Mibrary

Technical Report

Group:
Edwin Xeon Gutierrez
Debarshi Kundu
Lucy Liu
Austin Mager
Diemi Pham
Aaron Saldana

CS 373
Summer 2018
Glenn Downing

Contents

1 Motivation

The motivation of the Mibrary project is to provide an avenue for students to donate or exchange their textbooks. This will help students who have a financial barrier to their education, and in turn be of community service.

2 User Stories

Phase 1

User Story 1: There are many textbooks provided by our traders. When a user needs to know what textbooks we offer, the application displays the list of the textbooks exist in the system. Assuming that the user does not give any filter criteria for listing the textbook, so all of the textbooks will be shown.

Estimate time: 1 hour

Actual time: 1 hour

User Story 2: When a user clicks on a textbook from the textbook list displayed on the screen, the system will show the user information of that textbook includes its ISBN number, title, author, edition, and cover photo.

Estimate time: 1 hour

Actual time: 1 hour

User Story 3: Users are able to know what classes they can take at UT and information about the classes such as the classes' numbers, titles, and required textbooks. The system will display the list of classes and users can get any class's information by clicking on the class from the displayed list. Assuming users do not give any filter criteria for listing available classes.

Estimate time: 1 hour

Actual time: 1 hour

User Story 4: Users are able to know who is trading (donating or requesting) textbooks across the system and which textbooks are corresponding to each trader. Assuming traders are human.

Estimate time: 1 hour

Actual time: 1 hour

User Story 5: Users are able to know the methods to contact traders via contact information that traders post in the system. Users can get any trader's contact information when they click on the trader from the displayed user list.

Estimate time: 1 hour

Actual time: 1 hour

Phase 2

User Story 6:

Allow the users functionality to create an account on the website. This account should come with both a username and password. There should be security measures that dictate how strong the password should be.

Implemented only in the backend. Frontend implementation would come in a later version of Mibrary, as user creation and password management are advanced features.

Estimated time: 5 hours

Actual time: 2 hours

User Story 7:

Users can add textbooks that they own and initiate trading with other users. Each user account will have a list of books in ownership under the user. Any user can search for this list of books by simply typing in the username.

Implemented the list of books offered and requested by the user. Users adding textbooks would modify the database and so is an advanced feature that would come in a later version of Mibrary.

Estimated time: 7 hours

Actual time: 2 hours

User Story 8:

A user should be able to enter his or her own textbook data on our website, regardless of whether or not the textbook is available already on our website. This textbook will then be added to the user's list of books, which can be searched by other users.

Implemented in the backend, but not in the frontend. Similar to the previous user story, adding a textbook would modify the database and so is an advanced feature that would come in a later version of Mibrary.

Estimated time: 8 hours

Actual time: 1 hour in backend, not implemented in frontend.

User Story 9:

Users should be able to add textbooks they are interested in purchasing to their own account. A given seller can then search for users with a specific textbook they are looking to purchase, and make an offer.

The ability to add books to a user account is implemented in the backend, but not currently implemented in the frontend until user accounts can be created.

Estimated time: 4 hours

Actual time: 1 hour in backend, not implemented in frontend

User Story 10:

Users should be able to report suspicious content from other users. This is similar to how Facebook posts can be reported, and users can be blocked by other users on Facebook.

Currently not implemented to full functionality.

Estimated time: 5 hours

Actual time: ongoing

Phase 3

User Story 11:

Given a subject by the user, return a list of books filtered by that subjects.

Rather than getting a list of all the books stored in the system, users can narrow down the search results by filtering the search result such as filtering by subject of a book.

This feature is currently fully implemented as an option for filtering.

Estimated time: 3 hours

Actual time: 3 hours

User Story 12:

Given the name of an author of a book, title of a book, isbn of a book, or course for a book, return a list of books fulfilling those parameters. The search can return one book or multiple books for given constraints.

This feature is fully implemented as an option for searching and filtering.

