COMP2011

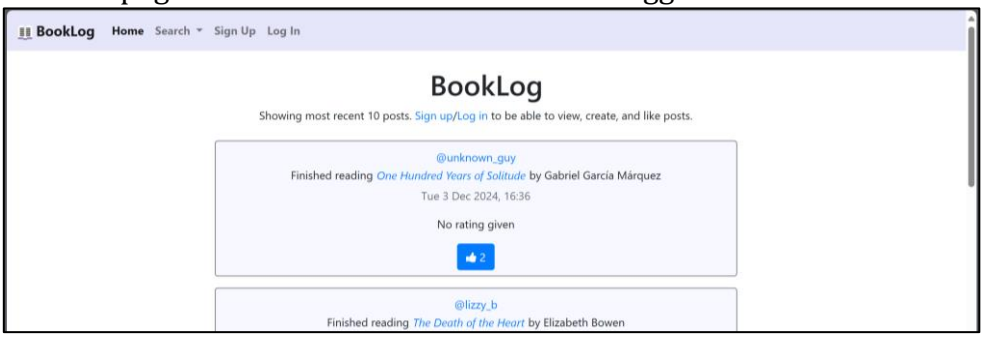
Hoe Yin Foong

My Website: BookLog

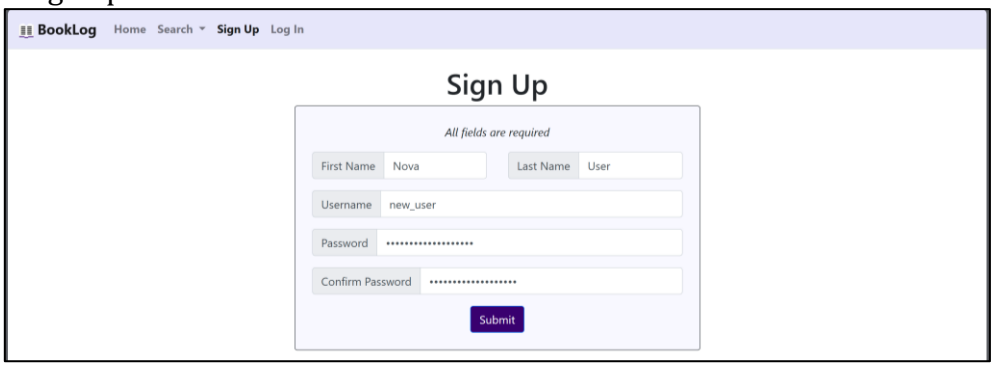
BookLog is a basic social media site that lets users rate and review books that they've read. Users look up different books and add them to a reading list or mark that they've read them. Users can also follow one another to view each other's activity, and can like each other's posts.

User Path

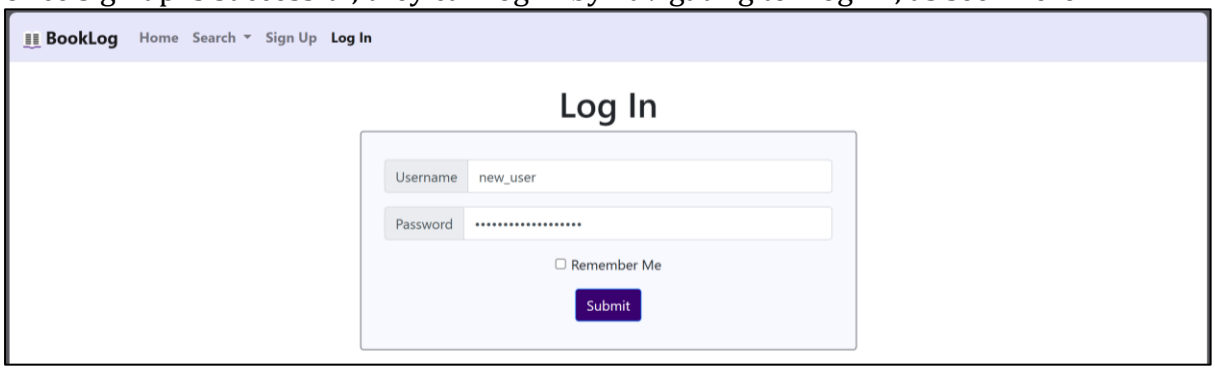
This is the home page of the site when the user is not logged in:



