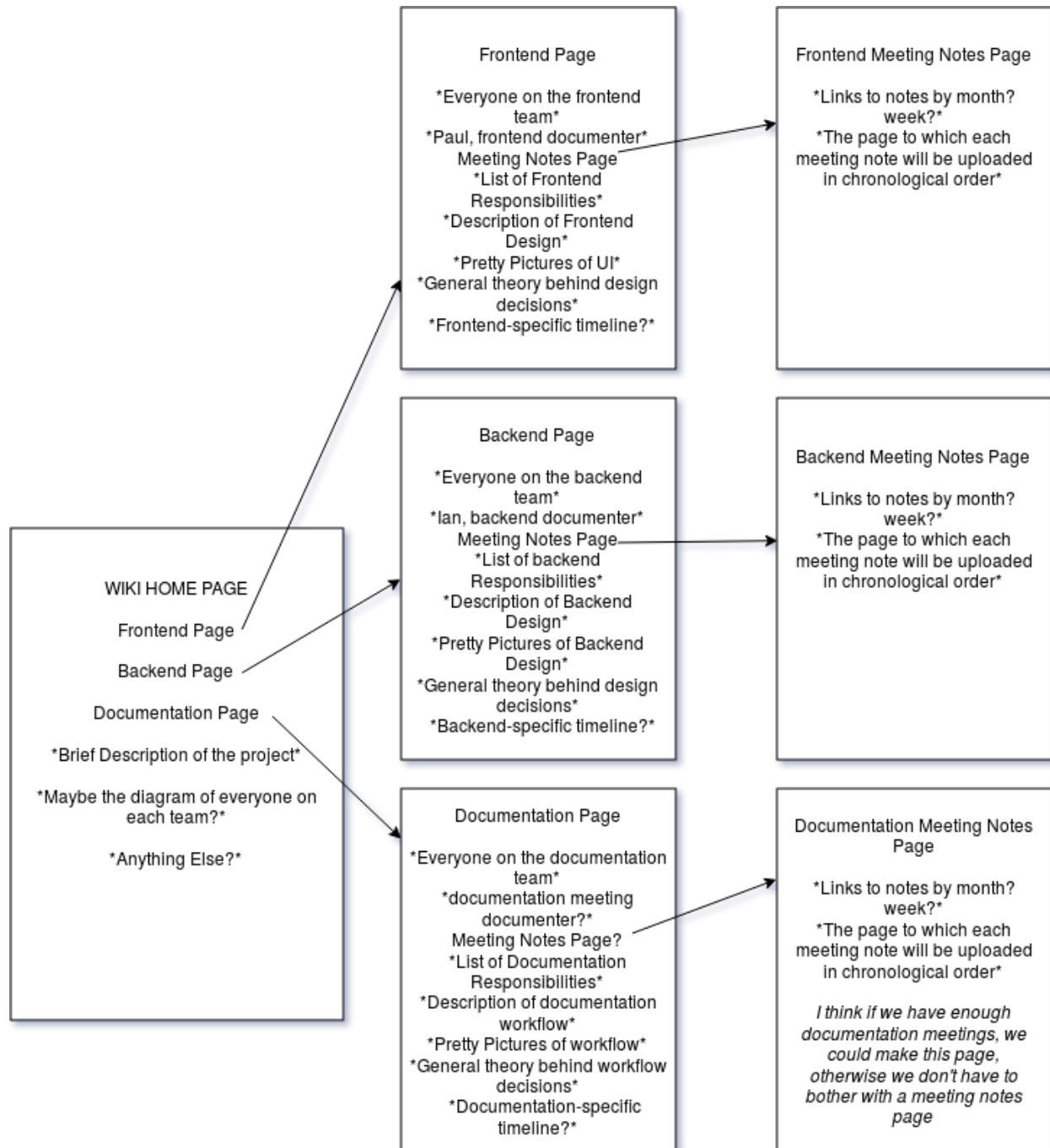
Meeting Structure

Meeting Structure for Tuesday Labs



Although the file structure has changed since the creation of the wiki flow chart below, the information on this image is still relevant in what each team's documenter should have in mind when writing team meeting notes.

Wiki Organization



Below are the team roles that were decided upon on January 30th.