Estimated time: 4 hours

Actual time: 5 hours

User Story 13:

Given a list of books or a list of search results, return the list in alphabetical order. This allows the user to quickly select the book he or she wants by simply scrolling down to the letters the book title begins with.

This feature is fully implemented as an option for sorting.

Estimated time: 3 hours

Actual time: 2 hours

User Story 14:

Given a department at the University of Texas, return a list of courses under that department, and links to their list of textbooks.

This is fully implemented on the website, for all the course data to which we currently have access.

Estimated time: 3 hours

Actual time: 4 hours

User Story 15:

Given a name of an individual or a major, search for the requested user(s).

This feature is fully implemented as an option for searching and filtering.

Estimated time: 4 hours

Actual time: 3 hours

3 RESTful API

The parameters of the API design requirement were vague and ambiguous at first. It was unclear if we were required to map the API calls that we were consuming to build our database, or the ones that will ultimately be provided to the front end and other users of Mibrary, or both. Once the requirements were cleared up, this part became relatively simple.

The Mibrary API is a RESTful API which can be used by developers to retrieve information about the books which are requested, offered, or traded by the users of Mibrary as well as information about the trades between the users.

Details of Mibrary RESTful API can be found in this documentation.

Endpoints

The API is hosted under https://mibrary.me/api

The internal API URL is: http://ec2-18-191-216-158.us-east-2.compute.amazonaws.com:5000/api/

Endpoint	Description
/users	Get all users in the system
/user/<username>	Get information of a user given the username
/user/create-user	Create a new user
/books	Get all books in the system
/book/isbn/<isbn>	Get a book detail given an ISBN
/book/<query>	Get books that match the given query
/book/author/<author>	Get books by given author
/book/title/<title>	Get books by given title
/book/course/<course_id>	Get books by given course_id
/book/delete-book	Delete a book from the system
/reviews/<isbn>	Get all reviews of a book given its ISBN
/meeting/<meeting_id>	Get a meeting detail given the meeting's id
/course/<course_number>	Get information of a course given its course number
/courses/institution/<institution>	Get all course offered by an institution given institution code
/course/department/<department>	Get list of course given department
/requests	Get all book requested by the users in the system
/requested-book/<isbn>	Get a requested-book detail given ISBN of the book
/requested-book/add-request	Create a new request for a book

/offers	Get all book offered by the users in the system
/offered-book/<isbn>	Get an offered-book detail given ISBN of the book
/offered-book/add-offer	Create a new offer for a book
/reports	Get all book reported-books by the users in the system
/reported-book/<isbn>	Get a reported-book detail given ISBN of the book
/reported-book/add-report	Create a new report for a book

Status codes and Error messages

In the JSON response, the field status indicates the status of the response as well as error messages which can be “OK” if everything is fine or “INVALID_ID” if the given id was not found in the database.

3.1. API requests for User

Get list of users

Example request:

```
curl --request GET --url https://mibrary.me/api/users
```

Response format

Returns a JSON file with a list of all users; each user will have the information shown in the table below.

Get details of a user given username

Example request:

```
curl --request GET --url https://mibrary.me/api/user/pdiemi
```

Get detail of a user given the user’s major

Example request:

```
curl --request GET --url https://mibrary.me/api/user/major/utcs
```

Add a new user

Example request:

```
curl --request POST --url https://mibrary.me/api/user/add-user
```

Response format

Returns a JSON file with information of a user.

Field	Description	Datatype
id	User ID	integer
username	Username of the user	string
email	Email of the user	string
major	Major of the user	string

3.2. API requests for Book

Get list of books

Example request:

```
curl --request GET --url https://mibrary.me/api/books
```

Get list of books required by a course given the course_id

Example request:

```
curl --request GET --url https://mibrary.me/api/book/course/2
```

Response format

Return a JSON file with a list of book; each book will have information as shown in the table below.