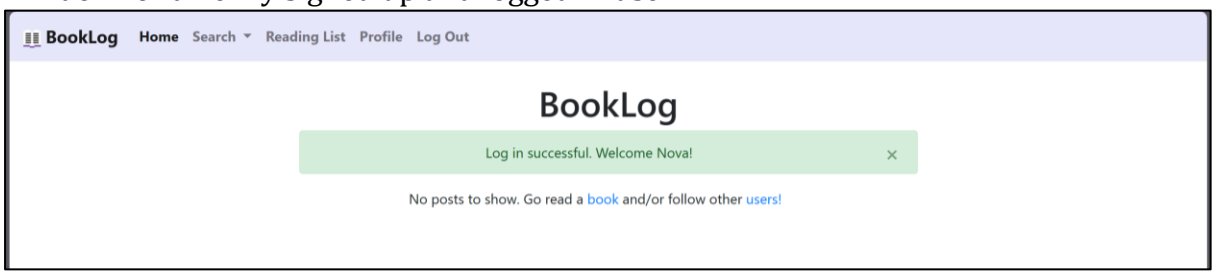
The user can then sign up by navigating to the 'Sign Up' page. This is an example of a filled out sign up form:



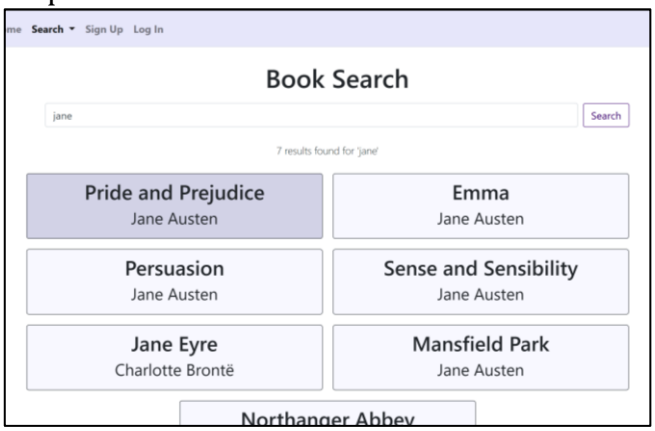
Once sign up is successful, they can log in by navigating to 'Log In', as seen here:

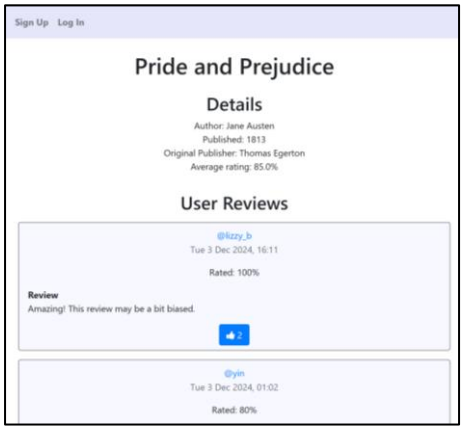


Window for a newly signed up and logged in user:

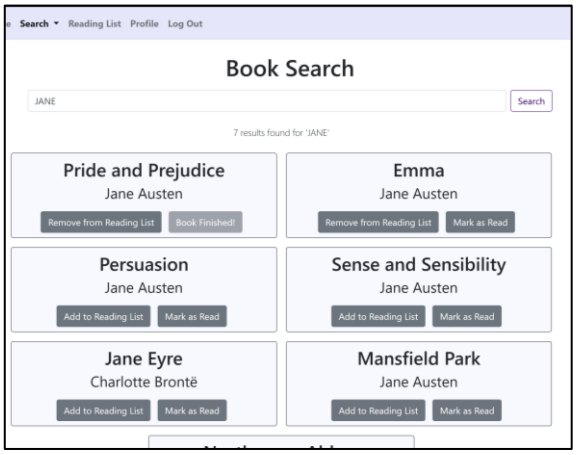


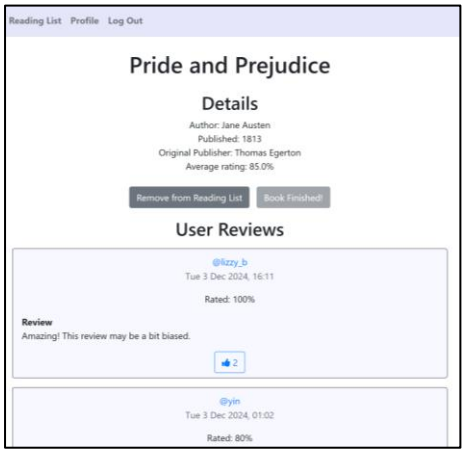
Any user (logged in or not) can open the sub-navigation “Search” and select “Books” to look up books to see their basic information and user reviews:





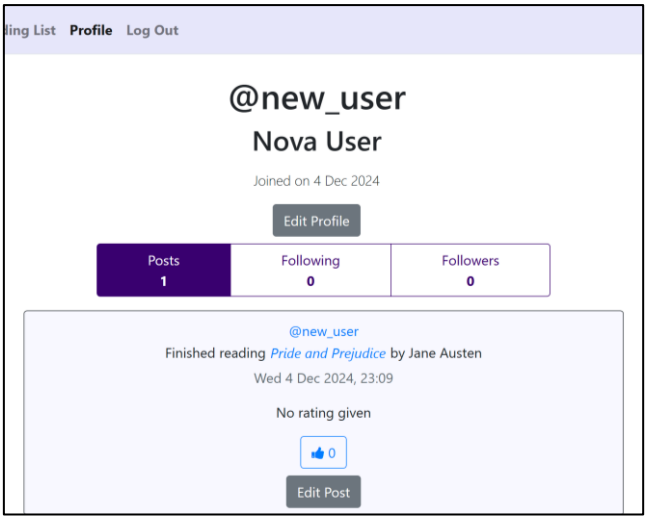
If the user is logged in, they can either add a book to their reading list, or mark it as finished. The respective views for a logged in user are below. Note that they can also like other users’ reviews now.



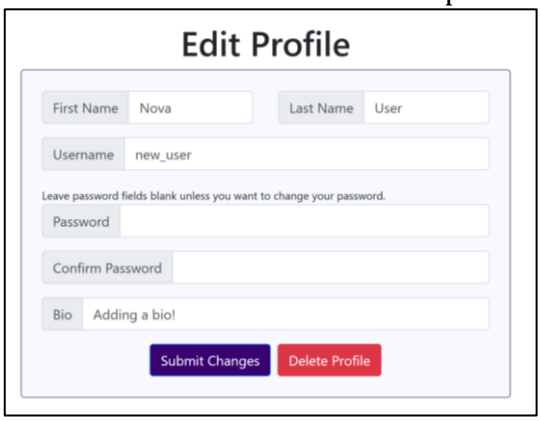


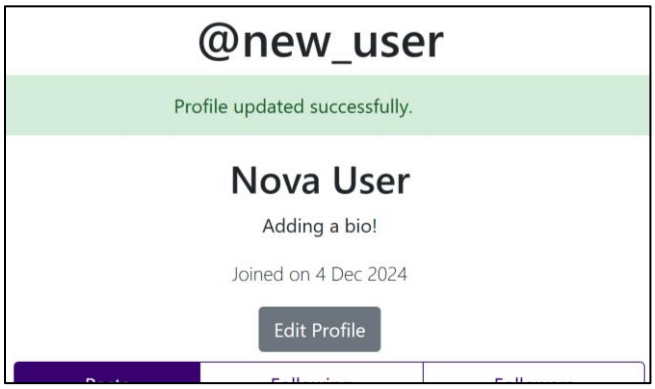
There is a similar situation for searching for different users, using “Search -> Users”. You can search and view user profiles when not logged in; you can follow, unfollow, and like posts when logged in.