Documentation Team Roles

Person	Responsibilities
Jeremy	team leader, wiki organizer
Paul	front-end documenter
Ian	back-end documenter
Kirk	documentation documenter (so meta), wiki manager
Kyler	schedule and issues manager
Tyler	special documents writer

11.6 Schedule

11.6.1 Master Schedule

As I continued working on the weekly schedules we (I) realized that people were more interested when large project issues were to be finished. I went around checking in with people on what sort of schedule they would like the most out of the Scheduling Manager. They really liked the Whitman College academic calendar. I made essentially the same schedule as the Whitman Academic Calendar and put in our own assignments and due dates. ~Kyler Dunn

Master Schedule

FEBRUARY														MARCH																									
record permitted thru February 23														Drop with 'W' grade permitted thru April 6																									
Week 4				Week 5				Week 6				Week 7				Week 8				Spring Break				Week 9															
F	M	T	W	Th	F	M	T	W	Th	F	M	T	W	Th	F	M	T	W	Th	F	M	T	W	Th	F	M	T	W	Th										
2	5	6	7	8	9	12	13	14	15	16	19	20	21	22	23	26	27	28	1	2	5	6	7	8	9	12	13	14	15	16	19	20	21	22	23	26	27	28	29
														Design Phase Due																									
														Implementation Phase Begins																									

Design Phase Due

Implementation Phase Begins

		APRIL														MAY																				
					Fall Pre-registration										Final Exams																					
Week 10					Week 11					Week 12					Week 13					Week 14					Week 15					Week 16						
F	M	T	W	Th	F	M	T	W	Th	F	M	T	W	Th	F	M	T	W	Th	F	M	T	W	Th	F	Sa	M	T								
30	2	3	4	5	6	9	10	11	12	13	16	17	18	19	20	23	24	25	26	27	30	1	2	3	4	7	8	9	10	11	12	14	15			
																	MVP Due					User Testing										Final Rollout				

MVP Due

User Testing

Final Rollout

Date	Assigned
Wednesday, March 7th, 2018	Design Phase Document Due
Monday, March 26th, 2018	Implementation Phase Begins
Tuesday, April 17th, 2018	MVP Due
Tuesday, April 24th, 2018	User Testing Begins
Thursday, May 10th, 2018	Estimated Final Rollout

11.6.2 Weekly Schedules

As the Schedule Manager for this project, I took it upon myself to sort of track what each group was doing over each week. This was to be used not only to aid in keeping people up to date but also for Mr. Banta to know where each team is at. The information tracked on the schedule would be used at overall project meetings so that everyone is aware where people are at during the design phase. ~Kyler Dunn

Week Four

	Total Tasks	I	A	C
Totals	9	3	3	3

Week Five

	Total Tasks	I	A	C
Totals	7	1	4	2

Week Six

	Total Tasks	I	A	C
Totals	7	1	3	3

[illegible]

- ISBN
 - This ISBN is used with a lookup service to gather data on:
 - * Title
 - * Author
 - * Edition
 - * etc.
 - Price
 - Condition

11.7.3 Search for Books

Searching for a book is a main feature of the Whitman Books Online service. A search bar will be prominently displayed on the home page and at the top of subpages, from which users can enter queries for the following attributes:

- Title
- Author
- Class
- Subject

The search will query the database and receive back a list of relevant books. The search will include an autocomplete feature to predict search terms as the user enters them. Once the list of results is displayed, users will be able to filter by Popularity, Subject, Date Added, and Price to further narrow down their search.

11.7.4 Reserve a Book

The process for reserving a book is as follows.

1. Locate book in the database
2. Choose copy you wish to buy
3. Click “Buy” button, triggering the launch of mail client to contact seller
4. Arrange to meet with seller outside of the app
5. Book remains listed until seller closes the transaction

11.7.5 Security Plan

Security questions within the application will likely be questions about authentication and user storage. While users will be stored in the database, our plan is to not store any authentication details, largely to minimize our security risks. This will be made possible by using Google OAuth 2.0 to identify users to the application, allowing us to only store something like an email address with which we can correlate Google users to application users. This will also allow us to limit logins to people with valid Whitman accounts.

Security from an operations perspective will try to provide for threats from generic unscrupulous users. Treating the application as a black box, such security will have two primary components: security for the server from unauthorized users and security between the application and the user. The first will be provided for with standard server security measures: key-based ssh auth, a maximally locked-down firewall, etc. If we get the budget for the required EC2 instances, we could even host the frontend service on an edge server, leaving backend+DB on another instance locked

down to the public. As for the security between the server and users, we will use HTTPS for all public connections, only allowing unencrypted HTTP URIs to redirect to their HTTPS counterparts. This should adequately provide security between the user and the application.

11.8 Testing

11.8.1 Front End

The Front-End will use the Jasmine testing framework to write and run our tests. We will focus our tests on the functions that interact with our Back-End and other services and will make sure all tests pass before opening pull requests to the master repo.

Jasmine.js

An example of the kind of test we would write.

```
describe("A suite is just a function", () => {  
  var a;  
  
  it("and so is a spec", () => {  
    a = true;  
  
    expect(a).toBe(true);  
  });  
});
```

11.8.2 Back End

Backend will use the unittest Python module to write unit tests. Just like the front end, we will ensure that all back-end tests pass before any merges into master.