Get details of a book given its isbn

Example request:

```
curl --request GET --url https://mibrary.me/api/book/isbn/9781934759
```

Get details of a book given its author

Example request:

```
curl --request GET --url https://mibrary.me/api/book/author/Downing
```

Get details of a book given its title

Example request:

```
curl --request GET --url https://mibrary.me/api/book/title/python
```

Get detail of a book given its subject

Example request:

```
curl --request GET --url https://mibrary.me/api/book/subject/java
```

Add a new book

Example request:

```
curl --request POST --url https://mibrary.me/api/book/add-book
```

Delete a book

Example request:

```
curl --request DELETE --url https://mibrary.me/api/book/delete-book
```

Response format

Return a JSON file with information of a book as following.

Field	**Description**	**Datatype**
isbn	ISBN number of the book	string
publisher	Name of book publisher	string
identifier	Type of ISBN	string
classification	System of organizing book	string
title	Name of the book	string

subtitle	Accompanying Title Name	string
url	Online location of the book	string
pages	Number of pages	string
subjects	Genre or topic of the book EG:Nonfiction, Art, etc…	string
date published	Date of publication	date
excerpts	Samples of writing from the book	string
author	The first writer of the book	string

3.3. API requests for Meeting

Example request:

```
curl --request GET --url https://mibrary.me/api/meeting/3
```

Response format

Return a JSON file information of a meeting specified by the meeting ID.

Field	**Description**	**datatype**
meeting_id	ID of the meeting	integer
time	Date and time of the meeting	string
location	Location of the meeting	string

3.4. API requests for Course

Get list of course

Example request:

```
curl --request GET --url https://mibrary.me/api/course/utaustin
```

Response format

Return a JSON file with a list of courses specified by the institution; each request has information as shown in the table below.

Get details of a course by course number

Example request:

```
curl --request GET --url https://mibrary.me/api/course/cs373
```

Get details of a course by department

Example request:

```
curl --request GET --url https://mibrary.me/api/course/undeclared
```

Response format

Return a JSON file with information of a course given its course name as shown below.

Field	**Description**	**Datatype**
Course ID	ID of Course	integer
Course #	Number of Course	string
Course Name	Name of Course	string
Institution ID	ID of the Institution	integer

3.5. API requests for Request

Get list of requests

Example request:

```
curl --request GET --url https://mibrary.me/api/requests
```

Response format

Return a JSON file that contains a list of requests; each request has the following information as shown in the table below.

Get details of a request given an isbn

Example request:

```
curl --request GET --url https://mibrary.me/api/requested-book/9781934759
```

Add a new request

Example request:

```
curl --request POST --url https://mibrary.me/api/requested-book/add-request
```

Response format

Return a JSON file information of a request as shown below.

Field	**Description**	**datatype**
date	Date of request	date
book_id	ISBN of the requested book	string
user_id	ID of user who requests the book	integer
meeting_id	ID of the meeting	integer

3.6. API requests for Offer

Get list of offers

Example request:

```
curl --request GET --url https://mibrary.me/api/offers
```

Response format

Return a JSON file that has a list of offers; each offer has the following information as shown in the table below.

Get details of an offer given an isbn

Example request:

```
curl --request GET --url https://mibrary.me/api/offered-book/9781934759
```

Add a new offer

Example request:

```
curl --request GET --url https://mibrary.me/api/offered-book/add-offer
```

Response format

Return a JSON file with an offer with following information as shown in the table below.

Field	**Description**	**datatype**
date	Date of offer	date
book_id	ISBN of the offered book	string
user_id	ID of user who offers the book	integer
meeting_id	ID of the meeting	integer

3.7. API requests for Report

Get list of reports

Example request:

```
curl --request GET --url https://mibrary.me/api/reports
```

Response format

Return a JSON file a list of report; each report has the following information as shown in the table below.

Get details of a report given an isbn

Example request:

```
curl --request GET --url https://mibrary.me/api/reported-book/9781934759
```

Add a new report

Example request:

```
curl --request GET --url https://mibrary.me/api/reported-book/add-report
```

Response format

Return a JSON file with a report with following information as shown in the table below.