When logged in, users can see all their profile’s information in their profile page (by navigating to ‘Profile’ in the navigation bar):

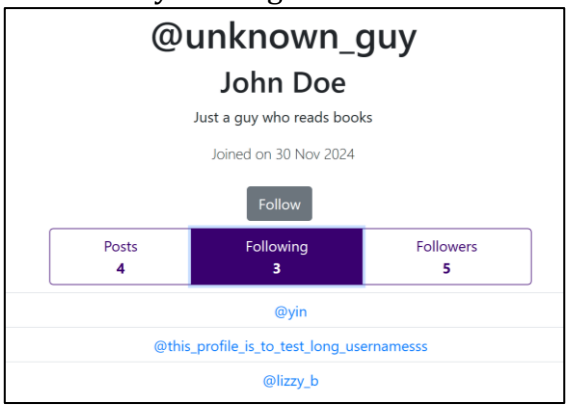


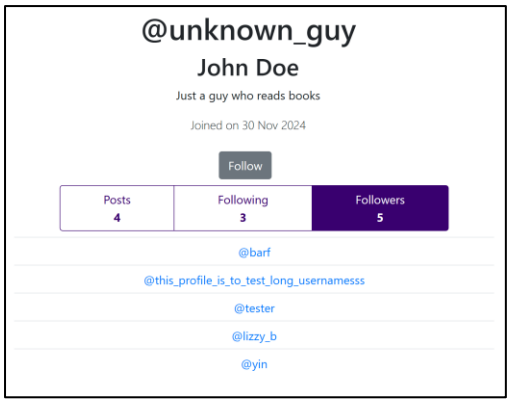
The user can edit or delete their profile by clicking on 'Edit Profile':



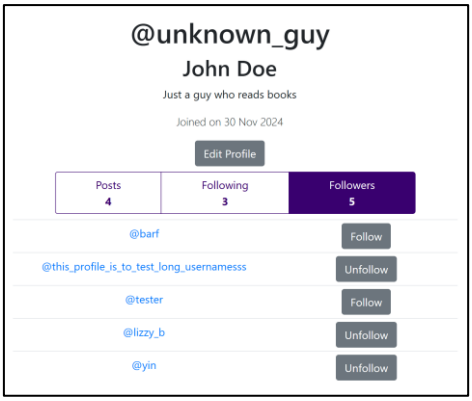


Anyone viewing the profile (logged in or not) can see the user's posts, followed users, and followers by clicking the radio buttons on their profile page. For example:

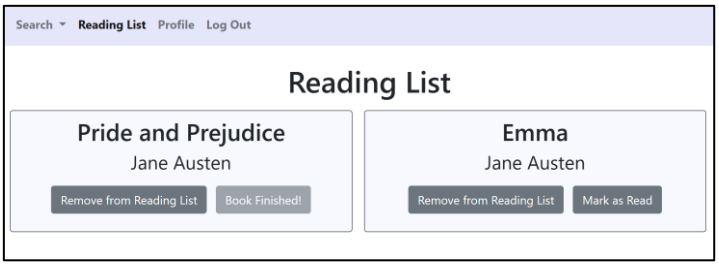




When logged in, users can unfollow and follow other users directly from their own profile page:

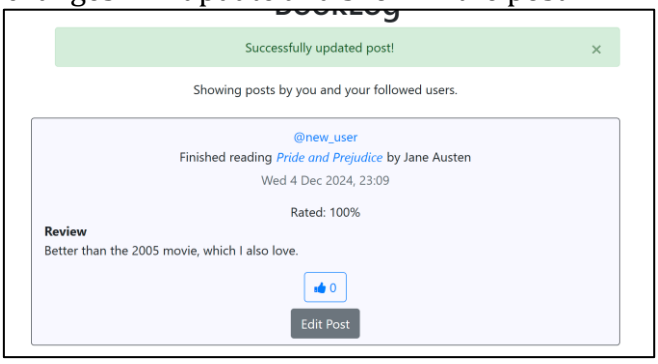


Logged in users can view their reading list by clicking on ‘Reading List’ in the navigation bar:



When a user marks a book as finished, a post is automatically created. Users can then edit or delete the post by clicking the ‘Edit Post’ button on it:

And then submitted changes will update and show in the post:



As seen above, pages for books and user profiles can be navigated to using the blue links in posts. Usernames are almost always displayed as a clickable link to indicate that the user’s profile can be seen by clicking on it.

Finally, a user can log out by clicking on ‘Log Out’ in the navigation bar.

Design

I wanted the design to be simple and straightforward. I tried to base the layout off standard industry practices, such as ordering posts in reverse-chronological order and how a user's profile is displayed. The intent is to make the site intuitive to use.

I also made it so that every time a user's username is shown, it can be clicked to view their profile. This was done using a hidden form that is submitted every time a relevant button (which is displaying the username) is clicked, passing along the correct user id:

```
<form class="d-none" action="/profile" id="profile-form" method='POST' aria-hidden="true"></form>
<button class="btn view-profile p-0" form="profile-form" name="view" value="{{ book.User.id }}">@{{ book.User.username }}</button>
```

This provides users with the convenience of being able to view a user's profile directly without having to search up their username in the User Search. It also matches common behaviour other social media sites, which helps the site feel more familiar to users. Something similar is also done with book titles.

I wanted the results of my Book Search to be 'clickable', as in the entire div should act as a button. I thought this would be more concise than having a separate 'View' button, and would make the results look neater. One challenge I had was that when a user is logged in, the results also display buttons within the div to add/mark a book. I had to solve it by excluding the buttons from the jQuery click event that runs when the div is clicked, shown below.

```
/* Redirect to view book when div is clicked
Learned from: https://stackoverflow.com/questions/10851312/document-click-not-in-elements-jquery*/
$("div.book-info").click(function(e){
    var id = $(this).attr("id");
    if(!$(e.target).is("button")){
        $(this).parent().children("button[value="+id+"]").trigger("click");
    }
});
```

Additionally, I had to change the CSS styling of the div so that it wouldn't change when the user was mousing over a button. This makes the option the user is hovering over less confusing.

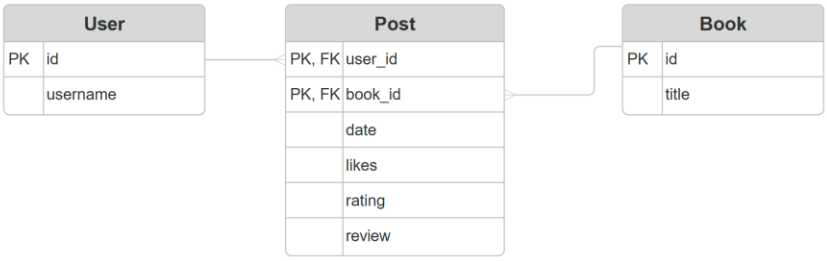
```
div.book-info:hover:not(:has(button:hover)) {
    background-color: #d2d2e6;
    cursor: pointer;
}
```

Finally, along with making each result tabbable using tabindex="0", I had to implement jQuery that redirected the user whenever they clicked 'Enter' while selecting a search result.

```
/* So that highlighted div can be clicked on 'Enter' press */
$(document).on('keydown','div.book-info',function(e){
    if(e.which==13)
        $(this).click()
});
```

Database

There are three many-to-many relationships in the relational model. There is a self-referential many-to-many relationship for users following other users, and two many-to-many relationships between users and books for their reading lists and posts. I will be detailing the latter.