CHAPTER 12

Ethical Concerns

Preamble: The CS-300-A Software Engineering Class of Spring 2018 is dedicated to developing functional and ethical software according to the guidelines of the IEEE/ACM Joint Task Force on Software Engineering Ethics and Professional Practices for the benefit of the students, staff, and faculty of Whitman College.

During the Analysis Phase of our Software Development process it was discussed as a class the primary goals and possible ramifications of developing a web app that would be used to buy and sell books on a peer-to-peer system. The discussion can be surmised in three main categories:

Demand: Is there an existing demand for such a software?

Yes. When taking into account the emails sent to the student listserv and the Facebook posts on the Whitman class pages it was unanimously agreed upon that there is a demand for such an app and that it would streamline a process that has become disorganized. Currently the only standardized solution to this problem is the Whitman College Bookstore that offers Textbook Buyback as a service. However, institutional trust in the current system is low and the Bookstore does not offer a cost efficient method. This webapp has the opportunity to circumvent such a system altogether by introducing a standardized and cost efficient solution for students.

Safety: Can we safely bring together two parties to complete a transaction?

Safety is one of our primary concerns. Although, Whitman students are already meeting at planned rendezvous locations in order to complete textbook transactions, this webapp has the opportunity to introduce a higher level of safety by standardizing the process and providing the information to already existing resources available to the students of Whitman College. Such as the Security Department and the Health Center. The webapp will also detail specific meeting spaces that are public and the timeframe in which they are publicly attended. Such as: - The Library (Open 24/7 and only accessible with a Whitman ID after 10 PM) - The Whitman Health Center (Open 24/7 and only accessible with a Whitman ID after 8PM) - Reid Campus Center (Open 8 AM - 11 PM)

Impact: What are the greater ramifications of such a software on the Whitman College Community?

It was discussed that the Whitman College Bookstore, and perhaps Whitman College Administration, may not be in support of an app that would allow students to more easily circumvent the bookstore and any revenue it may gain from the buying and selling of textbooks. As a result of this, the possibility of this webapp impacting the employment of individuals in the Whitman College Bookstore was considered. While this is an important concern, it was unanimously decided that the predatory nature of college textbook pricing far outweighed the possible ramifications of such an

impact on the Whitman College Bookstore. This webapp is by no means a solution to the predatory nature of college textbook pricing, a structural problem with higher education that lies beyond the scope of this document. However, we, as a team, hope that this webapp proves to be a solution to a symptom of the larger issue.

Cheat sheets, helpful guides, basic how-to's, and other stuff

13.1 Documentation

- [Markdown cheat sheet](#)
- [Sphinx tutorial](#)

13.2 Front-End

- [Testing for JS](#)
- [Cool JS stuff](#)
- [Cool React stuff](#)

13.3 Back-End

- [Python unittest](#)
- [Python logging](#)
- [Docstring standard for python](#)

13.4 Git/GitHub

- [Git cheat sheet](#)
- [How to add an image](#)

CHAPTER 14

Meeting Notes Workflow

Based on whiteboard drawings by Ian Hawkins and Tyler Phillips on 2018-01-30.

This document is intended to suggest to documentation team members how to update the wiki with new tasks after a meeting.

- Take notes from a meeting
- Make a new docs page and specify its location and date. For example:
 - `meeting/backend/2018-02-14.rst`
 - `meeting/general/2018-03-20.rst`
- Convert your meeting notes to [Markdown](#) formatting, if they aren't already.
 - Edit: Now instead convert to [reStructuredText](#).
- For each task in the meeting notes, create a new ticket in the appropriate project and assign it to the person most responsible for it. Add as much detail as possible from the meeting notes and specify any additional people working on that ticket.
 - For a proof of concept task, click the “Issues” button in the top right (on desktop) and add an issue to the ‘poc’ project.
- Notify Kyler about anything that would affect the big-picture schedule.

Requirements Document

The CS300 class will develop a web application to facilitate the sale and/or exchange of textbooks between students at Whitman College. This application will include, at minimum, the following features:

15.1 User account

A user will be able to create an account on the site that is linked to their Whitman email address. The application will allow users to login to and logout of this account. Users will not be able to create listings or view other users' contact information without creating an account. This will limit these permissions to Whitman-affiliated users only, for security and privacy reasons.

15.2 List a book for sale

Once a user has created an account, they will be able to list a book for sale. Book listings will be linked to the seller's user account and will include the following pieces of information: - The title of the book - The edition of the book - The book's ISBN number - The book's general subject matter - The sale price - A picture of the book, if the seller chooses to provide one, and - The condition of the book (level of wear and tear, presence of highlights and underlining, etc.)