Field	**Description**	**datatype**
date	Date of report	date
book_id	ISBN of the reported book	string
user_id	ID of user who reports the book	integer
meeting_id	ID of the meeting	integer

4 Models

4.1 Book

Attribute	**Description**
title	Name of the book
subtitle	Accompanying Title Name
isbn	ISBN number of the book
publisher	Name of book publisher
identifier	Type of ISBN
classification system	System of organizing book
url	Online location of the book
pages	Number of pages
subjects	Genre or topic of the book EG:Nonfiction, Art, etc…
date published	Date of publication
excerpts	Samples of writing from the book
author	The first writer of the book

4.2 Review

Attribute	***Description***

Review_id	ID of the review
Date	Date and time the review was written
Content	Content of the review.
Book_id	ISBN of book that the review is about
User_id	ID of user who wrote the review

4.3 Meeting

Attribute	***Description***
Meeting_id	ID of the meeting
time	Time of the meeting
location	Location of the meeting

4.4 User

Attribute	***Description***
User_id	ID of the user
Username	Username of the user
Password	Password of the user
Major	Major of the user

4.5 Course

Attribute	***Description***
Course_id	ID of the course
Course_number	Number of the course
Course_name	Name of the course
Institution_id	ID of the institution that offers the course
Department	Department that the course belongs to

5 Tools

5.1 Postman

Postman is the tool used to create documentation for the Mibrary API. Although Postman was a complicated tool in the beginning, its usefulness became apparent after working with it for longer. Postman allows us to easily document a restful API through creating and categorizing requests. These requests include the type, URL, description including sample usage, and parameters. A sample request can also be created and shown in a JSON or XML format.

5.2 Bootstrap

Bootstrap is the JavaScript and CSS framework that we used to enhance the user experience of our website. In order to implement Bootstrap, we integrated small pieces of scripting code into our html files. This code provides the link to a “CDN,” which is essentially a server with Bootstrap installed. Applying Bootstrap to all of our web pages ensured that the fonts were

aesthetically pleasing and easy-to-read. Furthermore, we were better able to organize our information.

5.3 Slack

Slack has proven essential for communication. Our entire team is available on a single slack channel, connected to our phones and computers, allowing us to be in constant communication. Through this mechanism, we are able to communicate, meet up, ask clarifying questions, and coordinate the shared assignment and division of tasks.

5.4 Git

Git has been an invaluable tool to allow multiple contributions to this project. We have set up a shared Mibrary repo with CI. We commit to the repository often, so we can always be working with the latest code. This allows all of us to update the same website codebase frequently, whenever individually we see a change that should be made.

5.5 MySQL

We used a MySQL database to store the data on the various models that are scraped from our API. MySQL commands allow our API to quickly and easily extract data from our database. A more detailed example of our models is found in the Models section.

5.6 Flask

Flask was the base for our backend, using Python. With Flask we spun up the backend of the site that connected to the database and exposed the Mibrary API for the React frontend to consume.

5.7 React

React is the JavaScript framework that we used for the frontend of the website. There was an initial time to learn how to use React, but after a while the site began to come together. After a few hours, the initial site was ported to React. React contains multiple components, allowing different pages and pieces of the website to be reused and displayed conveniently.

5.7.1 React-Router

React-Router is the library for React that allows linking to different pages. The site must be under a <BrowserRotuer> tag and should be loaded through a Component that has a series of Switch tags to load different pages depending on the path being linked to by Link tags in other pages. Once installed, and after looking through a tutorial, this library was relatively easy to understand and use.

5.8 Docker

We used Docker to create images for the front and back end of the website. This allowed us to all have a common version and set of tools for the website and allowing it to run consistently both on the Amazon hosting, and part of the GitLab CI.

5.9 Mocha

Mocha was the testing library used to test our frontend JavaScript code. Although it was rather confusing to set up, eventually tests were created and integrated with the GitLab CI.