Each user can read multiple books, and each book can be read by multiple users. The entity relationship diagram above shows this relationship. I made the choice to make every book that a user 'finishes' to automatically be a post. This is modelled using a one-to-many relationship between User and a Post entity, and then a many-to-one relationship from Post to Book. A user creates multiple posts (about different books) and a book has multiple posts (by different users).

The use of Post also allows for additional information to be stored, like exactly when the user marked it as complete or the number of likes it has. Other fields are initially null until the user manually edits a post to give a rating or review.
This relationship forms the core function of the site and enhances user experience by letting them add and interact with their posts in an uncomplicated manner. By using the Post entity to store all relevant data, it can easily be queried to produce relevant results. For example, viewing a book's page will show a book's average rating, and any posts that have a non-empty rating or review.

Accessibility

ARIA tags were added for elements that need them, such as hidden elements, buttons, forms, and navigation. This helps assistive technologies, such as screen readers, to render the pages. An example would be these forms used to pass different ids to their respective pages. They aren't actually displayed to the user, so they need the attribute aria-hidden="true":

```
<form class="d-none" action="/profile" id="profile-form" method='POST' aria-hidden="true"></form>
<form class="d-none" action="/pass_book_id" method="POST" id="pass-book-id" aria-hidden="true"></form>
```

Another important feature is using tabindex="0" (shown below) on certain elements to allow users to navigate to different posts or search results using the 'tab' button. This ensures that the site is functional using only keyboard input.

```
{% for book in read %}
    <div class="col-10 col-md-8 border border-dark rounded p-2 m-2 post" tabindex="0">
```

This jQuery detects when a user presses 'Enter' while selecting a book result in Book Search.
This allows a user to search and view for a book using only their keyboard.

```
/* So that highlighted div can be clicked on 'Enter' press */
$(document).on('keydown','div.book-info',function(e){
    if(e.which==13)
        $(this).click()
});
```

There is also sufficient contrast between text and background colours, as shown in all previous screenshots. This makes it so that users can read text clearly without strain.

Advanced Feature

I used AJAX and Flask-RESTful to implement a like system for posts. This allows the user to like a post and have the button visually update to reflect this, without needing the page to reload.

The HTML/Jinja for a pressed like button:

```
<button class="btn btn-primary like-button"
id="{{book.Post.user_id}}-{{book.Post.book_id}}"
aria-label="{{ book.Post.likes }} likes" name="unlike" aria-pressed="true">
<em class="fa fa-thumbs-up"></em> {{ book.Post.likes }}</button>
```

The important attributes are name=”unlike” and id=”{{book.Post.user_id}}-{{book.Post.book_id}}. These are retrieved by the jQuery event below in ‘ajax_js.js’:

```
/* Like/unlike post */
$(".like-button").click(function () {
    $.ajax({
        url: '/like_post',
        type: 'POST',
        data: JSON.stringify({ id: $(this).attr("id"), action: $(this).attr("name") }),
        contentType: "application/json; charset=utf-8",
        dataType: "json",
```

```
@app.route('/like_post', methods=['GET', 'POST'])
@login_required
def like_post():
    """Like/unlike post."""
    data = json.loads(request.data)
    ids = data.get('id').split("-")
    post_user_id = int(ids[0])
    post_book_id = int(ids[1])
    post = models.Post.query.get([post_user_id, post_book_id])

    if data.get('action') == "like":
        # Add to user's liked posts and increment likes
        insert = models.Liked(user_id=current_user.id,
                              post_user_id=post_user_id,
                              post_book_id=post_book_id)
        post.likes += 1
        db.session.add(insert)
    else:
        # Remove from user's liked posts and decrement likes
        delete = models.Liked.query.get([current_user.id,
                                         post_user_id,
                                         post_book_id])
        post.likes -= 1
        db.session.delete(delete)

    db.session.commit()

    return json.dumps({'status': 'OK', 'id': data.get('id'),
                       'action': data.get('action')})
```

The button’s id and name are then passed to ‘/like_post’ in ‘views.py’ in json format:

The button’s id consists of a User.id and Book.id that can be used to query the specific pot that is being liked.