15.3 Search for books

Any user will be able to search the collection of books that have been listed. Only users that have created accounts will be able to view the name of the seller of a book. Book listings will be searchable by title, ISBN number or subject matter.

15.4 Reserve a book

Users who have created an account will be able to reserve a book by clicking a reserve button on the book's listing page. This will send an email notification to the seller's Whitman email address notifying them that someone has reserved their book and including the Whitman email address of the user who has reserved it. Once the reserve button has been clicked, the book listing will no longer appear publicly. The listing will remain in the application's database until the seller indicates that the book has been sold. The buyer and the seller will both have the ability to unreserve the book in case of a mistake or in case the sale does not go through, in which case it will reappear publicly. It will then be up to the seller to contact the buyer via email, outside of this application's interface, to arrange the sale. The application will recommend that users meet in a public location on campus to complete the sale.

Note: Further consideration of the time lapse between a book taken out of circulation and an incomplete sale to be made in the future.

16.1 Jan 23 Tue - Initial Project Vote

Class voted for the app proposal Whitman Books Online. Original description:

An app that lets Whitman students buy and sell textbooks from other Whitman students. Currently, there is no good way to buy used textbooks from other students. If there was a simple app to search by book, then students would not be forced to use the bookstore. Students could also get more money back for their textbooks, and could also purchase them for cheaper.

16.2 Jan 24 Wed - Minimum Viable Product

16.2.1 Components

- textbook directory
- textbook listing
- user profile (barebones, Whitman email)
- item profile
- categories list/page

16.2.2 Actions

- add/remove listings
- sort by x
- compare
- search

- login/logout via Google Authentication
- buy item / agree to meet with seller
- search/filter

16.2.3 Other

- transactions occur manually between individuals (not virtually, at first)

16.3 Jan 30 Tue - Requirements Brainstorming

16.3.1 MVP Components

- Item
- Item List
- Login/Logout
 - Login/Logout/Google Auth (email, username)
- User Profile
- Item Profile
 - Subject, class, **name of the book**, book edition, **ISBN**, **Price**, Picture option, **username**, quality(bad->good->perf)

16.3.2 Non-MVP Components

- Category
- Category List
- Category Page

16.3.3 Account creation

(Lower barrier of entry, anyone can view, users can buy/sell/view usernames)

16.3.4 Story

Jane: Create>User Profile>Add listing>Logout John the Buyer: Create>User Profile>Searches for 'book'>Opens Jane's Item profile>Contacts Jane>Sets up meeting>Buys the book>End Jane: Login>User Profile>Delete listing>End

16.3.5 Tasks

- Contact Bookstore about course requirements (Jeremy) - Friday 2nd
- Requirements phase document (Documentation) - (Hard deadline:Friday 9th Feb)
- Google Auth POC (George) [Back-end] - Tues 6th Feb

- Schedule (Kyler) [Spiral] - Tues 6th Feb
- Further define roles.. (integrate with issues in Gitlab, wiki, transparency)
- Frameworks, languages, API (frontend, backend, documentation)
 - Documentation: Git, Gitlab, Markdown
 - Frontend: HTML, CSS, Java
 - Backend: SQLAlchemy (Databases), Flask (API)
- Database POC (Owen, Sean)
 - Whitman books group

16.4 Jan 31 Wed - Ethical Implications

Considered possible ethical implications of this app. The relevant parts of Jeremy's report after talking with the bookstore went as follows:

I talked to Janice King at the bookstore. [...] She definitely was suspicious of me though, and said that since she doesn't know what our project is, she wants to be careful about giving information that "slits her own throat", as she put it. Yikes. She also talked at length about being one of the last independent bookstores in the northwest, etc. She asked for details on the project, and I deflected saying we were all working as a group of 20+ people so we were still figuring that out. She also wanted to see the final project for the purposes of "being transparent both ways", so that's a problem. I guess the bookstore would have found out one way or another.

After discussion on Slack and in class, we decided to continue on the project anyway, because capitalism.

16.5 Feb 06 Tue - Planning and Workshopping

- Back-end OAuth discussion (see back-end meeting notes)
- Sub-team re-organization
- Workshop held on Git and GitLab

16.6 Feb 20 Tue - Wrapping Up Analysis

- The **importance of attendance** should be brought up to the whole group, especially that of team leaders.
 - **Team leaders should consider making a team planner** so in case of absence/sick-day their team is not completely lost as to what to do.
 - Anyone who will not show up to lab should communicate via Slack.
 - We will **begin attaching deadlines to our milestones/issues** in the hopes of encouraging stronger attendance and productivity.
- **We will be using Sphinx for code documentation.**
- Analysis document is coming along well. We mainly just need more info from the front-end.
- Documentation team needs a dedicated documenter (i.e. someone to write notes like these). Kirk Lange has taken on this role.
 - Who watches the watchmen?