5.10 unittest

Unittest is the Python unit testing framework to test the backend code. These tests will also be integrated with the GitLab CI.

5.11 Selenium

Selenium was used to test the GUI and interface. These tests were acceptance tests to verify that the site does what the users want. These weren't programmatic tests of individual program functions.

5.12 Visual Paradigm: Plant UML

PlantUML was used to generate and plan the features and characteristics of the data that is stored in the backend database.

6 Hosting

6.1 Namecheap

Our domain name was acquired from Namecheap. To use this domain with Amazon hosting, we used Namecheap's functionality to change the CNAME record to map our domain to the domain provided to us by AWS.

6.2 AWS

In the first phase, to host our static website, we used Amazon's AWS S3 service, a cloud-based static content storage service. There was some initial confusion regarding how to navigate Amazon's AWS service, but following the step by step process found online here was very helpful. After registering an account with AWS, the S3 console can be found at https://console.aws.amazon.com/s3/. We created buckets for both the mibrary.me and www.mibrary.me domains, and uploaded the same files to both. When uploading files, all the permissions must be set to public readable by everyone. Finally, to enable website hosting, static website hosting must be enabled in the properties of each bucket.

For the second phase, we used an Amazon EC2 instance to which we deployed 2 docker images, one for our Flask backend, and the other for our React frontend. We had the two images point to different ports, 3000 for the frontend, and 5000 for the backend API. This way we can access both of our sites from the same instance. We then consumed data from the backend port in our frontend code. Using namecheap, we redirected the two ports to two different URLs, mibrary and mibrary/api.

8 Pagination

Pagination was relatively simple to implement through the use of the React framework. Each page has a state that can be updated in different functions, and the variables included in the state can be rendered as part of the HTML rendered in the render call. Our state for the model pages included a list: models, an int: current page, and an int: pageModelCount. When the page number was clicked, we updated the currentPage count. pageModelCount was the amount of model links that would be displayed on each page. When the page itself was loaded, we would get the list of

models from the Mibrary API and give those to the page state as the models list. When a search or filter is concluded, the user will be returned to the first page of results, and the total number of pages will be updated.

10 Filtering

We implemented filtering across the 3 models in a fairly generalized way across the three models: Book, User, and Course. Currently each model has two filters from which users of the site can filter the models.

Book:

- Subject
- Author

Course:

- Department
- Course Number

User:

- Email
- Major

The user can leave the filters blank in order to not filter any models.

When the filter button is pressed, the Model class performs a procedure similar to the search function which returns a new list of all models for which the filterCondition function returns true. Each Model class has a specific filterCondition function that returns true if the property of the model being filtered matches with the user's given filter term. This new list of models is then assigned to the list of models to display in the state and rendered into the page.

11 Searching

Searching was implemented in a Google like manner in the individual model pages and the home page. As the user types in the searchbox, we update the list of displayed models by checking to see which names contain the phrase being searched by the user. Since we check the objName property, which exists for every type of Model, the search code is written once and works for Book, Course, and User. All of the models have the search phrase in their name highlighted in bold text. On the main page, each model name is preceded by the type of model, allowing the user of the site to quickly and easily differentiate between the different types of results for their search phrase. The Model state contains references to two lists: a list containing the list of every possible model in the database, and a second list containing the list of models to be displayed. The list of models to be displayed is continuously updated with models from the first list that pass the searchCondition test as the user searches.

12 Sorting

We implemented sorting in both ascending and descending order for each of the model pages, as well as for the mix of all 3 models on the home page. This sort is generalized for every model, the code for which is all contained in the Model class, which is a superclass of Book, Course, and User. Our sorting mechanism sorts the list of displayed models using the built in JavaScript array function, and one of two comparator comparison functions, depending on what the user selected. The Ascending and Descending comparators take two individual models and compare the two based on the "objName" property, which exists for every model. Books are sorted by their title. Courses are sorted by their course name. Users are sorted by username. To accomplish this, the "objName" property points to the title for a Book, to the course name for a Course, and to the username for a User. This way, Models of different specific types can all be sorted together by their "name".