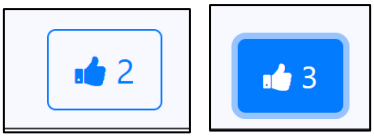
The button’s name indicates whether clicking it will add or retract a like. The code will either insert or delete the current user’s like for this specific post into the Liked model. The code then either increments or decrements a like from the post.

```
success: function (response) {

    // Change like button and update count
    var likes = Number($("#" + response.id).html().slice(-1));
    if ($("#" + response.id).attr("name") == "like") {
        $("#" + response.id).removeClass("btn-outline-primary").addClass("btn-primary");
        $("#" + response.id).attr("name", "unlike");
        $("#" + response.id).attr("aria-pressed", "true");
        likes = likes + 1;
    } else {
        $("#" + response.id).addClass("button-outline-primary");
        $("#" + response.id).removeClass("btn-primary").addClass("btn-outline-primary");
        $("#" + response.id).attr("name", "like");
        $("#" + response.id).attr("aria-pressed", "false");
        likes = likes - 1;
    }
    $("#" + response.id).html("<em class='fa fa-thumbs-up'></em> " + likes);
```

The ajax function back in ‘ajax_js.js’ then switches the button to its other state and updates the number within it so the user can see the visual update. All relevant attributes are also updated, such as the button’s name and aria-pressed value.

Example of clicking a like button:



A like button is a fun staple of a social media site. It allows users to easily interact with one another in a minimal but positive way.

Other AJAX features implemented:

- Clicking the 'Follow' or 'Unfollow' buttons will change their appearance and update the database without reloading the page. The user's profile will also update the following/follower sections of its page if needed.
- Similarly, the user can add a book to their reading list or mark the book as finished without needing to reload the page.

AJAX allows for a more responsive user experience and makes the site feel faster, as the browser does not have to reload the page to process and update any changes made.

A final advanced feature was the search function implemented for both books and users.

```
def format_title(search):
    """Format a search to follow standard book title format."""
    lowercase = ["a", "an", "the", "but", "for", "or", "nor", "of", "at", "on",
                 "yet", "in", "and"]
    words = search.split(" ")
    words[0] = words[0].lower().capitalize()
    words[len(words)-1] = words[len(words)-1].lower().capitalize()
    for i in range(1, len(words)-1):
        words[i] = words[i].lower()
        if words[i] not in lowercase:
            words[i] = words[i].capitalize()
    return words
```

The method format_title() shown to the left will format each word in a string (which was entered into the search bar) to match standard book title format.

```
# Query by exact title
results = models.Book.query.filter_by(title=" ".join(words))

# Query by if author contains exact search term when longer than 2
if len(request.form['search']) > 2:
    authors = models.Book.query.filter(
              Book.author.like("%"+request.form['search']+"%"))
    results = results.union(authors).group_by(Book.id)

# Query by if title or author contains a significant search term
for i in range(len(words)):
    if len(words[i]) > 3:
        title = models.Book.query.filter(
                Book.title.like("%"+words[i]+"%"))
        author = models.Book.query.filter(
                 Book.author.like("%"+words[i]+"%"))
        results = results.union(title).union(author).group_by(Book.id)
```

The '/books' view can then perform different queries (shown to the left) and join the together to generate a final list of all books that contain a relevant match to the user's search. There is use of SQLAlchemy's like() function to find matching strings in the database.

A similar method is implemented for the user search. These methods give the user the convenience to search for results they want without needing to display either every single entry or only exact matches.