16.7 Feb 27 Tue - GitHub Migration

16.7.1 GitHub

- Might as well make the repo public?
 - It's going public eventually anyway

16.7.2 Migration Process

1. Create GitHub repo
2. Setup Sphinx
3. Migrate existing content to GitHub (such as converting our current wiki to .rst)
4. Link ReadTheDocs to the GitHub repo

16.8 Apr 03 Tue - Funding Update

- Asked for either hosting on Whitman's servers or money to get an AWS instance
- Questions Mooko asked:
 - How we know this app does not already exist?
 - What will it do that Whitman's buy-back system doesn't do?
 - Vouched for the Whitman bookstore because it's non-profit
- Need to negotiate with bookstore if want to gain access to Whitman servers
- Will meet with and solicit funding from Finance Department

16.9 Jan 26 Fri - UI Design

Model

- Database of books/users
- Constructed by backend (not sure which language yet)
- Holds all info

View(UI)

- Constructed using HTML/CSS
- Display textbooks on sale, Craigslist style organization
- Projecting what is given by Controller

Controller (UX)

- Any change to view happens in controller
- Get data from Model, make stuff, give to View
- Everything must pass through Controller

GOAL: Keep Model, View and Controller files distinct from one another

- Go over CSS/Javascript before lab on Tuesday
- Model will be largely handled by backend
- View and Controller almost entirely frontend

Lots of communication necessary with backend to keep View/Controller in line with Model

- Compromise, meet halfway, flexibility is key

16.10 Jan 30 Tue - Front-End Tools and Design

Redux = Library to use on top of React

- Will act as Controller
- Changes will be reflected in state and then reflected back in app
- Complicated so we'll go over it more later

Consider doing testing, dependent on scope of project

- If time allows, should be doable
- Could use Jasmine framework
- Documentation team writing testing?

Back-end requests logging, would be helpful

- Running through the app
 - User prompted to login by start page
 - User tries to login
 - * Whitman email = controller presents view for app
 - * Non-Whitman email = controller presents failure page
- List of books in app
 - Each book has drop down showing all sellers of book/all copies on sale
 - * Drop down or dedicated page? Disagreement for now

16.11 Feb 2 Fri - Overall Architecture

- Develop website first then port to mobile app
- Start with list view on site
 - Add grids afterwards?
- System like Amazon or Ebay?
 - Amazon = Click on book, pulls up list of sellers
 - Ebay = All entries that match search listed separately
- Clicking on an entry should take you to a different page
- Want to be able to search for a book and see cheapest option

- Sort by option in sidebar

16.12 Feb 05 Mon - Details and Desired Functionality

- Listings showing title of book and price range
 - Selecting listing creates dropdown
 - Dropdown contains list of sellers ranked price/book condition
 - Disagreement on dropdown or new page
 - * What are the advantages of dropdown vs. new page? Vice versa?
- Book condition, select from list of preset conditions
 - Can add additional description of condition, click on “More” to see
- Can upload any book, not only Whitman books
 - Add ISBN number when uploading new book?
- **Seller’s perspective:** Adding books to sell
 - Adding by title
 - Adding by ISBN
 - * Necessary to be accurate on editions of textbooks
 - * Could prompt user to ask to check ISBN
 - * Original upload of book could force adding ISBN
 - Subsequent adds would ask to check if ISBN is same
 - Ideally, would be able to add by either
 - Pulling info from Amazon?
 - How do people know how much to sell for?
 - * Free market will decide
 - Option of adding picture(s) of book, not required to upload

16.13 Feb 06 Tue - Tool Tutorials

- Start writing code today?
 - Start with basics of web development
 - * Code Academy links in front-end webpage
 - Get proper software installed on every front-end member’s laptop
- Time spent working with Code Academy to familiarize with languages
- FlyChecker to maintain style guidelines
 - Automatically corrects coding style based on presets
 - Could be helpful in maintaining consistent coding style
- React Tutorial

- Public folder contains index.html file
 - * Template for app
- Use ReadMe to understand create-react-app
- Package.json
 - * Describes basic configuration of app (Name, version etc.)
- Changes to React environment create immediate changes to webpage
- Importing functions similarly to Python
- Define components as class containing attributes and methods
 - * Almost every component will have render() method
- Sizing and Formatting
 - * Use px for elements you want to keep the same size and % for elements you want to scale with size of the page
 - * % is based on space given, not on true size of page
- Inspecting a webpage: Ctrl + Shift + I (learn this shortcut)
- Making a component for Login page
 - * Renders login button on screen
 - * Inserted into render() method of app
 - * Login function sends alert for logging in when clicked (temporary)

16.14 Feb 09 Fri - Random

- Black Pearl or USS Enterprise?
 - Enterprise: No drinking, sterile environment, but SPACE
 - Black Pearl: Lots of drinking, swashbuckling (great), grog (GREAT), scurvy...
- Nikhil cut me from the team
- Prioritize finishing React tutorial

16.15 Feb 06 Tue - Team Organization and OAuth

Making sure everyone is on the same page; Affirming what backend is and does.