13 DB

The UML diagram shown below describes how the tables are laid out in our database.

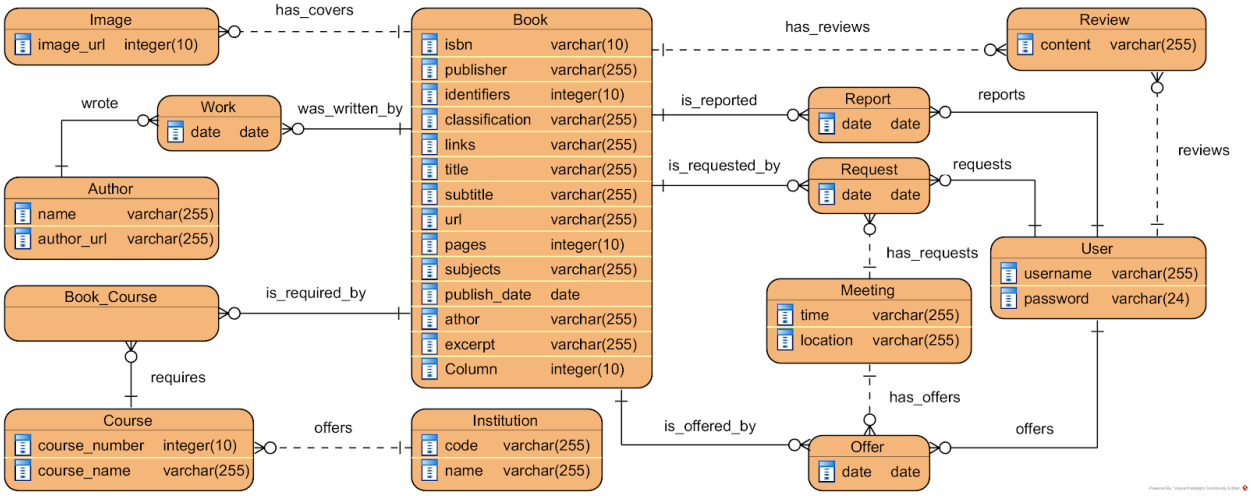

Figure: Mibrary UML model

Our database is formed by eight main models such as Author, Book, Course, Image, Institution, Meeting, Review, and User. Those models contain information that will be displayed on the model pages of the website and hold relationships to other models. Besides those eight models, our database also includes five other models that are Course_Book, Offer, Report, and Work. These five models are association classes which describe the many-to-many relationships between the eight main models.

Association classes explanations:

- Course_Book: Each book can be required by different courses and vise versa, each course is able to require more than one book.
- Offer: An offer is a connection between a meeting, a user, and a book. Indeed, a meeting consists of two users and books; each offer gives information about a book and the user who offers the book as well as the meeting that is involved
- Report:
- Request: A request, like an offer, is a connection between a meeting, a user, and a book. Indeed, a meeting consists of two users and books; each request gives information about a book and the user who requests the book as well as the meeting that is involved.

Work: In fact, a book can be a collaborated work of many authors and an author writes many different books. A work implies the relationship between an author and a book; that means this specific book was written by this author and this author wrote this book.

14 Testing

We used four different testing mechanisms to test our website. To unit test our API we used Postman, as well as testing the results from the frontend. To unit test our backend we used the unittest framework for Python. This involved testing our API scraping calls and our own API, as well as accessing the database. To unit test the frontend we used mocha with JavaScript. We tested our calls to our API and our loading of data from that API on the frontend. Lastly we made acceptance tests for our GUI using Selenium. These test the general GUI that the user will interact with. Additional mocha tests were created to deal with testing the searchCondition and filterCondition functions under various scenarios to ensure that no matter what Model type is used, the filters and searches will return the correct results.

15 Charts