Testing & Critical Analysis

Testing

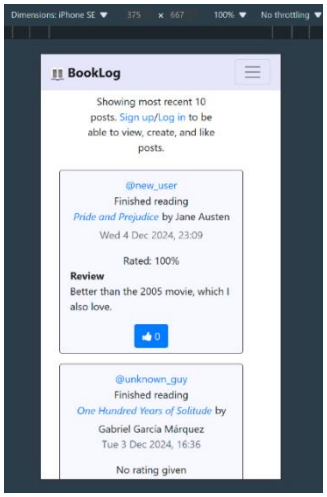
Functionality

I came up with a list of test cases and intended behaviours to check that my site was functioning properly. Below is an example of just testing the Log In page.

Test	Expected
Create a new user account correctly	New user account created.
Create an account with a username that's already in use	No user account created.
Create an account with a username that has whitespace.	No user account created.
Create an account with empty fields	No user account created.
Create a new user account with short password (<12 length)	No user account created
Create user account with passwords not matching	No user account created
Trying to access the Sign In page when already logged in	Redirected to Home page.

I also used flask-admin to easily modify and view my database when testing my code. This let me check and compare what I wanted my queries to do with what was really happening. I had to implement specific model views to view my many-to-many relationships as well. Flask-admin has been removed in the final application to maintain database security.

Layout & responsiveness



As shown in the image in the right, I used Microsoft Edge's developer tools to emulate different devices; this let me test how the site would work on different screen sizes and with only touchscreen input.

I also resized my browser's window to test how the site would react with live changes in display.

Most elements reacted fine as I used Bootstrap to design my site, which is made to be responsive by nature. The main edits were to add breakpoints for display elements so they would take up an appropriate amount of space at different screen sizes.

Accessibility

I used https://webaim.org/resources/contrastchecker/ to make sure there was sufficient contrast between all text and background colours.

I also ran through my test cases for functionality using only keyboard input to make sure that the site was functional without a mouse. This let me recognize the need to make certain elements 'tabbable' using the HTML attribute "tabindex='0'" so that users can select them when using a keyboard.

Finally, I downloaded NVDA to test how text-to-speech software would render the site, which helped me identify which elements needed better ARIA labelling to reduce confusion and make the site navigable using the software. For example, I had to add `aria-label="{{ post.Post.likes }} likes"` to each like button. This lets the screen reader read out the number of likes (e.g. “5 likes”) in a more natural way than without it, since the actual like button uses a picture to represent likes.

Analysis

What Went Well

As described above, I managed to implement the results of the Book Search to display as a bunch of s behaving as buttons. The result is clean and simple, which allows users to parse results quickly without the distraction of another visual button.

I am also pleased with the like system. It was a good way to get familiar with AJAX and Flask-RESTful implementation. It provides good user interactivity, and is also an integral part of any social media site; the site would feel like it lacked something without it, so I’m glad that it was executed well.

What Went Badly

It took me a while to implement the AJAX for the ‘Follow’ and ‘Unfollow’ mechanics. The initial struggle was implementing the actual self-referential many-to-many relationship in my database, which was confusing. Then I had to implement AJAX to check for a variety of conditions, which would then hide or display certain pre-loaded HTML elements in the page. It took a while to cover every case where a user could follow or unfollow another user. Reading the various documentation and help online carefully allowed to eventually learn from and fix my issues.

I populated my Book table manually, instead of finding a pre-existing database to insert into it. It was time consuming, and I ended up with less books (only roughly 300) in my database than I would have liked. Using Python to insert another database would not have been hard or take nearly as much time.

Improvements

The app could be easily expanded to include a variety of things. I would like to add the ability to comment on a post. This would allow for more interactivity between users on the social media site and provide a place for discourse.

I would also add better data regarding the books, such as their genre. This could be used to implement a better search function, allowing users to search by genre. A recommendation system could also be made that gives users suggestions based on what they’ve previously liked reading. Adding a proper ISBN code could also allow users to identify the specific book they want, and find where to buy or borrow it.