Backend has all their current tasks put into issues. Jesse will make more tickets for his own tasks.

George has a paid server option that sounds good.

Jesse will probably take over Raj's ticket of determining what API is suitable for this project. Flask is the most likely pick.

We have access to online Flask lessons through Udemy, as discussed in Slack.

16.15.1 OAuth

Do we want OAuth to send the token directly? That's a question for later, but Jesse thinks the token should be sent to us along with the full name and email address.

Since this is looking more like a frontend matter, Jesse will talk to Richie about passing authentication responsibility to frontend and the OAuth PoC ticket will be closed.

George has been using Flask for part of the OAuth PoC.

16.16 Feb 28 Wed - API Basics

OAuth being handled by frontend, because of how Google handles its Auth stuff

16.16.1 Frontend/API Actions

- login
- create/update/remove user profile
 - status, object for create/update
 - status for remove
- get user profile
- create listing
 - insert listing record
- create/update/remove listing:
 - status, object for create or update
 - status for remove
- search/filter:
 - listing (fuzzy search?)
- get user
- get listings

16.16.2 Database Structure

- user profile:
 - name
 - email
 - last login date/time
- listings:
 - seller, price

Anytime you want to add something it would be a put request

Richie wants to update data rather than add whenever possible

Designing super basic api for now, specifics will depend heavily on basic structure, so no point getting to specifics now

16.17 Mar 27 Tue - API Endpoints

Owen, Sean, and Richie

Small meeting to talk about the functions

Richie brings up the design phase document to show the diagram.

Richie wants a function for each endpoint. For example, there will be an endpoint called `/users` and it will contain key-value pairs where the key is a user ID and the values are data about the user. There may be 2 functions for `/books` so that they can be accessed by ID or keywords.

Endpoints:

Think of the endpoints as their own classes.

Here's a representation of a possible database. There may be some bits missing, for example, a book has more information than just title, author, and ID.

- */users*
 - *\$user_id*
 - *name*
 - *email*
 - list of listing IDs for listings created by this user
- */books*
 - *\$book_id*, perhaps ISBN
 - *title*
 - *author*
 - list of listing IDs for listings for this book
 - * *\$listing_id* for each listing
- */listings*
 - *\$listing_id* <- this is one particular listing
 - *price*
 - *condition*
 - *book_id*
 - *user_id*
 - *status*

Frontend will send a request with either a query or some number of book IDs. When they send a query, backend needs to do some kind of search. When they send book IDs, just look up those books.

Things that frontend needs from the API:

In any of the “get” actions above, provide all information about the object being requested.

16.18 Apr 03 Tue - API Meeting

Sean and Owen talked to Richie [reference the notes for this meeting] and they want to get everyone caught up on what they know.

By the end of this meeting, everyone not working on something else will have

In Flask, you declare a port number and an endpoint like */books*.

The endpoints have methods *get*, *put*, and *delete*. This will look like */book(ID)* where *ID* is a unique identifier for the book. Books can also be accessed with a search term.

There will be 3 Python modules: *book.py*, *listing.py*, and *user.py*, to be divided among backend team members. A class will be written for each and then the Flask will be implemented afterward.

16.19 Jan 30 Tue - Task Management

Tasks will be assigned to individual/multiple people will be handled via issues

16.19.1 Roles as of Now

Person	Responsibilities
Jeremy	wiki organizer, front-end secondary
Paul	front-end primary
Ian	back-end primary
Kirk	documentation team documenter (so meta), doc binder, wiki manager
Kyler	scheduler/task manager, back-end secondary
Tyler	document writer

16.20 Feb 05 Mon - Take Me To Your Leader

- Jeremy Davis becomes documentation leader
 - roles: vision, setting goals, documenting team leader meetings
- Nelson Hayes (back-end) drops the class

16.21 Feb 06 Tue - Analysis Document Outline

Jeremy Davis’s notes from talking to Prof. Klein about the analysis document

16.21.1 Risk analysis

Risk of not getting project done on time and what can be done to mitigate that risk (include timeline for example) Risk of litigation, talk to Whitman legal counsel Personnel risks such as bad relationship with bookstore

16.21.2 Tools

Analysis of software tools to be used to accomplish project such as MySQL for database, html/css and Javascript and react for frontend, etc. Doxygen? All tools we will be using Make sure all the interfacing between languages is doable

16.21.3 Backend

python with flask microframework SQLALCHEMY Object-relational mapper for use with MYSQL databate Anything we have to buy? Domain name, server hardware? Time analysis, do we have enough time to complete everything?

16.21.4 Marketing Strategy

Catchier name? Survey of student to see if they would even be interested? Design of ui to make it salable

16.21.5 Other

Software quality assurance analysis, such as what we do to make sure code is up to par, anticipating possible problems to mitigate them. Anticipating future sub-divisions of teams Different organization charts of sub-teams, more up to date for exactly what everyone is working on

16.22 Feb 27 Tue - New Documentation Tools

- For sure using Sphinx
- Probably using readthedocs.org, especially if the repo is public
 - Look into ReadTheDocs private documentation
 - Link to ReadTheDocs in the top directory's README.md

16.23 Apr 11 Wed - Documentation Tasks

- After doc leader feedback survey, Jeremy will now take a more hands-on approach
- Tentative deadline for user tutorials is April 20th
- JavaScript autodoc work-around * Use JS' autodoc -> html * Move that html into an rst file * Have that rst file be an html block
- Continue to reorganize meeting notes into their seperate teams

CHAPTER 17

Privacy Policy

This privacy policy discloses the privacy practices of “whitmanbooks.online” A.K.A. “whitmanbooksonline.com”. This privacy policy solely applies to information collected by these domains.

It will notify you of the following:

1. What personally identifiable information is collected from you through the website, how it is used and with whom it may be shared.
2. The security procedures in place to protect the misuse of your information.
3. How you can correct any inaccuracies in the information.

Information Collection, Use, and Sharing

The information that *whitmanbooks.online* gathers is minimal because we use Google Authentication to verify that all users have a *whitman.edu* gmail address.

Here is what we have access to:

- your IP address
- the date and time
- your browser
- your operating system
- your listed books
- basic information related to your google account:
 - your name
 - your profile picture
 - your email address that you use to login

Here is what we store in our database: - your google token associated with your Google+ profile (this is publicly available information) - any books you are selling, their price, and their condition

All of this information is used to generate your profile and connect your listings to your *whitmanbooks.online* profile. Additionally, your email will be attached to the books you are selling for ease of contact.

Your Access and Control of Your Information

At any point you may contact us via the email address or phone number given below for any of the following:

- see all data we have related to your account
- have us delete any data related to your account
- express any concerns you may have about our use of your data

Security

Our website protects all information by encrypting and transmitting it in a secure way. You can verify this by checking the left hand side of your URL bar and noticing the green padlock and 'Secure' text denoting a secure https connection.

Our use of AWS and Google Auth also allows us to take advantage of the security of two of the largest tech companies in existence. This means that your data is more likely to be kept secure.

Notification of Modification

Should this privacy policy change, all users will be notified of its change through email or our website.

If you feel that we are not abiding by this privacy policy, please contact us immediately via phone or email: (619) 495-2111, bantaib@whitman.edu .

a

`api.book`, [23](#)
`api.listing`, [26](#)
`api.user`, [30](#)

A

allListings (class in api.listing), 29
 api.book (module), 23
 api.listing (module), 26
 api.user (module), 30
 as_view() (api.book.Book class method), 23
 as_view() (api.book.BookList class method), 24
 as_view() (api.listing.allListings class method), 30
 as_view() (api.listing.Listing class method), 26
 as_view() (api.user.User class method), 30
 as_view() (api.user.UserList class method), 31
 authors (api.book.BookModel attribute), 25

B

bare_json() (api.book.BookModel method), 25
 bare_json() (api.listing.ListingModel method), 28
 bare_json() (api.user.UserModel method), 32
 book (api.listing.ListingModel attribute), 28
 Book (class in api.book), 23
 book_json_w_listings() (api.book.BookModel method), 25
 book_json_wo_listings() (api.book.BookModel method), 25
 BookList (class in api.book), 24
 BookModel (class in api.book), 24
 bu_bare_json() (api.listing.ListingModel method), 28

C

canonicalVolumeLink (api.book.BookModel attribute), 25
 categories (api.book.BookModel attribute), 25
 condition (api.listing.ListingModel attribute), 28

D

decorators (api.book.Book attribute), 23
 decorators (api.book.BookList attribute), 24
 decorators (api.listing.allListings attribute), 30
 decorators (api.listing.Listing attribute), 27
 decorators (api.user.User attribute), 30

decorators (api.user.UserList attribute), 31
 delete() (api.book.Book method), 23
 delete() (api.listing.Listing method), 27
 delete() (api.user.User method), 30
 delete_from_db() (api.book.BookModel method), 25
 delete_from_db() (api.listing.ListingModel method), 28
 delete_from_db() (api.user.UserModel method), 32
 dispatch_request() (api.book.Book method), 23
 dispatch_request() (api.book.BookList method), 24
 dispatch_request() (api.listing.allListings method), 30
 dispatch_request() (api.listing.Listing method), 27
 dispatch_request() (api.user.User method), 30
 dispatch_request() (api.user.UserList method), 31

E

email (api.user.UserModel attribute), 32

F

familyName (api.user.UserModel attribute), 32
 find_by_email() (api.user.UserModel class method), 32
 find_by_google_tok() (api.user.UserModel class method), 32
 find_by_isbn() (api.book.BookModel class method), 26
 find_by_isbn() (api.listing.ListingModel class method), 28
 find_by_listing_id() (api.listing.ListingModel class method), 29

G

get() (api.book.Book method), 23
 get() (api.book.BookList method), 24
 get() (api.listing.allListings method), 30
 get() (api.listing.Listing method), 27
 get() (api.user.User method), 30
 get() (api.user.UserList method), 31
 get_listings() (api.book.BookModel method), 26
 get_listings() (api.user.UserModel method), 32
 givenName (api.user.UserModel attribute), 32, 33
 google_tok (api.listing.ListingModel attribute), 28, 29

google_tok (api.user.UserModel attribute), 32, 33

I

imageURL (api.user.UserModel attribute), 32, 33

infoLink (api.book.BookModel attribute), 25, 26

isbn (api.book.BookModel attribute), 25, 26

isbn (api.listing.ListingModel attribute), 28, 29

L

Listing (class in api.listing), 26

listing_id (api.listing.ListingModel attribute), 28, 29

listing_json_w_book() (api.listing.ListingModel method), 29

listing_json_w_book_and_user() (api.listing.ListingModel method), 29

listing_json_w_user() (api.listing.ListingModel method), 29

ListingModel (class in api.listing), 28

listings (api.book.BookModel attribute), 26

listings (api.user.UserModel attribute), 32, 33

M

metadata (api.book.BookModel attribute), 26

metadata (api.listing.ListingModel attribute), 29

metadata (api.user.UserModel attribute), 33

method_decorators (api.book.Book attribute), 24

method_decorators (api.book.BookList attribute), 24

method_decorators (api.listing.allListings attribute), 30

method_decorators (api.listing.Listing attribute), 27

method_decorators (api.user.User attribute), 31

method_decorators (api.user.UserList attribute), 31

methods (api.book.Book attribute), 24

methods (api.book.BookList attribute), 24

methods (api.listing.allListings attribute), 30

methods (api.listing.Listing attribute), 27

methods (api.user.User attribute), 31

methods (api.user.UserList attribute), 31

N

name (api.user.UserModel attribute), 32, 33

P

parser (api.book.Book attribute), 24

parser (api.listing.Listing attribute), 27

parser (api.user.User attribute), 31

post() (api.book.Book method), 24

post() (api.listing.Listing method), 27

post() (api.user.User method), 31

previewLink (api.book.BookModel attribute), 25, 26

price (api.listing.ListingModel attribute), 28, 29

provide_automatic_options (api.book.Book attribute), 24

provide_automatic_options (api.book.BookList attribute), 24

provide_automatic_options (api.listing.allListings attribute), 30

provide_automatic_options (api.listing.Listing attribute), 27

provide_automatic_options (api.user.User attribute), 31

provide_automatic_options (api.user.UserList attribute), 31

publishedDate (api.book.BookModel attribute), 26

publishedDate (api.book.BookModel attribute), 25

put() (api.listing.Listing method), 27

Q

query_class (api.book.BookModel attribute), 26

query_class (api.listing.ListingModel attribute), 29

query_class (api.user.UserModel attribute), 33

R

representations (api.book.Book attribute), 24

representations (api.book.BookList attribute), 24

representations (api.listing.allListings attribute), 30

representations (api.listing.Listing attribute), 27

representations (api.user.User attribute), 31

representations (api.user.UserList attribute), 31

S

save_to_db() (api.book.BookModel method), 26

save_to_db() (api.listing.ListingModel method), 29

save_to_db() (api.user.UserModel method), 33

smallThumbnail (api.book.BookModel attribute), 25, 26

status (api.listing.ListingModel attribute), 28, 29

subtitle (api.book.BookModel attribute), 24, 26

T

thumbnail (api.book.BookModel attribute), 25, 26

timestamp (api.listing.ListingModel attribute), 28, 29

title (api.book.BookModel attribute), 24, 26

U

user (api.listing.ListingModel attribute), 28, 29

User (class in api.user), 30

user_json_w_listings() (api.user.UserModel method), 33

user_json_wo_listings() (api.user.UserModel method), 33

UserList (class in api.user), 31

UserModel (class in api.user), 31